M68SOSEK

Motorola Microcontroller Families

OSEK

Operating System

USER'S MANUAL

MOTOROLA

OSEK Operating System

USER'S MANUAL

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Employment Opportunity/Affirmative Action Employer.

Legal Notices

The information in this document has been carefully checked and is believed to be entirely reliable, however, no responsibility is assumed for inaccuracies. Furthermore, Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Motorola does not assume liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights or the rights of others.

The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement.

© Copyright Motorola, Inc., 1997. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means - graphic, electronic, electrical, mechanical, chemical, including photocopying, recording in any medium, taping, by any computer or information storage retrieval systems, etc., without prior permissions in writing from Motorola Inc.

Permission is granted to reproduce and transmit the Problem Report Form, the Customer Satisfaction Survey, and the Registration Form to Motorola.

IMPORTANT NOTICE TO USERS

While every effort has been made to ensure the accuracy of all information in this document, Motorola assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Motorola further assumes no liability arising out of the application or use of any product or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. Motorola disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties of merchantability or fitness for a particular purpose.

TRADEMARKS

Microsoft, MS-DOS and Windows are trademarks of Microsoft.

UNIX is a trademark of AT&T Bell Laboratories.

Cosmic is a trademark of COSMIC Software.

Table of Contents

Paragraph Number **Title** **Page Number**

Table of Contents

Paragraph Number **Title** **Page Number**

Table of Contents

Table of Contents

Paragraph Number **Title** **Page Number**

Table of Contents

Paragraph Number | **Title** | **Page Number**

Table of Contents

Paragraph Number **Title** **Page Number**

Table of Contents

Table of Contents

List of Illustrations

Figure Number **Title** **Page Number**

List of Illustrations

Figure Number	Title	Page Number

List of Tables

List of Tables

Table Number	Title	Page Number

SECTION 1
OVERVIEW

OSEK[1] Operating System (OSEK OS) is a real-time operating system conforms the specification of the OSEK Operating System v.1.00.

OSEK OS conforms the following requirements:

- OS is fully configured and scaled statically;
- OSEK OS is a ROM-able system, i.e. the OS code may be executed from Read-Only-Memory. OSEK OS may be placed into the chip memory during manufacturing time and users' applications may be added during development time;
- OS performance parameters are well known;
- Being written in strict correspondence with ANSI C standard, the OS and application on its basis can be easily ported from one platform to another.

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

The OSEK 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 granted to 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.

---

1. The term OSEK means "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" (Open systems and the corresponding interfaces for automotive electronics). A real-time operating system, software interfaces and functions for communication and network management tasks are thus jointly specified within the OSEK standard.

Interrupt Management

- Services for hardware interrupt flag manipulations;
- Frames for interrupt service routines.

Resource Management

- Mutually exclusive access control for inseparable operations to jointly used resources or devices, or for control of a program flow.

Communication

- State Message: Data exchange without buffering;
- Event Message: Data exchange with buffering.

Counter and alarm management

- Counter management provides services for execution of recurring events;
- Alarm management is based on counter management. It enables entry of (cyclic) alarm requests. The expiration of a preset relative counter value, or the fact that a preset absolute counter value is reached, results in activating of a task, or setting a task event.

Error treatment

- Mechanism supporting the user in case of various error classes.

OSEK Operating System is scaled in two ways - either by changing the set of system services or via the so-called Conformance Classes. They are available to satisfy different requirements concerning functionality and capability of the OS. These Conformance Classes do not only differ concerning the number of services they provide, but also with regards to their capabilities and scalability. The classes are based on one another in upwardly compatible fashion. The Conformance Classes are generated by meaningful grouping of services (see section **3.2 Conformance Classes**).

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

OSEK Operating System is well documented and measured. In the User’s Manual all system mechanisms, particularities, services and programming techniques are described in detail with numerous examples. Numbers for performance characteristics and memory requirements are provided.

SECTION 2
NOTATION

2.1 Manual Structure

This User's Manual consists of the following sections:

SECTION 1 Overview describes what the OSEK OS is and highlights its basic features.

SECTION 2 Notation contains the description of the Manual structure, typographical conventions and the list of acronyms.

SECTION 3 Operating System Architecture gives the high level description of OS architecture and presents OS Conformance Classes.

SECTION 4 Task Management explains the task concept in OSEK and all other questions related to tasks.

SECTION 5 Scheduler provides the description of scheduling policies in OSEK OS.

SECTION 6 Interrupt Processing highlights OSEK approach to interrupts handling.

SECTION 7 Resource Management describes resource management and task coordination by resources.

SECTION 8 Counters and Alarms describes usage of these control mechanisms in OSEK OS.

SECTION 9 Events is devoted to event management and task coordination by events.

SECTION 10 Communication describes message concept in OSEK and their usage.

SECTION 11 Error Handling and Special Routines describes support provided to the user to debug an application and handle errors.

SECTION 12 System Configuration describes possible OSEK OS versions, configuration options and the configuration mechanism.

SECTION 13 Building of Application contains information on how to build an user's application using OSEK OS. It also describes memory requirements.

SECTION 14 Platform-Specific Features discusses special OSEK OS features for different MCU types and issues connected with porting applications to these MCUs.

SECTION 15 Application Troubleshooting contains useful information for debugging applications developed using OSEK OS.

SECTION 16 System Services provides a detailed description for all OSEK Operating System run-time services, with appropriate examples.

APPENDIX A Sample Application contains the text and listing of a sample customer applications developed using OSEK OS.

APPENDIX B System Service Timing provides information about OS services execution time.

APPENDIX C Memory Requirements provides information about the amount of ROM and RAM directly used by various versions of the OSEK OS.

APPENDIX D System Generation Error Messages explains OSEK OS System Generator error messages.

APPENDIX E System Services Quick Reference briefly lists all OSEK OS run-time services, with entry and exit conditions.

2.2 Typographical Conventions

This manual employs the following typographical conventions:

boldface type

Bold is used for important terms, notes and warnings.

Italics

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

Courier font

Courier typeface is used for code examples in the text.

Courier Italic

Courier Italic typeface is used for OSEK terms when these are first introduced.

2.3 Definitions, Acronyms and Abbreviations

API	Application Program Interface (a set of data types and functions)
BCC	Basic Conformance Class, a defined set of functionality in OSEK, for which the waiting state of tasks is not permitted
BT	Basic task (task, which never has the waiting state)
CPU	Central Processor Unit
ECC	Extended Conformance Class, a defined set of functionality in OSEK, for which the waiting state of tasks is permitted
ECU	Electronic Control Unit (similar to MCU)
ET	Extended Task (task, which may have the waiting state)
ID	Identifier, an abstract identifier of a system object
ISF	Interrupt Stack Frame, a stacked model of CPU registers, produced by CPU hardware and/or software instructions during CPU interrupt
ISR	Interrupt Service Routine
MCU	Microcontroller Unit (Motorola’s microcontrollers)
MO	Message object
OS	Operating System, a part of the OSEK
OSEK	Open systems and their corresponding interfaces for automotive electronics (in German)
RAM	Random Access Memory
ROM	Read Only Memory
SG	System generator

2.4 Installation Instructions

This is OSEK Operating System version 1.0 RP release.

2.4.1 Required Environment

This version of the product is to be used on IBM PC 486 (and higher) compatible. To install and evaluate the product about 8 MB disk space is required.

The PC has to work under MS Windows 3.1x, MS Windows 95 or Windows NT 3.51 during OSEK installation.

To use OSEK OS after installation the Cosmic Software version 4.0p should be installed on the computer. It is necessary to call the DOS prompt under Windows NT to run the MAKE utility.

All supplied makefiles are developed for the Microsoft C++ NMAKE utility. For Windows 3.1x it is possible to use Microsoft C++ v1.5 NMAKER utility instead NMAKE.

2.4.2 Installation

To setup OSEK on your hard disk:

1. Insert the installation disk into a 3.5” floppy drive. The following instructions assume that the drive letter is **a:**. If another drive is chosen, substitute that drive letter where appropriate.
2. Run the A:\SETUP.EXE program either from the File Manager or Program Manager.
3. Follow the prompts and instructions of the installation program.
4. After installation verify the consistency of the package by means of comparing the real set of the files and directories with the list in FILELIST.TXT file.

After installation, the hard drive should contain the root directory of OSEK which will contain a set of files in the following subdirectories:

• BIN	System Generator
• INC	Operating System header files
• SRC	Operating System source files
• SAMPLE	OSEK Operating System Sample application
• MAN	User’s Documentation

There is FILELIST.TXT file in the OSEK root directory. This file contains a complete list of files which are included in this release.

Besides, the root directory contains README.TXT file, which provides additional information for the user.

2.5 Technical Support Information

To order any additional information or to resolve arising problems contact with Motorola ZAO - St. Petersburg Branch Office, Russia:

Europa House, 1 Artilleriyskaya Street,
St.Petersburg, 191104, Russia

Phones: (089) 92 103 308 (Munich, Germany)
+7 (812) 329-1910 (St.Petersburg, Russia)
Fax: (089)92 103 101 (Munich, Germany)

+7 (812) 329-1912 (St.Petersburg, Russia)

E-mail: R21898@email.sps.mot.com
rfrance@sdt.sps.mot.com
rte-users@sprl.sps.mot.com

Developers: Maxim Chervinsky, Irene Bratanova, Ivan Gumenyuk, Alexander Semjonov, Vladimir Belov

SECTION 3
OPERATING SYSTEM ARCHITECTURE

3.1 Processing Levels

The OSEK 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 OSEK OS enables a controlled real-time execution of several processes which virtually run in parallel.

The highest processing priority is assigned to the interrupt level, where interrupt service routines (ISR) are executed. Interrupt services may call a number of operating system services. The processing level of the operating system has a priority ranking immediately below the former one. This is the level on which the operating system works - task management procedures, scheduler and system services. Just below this 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 **4.1 Task Concept**). The run time context refers to resources which are occupied at the beginning of execution time and are released again once the task is finished.

3.2 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) demand different stages of the operating system. These operating system stages are described as *`Conformance Classes`* (CC). They differ in the number of services provided, their capabilities and different types of tasks.

An application which has been written for a certain Conformance Class is executable in any higher Conformance Class (upwardly compatible). The Conformance Classes are based upon one another. This applies to the functionalities provided and to the availability of a minimum number of resources (e.g. minimum number of available priorities). Any higher Conformance Class is defined so that it also covers the functionalities and minimum equipment of the previous one.

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

The definition of the functionalities provided by each Conformance Class depends on the properties of the tasks and on the scheduling behavior. As the task properties (Basic or Extended, see **4.2 Task State Model**) have a distinct influence on CC, they also resume part of their names. There are Basic-CC and Extended-CC, and each of these groups can have various “derivatives”. The following Conformance Classes, which are based upon one another, are defined: *`BCC1, BCC2, BCC3, ECC1, ECC2`*.

- **BCC1** - only Basic tasks, limited to one request per task and one task per priority, while all tasks have different priorities;
- **BCC2** - like BCC1, more than one task per priority possible;
- **BCC3** - like BCC2, additionally multiple activation admissible;
- **ECC1** - like BCC3, plus Extended tasks without multiple activation admissible;
- **ECC2** - like ECC1, plus multiple activation possible for Extended tasks.

BCC1 defines the minimum Conformance Class of the OSEK OS, ECC2 defines its maximum CC.

`Multiple activations` means that the OSEK OS receives and records activations (even multiple activations) of a task already activated. On terminating the task, the system checks whether activations have already been noted. If so, the task being terminated is reactivated immediately.

Table 3–1 presents minimum resources to which an application may resort and which have been determined for each Conformance Class.

Table 3–1 OSEK Operating System Conformance Classes

	BCC1	BCC2	BCC3	ECC1	ECC2
Multiple activation of tasks	no		yes	BT: yes, ET: no	YES
Number of tasks which are not in the *suspended* state	≥8			≥ 16, any combination of BT/ET	
Number of tasks per priority	1	>1		BT: >1, ET: 1	>1
Number of events per task	-			BT: no ET: ≥ 8	
Number of priority classes	≥8				
Resources	only Scheduler		≥ 8 resources (including Scheduler)		
Alarm	≥ 1 single or cyclic alarm				
Messages	possible				

The system configuration option ***ConformanceClass*** (specified by the user) defines the class of the overall system. This option can have values *BCC1*, *BCC2*, *BCC3*, *ECC1*, *ECC2* (see **12.3 System Properties Definition**).

It is impossible to have tasks of different Conformance Classes in one application! All tasks must strictly conform the Conformance Class specified at the system configuration stage.

3.3 OSEK OS Overall Architecture

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

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

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

As you have seen in **Table 3–1** , Conformance Classes, in general, differ in the degree of services provided for the task management and scheduling (number of tasks per priority, multiple requesting, Basic/Extended Tasks). In higher CC advanced functionality is added for resource management and event management only. But even in the lowest BCC1 the user is provided with merely all OSEK OS service mechanisms.

The OSEK Operating System is scaled not only via Conformance Classes but it also has many various extensions which can be in any Conformance Class. These extensions affect memory requirements and overall system performance. The extensions can be turned on or turned off with the help of the corresponded system configuration options. All them are described in **SECTION 12 System Configuration**.

Since the OSEK 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 the special file. These statements fully describe the desired system features and application object's parameters. SG produces C-code that is to be compiled together with other user's source code. The produced code consists of C-language definitions and declarations of data as well as C-preprocessor directives. See **SECTION 12 System Configuration** and **SECTION 13 Building of Application** for details about system generation.

3.4 Application Program Interface

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

OSEK OS data types are described in subsections dedicated to the corresponded mechanisms. Syntax of system calls and system configuration statements are described briefly in corresponded subsections and in detail in **SECTION 12** and **SECTION 16** .

NOTE:

> The user’s source code shall strictly corresponds to the rules presented in this Manual.

The OSEK OS can has the `Extended Status`. It means that additional checking is made inside all OS activities and extended return codes are returned by all OS services to indicate errors if they are occurred. See **SECTION 16 System Services** and **11.2 Error Handling** about Extended Status return values. To provide the Extended Status in the system the configuration option ***ExtendedStatus*** must be turned on at configuration stage.

SECTION 4
TASK MANAGEMENT

4.1 Task Concept

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

OSEK OS provides a set of tools for the user to manage tasks. They are activated before their execution - memory resources needed for a task are allocated. Tasks may be terminated after needed actions were performed - memory resources are released. After a task is activated it may run and terminate or it may be implemented in an endless loop with at least one synchronizing system call. It is not possible to run several parallel endless loops without switching mechanism (*`scheduler`*). Tasks can be switched during their execution from one to another, or may be interrupted by ISR. If no task is active, only the scheduler idle loop runs (see **5.1 General**). Task states and transitions between them are discussed in section **4.2 Task State Model**.

Two different task concepts are provided by the OSEK OS:

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

Basic Tasks only release the processor, if:

- they are being terminated,
- the OSEK OS is executing higher-priority tasks, or
- interrupts occur which cause the processor to switch to an interrupt service routine.

Extended Tasks are distinguished from Basic Tasks by being allowed to use additional operating system services which may result in a *waiting* state. The *waiting* state allows the processor to be freed and to be reassigned to a lower-priority task without the need to terminate the Extended Task.

Both kinds of tasks have their advantages which must be compared application dependent. They are both justified and are supported by the OSEK operating system.

Each task have a set of data related to it - a task descriptor located in ROM (*`task configuration table`*) and a task control block in RAM (*`task node`*) for activated tasks. Also each task has its own stack assigned. See section **4.6 Task Related Resources** about the task control block, the task configuration table and the task stack.

Every running task is represented by its *`run time context`*. This refers to CPU registers and some compiler-dependent "pseudoregisters" in RAM. When the task is interrupted or preempted by another task the run time context is saved in the task's stack. Run time context differs for non-preemptive and preemptive tasks - for preemptive tasks more RAM is required to save the context. Therefore, for mixed-preemptive system (it means that both non-preemptive and preemptive tasks can exist in the system, see section **5.2 Scheduling Policy**) a distinction can be made by the Operating System between these types of contexts. It is controlled by the ***UseSameContext*** configuration option. In case if different contexts are saved for non-preemptive and preemptive tasks OS has to perform additional code to distinct the task type. Otherwise, some more RAM is required to store the "full" context for non-preemptive tasks too. The use of this option is application dependent.

4.2 Task State Model

A task can be in several states, as the processor can only execute one instruction of a task at any point in time, while several tasks may be competing for the processor at the same time. The OSEK OS is responsible for saving and restoring task context in conjunction with state transitions whenever necessary.

4.2.1 Extended Tasks

Extended Tasks have four task states:

• ***running***	In the *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 any point in time, while all the other states can be adopted simultaneously by several tasks.
• ***ready***	All functional prerequisites for a transition into the *running* state exist, and the task only waits for allocation of the processor. The scheduler decides which *ready* task is executed next.
• ***waiting***	A task cannot be executed (any longer), because it has to wait for at least one event (see **SECTION 9 Events**).
• ***suspended***	In the *suspended* state, the task is passive and does not occupy any resources, merely ROM.

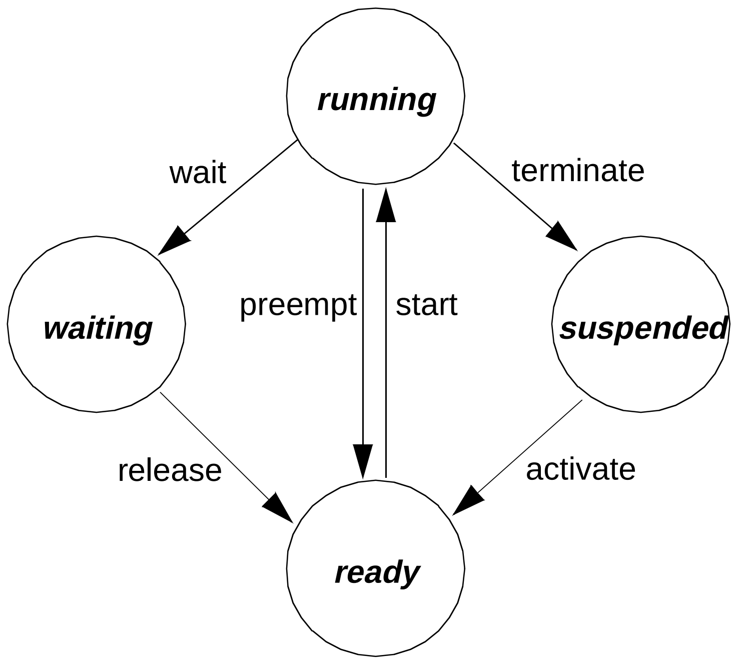

Figure 4–1 Task status model of an Extended Task with its task transitions.

Table 4–1 States and status transitions in the case of Extended Tasks

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 operation, the *running* task requires an event. It causes its transition into the *waiting* state by using a system service.
release	waiting	ready	Events have occurred which a task has waited on.
preempt	running	ready	The scheduler decides to start another task. The *running* task is put into the *ready* state.
terminate	running	suspended	The *running* task causes its transition into the *suspended* state by a system service.

Termination of tasks is only possible if the task terminates itself ("self-termination"). This restriction is to avoid complex book-keeping of resources dynamically allocated by the task.

There is no provision for a direct transition from the *suspended* state into the *waiting* state. This transition is redundant and would add to the complexity of the scheduler. The *waiting* state is not directly entered from the *suspended* state, as the task starts and explicitly enters the *waiting* state on its own.

4.2.2 Basic Tasks

The state model of Basic Tasks is nearly identical to the Extended Tasks state model. The only exception is that Basic Tasks do not have a *waiting* state.

• ***running***	In the *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 any point in time, while all the other states can be adopted simultaneously by several tasks.
• ***ready***	All functional prerequisites for a transition into the *running* state exist, and the task only waits for allocation of the processor. The scheduler decides which *ready* task is executed next.
• ***suspended***	In the *suspended* state, the task is passive and does not occupy any resources, merely ROM.

Table 4–2 States and status transitions in the case of Basic Tasks

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 the *ready* state.
terminate	running	suspended	The *running* task causes its transition into the *suspended* state by a system service.

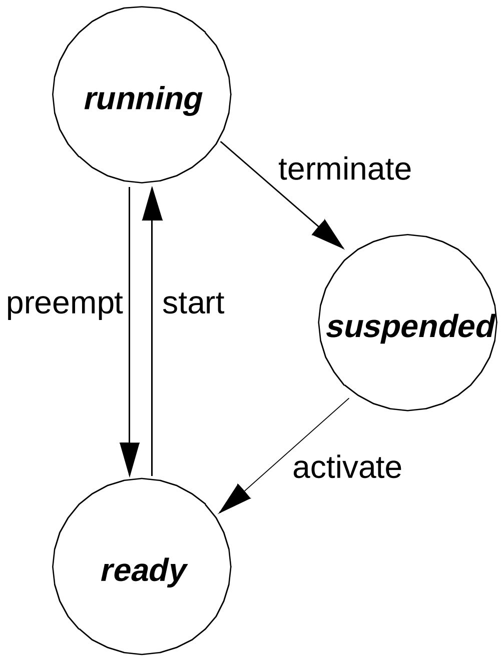

Figure 4–2 Task status model with task transitions for Basic Tasks.

4.3 Task Activation and Termination

Depending on the Conformance Class a task can be activated single or multiple. The difference between `single` and `multiple activation` is that:

- In the case of a task suited for **multiple activation**, all task activations are noted down. It means, that whenever a task is activated by the application, the appropriate task is started at the next possible time, even if it is already running while the request is issued. In this case, the request is saved by the operating system, so the task can be started again by the operating system once it has been terminated. This procedure is repeated several times until of all requests has been completed.
- In the case of a task suited for **single activation**, either the task activation is noted down, or, if the task has already been requested, nothing happens.

Multiple activation property exists only for BCC3, ECC1 and ECC2 Conformance Classes. It is possible to turn it off via the system configuration option ***MultiplyActivation***.

OSEK OS uses the scheme of task activation and termination like presented on **Figure 4–3**. This idea allows dynamic memory reallocation which provides economical resources consuming.

During task activation the *ActivateTask* service analyzes the task configuration table of the task to be activated, and allocates a stack buffer and a task node as it is specified in the task configuration table. In the figure the typical allocation method is shown - a stack buffer is retrieved from the stack pool and a task node from the scheduler's list of free task nodes. If these resources are available, then the task node is initialized and inserted into the scheduler's queues to be dispatched.

The *TerminateTask* service releases both the task node and the stack buffer and returns them to the system for re-use.

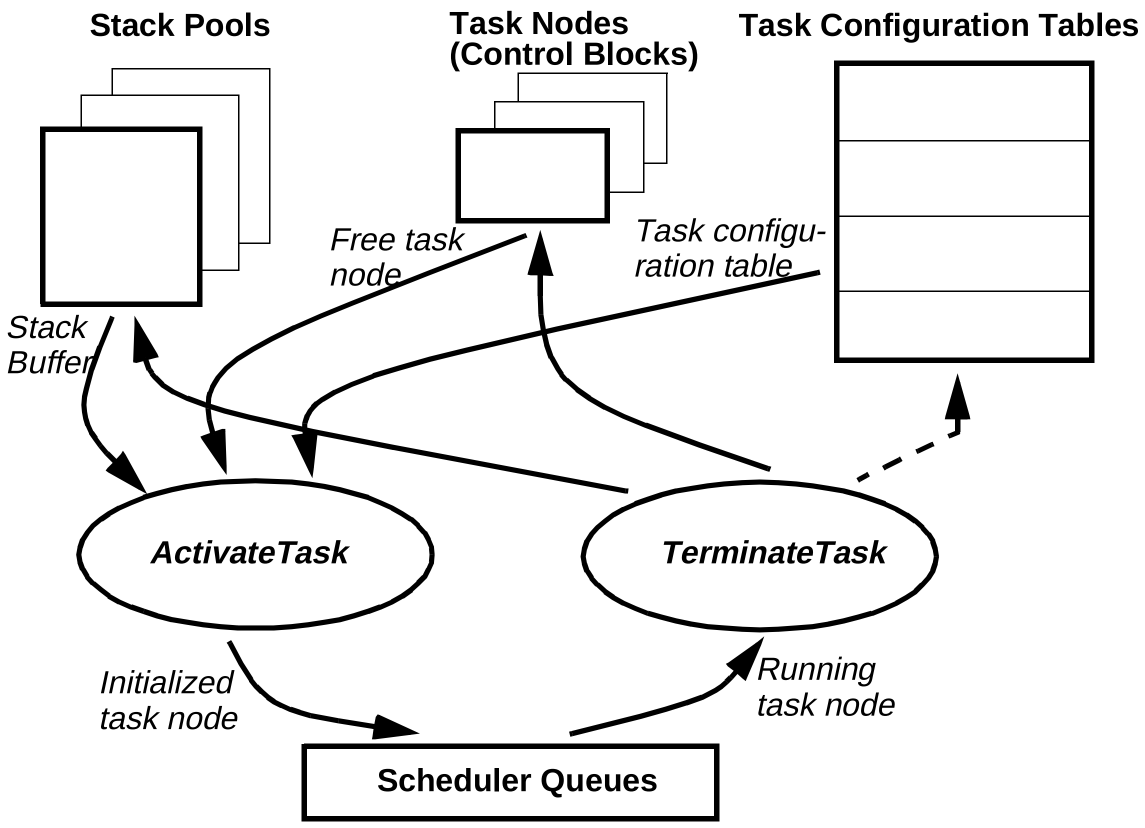

Figure 4–3 Task management architecture

This way task activation may be considered as dynamic resource allocation, because task control blocks and stack pools are allocated dynamically. This approach allows the optimization of using resources, because one resource (for instance, a task control block) may be used by different tasks at different moments of time.

But in case when requested resources aren't available, it may lead to indeterministic behavior of the operation. This situation may occur, when too many tasks are activated. To avoid that weakness, the user is also allowed to statically assign task control blocks and stack areas for critical tasks. Such explicitly defined resources are hold by a task for the application life time. In this case the task control block and/or the task stack are not released and cannot be re-assigned for other tasks.

4.4 Task Properties

In OSEK OS every task is characterized by the set of properties. These properties define the task behavior and resource allocation method. Each task property has its own name, and the user defines the task features by placing the desired symbolic constants in the *DefineTask* statement. These constant can be united via the 'OR' operator in *DefineTask*. See **SECTION 12** about task definition. The list of possible task properties is provided in **Table 4–3** .

Table 4–3 Task Properties

Property Name	Description
BASIC	Basic task
EXTENDED	Extended task
NONPREEMPT	Non-preemptive task
PREEMPT	Preemptive task
ACTIVATE	Activate the task at system start-up
ASSIGNNODE	A persistent task control block is assigned (linked with the task configuration tables)
POOLSTACK	A stack should be allocated to the task during task activation - dynamic stack allocation from the stack pool
OWNSTACK	A stack is explicitly assigned to the task (address of the top of the stack is included into the task configuration table)
NODESTACK	A task node stack is implicitly assigned to be used by the task (during the task activation)
ASSIGNSTK	A persistent stack from the stack pool (the pool identifier is included into the task configuration table)

4.5 Task Priorities

In OSEK OS a priority is statically assigned to each task and it cannot be changed by the user at the execution time. Some Conformance Classes permit several tasks of the same priority level (see **Table 3–1**). A dynamic priority management is not supported. However, in particular cases the operating system can treat a task with a defined higher priority. In this context, please refer to **7.2.1 Priority Ceiling Protocol**.

On the basis of the task priority the scheduler decides which is the next of the *ready* tasks to be transferred into the *running* state. Tasks are started depending on their order of activation according to the FIFO mechanism, whereby Extended Task in the *waiting* state do not block the start of subsequent tasks of identical priority.

A preempted task is considered to be the oldest task in the *ready* list of its current priority. A task being released from the *waiting* state is treated like the newest task in the *ready* list of its priority.

Figure 4–4 illustrates how tasks priorities affect on the order of their execution.

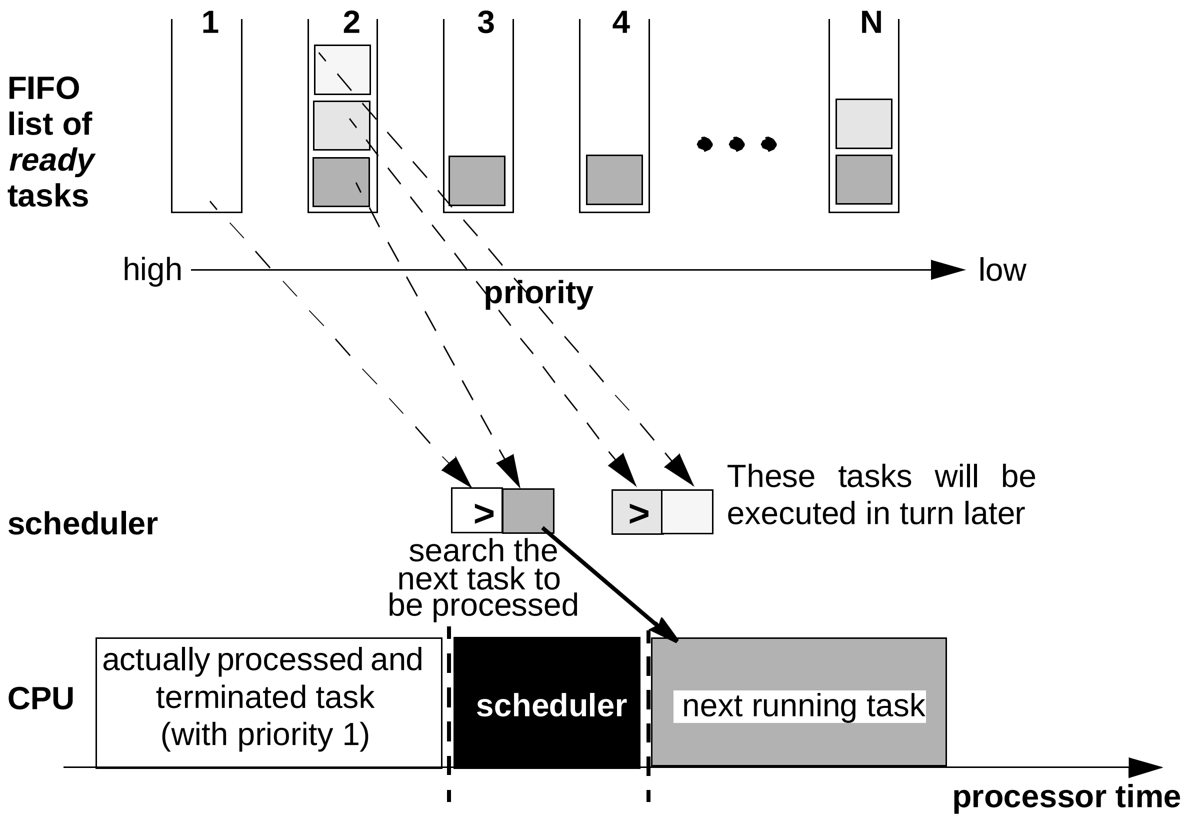
1
2
3
4
N
FIFO list of ready tasks
high
priority
low
scheduler
These tasks will be executed in turn later
search the next task to be processed
CPU
actually processed and terminated task (with priority 1)
scheduler
next running task
processor time

Figure 4–4 Task priorities

Several tasks of different priorities are in the *ready* state - three tasks of priority 2, one priority 3 and 4 tasks, and two priority N tasks. The tasks which has waited the longest time, depending of its order of requesting, is shown at the bottom of each FIFO queue. CPU has just processed and terminated a task with priority 1. The scheduler selects the next task to be processed (priority 2, first FIFO location). Before priority 3 tasks can be processed, all tasks of higher priority must have been executed completely.

Thus, the following three steps are made to switch to the next running task:

5. The scheduler searches for all tasks in the *ready* state.
6. From the set of *ready* tasks the scheduler determines the set of tasks with the highest priority.
7. Within the set of tasks in the *ready* state and of the highest priority the scheduler finds the task which has been in the *ready* state the longest time ("oldest" task).

4.6 Task Related Resources

Each task has a *task configuration table* in ROM and *task control block (task node)* and *task stack* in RAM (during the task run time, if they are not assigned statically). These resources uniquely define task properties and the task run-time context. Also at run-time the *task link table* exists in RAM to provide the correspondence between the task control block and the task configuration table.

4.6.1 Task Configuration Table

The ROM-based *task configuration table* contains a static task description. This table holds all information describing the tasks's properties and serves to initialize the RAM-based task control block during task activation.

The task configuration table contains the following data:

- Task properties (see **Table 4–3**) which define a run-time behavior of the task, as well as task activation parameters. The main properties are whether it is a Basic/Extended task (for the extended Conformance Classes only) or a preemptive/non-preemptive task (for a mixed-preemptive scheduling policy). The properties also specify the source of the stack for the task, and the assignment of the task control block;
- The starting priority of the task;
- The hardware memory bank of the task. The user is responsible to switch the hardware bank by means of using hooks (See section **11.1 Hook Routines**). If bank switching is not supported (the ***HCBankCode*** system configuration option is turned off) this field is absent in the task configuration table;
- The starting interrupt mask of the task. The field is absent if interrupt mask control is not supported;
- The program counter value (the entry point) of the task;

- The address of the persistent task node, that is a task node, which is permanently assigned to the task (see **4.6.2 Task Control Block**). This field is absent if persistent task control blocks are not supported in the system;
- The address of the task stack source, i.e. the address of the stack pool or the constant address of the own task stack (see **4.6.4 Task Stack**). This field is absent if all tasks use only node stacks.
- The address of the element of the task link table (see **4.6.3 Task Link Table**). This field is absent if the simplified scheduler is used in the system (see **5.1.1 Simple Scheduler**) or the system configuration option *TaskIndexMethod* is off.

Some of these parameters are specified by the user during system configuration by means of the *DefineTask* statement while others are generated automatically. See **SECTION 12** and **SECTION 13** about system configuration and building an application.

4.6.2 Task Control Block

A RAM-based `task control block (task node)` serves as a dynamic descriptor of the task. It is valid only during task execution, that is when the task is not in the suspended state. Except persistent node assignment (see **4.6.2.1**), this task control block is linked to the scheduler's list of free nodes when it is not assigned to the task. In general, this block is allocated to the task during task activation, and it is initialized at that moment (see **Figure 4–3**).

The task control block contains the following data:

- A pointer for linking the task control block into scheduler queues. There is no this field if the simplified scheduler is used in the system;
- The current task status. Some of the status bits are copied from the task properties bits of the task configuration table. Others bits are:

  — a task waiting bit, which is set when the task is waiting for an event(s);

  — a scheduler bit, which is set when the task occupies the scheduler.
- The current task priority. In general, the current priority may differ from the starting priority (located in the task configuration table) during the periods of time, when the task occupies the resources, see **7.2.1 Priority Ceiling Protocol**.This field is absent if the simplified scheduler is used in the system;
- A pointer to the task configuration table;
- The task context (for non-preemptive tasks), or a pointer to the stack frame, where the context is saved (for preemptive tasks). Note, that for mixed-preemptive scheduling systems both non-preemptive context and a pointer to the preemptive context are defined if the system configuration option ***UseSameContext*** is turned on. Otherwise, the preemptive context is used for non-preemptive tasks also;

- Two fields for event control in Extended Classes of the system. The first field contains the mask of the events the task is waiting for, while the second field contains the mask of the events, which are set for the task (see **SECTION 9 Events**). If event management is not supported, these fields are absent;
- If the system supports multiple activations of the task, the task control block contains the counter of times of activation. That is, the content of this field makes an increment each time, when the task to be activated is in the non-suspended state, and makes decrement each time, when the task is re-activated after terminating (if it is greater than zero during task termination);
- The pointer to the head of the list of resources occupied by the task (if resource management is supported and the system property ***FastResource*** is turned off, see **SECTION 7 Resource Management**);
- The address of the stack buffer, allocated to the task. It is used during task termination when task management returns this stack buffer into the stack pool. If stack pools are not supported in the system then this field is absent;
- The address of the task node stack or address of the persistent task stack if persistent node assignment is supported by the system. This field is absent if neither task node stacks nor persistent stacks are used;
- The real address of the top of the task stack. This field is absent if multiple activation is not supported and the system configuration option ***ChainTaskItself*** is turned off (see **12.3 System Properties Definition**).

The task control block size depends on the system configuration and task properties. Generally, the minimal task control block size is 5 bytes and the maximal size is 19 bytes.

4.6.2.1 Persistent Node Assignment

The task control block may be statically assigned to the task. This mechanism prevents a task activation failure due to missing free task nodes and increases the speed of task activation/termination operations by means of excluding a task node queues manipulation. This mechanism is called persistent task node assignment, and it is illustrated in **Figure 4–5** . To define persistent node assignment for the task the task property *ASSIGNNODE* shall be set in the *DefineTask* statement for this task and the system configuration option *PersistentNode* must be turned on.

As it is in the figure, task#2 has persistent task node#3, assigned to this task. Therefore, this task control block cannot be used by any other task in the system, and is not linked into the scheduler’s list of free nodes when task#2 is not activated (not used). This task node is exclusively assigned for use by task#2. On the other hand, task#1 may use any of task nodes #1, #2 or #4, if they are free while task#2 is activated.

Persistent task node assignment may be used in combination with any of the stack allocation options (see **4.6.4 Task Stack**).

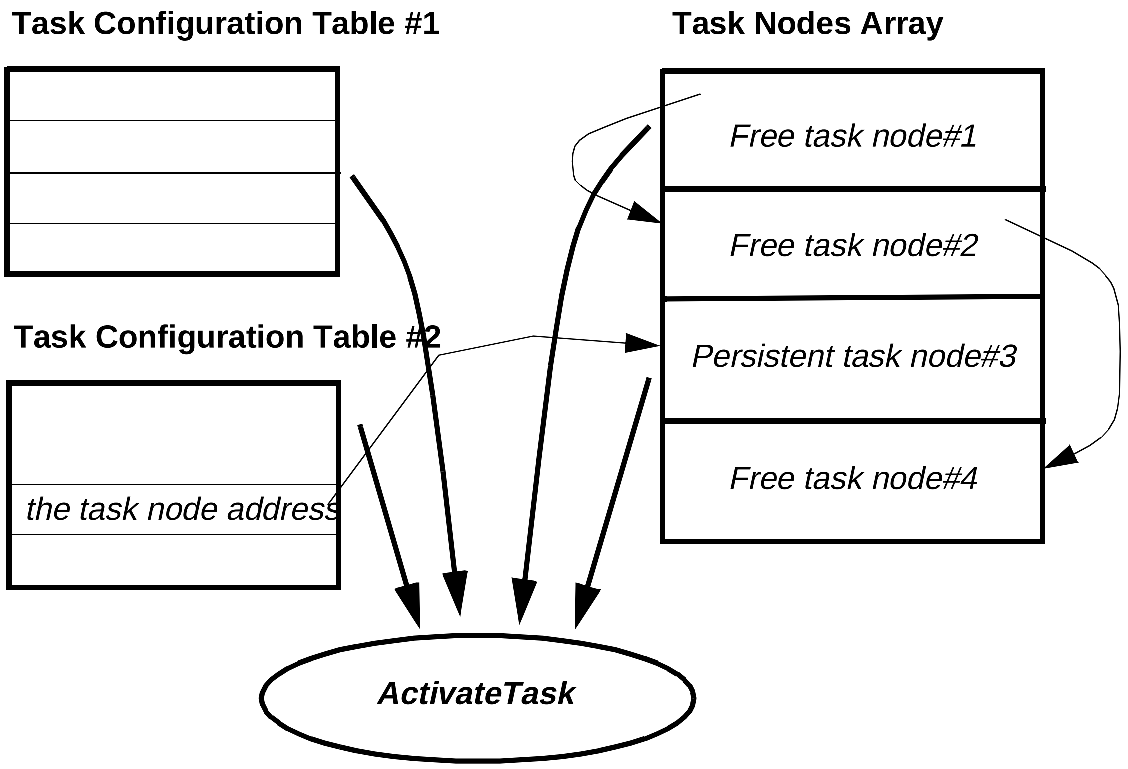

Figure 4–5 Persistent task node assignment

4.6.3 Task Link Table

Every task in the system is defined via the ROM-based task configuration table. When a task is activated, the task allocates a task node, and the address of the task configuration table is stored in the task node. To accelerate the procedure of run-time task referencing, the system supports a special RAM-based vector of the links between the task configuration tables and task nodes. This vector is called the *task link table*. It serves for fast access to the task configuration table and the corresponding task control block during operations which reference to a task.

The task link table is illustrated in the figure below:

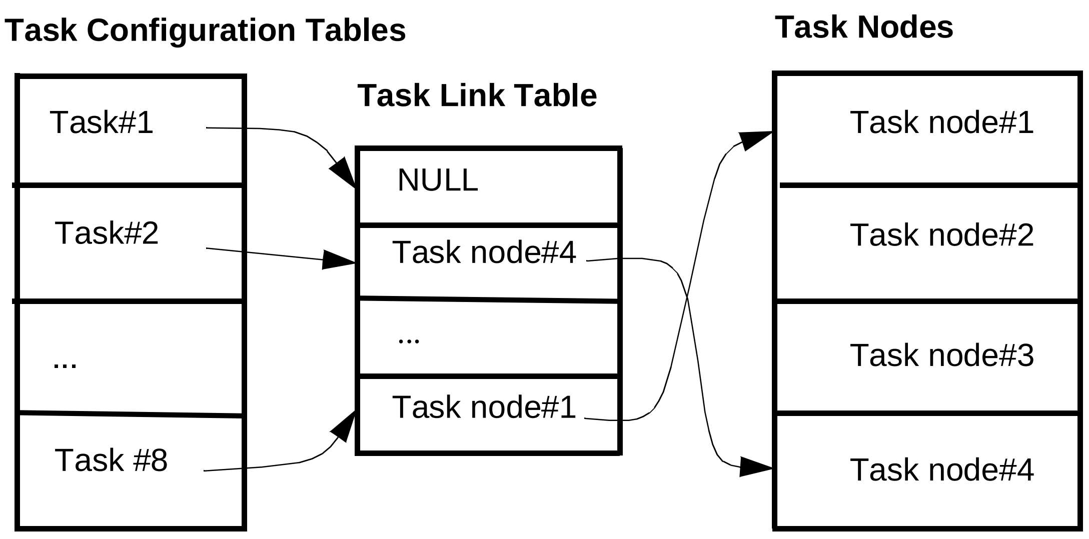

Figure 4–6   Task link table

The task link table contains one element per each task, configured in the system. The element contains a NULL-pointer, if the task is in the suspended state, or the task node address, if the task is in the non-suspended state (it is saved there during task activation). For instance, in **Figure 4–6** task#1 is not activated, while task#2 and task#8 are activated. Task#2 uses task node#4, and task#8 uses task node#1. The task configuration table contains the address of the task link element. The system uses this address when the task is referenced, and looks at the contents of the element of the task link table to define the current state of the task.

4.6.4 Task Stack

4.6.4.1 Stack allocation

During run-time every task has its own stack. The stack is either dynamically assigned to a task during its activation or statically allocated for a task during system initialization. If a stack is allocated dynamically, it is got from a *stack pool*. Several options are possible for static stack assignment during system start-up.

Stack pools are used by task management to dynamically allocate stack area for a task during task activation. To have stack pools in the system the configuration option *StackPool* must be turned on. The user should specify the desired number of stack pool with a defined number of buffers of needed size in them. This is done with the help of the *DefineStackPool* configuration statements, see **SECTION 12** and **SECTION 13** . The stack pool contains a queue of stacks of a fixed size. The task control block contains the address of the stack pool control block, and the stack buffer is unlinked from the pool and assigned to the task during task activation. During task termination the stack buffer is returned back to the stack pool.

The simplest method of stack allocation is to use task node stacks. Each task in the OSEK Operating system has the stack linked with a task node. The size of this stack is the same for all tasks, it is defined via the *DefineScheduler* configuration statement. See **4.6.4.2** for this option.

The following options are available for task stack allocation:

1. Allocation of fixed stack linked with the task node.
2. Dynamic stack allocation from the stack pool.
3. Persistent stack allocation from the stack pool.
4. Explicit stack allocation.

These options may be used simultaneously in the system, but one task must use only one of them. These options allow the user to optimize stack usage in the application.

The use of the stack allocation options is illustrated in **Figure 4–7** , **Figure 4–8** , **Figure 4–9** , **Figure 4–10** and explained below.

4.6.4.2 Allocation of fixed stack linked with the task node

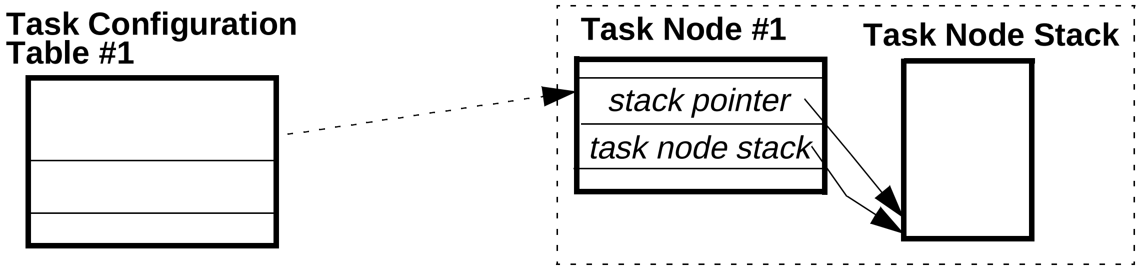

Figure 4–7 Fixed stack linked with the task node

To allocate a task stack in this manner the option *NODESTACK* should be specified in the *DefineTask* statement. Every task control block in the system has an associated stack. The size of this stack is defined via the *DefineScheduler* configuration statement and is the same for all task nodes. During task activation, the system allocates a task control block and the linked stack, and uses this stack as a task stack. Because all task control blocks have an associated stack, it is the simplest method of stack allocation. But in this case all tasks will use a stack of the same (fixed) size regardless of the stack space, really needed for tasks. To remove such type of stack the system configuration option ***NodeStack*** should be turned off.

4.6.4.3 Dynamic stack allocation from the stack pool

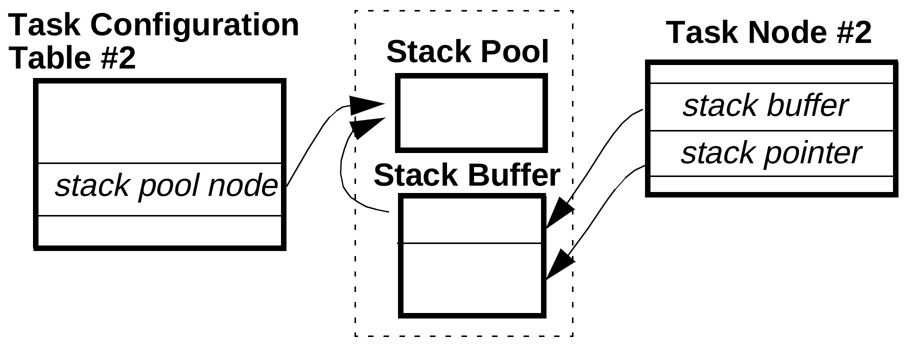

Figure 4–8   Dynamic stack allocation

To force a task to use dynamic stack allocation the option *POOLSTACK* should be defined in the *DefineTask* statement and the reference to the stack pool. The stack pool is defined separately and it is statically assigned to the task. Task #2 is configured to allocate a stack from the stack pool during task activation. The task configuration table contains the address of the stack pool, and the system allocates a stack buffer from this pool, and saves the address of the stack buffer in task node #2. During task termination the stack buffer is returned to the stack pool queue. The user is allowed to configure a stack pool of different sizes and the number of stack buffers in each to satisfy different stack space requirements for various tasks.

NOTE:

The user is responsible for proper quantity of stack buffers to avoid a possible failure of task activation, if there is no stack buffers left. Also the increased time of task activation/termination due to stack pool manipulations timing should be taken into consideration. This method allows most efficient RAM distribution for task stacks.

4.6.4.4 Persistent stack allocation from the stack pool

To use this option the user should define the *ASSIGNSTK* and *ASSIGNNODE* properties in the *DefineTask* statement. The system configuration option *PersistentStack* must be turned on. In this case the task stack is allocated from the stack pool during system initialization and the address of the stack is written in the task control block, which is persistently designated to the task too. In **Figure 4–9** task #3 is configured so that the persistent task node and a stack buffer from the specified pool are allocated for the task at system start-up, and the address of the allocated stack buffer is placed in the task node.

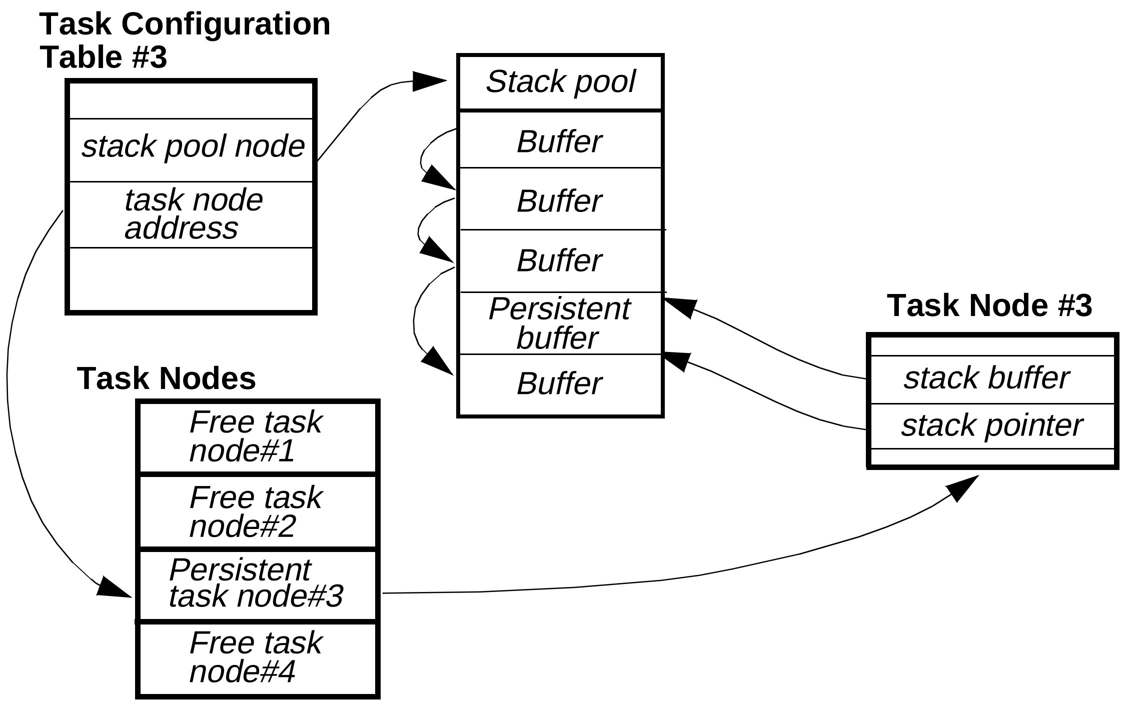

Figure 4–9 Dynamic stack allocation

NOTE:

Persistent stack from a stack pool may be assigned only for the task with assigned persistent task node and with assigned stack pool.

4.6.4.5 Explicit stack allocation

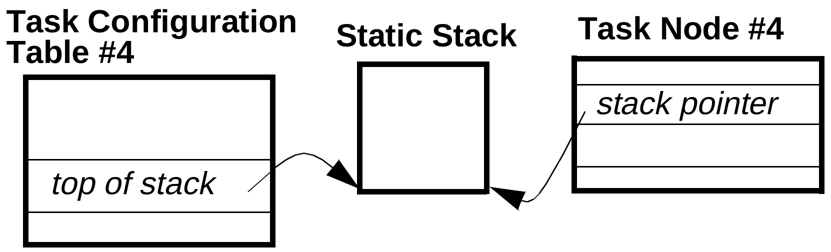

Figure 4–10 Static stack allocation

Persistent stack allocation is selected via the option *OWNSTACK* in the task definition. The system configuration option *TaskOwnStack* must be turned on. The user must also specify the stack size. In the figure above the task #4 has a stack permanently assigned to the task. The user is allowed to assign one static stack area to different tasks, if these tasks are not activated simultaneously. But with such stack allocation there is the overhead connected with non-used RAM areas when the task is not activated.

4.6.4.6 Stack size

The minimal size of the task stack depends on:

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

The recommended values of the minimal task stack size is provided in **SECTION 14 Platform-Specific Features**.

NOTE:

If the task stack is less than the minimal value for the given configuration, it may lead to the unpredictable behavior of the task and to the system crash.

4.7 Programming Issues

4.7.1 Configuration Options

The following system configuration options affect the task management:

• ***MultiplyActivation***	The option controls the multiply activation ability for Conformance Classes BCC3, ECC1 and ECC2. If the option is turned off multiply activation is disabled for tasks of these Conformance Classes.
• ***TaskIndexMethod***	If the option is turned on then the intermediate vector of the pointers to the tasks control blocks is used (fast and deterministic access to task control blocks).
• ***NodeStack***	The option defines the presence of task node stacks in the system. If it is turned off there are no task node stacks implemented.
• ***StackPool***	The option defines the presence of stack pools in the system. If it is turned off there are no stack pools implemented.
• ***PersistentNode***	If the option is turned on a persistent task control block may be assigned to the task.
• ***PersistentStack***	If the option is turned on a persistent stack buffer may be assigned to the task.
• ***TaskOwnStack***	The option defines that a task may have its own stack.

• ***UseSameContext***	If the option is turned on the same run time context frame is used both for non-preemptive and preemptive tasks in mixed-preemptive systems.
• ***TaskBasePage***	If the option is turned on, the task control blocks are placed into the base page memory. It increases the system performance since CPU accesses the base page faster than extended memory. In this case the user is responsible for the needed amount of RAM in the base page for the desired number of task control blocks.

4.7.2 Data types

The OSEK Operating System establishes the following data types for the task management:

• ***TaskRefType***	The abstract data type for task identification;
• ***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.

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

4.7.3 Task Definition

Each task in an application is generated by means of using the *DefineTask* system generation statement like the following:

```
DefineTask( <TaskName>, <TaskProperties>, <TaskPriority>,
            <EntryPoint> [,<TaskBank>] [,<InterruptMask>]
            [,<TaskStack> [,<TaskStackSize]]);
```

The application definition file contains one such statement per task. In detail task generation statement is described in **SECTION 12 System Configuration**.

To refer to a task the constructional statement should be used to declare the task before references to it:

```
DeclareTask( TaskRefType <TaskName> )
```

This declaration is equivalent to the external declaration of variables.

4.7.4 Run-time Services

OSEK OS grants a set of services for the user to manage tasks. Detailed description of these services is provided in **SECTION 16 System Services**. Here only the brief list of them is given.

Table 4–4 Task Management Run-time Services

Service Name	Description
ActivateTask	Activates the task, i.e. put it from the *suspended* into the *ready* state
TerminateTask	Terminates the task, i.e. put it from the *ready* into the *suspended* state
ChainTask	Terminates the task and activates a new one immediately
Schedule	Yields control to a higher-priority ready task (if any exists)
GetTaskId	Gets the identifier of the running task
GetTaskState	Gets the status of the specified task

Examples of using the run-time services are provided in section **16.2.11 Examples for Task Management Services**.

4.7.5 Constants

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

• ***RUNNING***	Constant of data type *TaskStateRefType* for task state *running*
• ***WAITING***	Constant of data type *TaskStateRefType* for task state *waiting*
• ***READY***	Constant of data type *TaskStateRefType* for task state *ready*
• ***SUSPENDED***	Constant of data type *TaskStateRefType* for task state *suspended*

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

4.7.6 Conventions

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

```
TASK EntryPoint( void )
{
...
}
```

The keyword ***TASK*** is evaluated by the system generation to clearly distinguish between functions and tasks in the source code.

SECTION 5 SCHEDULER

5.1 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 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 **4.5 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 ready task to be running. 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 while ISR awake a task to be executed. If it is supported by hardware the scheduler's idle loop may be replaced by an instruction that puts CPU in low power mode to reduce power consumption. This property is turned on via the system configuration option ***HCLowPower***.

The scheduler has its own small stack (the *scheduler's stack*) which is needed for the endless idle loop. The stack size is specified by the user, see section **12.4.1 DefineScheduler**. The system property ***UseMainStack*** can be specified by the user. In this case the same stack is used by the `main()` function, by the scheduler and by ISRs. This option saves RAM consumption but the user is responsible for the required memory amount for the stack.

The scheduler can be treated as a specific resource that can be occupied by any task. See **7.2.2 Scheduler as a Resource** for details.

The scheduling policy and some scheduler-related parameters are defined by the user, see section **12.4.1 DefineScheduler**.

5.1.1 Simple Scheduler

If each task in the application has an unique priority the simplified scheduler may be used to reduce memory and time consuming. In this case the scheduler uses a table of tasks instead of the queues. This system property can be turned on via the system configuration option ***SimpleScheduler***. By default this option is turned off.

The option does not depend on Conformance Classes so it is possible to use this system property in ECC2 as well as in BCC1, only unique task priorities are required (one task per priority).

The simplified scheduler can not be used with resource management. Therefore, if the *SimpleScheduler* and the *Resources* options are both turned ON in the system configuration file, then *SimpleScheduler* is ignored by the System Generator, and a warning message is produced (see **SECTION 12 System Configuration**).

5.2 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, but in case of the mixed-preemptive policy each task in an application may use only one of the three policies. It is defined via the appropriate task property (preemptive/non-preemptive).

Note that the interruptibility of the system depends neither on the Conformance Class, nor on the task type.

The desired scheduling policy is defined by the user via the system configuration option *SchedulerPolicy*. The valid values are - *NONPREEMPT, FULLPREEMPT, MIXPREEMPT*.

5.2.1 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 non-preemptive section of a running task with lower priority 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:

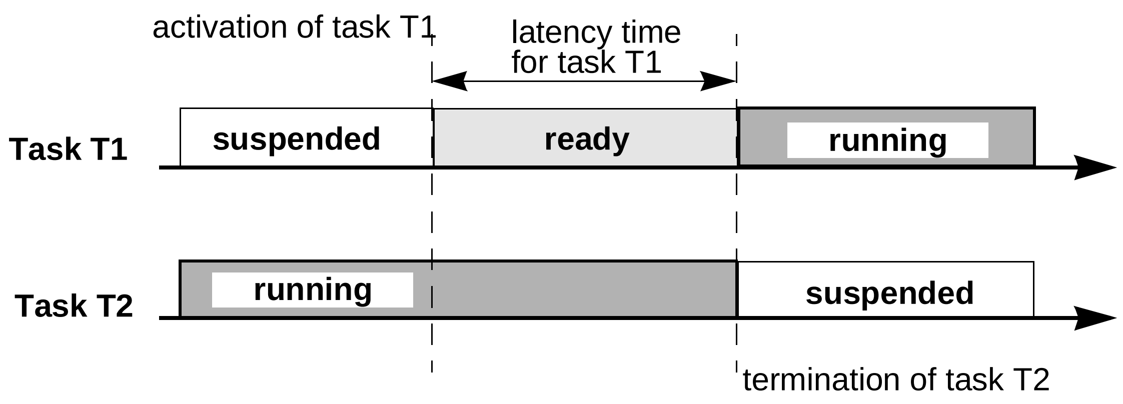

Figure 5–1 Non-preemptive scheduling

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

The following **points of rescheduling** exist in the OSEK Operating System:

- Successful termination of a task (via the *TerminateTask* system service);
- Successful termination of a task with explicit activating 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-preemptive and the task switching will take place exactly in the listed cases.

5.2.2 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 increased RAM space required for saving the context, and the enhanced complexity of features necessary for synchronization between tasks. As each task can theoretically be rescheduled at any location, access to data which are used jointly with other tasks must be synchronized.

In full-preemptive system all tasks are preemptive.

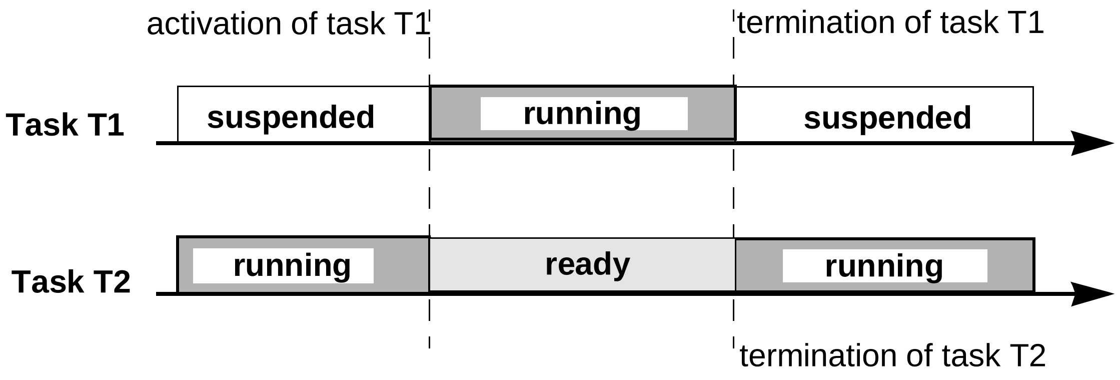

Figure 5–2   Full-preemptive scheduling

5.2.3 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 (preemptive/non-preemptive).

The definition of a non-preemptive 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 of the time of a task switch,
- if RAM is to be used economically to provide space for saving the task context,
- if the task must not be preempted.

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

NOTE:

Tasks can be ported between preemptive, non-preemptive and mixed-preemptive OSEK applications if they do not any assumption of non-preemptability. That means, that critical data must be protected using resources and communication must be done using messages.

5.3 Programming Issues

5.3.1 Configuration Options

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

• ***SimpleScheduler***	If the option is turned on the simplified scheduler will be used in the system, if each task has an unique priority. It reduces the OS code and increases system performance.
• ***SchedulerPolicy***	The option defines which scheduling policy - non-preemptive, full-preemptive or mixed-preemptive will be used in the application.
• ***UseMainStack***	If the option is turned on the same stack area is used for the `main()` function (during start-up), for the scheduler stack and for the ISR stack.
• ***HCLowPower***	If the option is turned on an instruction that puts CPU in low power mode is used instead of the scheduler's idle loop.

5.3.2 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. That leads to task rescheduling.

The scheduler can be used by the programmer as a resource (in all Conformance Classes). 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 *ReleaseScheduler* will be performed. While the task occupies the scheduler it has the highest priority and, therefore, cannot be interrupted 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 - the code here cannot be interrupted by any
other task */
...
ReleaseResource( RES_SCHEDULER ); /* End of the critical section */
```

5.3.3 Scheduler Definition

The scheduler and some parameters must be defined by the user in the configuration file via the *DefineScheduler* statement.

```
DefineScheduler( <NumberOfTasks>,<NumberOfPriorities>,
                 <SchedulerStackSize>, <SchedulerStackAddress>,
                 <TaskNodeStackSize>, <TaskNodesStackAddress> );
```

SECTION 6
INTERRUPT PROCESSING

6.1 General

Interrupt processing is the 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. Addresses of ISRs should be pointed in the vector table.

In OSEK OS all ISRs should use the separate stack (`ISR stack`) which is used only by ISRs during their execution. The size of the ISR stack is defined by the user. At the beginning of an Interrupt Service Routine the user should switch to this stack using the system service *EnterISR*. After the ISR completion the corresponded service *LeaveISR* should be performed to switch back to the previous stack. The instruction sequence between the *EnterISR* and *LeaveISR* calls is considered as `ISR frame`.

When interrupt occurs either current stack can be used or the ISR stack. The current stack is:

- the stack of the interrupted task,
- the scheduler's stack if an interrupt occurred during scheduler execution,
- ISR stack in case of nested interrupts.

OSEK OS supports nested interrupts, theoretically up to 255 levels. Special system counter tracks the number of nested interrupts. Since OS provided means to switch the stack and to control the interrupt mask, such nested interrupts, if they were written correct, can be treated as the single one. To not waste a task stack space in case of nested interrupts or complicated ISR, the ISR stack is used.

OS also controls the state of an interrupt mask. The user defines values of the interrupt masks for disabling and enabling all interrupts, and the default interrupt mask for task execution. See section **12.4.2 DefineInterrupts** for details.

ISRs can communicate with tasks by the following means:

- ISR can activate a task;
- ISR can send a state or an event message to a task;
- ISR can trigger a counter.

Interrupts cannot use any OS services except those which are specially allowed to be used within ISRs. In case of using other services the system behavior will be unpredictable. In the Extended (debugging) status of the Operating System the error will be reported in such case. See **Table 6–1** and **SECTION 16** for details.

6.2 ISR stack

The purpose of the ISR stack is to save the memory. Since interrupts can be nested, it means, that every task stack has to be big enough to store several interrupt stack frames (in addition to task needs for local variables, function calls, etc.). To avoid this overhead, the separate ISR stack is used in the OSEK Operating System. Switching to this stack is performed by the *EnterISR* service at the beginning of ISR. This stack is used only by ISRs, and if nested interrupts occur after the stack has been switched, they will use this stack too. Before leaving the ISR switching back to the interrupted task stack have to be done by means of *LeaveISR*.

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

Most compilers use function modifiers (like 'interrupt') to generate stack frame. In turn, the ***ISR*** keyword, specified in OSEK (see section **6.6.4 Conventions**), is a macro for this modifier.

6.3 ISR Categories

In the OSEK Operating System three types of Interrupt Service Routines are considered.

6.3.1 ISR category 1

ISRs of this type does not use any operating system service. These ISR does not use OS services *EnterISR* and *LeaveISR*, consequently, ISR of these type are executed on the current stack. In this case, if ISR uses the stack space for its execution, the user is responsible for the appropriate stack size. Moreover, if interrupts are reenabled inside the ISR, nested interrupts are possible which will use the same task stack. The general recommendation for ISRs category 1 may be the following:

RULE:

ISR category 1 can not use any OS services. These ISRs have to disable interrupts inside itself (at the beginning of the routine).

After the ISR is finished, processing continues exactly at the instruction where the interrupt has occurred, i.e. the interrupt has no influence on task management.

```
ISR ISR_handler
{
...
/* the code without any OS service calls */
...
}
```

6.3.2 ISR category 2

In ISR category 2 the OSEK Operating System provides the ISR frame to execute more complicated user code. The ISR frame is instructions between OS services *EnterISR* and *LeaveISR*. Such ISRs must have the *EnterISR* call at their beginning to switch to the ISR stack and save the initial interrupt mask. After that any user's routine can be executed, include allowed OS calls (to activate a task, send a message or trigger a counter). See section **6.6.3 Run-time Services** for the list of services allowed for ISR. At the end of the ISR the *LeaveISR* service must be executed to switch back to the task stack and restore the interrupt mask.

```
ISR ISR_handler
{
EnterISR();
...
/* the code with allowed OS calls */
...
LeaveISR();
}
```

Inside the ISR no rescheduling will take place. Rescheduling may **only** take place on termination of the ISR if a preemptive task has been interrupted.

6.3.3 ISR category 3

Such ISR's are similar to those of category 2. But the location of the ISR-frame in the code segment is application dependent and user defined. The code outside the ISR frame can be used, e.g., to access global variables or hardware registers. The user is responsible for operations outside the ISR frame, since they are executed on the task stack, and interrupts may be enabled (see section **6.3.1 ISR category 1**). No OS service calls can be executed outside the ISR frame.

```
ISR ISR_handler
{
/* user's code */
...
EnterISR();
/* the code with allowed OS calls */
...

LeaveISR();
}
```

To avoid possible errors the following rule for ISR category 3 is reasonable:

RULE:

In ISR category 3 all operations with using OS services and/or with enabled interrupts have to be performed only inside the ISR frame.

The OSEK Operating System have no means to distinct ISR category 2 and 3 at run-time. So such distinction should be made by the user at the stage of source code writing, and it has a rather conventional nature.

6.4 Interrupt Flag Manipulation

OSEK OS provides services to control the interrupt flag - to disable or enable interrupts and get the current state of the interrupt mask. These services are implemented to allow the user to control interrupts in case of several interrupt levels (for instance, Motorola MC68HC16 MCU has 8 interrupt levels). In this case it is possible to set the desired interrupt mask which disables some interrupts and enables other ones. In the interrupt definition statement the user should select interrupt masks for all interrupts disabled, all interrupts enabled and some desired medium state (by default a task gets this interrupt mask when it is activated if it has no its own starting interrupt mask specified). The special data type *IntMaskType* is defined within the OSEK Operating System to control interrupt masks.

This version of the OSEK Operating System is intended for Motorola MC68HC08 MCU family which has only two states for interrupts - all disabled or all enabled. Therefore for HC08 IS services for interrupt flag manipulation simply disables and enables interrupts and inquires the current status if the interrupt flag. If a task is preempted by other task or suspended its interrupt mask is saved. When the task is resumed its saved interrupt mask is restored. When a task is activated it has either the interrupt mask specified in the task definition statement (see **12.4.4 DefineTask**) or the interrupt mask specified in the interrupt definition statement (see **12.4.2 DefineInterrupts**).

6.5 Local Variables Considerations

The *EnterISR* and *LeaveISR* functions assume, that the stack frame is defined on the entry. Moreover, the stack pointer may be changed during the execution of these functions. This way, the using of the local variables inside ISR may lead to unpredictable behavior and crash. Therefore, the user is not allowed to use local variables inside ISR. Instead, the user may split the interrupt service routine into two functions. The nested function should perform useful work, and may have local variables. The outermost function shall call the nested function framed by the calls to *EnterISR* and *LeaveISR* services.

The example of the right code:

```
int _SciHandler(void)
{                 /* The inner function with local variables */
char      status;        /* local variable to hold SCI status*/
int       num;           /* local variable to byte counter*/
...                      /* useful user’s code*/
}

ISR       SciHandler(void)      /* Interrupt service routine*/
{                               /* No local variables defined*/
       EnterISR();    /* Switch to the ISR stack */
       _SciHandler(); /* Perform useful work to handle interrupt*/
       LeaveISR();
}
```

The example of the wrong code - local variables are allocated on the task stack, but the stack pointer is changed by *EnterISR* after, therefore references to the variables are invalid:

```
int _SciHandler(void)
{                 /* The inner function without local variables */
...               /* useful user’s code*/
}

ISR       SciHandler(void)      /* Interrupt service routine*/
{
char      status;      /* local variable to hold SCI status*/
int       num;         /* local variable to byte counter*/
       EnterISR();    /* Stack Pointer is changed ! */
       _SciHandler(); /* Perform useful work to handle interrupt*/
       LeaveISR();
}
```

6.6 Programming Issues

6.6.1 Configuration Options

The following system configuration options affects the interrupt management:

• ***EntryExitISR***	if the option is turned off it is assumed that there are no interrupts in the system at all and the Operating System does not have any interrupt handling mechanisms. No interrupt management services are implemented.

- ***InterruptMaskControl*** if the option is turned off (while the *EntryExitISR* option is turned on) interrupt masks are not controlled by the operating system. Services *EnableInterrupt*,*DisableInterrupt*, *GetInterruptMask* are not implemented.

6.6.2 Data Types

The OSEK Operating System uses the special data type ***IntMaskType*** for interrupt masks. For reference to variables of this type the data type ***IntMaskRefType*** is introduced in OSEK OS.

6.6.3 Run-time Services

OSEK OS provides the set of services for interrupt management. Also some services may be used both on the task level and on the ISR level. These services are shown in the **Table 6–1** .

Table 6–1 Interrupt Management Services

Service Name	Description
Interrupt Management Services	
EnterISR	Registers the switching to the interrupt level and switch context to the ISR stack
LeaveISR	Registers the leaving of the ISR level
EnableInterrupt	Enables interrupts in accordance with the given mask
DisableInterrupt	Disables interrupts in accordance with the given mask
GetInterruptMask	Returns the current state of interrupts
Services allowed for use in ISR	
ActivateTask	Activates the specified task (puts it into the *ready* state)
SendStateMessage	Sends a state message to the specified task
SendEventMessage	Sends an event message to the specified task
CounterTrigger	Increments a counter value

6.6.4 Conventions

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

```
ISR IsrName( void )
{
...
}
```

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

6.6.5 ISR definition

To define common ISR parameters like ISR stack size and predefined interrupt masks the *DefineInterrupts* statement should be specified in the configuration file:

```
DefineInterrupts(<DisableMask>, <EnableMask>, <TaskMask>,
                 <ISRStackSize>, <ISRStackAddress> );
```

See **12.4.2 DefineInterrupts** for details.

SECTION 7
RESOURCE MANAGEMENT

7.1 General

The resource management is used to coordinate concurrent accesses of several tasks to shared resources, e.g. management entities (scheduler), program sequences (critical sections), memory or hardware areas. In general, the resource management is provided in all Conformance Classes, but it is fully supported only beginning from the *BCC3* Conformance Class. In *BCC1* and *BCC2* only the scheduler is treated as the specific system resource which can be used by tasks.

Resource management ensures that

- two tasks cannot “own” the same resource at the same time,
- priority inversion cannot arise while resources are used,
- deadlocks do not occur by use of these resources,
- access to resources never results in a *waiting* state.

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

- full- or mixed-preemptive scheduling;
- non-preemptive scheduling, if resources are also to remain occupied beyond a scheduling point (except the scheduler resource);
- non-preemptive scheduling, if the user intends to have the application code executed under other scheduling policies too.

Resources cannot be occupied by more than one task at a time. The resource that is now by a task must be released before another tasks can it. The OSEK operating system ensures that tasks are only transferred from the *ready* state into the *running* state, if all resources which might be occupied by that task during its execution have been released. Consequently, no situation occurs in which a task tries to access an occupied resource. The special mechanism is used by the OSEK Operating System to provide such behavior, see **7.2.1 Priority Ceiling Protocol** for details.

The *waiting* state is not admissible for Extended Tasks while a resource is occupied. It means that the task occupying a resource is not allowed to call the *WaitEvent* service.

In case of multiple resource occupation, the task has to request and release 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 */
```

OSEK 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).

7.2 Access to Resources

Before their using resources must be defined by the user at system configuration stage via the configuration statement *DefineResource*, see section **12.4.6 DefineResource**. Then the resource is declared in the source file where it will be used by means of system declaration statement *DeclareResource* (see section **16.4 Resource Management Services**). After that the task 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 are executed, this resource cannot be requested by another task.

In the OSEK Operating System resources are ranked by priority. Each resource is assigned statically to a user defined priority which is called *Ceiling Priority*. It is possible to have resources with the same priorities, but the resource Ceiling Priority has to be identical or higher to the highest task priority with access to this resource. This resource feature supports the *Priority Ceiling Protocol*.

7.2.1 Priority Ceiling Protocol

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

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

The example shown in **Figure 7–1** illustrates the mechanism of the Priority Ceiling Protocol.

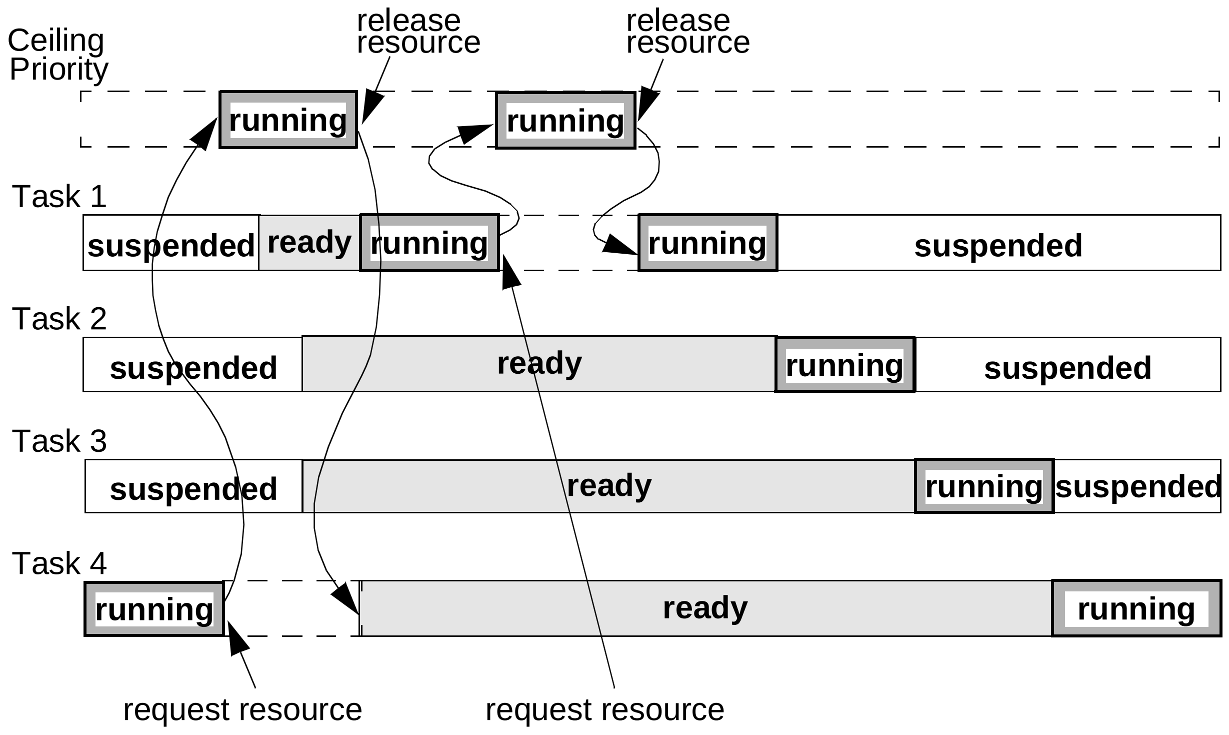

Figure 7–1 Priority Ceiling Protocol

In the figure above Task 1 has the highest priority, Task 4 has the lowest Priority. The resource has the priority greater than or equal to the Task 1 priority. When Task 4 occupies the resource it gets the priority not less then Task 1, therefore it cannot be preempted by *ready* Task 1 until it release the resource. Just after 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 its turn, occupies the resource, its priority is also changed to the Ceiling Priority.

7.2.2 Scheduler as a Resource

The OSEK 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 section **7.1**.

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 rescheduling of tasks. It is not allowed for a task to get the scheduler and then yield CPU via the *Schedule* service. The application behavior will be unpredictable in this case.

NOTE:

If a non-preemptive task gets the scheduler as a resource it must to release it before the point of rescheduling!

7.3 Programming Issues

7.3.1 Configuration Options

The following system configuration options control the resource management in the OSEK OS:

• ***Resources***	This options defines whether resource management is provided by the OS or not.
• ***FastResource***	The option can be specified by the user to increase the system performance. If it is turned on the system will work faster. But this option may be used only for debugged applications, because errors related to incorrect access and priority are not signalled. If this option is turned on less amount of ROM and RAM is needed for resources. But, if resources priorities have a big difference (e.g. first resource has priority 1 and the second resource has priority 20) this option does not lead to RAM saving.

7.3.2 Data types

The OSEK Operating System establishes the following data type for the resource management:

• ***ResourceRefType*** the abstract data type for referencing a resource;

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

7.3.3 Run-time Services

OSEK OS grants a set of services for the user to manage resources. Detailed description of these services is provided in **16.4 Resource Management Services**. Here only the brief list of them is given.

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

Service Name	Description
ReleaseResource	Releases the resource assigned to the critical section (to leave the critical section)

7.3.4 Resource Definition

To define a resource the following definition statement should be specified in the generation file:

```
DefineResource( <ResourceID>, <ResourcePriority> );
```

For more details see section **12.4.6 DefineResource**.

To refer to a resource the declaration statement should be used to declare the resource before its using:

```
DeclareResource( ResourceRefType <ResourceID> );
```

This declaration is equivalent to the external declaration of variables.

SECTION 8
COUNTERS AND ALARMS

The OSEK operating system comprises a two level concept to make use of recurring events like periodic interrupt of timers, interrupt of the sensors on rotating angles, or any recurring software events. To manage such situation counters and alarms are provided by the OSEK Operating System. The recurring events (sources) can be registered by counters. Based on counters, the OSEK OS offers alarm mechanism to the application software. Counters and alarms are provided by the OSEK OS in all Conformance Classes.

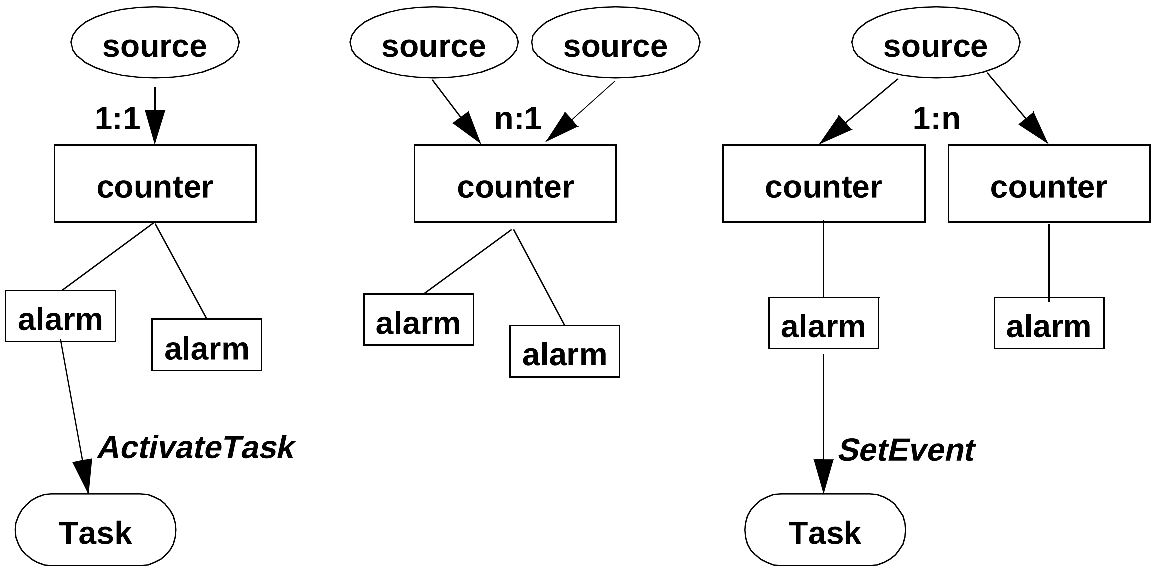

Figure 8–1 Counters and alarms

8.1 Counters

Any event in the system can be linked with a counter. It means, when the event is 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.

A counter is represented by a current counter value and some counter specific parameters. These parameters are the *`counter initial value`* , the *`conversion constant`* and the *`maximum allowed counter value`*. They are defined by the user. The latter 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 *CounterTrigger*.

The maximum allowed counter value specifies the number after that the counter rolls over. When a counter reaches its maximum allowed possible value (or rolls over the predefined size - byte etc.), it starts counting again from zero.

NOTE:

> The maximum allowed counter value is never really reached by a counter. It means, that if, for instance, value 5 is specified as the maximum allowed one, 5 is never can be read as a current counter value. If an alarm should be set for value 5, the maximum allowed counter value must be 6.

The user also defines the maximal size of all counters in the system - they can be byte-, word- or longword-sized. This is defined via the system properties ***CounterSize***.

The conversion constant 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 this parameter can be treated as a counter-specific reference value.

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

At least one counter always exists in the system. This counter is used as a *`system timer`* (the internal system clock). The system timer is a standard counter with the following additions:

- the user must always define the system timer in an application;
- special constants are defined to describe counter parameters and to decrease access time;
- the user defines the source of hardware interrupts for the system counter.

In the system definition statement for the system 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 **SECTION 14 Platform-Specific Features** for details).

If hardware related parameters are defined, the code to initialize the system timer hardware and the interrupt handler are automatically provided for the user as a part of OSEK OS. In that case the user does not have to care about handling of this interrupt and he/she can not change the provided code. If the parameters are not defined the user has to provide the code to initialize the hardware and handle the interrupt. In this case in ISR for the specified interrupt the service *CounterTrigger* must be used to advance the counter.

NOTE:

The system timer will not be triggered if the EntryExitISRproperty is turned OFF!

The system timer has a predefined conversion constant that equals to the number of ticks required to reach 10 milliseconds.

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 event should call the system service *CounterTrigger*. This service must be called within the ISR frame created by the *EnterISR* and *LeaveISR* services. It is not allowed to use *CounterTrigger* in ISR category 1 (see section **6.3 ISR Categories**).

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 8–1 Counters and alarms**. It means that it is possible to advance the same counter in different software routines.

8.2 Alarms

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

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

Alarms are defined statically as all other system resources. The assignment of alarms to counters, as well as the action to be performed when an alarm expires (task and event) are defined statically, too. After the alarm was defined and assignment to a counter an application can use this alarm. 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 start 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 tick elapses starting from the current counter value at the moment of alarm setting.
- **Absolute alarm** expires when the counter reaches the specifies number of ticks starting from zero counter value, no matter which value the counter have at the moment of alarm setting. 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. It is illustrated by **Figure 8–2** . In the latter case the total time until the alarm expires is the sum of T_1 and T_2.

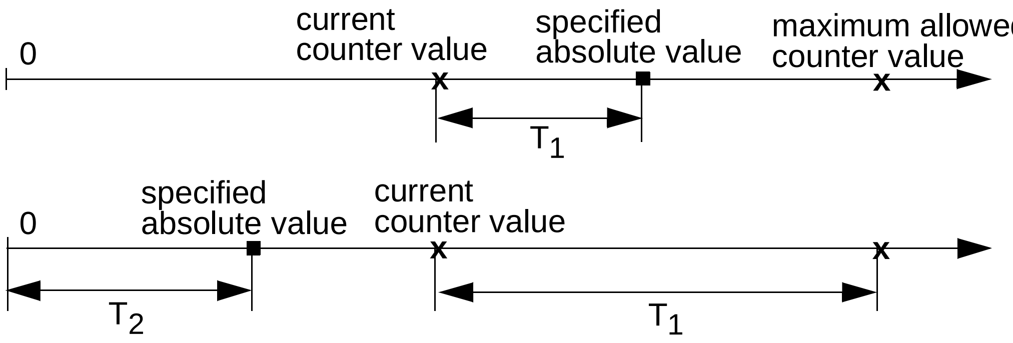

Figure 8–2   Two cases for the absolute alarm

If a cycle value is specified for the alarm it is logged on again immediately after expiry with this relative value. Specifies 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 single one.

8.3 Programming Issues

8.3.1 Configuration Options

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

• ***Counters***	This options defines whether counters are providedby the OS or not.
• ***CounterSize***	This option defines the size of all counters. The valid values are 8, 16 and 32 which conform to byte, word or long word size of counters.
• ***Alarms***	This options defines whether alarms are provided by the OS or not.
• ***AlarmList***	If the option is turned on the running alarms are linked into a list which decreases the time for alarm handling.

8.3.2 Data Types

The following data types are established by OSEK OS to work with counters and alarms:

- ***CtrRefType*** - 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*
- ***CtrInfoType*** - the data type represents a structure for storage of counter characteristics. This structure has the following fields:
  - ***maxallowedvalue*** maximum possible allowed count value;
  - ***tickperbase*** number of ticks required to reach a counter-specific significant unit;
  - ***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.tickperbase;
cons = CntData.mincycle;
```

- ***CtrInfoRefType***- the data type references data corresponding to the data type *CtrInfoType*
- ***AlarmRefType***- the data type represents the reference to an alarm element.

8.3.3 Counters and Alarm Generation

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

```
DefineCounter( <CounterID>, <maxallowedvalue>,
               <ticksperbase> [,<mincycle>] );
```

The system counter (system timer) has always to be defined in the system in the following manner:

```
DefineSystemTimer( <CounterID>, <maxallowedvalue>,
                   <ticksperbase>, <tickduration> [,<mincycle>]
                  [, <HardwareType> [, <HardwareParams>]] );
```

An alarm is generated by means of the *DefineAlarm* statement:

```
DefineAlarm( <AlarmID>, <CounterID>, <TaskID> [,<Event>]);
```

The application definition file must contain at least the system counter definition. In detail counter and alarm generation statements is described in section **12.4.9 DefineAlarm**.

To refer to a counter or alarm the declaration statements should be used to declare the element (counter or alarm) before their using:

```
DeclareCounter( CtrRefType <CounterID> );
DeclareAlarm( AlarmRefType <AlarmID> );
```

These declarations are equivalent to the external declaration of variables.

8.3.4 Run-time Services

OSEK OS grants a set of services for the user to manage counters and alarms. Detailed description of these services is provided in **SECTION 16 System Services**. Here only the brief list of them is given.

Table 8–1 Counter and Alarm Management Run-time Services

Service Name	Description
InitCounter	Sets the initial value of the counter
CounterTrigger	Increments the counter value
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

Examples of the run-time services usage are provided in **SECTION 16** .

8.3.5 Constants

For system counter, which is always a time counter, the special constants are provided by the operating system:

- ***OSMAXALLOWEDVALUE*** **-** maximum possible allowed value of the system timer in ticks;
- ***OSTICKSPERTIME*** **-** number of ticks that are required to reach 10 milliseconds in the system counter;
- ***OSTICKDURATION*** - duration of a tick of the system counter in nanoseconds;
- ***OSMINCYCLE*** **-** minimum allowed number of ticks for a cyclic alarm (only for system with Extended Status).

SECTION 9
EVENTS

9.1 General

Within the OSEK operating system tasks can be synchronized via occupation of a resource (see **SECTION 7**). Another means of synchronization is the event mechanism, which is provided for Extended Tasks only. Special fields in the task node of an Extended Task are provided for event management in the OSEK OS (see **4.6.2 Task Control Block**). Events are the only mechanism allowing a task to enter the *waiting* state.

An `event` is an object managed by the OSEK Operating System, which is able to store binary data. The interpretation of the event is up to the user. Examples are: the signalling of a timer's expiry, the availability of a resource, the receipt of a message, etc.

Within the operating system, events are not independent objects, but allocated to Extended Tasks. Each ET has a definite number of events - 8 or less (in fact, the byte size are used for event management in the current OSEK OS implementation.). Events are represented by two event masks - byte-sized fields in the task node (See **4.6.2 Task Control Block**). One field is the mask of events the task is waiting for, this mask can be set and cleared only by the task-"owner“ (a “private” mask). The second field (a “public” mask) contains the mask of the events, which are set for the task by other tasks. When activating an Extended Task, 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 “private” event mask are set. The system service *WaitEvent* is designed to force a task to wait for an event. When other task sets an event it sets the specified bits of the “public” event mask, and if some bits in both “private” and “public” masks are the same, the task is transferred into the *ready* state. The task can clear its own events by clearing the “private” event mask.

Various system services are available to manipulate events, depending on whether the dedicated task is the “owner” of the event or another task which does not necessarily have to be an Extended Task. 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 Extended Task which sets an event at a certain time. Thus, the Extended Task can delay itself (see example in section **16.6.6 Examples of using events**).

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 (but not an interrupt) can set an event for an Extended Task, and thus inform the appropriate Extended Task (its identification must be known) about any status change via this event.

To have events in the system the configuration option ***Events*** must be turned on.

9.2 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 Tasks are in the *waiting* state, if an event for which the task is waiting has not occurred. If an Extended Tasks tries to wait for an event and this event has already occurred, the task remains in the *running* state.

Figure 9–1 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.

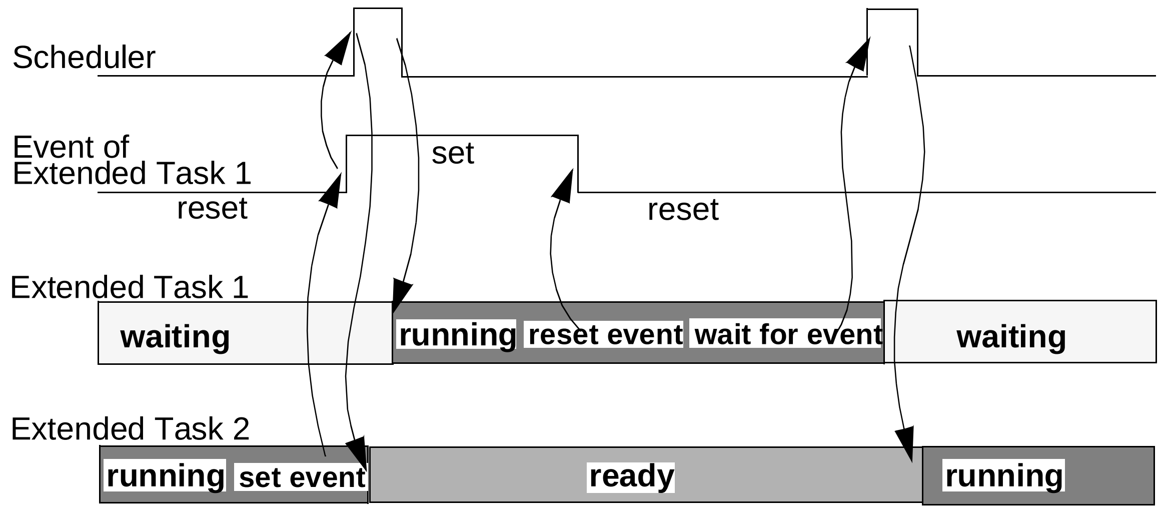

Figure 9–1 Synchronization of Extended Tasks by setting events in case of full-preemptive scheduling.

If non-preemptive scheduling is supposed, rescheduling does not take place immediately after the event has been set as it is shown in **Figure 9–2**.

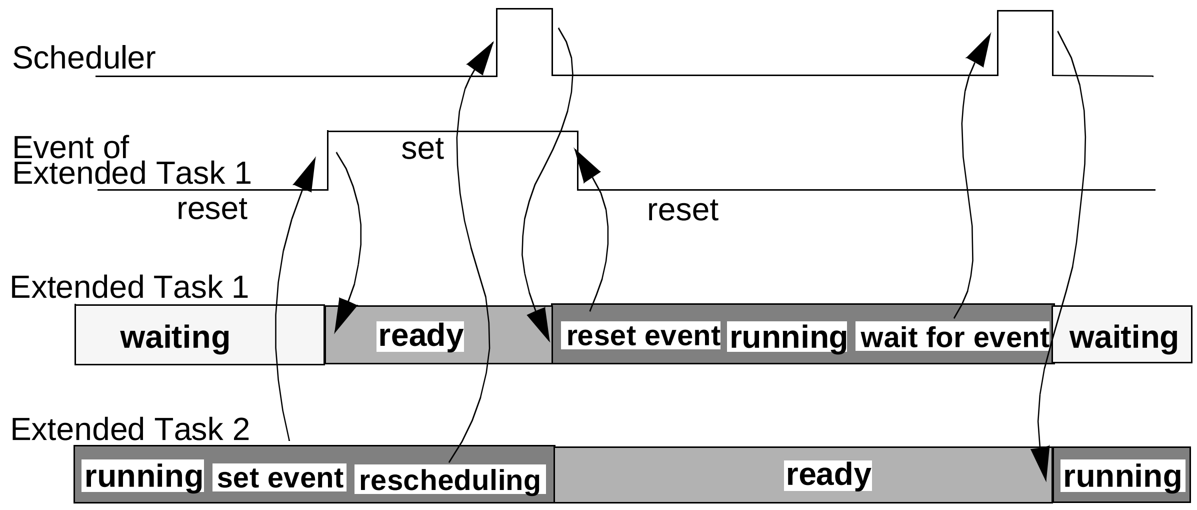

Figure 9–2 Synchronization of Extended Tasks by setting events in case of non-preemptive scheduling.

9.3 Programming Issues

9.3.1 Configuration Options

The only system configuration option ***Events*** controls event management in the system. If it is turned off events are not implemented.

9.3.2 Data Types

The OSEK 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.

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

9.3.3 Events Definition

Events are not defined by the user at the system configuration stage.

9.3.4 Run-time Services

OSEK OS grants a set of services for the user to manage events. Detailed description of these services is provided in **16.6 Event Management Services**. Here only the brief list of them is given.

Table 9–1 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 **16.6 Event Management Services**.

9.3.5 Additional usage

Note that it is possible to use bits in memory fields assigned for events for some internal task’s needs, as bit flags. In case of using events in the system in every Extended task’s control block special fields are created. If a task uses less than 8 events it is possible to use *SetEvent* and *ClearEvent* system services with appropriate masks to manipulate internal task’s bit flags. See the following example:

```
#define X_FLG  0x80    /* define masks for internal flags */
#define Y_FLG  0x40
#define Z1_FLG  0x20
#define Z2_FLG  0x10

DeclareTask( TASK_A )
DeclareTask( TASK_C )

TASK taskA_ext
{
EventMaskType x, y, z1, z2;
z1 = Z1_FLG; z2 = Z2_FLG;
int speed;

...
if (speed == LIMIT)
    {
      x = X_FLG;
      SetEvent( TASK_A, x );
    }

GetEventMask( TASK_A, &x );
if ((x & X_FLG) != 0 ) ClearEvent( z1 );
else SetEvent( TASK_A, z2 );
if ((x & Y_FLG) == 0 ) ChainTask( TASK_C );
...
}
```

In the example the task uses 4 most significant bits of the event field as its internal bit flags. Least significant bits are free and they can be used for “external” OSEK OS events. But such approach requires more attention from the user to avoid occasionaly changing of “internal” events instead of “externa” ones and visa versa.

Note that access to that binary data is performed only via the system services for event management.

SECTION 10
COMMUNICATION

10.1 Message Concept

In OSEK Operating System communication between application tasks takes place via messages. Messages are stored in `Message Objects (MO)` which are handled by the operating system. A distinction is made between the following:

- *`State Messages`* and
- *`Event Messages`*

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

In contrast, an Event Message contains an event information, e.g. “engine temperature exceeds a certain limit”. Event Messages are buffered with the send operation and consumed with a receive operation.

In OSEK OS message objects are referenced by tasks via these unique identifiers defined by the user at the configuration stage.

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

OSEK supports two types of communication between tasks: 1:1 and 1:N communication.

- 1:1 communication means that only one task receives the message;
- 1:N communication means that N tasks receive the same message.

Both types of messages, State and Event Messages can be used for 1:1 and 1:N communication, for local (ECU-internal) and network communication.

As an option, ***task activation*** or ***event signalling*** 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 which want to react immediately on new message information. There is no special operating system service to wait for messages, but normal event mechanism is used. Only one notification method can be assigned for certain message.

Alarms can statically be assigned to message objects (only one alarm per message object). If the message does not arrives during the time period specified in the *DefineMessageAlarm* configuration statement, the alarm counter expires and an associated task is activated or an event is signalled. Whenever a message arrives, the alarm is restarted again. This feature can be used to supervise if State Messages are updated or Event Messages arrive on time.

It is possible to assign both an alarm and task notification (task activation or event signalling) to a certain message object.

Table 10–1 Features of the Message Concept

State Message	Event Message
No buffering	Buffering in a FIFO-queue
No consumption of message	Consumption of messages
Direct access possible in non-preemptive systems (no copies)	No direct access possible (always copies)
Static definition of task activation or event signalling	
Static definition of a message alarm	

Each message in OSEK OS may have a so-called `timestamp` - the special field which is updated by the user before message sending. The timestamp is designed to notify the message receiver about the time when the message was sent. The user defines statically whether a message has a timestamp or not with the help of configuration options ***StateMsgTimeStamp*** and ***EventMsgTimeStamp***. If a message is to be with a timestamp, the following code should be written:

```
typedef MSGA tagMsgA
struct tagMgsA
{
  TickType timeStamp;
  int x;
};
```

OSEK OS communication services provide all means for local message transfer, within single ECU. To transfer data over the network the OSEK Communication System (COM) will be used, which is designed to handle all other types of communication through the network. And OSEK OS communication services provide an interface for application tasks to exchange data. Thus, messages serve as interface for both, ECU - internal and network communication. Uniform services with identical interfaces are offered (network transparency).

10.2 State Messages

State Messages represent the current value of a state variable. Tasks can have three various access type to State Messages - read only (receive), write only (send) and full read/write access. The send operation overwrites the current value of a message, i.e. State Messages are not buffered. The receive operation reads the current value of a State Message whereby the message data is not consumed.

These services ensure:

- the consistent writing and reading of message data within the send and receive operation (also in preemptive systems) and
- the consistency of data while tasks and functions (subprograms) use the message data.

A State Message can have a default value which is assigned to the message at the configuration stage statically. This value is returned when the message is received first time without prior send operation.

State Messages may be either with local copy of a message or without one. It indicates with qualifiers ***WithCopy*** and ***WithoutCopy*** when message is declared. The distinction between these kinds of State Messages is the following:

- When the ***WithCopy*** qualifier is used, the State Message item is copied from the task data space into the message item for send operation, and from the message item to the task data space for receive operation. The message body is copied consistently, i.e. the operating system provides atomic behavior of the send-receive operations. The default value is copied from the user's-defined area into the message body during system start-up.
- When the ***WithoutCopy*** qualifier is used, the State Message item is not copied from the task data space into the message item. Instead, the task-sender updates the message body directly via de-referencing of the pointer and writing the data at the pointer address, and the task-receiver uses the message body by means of de-referencing the pointer and reading the data. The operating system doesn't provide atomic behavior of the send-receive operations, and this consistency is to be provided by the user's code.

State Messages are used similar to C-language variables (direct access e.g. in a C-language assignment or in a C-language expression). The access privileges (send, receive or send/receive) defined for a message also determine which direct access operations are allowed.

1:N communication for State Messages does not have any difference from 1:1 communication, since any task can read the State Messages if its identifier is known.

To have State Messages in the system the configuration option ***StateMessage*** must be turned on.

10.3 Event Messages

In contrast to State Messages, Event Message objects can temporarily store several messages. The temporary storage follows the FIFO principle, i.e. messages are received in the same order as they were sent. During the send operation the message item is copied from the task data space into the message FIFO area at the write pointer. During the read operation the message item is copied from the FIFO area at the read pointer into the receiving task space. The data is copied consistently, i.e. the operating system provides atomic behavior of the send-receive operations. The FIFO area is circular. It is illustrated in **Figure 10–1**. Message “msg1” is consumed from the Event Message object by a task, and other message items are “moved” in the Event MO towards the top (in fact, read and write pointers are changed). After the message item “msg1” was consumed, new message “msg5” is written by a task at the empty location (the last one). If an Event Message object has no memory capacity left to store the new message, the oldest message is overwritten - in **Figure 10–1** “msg2” is overwritten by the message item “msg6” in spite that “msg2” has not been consumed. The overwriting process is indicated both to sender and receiver in systems with Extended Status by means of return codes while send or receive services are executed by a task.

An Event Message is characterized by the length of a single message item and by the depth of a queue. These parameters are defined by the user at the configuration stage.

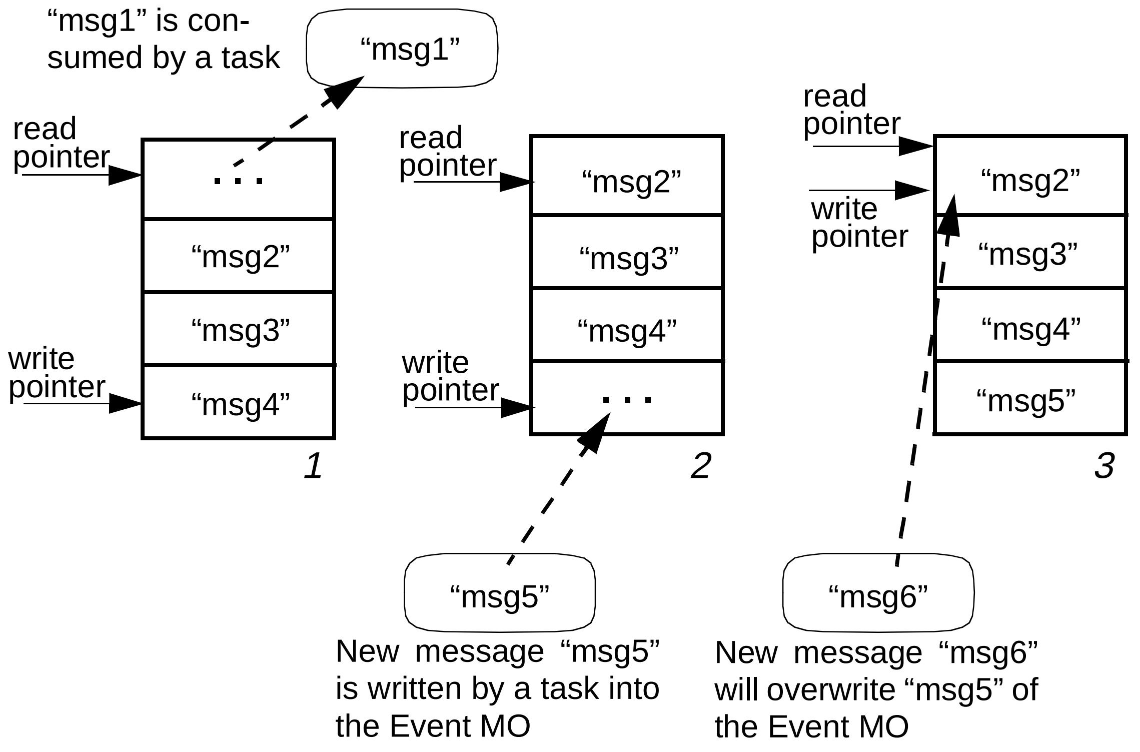

Figure 10–1 Operations with Event Messages

Event communication includes an implicit synchronization of the communication partners since an Event Message must be sent before it can be received. When the receive operation is performed, an Event Message is removed from the message object and is thus consumed.

In case of multiple receivers (1:N communication) OSEK OS ensures that no receiver may receive a new message, until all receivers will complete the reception of the current message. The last receiver consumes the message.

To have Event Messages in the system the configuration option ***EventMessage*** must be turned on.

10.4 Programming Issues

10.4.1 Configuration Options

The following system configuration options are intended to control communication features:

- ***StateMessage*** The option allows State Messages in the system;

• ***StateMsgDefaultValue***	The option allows State Messages to have default values;
• ***StateMsgTimeStamp***	The option allows State Messages to have a time stamp;
• ***EventMessage***	The option allows Event Messages in the system;
• ***EventMsgTimeStamp***	The option allows Event Messages to have a time stamp;
• ***EventMessageOneToN***	If the option turned on, 1:N Event Messages are allowed;
• ***ActivateOnMsg***	If the option turned on task activation on message arrival is supported;
• ***AlarmOnMsg***	If the option turned on an alarm can be linked to message arrival;
• ***SetEventOnMsg***	If the option turned on event setting is allowed on message arrival.

10.4.2 Identifiers

The following names are used in the OSEK Operating System for work with messages:

• ***SymbolicName***	This is a unique name representing a message. It only can be used in conjunction with calls of the message service. A SymbolicName need not be a data type. Variables or constants of SymbolicName can be declared or used.
• ***AccessModeName***	This is a unique name defining access to a message object. Legal names are Send, Receive and SendReceive.
• ***CopyQualifierName***	This is a unique name defining whether local copies of a state message will be created. Legal names are WithCopy and WithoutCopy.

10.4.3 Message Definition

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

```
DefineStateMessage( <MsgName>, <Type>, <Size>, <TStamp>
                    [, <DefaultValue>] );
DefineEventMessage( <MsgName>, <Type>, <Size>, <BufferSize>,
                    <NRec>, <OCheck>, <TStamp> );
ActivateOnMessage( <MsgName>, <TaskId> );
```

```
SetEventOnMessage( <MsgName>, <TaskId>, <EventMask> );
DefineMessageAlarm( <MsgName>, <AlarmId>, <TimeOut> [, <Start>] );
```

The *DefineStateMessage* and *DefineEventMessage* statements defines State and Event message correspondingly. The *ActivateOnMessage* statement is designed to link activation of the specified task with message arrival. The *SetEventOnMessage* is intended to link event setting for the specified task on message arrival. The *DefineMessageAlarm* statement is designed to link an alarm with message arrival - when the message arrives the alarm is restarted. In detail message configuration statements is described in **SECTION 12 System Configuration**.

To refer to a message the constructional elements should be used to declare messages, State and Event ones correspondingly:

```
UsesStateMessage[AsParameter](SymbolicName <Msg>,
                              AccessModeName <Access>,
                              CopyQualifierName <Copy>);
UsesEventMessage[AsParameter](SymbolicName <Msg>,
                              AccessModeName <Access>);
```

This declaration is equivalent to the external declaration of variables. If the syntax `UsesStateMessageAsParameter` or `UsesEventMessageAsParameter` is used it means that a message is the function parameter. Such declaration elements are used in function declarations, see example in **10.4.5 Usage of Messages**.

10.4.4 Run-time Services

OSEK OS grants a set of services for the user to manage tasks. Detailed description of these services is provided in section **16.7 Communication Management Services**. Here only the brief list of them is presented.

Table 10–2 Task Management Run-time Services

Service Name	Description
SendStateMessage	Updates the state message
ReceiveStateMessage	Gets the state message
SendEventMessage	Send the event message
ReceiveEventMessage	Receives (consumes) the event message

Examples of the run-time services usage are provided in **SECTION 16**.

10.4.5 Usage of Messages

Messages are identified via a symbolic name. This symbolic name is used as a variable or a pointer name for the message within the user’s code. This identifier is used for references to the message when the system service is used.

If the message is used with the *WithCopy* qualifier, then the variable is defined within the user’s code by means of using the *UsesStateMessage* or *UsesEventMessage* directives. The message item is referenced as *Msg*.

For example, if the user defines the message `MsgA` having the type `int`, and user’s code contains the directive:

```
UsesStateMessage( MsgA,SendReceive,WithCopy )
```

then the user may access the message data by means of the following statements:

```
ReceiveStateMessage( MsgA, WithCopy );
if( MsgA == 2 ) { MsgA = 1; }
SendStateMessage( MsgA, WithCopy );
```

If the user defines the message `MsgA` with a timestamp and one field having the type `int`, and the user’s code contains the directive:

```
typedef struct tagMYMSGA MYMSGA;
struct tagMYMSGA
{
  TickType timeStamp;
  int x;
};
UsesStateMessage( MsgA, SendReceive, WithCopy )
```

then the user may access the message item fields by means of the following statements:

```
ReceiveStateMessage( MsgA, WithCopy );
if( MsgA.x == 2 ) { MsgA.x = 1; }
MsgA.timeStamp = 100;
SendStateMessage( MsgA, WithCopy );
```

If the message is used with the *WithoutCopy* qualifier, then the pointer is defined within the user’s code by means of using the *UsesStateMessage* directive. The message item is referenced as **Msg* in this case.

For example, if the user defines the message `MsgA` having the type `int`, and user’s code contains the directive:

```
UsesStateMessage( MsgA, SendReceive, WithoutCopy )
```

then the user may access the message item fields by means of the following statements:

```
ReceiveStateMessage( MsgA, WithoutCopy );
if( *MsgA == 2 ) { *MsgA = 1; }
SendStateMessage( MsgA, WithoutCopy );
```

If the user defines the message `MsgA` with a timestamp and one field having the type `int`, and the user's code contains the directive:

```
typedef struct tagMYMSGA MYMSGA;
struct tagMYMSGA
{
  TickType timeStamp;
  int x;
};
UsesStateMessage( MsgA, SendReceive, WithoutCopy )
```

then the user may access the message item fields by means of the following statements:

```
ReceiveStateMessage( MsgA WithoutCopy );
if( MsgA->x == 2 ) { MsgA->x = 1; }
MsgA->timeStamp = 100;
SendStateMessage( MsgA, WithoutCopy );
```

In fact, the user's code will always contain the identifier `MsgA`, which has the type defined by the user in the system generation statements *DefineStateMessage* or *DefineEventMessage*, or a pointer to the variable of this type, if the *WithoutCopy* qualifier is used.

The following examples demonstrate the use of references to the messages for external declarations and as a function parameter.

```
extern UsesStateMessage( MsgA, Send, WithCopy );
int Function1( UsesStateMessageAsParameter( MsgA, Receive,
                                            WithCopy));
TASK TaskA(void)
{  ...
   MsgA = 1;
   SendStateMessage( MsgA, WithCopy) ;
   Function1( MsgA );
   ...
}
```

```
int Function1( UsesStateMessageAsParameter(MsgA, Receive,
                                          WithCopy))
{  ...
   ReceiveStateMessage( MsgA, WithCopy );
   if(MsgA == 1) { ... }
   ...
}
```

SECTION 11
ERROR HANDLING AND SPECIAL ROUTINES

The OSEK Operating System provides for the user tools for error handling and simplest debugging at run time. These are special hook routines with names specified by OSEK OS which are to be developed by the user. In this section error handling at the system configuration stage does not consider, it is described in **SECTION 12 System Configuration**.

11.1 Hook Routines

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

These hook routines are

- called by the operating system, in a special context provided by the operating system, using a special interface;
- implemented by the user, not standardized in functionality;
- not allowed to use OSEK OS run-time services.

In the OSEK OS hook routines are intended for

- tracing or application dependent debugging purposes;
- user defined extensions of the context switch;
- error handling.

The OSEK OS provides the following hook routines - *OSError*, *OSPreTask*, *OSPostTask*. The user must create the code of these routines, OSEK OS only provides description of function prototypes.

- ***OSError*** - 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 **11.2.1 Error Interface**). It is called before returning to the task level or the ISR level.
- ***OSPreTask*** - this hook is called before the operating system enters the context of the task. 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. Also the routine can be used to switch the hardware memory bank if needed.
- ***OSPostTask*** - This hook is called after the operating system leaves the context of the task. 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. Also the routine can be used to switch the hardware memory bank if needed.

If an application supports the memory bank switching (the system property ***HCBankCode***, see **12.3 System Properties Definition**) these routines can be used to switch the desired memory bank before entering/leaving the task code.

Time stamps can be integrated individually into the application software with the help of hook routines *OSPreTask* and *OSPostTask*. The user can set time stamps enabling him to trace the program execution at the following locations before calling operating system services:

- When activating or terminating tasks;
- At explicit points of rescheduling (*ChainTask*, *Schedule*);

The Operating System needs not include a time monitoring feature which ensures that each or only, e.g. the lowest-priority task has been activated in any case after a defined maximum time period. The user can optionally use the hook routines or establish a watchdog task that takes “one-shot displays” of the operating system status.

See examples of programming techniques using the hook routines in section **16.8 Error Handling and Debugging Services**.

11.2 Error Handling

11.2.1 Error Interface

The hook routine *OSError* is provided to handle temporarily and permanently occurring errors within the OSEK Operating System. Its basic framework is predefined and has to be completed by the user. This gives the user a choice of efficient centralized or decentralized error handling.

Three different kinds of errors are distinguished:

- **mild errors** - the Operating System achieved the requested service but has some doubt on the correctness of the application. In this case centralized error treatment is called. Additionally OSEK OS returns the error by the status information for decentralized error treatment.
- **severe errors** - the Operating System could not achieve the requested service, but assumes the correctness of its internal data. In this case centralized error treatment is called. Additionally OSEK OS returns the error by the status information for decentralized error treatment.
- **fatal errors** - the Operating System can no longer assume correctness of its internal data. In this case OSEK OS calls the centralized system shutdown.

The special system routine ***OSShutDown*** is intended to shut down the system in case of the fatal error. This routine aborts the overall system by calling the standard ANSI C function `exit()`. *OSShutDown* may be called both by the user and by the system in case of a fatal error. This service routine is provided by the OSEK Operating System in distinction to the *OSError* routine which should be written by the user.

The OSEK OS error service is called with a parameter that specifies the error. It is up to the user to decide what to do depending on which error has occurred. The OSEK Operating System specifies the following errors:

Table 11–1 OSEK OS Error codes

Error Name	Value	Description
Common Error Codes		
E_OK	0	No error, successful completion
E_NOFUNC	1	The object is not used, the service is rejected
E_NOMSG	2	There is no message available
E_STATE	3	The state of the object is not correct for the required service
E_ACCESS	4	Access to the service/object denied
E_CALLEVEL	5	Access to the service from the ISR is not permitted
E_ID	6	The object ID is invalid
E_LIMIT	7	The limit of services/objects exceeded
E_RESOURCE	8	The task still occupies the resource
E_VALUE	9	A value outside of the admissible limit
E_STACK	10	Internal stack overflow

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.

11.2.2 Extended Status

The OSEK Operating System version with *`Extended Status`* requires more execution time and memory space than the run time 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 the run time version.

The following example can illustrate Extended Status usage:

- If a task is activated in the run time, either “OK” or “Task already activated” is returned. Moreover, in the Extended Status version, the additional status like “Task not defined”, “Maximum number of tasks already activated” or “Stack overflow”, etc. 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 run time version.

11.2.3 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.

11.3 Start-up Routine

The special system routine ***StartUp*** is implemented in the OSEK 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 and it is never returned back to the caller (except start-up errors) but passes the control to the scheduler to schedule the first task to be running. See **APPENDIX A Sample Application** for details.

11.4 Programming Issues

11.4.1 Configuration Options

The following configuration options affect error handling and hook routines:

ErrorHandler	If the option is turned on the user’s hook is called by the system for error handling
ContextSwitchRoutine	If the option is turned on the user’s hooks are called by the system during context switching

11.4.2 Hook Routine Definition

OSEK OS hook routines have to be defined by the user in the system configuration file. The special configuration statement is used for this purpose:

```
DefineHooks( OSError, OSPreTask, OSPostTask )
```

If some name is absent in the definition, this routine will never be called in the system. For details see **SECTION 12 System Configuration**.

SECTION 12
SYSTEM CONFIGURATION

12.1 General

The OSEK Operating System is fully statically configured one. All system properties, the number of system objects and their parameters (characteristics of tasks, counters, alarms, messages, 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.

All application parameters are defined in the special configuration file. This file must conform some grammar rules. It is processed by the separate *`System Generator utility (SG)`* [1]. The System Generator analyzes statements in the configuration file and build output C-language files needed 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 source code.

12.2 Application Configuration File

`Application configuration file` contains the statements which define the system properties and objects. Such file can has any extension and the extension ".def" is suggested by default. The file of this format is processed by the SG utility.

As a result of application configuration file processing SG produces three types of standard C-language files as it is described in **13.2.1 Application Configuration**. SG produces two header files and one or several source files. These files provides the code for all system tables, descriptors, arrays etc. both in ROM and RAM according to the user specified application configuration.

The following command line options are intended to control SG:

---

[1]. Two versions of SG are delivered - the 16-bit version ("sg.exe") and the 32-bit version ("sg32.exe") for Windows NT and Windows 95.

Table 12–1 System Generator command line options

Option	Description	Default value
-c*	Defines * as the data file name	Input file name with ".c" extension
-h*	Defines the header file name	Input file name with ".h" extension
-i*	Defines the path for include files	No
-p*	Defines the property file name	"osprop.h"

12.2.1 Configuration File Grammar

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

- All statements must be written without errors;
- Statements must be in the predefined order;
- It is recommended to avoid conflicting statements (e.g., the system property *TaskOwnStack* is not set, but the own task stack is defined for a task) since it leads to error or warning messages.

12.2.1.1 Statements Types

The definition file can contain the following types of statements:

- comments;
- section declaration;
- include statements;
- file name definition;
- system property definition statements;
- system object definition statements.

The source file may contain comments like used in C-language. Characters enclosed between "/*" and "*/" or started after "//" to the end of line are considered as comment lines and ignored by the System Generator.

The definition file is divided into sections. Each section begins with a section declaration that is the section name enclosed in square brackets. The section must be in the order which is presented below:

• **[Property]**	Defines OS properties (includes statements like <property name> = <property value>)
• **[Scheduler]**	Defines the scheduler via the *DefineScheduler*

	statement
• **[Interrupt management]**	Defines the interrupt masks and the interrupt stack parameters via the *DefineInterrupts* statement
• **[User’s hooks]**	Defines user hooks via *DefineHooks*
• **[Tasks]**	Defines tasks via *DefineTask* and stack pools via the *DefineStackPool* statements
• **[Resources]**	Defines resources via the *DefineResource* statement
• **[Counters]**	Defines the counters presented in the system via the *DefineSystemTimer* and *DefineCounter* statements
• **[Alarms]**	Defines alarms via the *DefineAlarm* statement
• **[State Messages]**	Defines State messages via *DefineStateMessage*, *DefineMessageAlarm*, *ActivateOnMessage* and *SetEventOnMessage*
• **[Event Messages]**	Defines Event messages via *DefineEventMessage*, *DefineMessageAlarm*, *ActivateOnMessage* and *SetEventOnMessage*

The configuration file must always contain sections *[Property]*, *[Scheduler]*, *[Tasks]* and *[Counters]*. Other sections can be empty or omitted.

The configuration file can contain include statements according to ANSI C language rules. Such statements may look like the following ones:

```
#include <file name>
#include "file name"
#include "path\file name"
```

For example:

```
[Tasks]
DefineTask( TaskB, BASIC|ACTIVATE|OWNSTACK,2, TaskBStart,,,,64 );
#include <task.def>

[Resources]
#include "c:\app\src\resource.def"
```

Include files can be anywhere in the configuration file, but they must conform the configuration file format. Statements from include files are processed as statements of the main configuration file. The System Generator searches include files either in the current directory or in the directories specified in the SG command line.

12.2.2 Separate Output Files

The possibility to have separate C-language source files for different system objects is important for the user. These files are intended to allocate RAM areas and to initialize some parameters. Separate files allow the user to place different data in different memory areas by means of separate compilation.

The initialization part of a different object may be written into different data files. To create separate output files for some sections the first statement after the section header must be the definition of the output file name. If file name is undefined then the common data file will be used. The file name definition has the following form:

```
>><file name>;
```

The initialization part of a different object may be written into different data files. The statements like

```
[Tasks]
>> c:\usr\srs\task.c;
```

are used to output the *[Tasks]* section into the separate file "task.c". All other sections will be output into the default file.

Statements for system property definition and system objects definition are described in subsections below.

12.3 System Properties Definition

All property definitions are placed in the *[Property]* section of the configuration file. Operating System version will be built according to the specified values (e.g., the certain Conformance Class). The property definition must have the following form:

```
<Property name> = <Property value>;
```

Property definition statements can be placed in any order. There are two types of property definitions:

- **two state property -** the property can have only the values ON or OFF which defines whether this OS option is supported in this configuration or not. This property could have a default value;
- **list-of-values property** - the property has a set of possible values that are chosen by the user. This property could have a default value.

If a property is not defined, then the default value is used.

The list of properties is presented in the table which follows.

Table 12–2 OSEK Operating System Properties

Property name	Description
General properties	
TargetMCU	Specifies the CPU type. Now only HC08 is the valid value
HCBasePage	Defines that the base memory page (0x00 - 0xFF) will be used for system variables
HCLowPower	Defines that Low Power Mode will be used instead of the scheduler idle loop when there no active task
HCBankCode	Defines that memory bank switching is supported by the system
OS properties	
ConformanceClass	Defines the functionality of OSEK OS according to the specified CC
SchedulerPolicy	Defines the scheduling policy
SimpleScheduler	The ON option defines that the simplified scheduler will be used in the system. In this case each task must have its unique priority.
ExtendedStatus	Defines whether additional checks of system services execution are performed by OS or not; extended return codes are provided in case of Extended Status
UseMainStack	If the option is turned ON, the same memory area is used for the stack of the `main()` function, for ISR stack and for the scheduler stack. It saves memory but the user should provide enough RAM for the stack.
UseSameContext	Defines whether the same run time context frame is used both for non-preemptive and preemptive tasks (for mixed-preemptive scheduling only)
MultiplyActivation	The OFF option in Conformance Classes BCC3, ECC1, ECC2 this option disables task multiply activation.
StackPool	Defines whether stack pools are supported or not
PersistentNode	Allows persistent task node allocation
PersistentStack	Allows persistent task stack allocation
NodeStack	Defines the presence of task node stacks in the system
TaskOwnStack	Allows explicit task stack allocation

Table 12–2 OSEK Operating System Properties

Property name	Description
TaskIndexMethod	If the option is turned on then the intermediate vector of the pointers to the tasks nodes is used (fast and deterministic access to task nodes)
EntryExitISR	Defines whether system services to enter and exit ISRs are implemented or not
InterruptMaskControl	Defines whether interrupt masks are controlled by the Operating System or not
ErrorHandler	If the option is turned ON, the user’s hook is called by the system for error handling
ContextSwitchRoutine	If the option is turned on the user’s hooks are called by the system during context switching
InternalErrorHandler	Defines whether the internal error handler is implemented
Resources	Defines is resource management implemented or not
FastResource	The ON option accelerates work with resources; this option is strongly recommended only for debugged applications
Events	Defines are events implemented or not
Counters	Defines whether counters are implemented or not
CounterSize	Defines the counter size (byte, word or long word)
Alarms	Defines whether alarms are implemented or not
AlarmList	The option accelerates work with alarms
StateMessage	Defines whether state messages are implemented or not
StateMsgDefaultValue	Allows state messages to have a timestamp
StateMsgTimeStamp	Allows state messages to have a default value
EventMessage	Defines whether event messages are implemented or not
EventMsgTimeStamp	Allows event messages to have a timestamp
EventMsgOneToN	Allows 1:N communication for event messages
ActivateOnMsg	Allows task activation on message arrival
AlarmOnMsg	Allows alarm resetting on message arrival
SetEventOnMsg	Allows event setting on message arrival

Table 12–2 OSEK Operating System Properties

Property name	Description
Properties to exclude particular OS services	
ActivateTask	The OFF option excludes the service *ActivareTask* from the OSEK OS code
TerminateTask	The OFF option excludes the service *TerminateTask* from the OSEK OS code
ChainTask	The OFF option excludes the service *ChainTask* from the OSEK OS code
ChainTaskItself	The option can be turned OFF if no tasks that chain itself. It decreases the task control block size.
Schedule	The OFF option excludes the service *Schedule* from the OSEK OS code
GetTaskId	The OFF option excludes the service *GetTaskId* from the OSEK OS code
GetTaskState	The OFF option excludes the service *GetTaskState* from the OSEK OS code
GetResource	The OFF option excludes the service *GetResource* from the OSEK OS code
ReleaseResource	The OFF option excludes the service *ReleaseResource* from the OSEK OS code
EnterISR	The OFF option excludes the service *EnterISR* from the OSEK OS code
LeaveISR	The OFF option excludes the service *LeaveISR* from the OSEK OS code
EnableInterrupt	The OFF option excludes the service *EnableInterrupt* from the OSEK OS code
DisableInterrupt	The OFF option excludes the service *DisableInterrupt* from the OSEK OS code
GetInterruptMask	Being turned off the option excludes the service *GetInterruptMask* from the OSEK OS code
InitCounter	The OFF option excludes the service *InitCounter* from the OSEK OS code
CounterTrigger	The OFF option excludes the service *CounterTrigger* from the OSEK OS code

Table 12–2 OSEK Operating System Properties

Property name	Description
GetCounterValue	The OFF option excludes the service *GetCounterValue* from the OSEK OS code
GetCounterInfo	The OFF option excludes the service *GetCounterInfo* from the OSEK OS code
SetRelAlarm	The OFF option excludes the service *SetRelAlarm* from the OSEK OS code
SetAbsAlarm	The OFF option excludes the service *SetAbsAlarm* from the OSEK OS code
CancelAlarm	The OFF option excludes the service *CancelAlarm* from the OSEK OS code
GetAlarm	The OFF option excludes the service *GetAlarm* from the OSEK OS code
SetEvent	The OFF option excludes the service *ActivareTask* from the OSEK OS code
ClearEvent	The OFF option excludes the service *SetEvent* from the OSEK OS code
GetEvent	The OFF option excludes the service *GetEvent* from the OSEK OS code
WaitEvent	The OFF option excludes the service *WaitEvent* from the OSEK OS code
SendStateMessage	The OFF option excludes the service *ActivareTask* from the OSEK OS code
ReceiveStateMessage	The OFF option excludes the service *SendStateMessage* from the OSEK OS code
SendEventMessage	The OFF option excludes the service *SendEventMessage* from the OSEK OS code
ReceiveEventMessage	The OFF option excludes the service *ReceiveEventMessage* from the OSEK OS code

Table 12–2 shows default values of system properties.

Table 12–3 OSEK Operating System properties default values

Property name	Value set	Default value
General properties		
TargetMCU	HC08	HC08
HCBasePage	ON/OFF	OFF
HCLowPower	ON/OFF	OFF
HCBankCode	ON/OFF	OFF
OS properties		
ConformanceClass	BCC1 BCC2 BCC3 ECC1 ECC2	ECC2
SimpleScheduler	ON/OFF	OFF
SchedulerPolicy	NONPREEMPT FULLPREEMPT MIXPREEMPT	NONPREEMPT
ExtendedStatus	ON/OFF	OFF
UseMainStack	ON/OFF	OFF
UseSameContext	ON/OFF	OFF
MultiplyActivation	ON/OFF	ON
StackPool	ON/OFF	OFF
NodeStack	ON/OFF	ON
PersistentNode	ON/OFF	OFF
PersistentStack	ON/OFF	OFF
TaskOwnStack	ON/OFF	OFF
TaskIndexMethod	ON/OFF	OFF
EntryExitISR	ON/OFF	ON
InterruptMaskControl	ON/OFF	ON
ErrorHandler	ON/OFF	OFF

Table 12–3 OSEK Operating System properties default values

Property name	Value set	Default value
ContextSwitchRoutine	ON/OFF	OFF
InternalErrorHandler	ON/OFF	OFF
Resources	ON/OFF	ON (OFF for BCC1,BCC2)
FastResource	ON/OFF	OFF
Events	ON/OFF	ON (OFF for BCC1-BCC3)
Counters	ON/OFF	ON
CounterSize	8 16 32	32
Alarms	ON/OFF	ON
AlarmList	ON/OFF	OFF
StateMessage	ON/OFF	ON
StateMsgDefaultValue	ON/OFF	OFF
StateMsgTimeStamp	ON/OFF	OFF
EventMessage	ON/OFF	ON
EventMsgTimeStamp	ON/OFF	OFF
EventMsgOneToN	ON/OFF	OFF
ActivateOnMsg	ON/OFF	ON (OFF for BCC1,BCC2)
AlarmOnMsg	ON/OFF	ON (OFF for BCC1,BCC2)
SetEventOnMsg	ON/OFF	ON (OFF for BCC1-BCC3)
Properties to exclude particular OS services		
Any such property (see **Table 12–2**)	ON/OFF	ON

The examples of system properties definitions are:

```
TargetMCU = HC08;
ConformanceClass = ECC2;
```

```
ExtendedStatus = ON;
TaskIndexMethod = OFF;
PersistentNode = OFF;
PersistentStack = OFF;
TaskOwnStack = OFF;
StackPool = OFF;
SchedulerPolicy = MIXPREEMPT;
```

12.4 System Objects Definition

All objects that are controlled by the Operating System - tasks, resources, alarms, messages, counters, ISRs and even the scheduler - 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 according the order in which they are placed in the configuration file. Parameters are delimited by a comma. If some parameters in the middle of the list are omitted the corresponded commas must stay in place; the last parameters can be removed with their commas.

Some parameters represents memory addresses. They are defined either automatically or by the user. If the user does not need to control memory address such parameters should be omitted in definition statements. In this case SG automatically creates the code to allocate memory areas. If the user explicitly specifies the address then the symbolic name (e.g. `MyTaskStack`) should be provided in the corresponded position. This name must be defined in a user's file (e.g. an array can be defined for memory area) and declared as external name before the generated source files will be used. Such techniques is demonstrated in sample application, see **APPENDIX A Sample Application**.

12.4.1 DefineScheduler

Syntax:

```
DefineScheduler( <NumberOfTasks>, <NumberOfPriorities>,
               [<SchedulerStackSize>], [<SchedulerStackAddress>],
               [<TaskNodeStackSize>] ,[<TaskNodesStackAddress>]);
```

Description:

This statement is used to define scheduler parameters. Only one such statement must be in the configuration file. The following parameters are used:

- *<NumberOfTasks>* specifies the maximum number of tasks in the non-suspended state, available in the system. In fact, this parameter specifies the number of task control blocks, allocated in the system;
- *<NumberOfPriorities>* specifies the number of priority classes in the system. The priorities range from 0 to *<NumberOfPriorities>* minus 1. In fact, this

parameter specifies the number of elements in the scheduler queues. This parameter is ignored if the system configuration option *SimpleScheduler* is turned on, since in this case it must be equal the number of tasks;

- *<SchedulerStackSize>* specifies the size (in bytes) of the scheduler's idle loop stack. It should not be less than the interrupt stack frame. If the system property *UseMainStack* is set this parameter (and <*SchedulerStackAddress*> too) can be omitted;
- *<SchedulerStackAddress>* specifies the bottom of the scheduler stack. This address is used explicitly to provide a way to optimize stacks allocation. This parameter can be omitted;
- *<TaskNodeStackSize>* specifies the size of the stack (in bytes) per each task node. The total size of the memory area for task nodes' stacks is defined as multiplication of this parameter by the <*NumberOfTasks*> parameter. This parameter is ignored if the system configuration option *NodeStack* is turned off;
- *<TaskNodesStackAddress>* specifies the bottom of task nodes' stacks. This address is used explicitly to provide a way to optimize stacks allocation. This parameter is ignored if the system configuration option *NodeStack* is turned off. This parameter can be omitted.

Example:

```
DefineScheduler( 7, 5, 16, SchedStack );
```

This statement defines that there are 7 task nodes and 5 task priorities in the system. Scheduler stack size equals 16 bytes and has the bottom at the address referenced as `SchedStack` (`SchedStack` is defined as an array of 16 bytes size). There are no task node stacks.

12.4.2 DefineInterrupts

Syntax:

```
DefineInterrupts( <DisableMask>, <EnableMask>, <TaskMask>,
                  [<ISRStackSize>,] [<ISRStackAddress>] );
```

Description:

This statement is used to define interrupt management data. Only one such statement must be in the configuration file. The following parameters are used:

- *<DisableMask>* specifies the value of the interrupt mask, which corresponds to all interrupts disabled;
- *<EnableMask>* specifies the value of the interrupt mask, which corresponds to all interrupts enabled;
- *<TaskMask>* specifies the value of the interrupt mask, which, typically,

corresponds to the middle level of enabled interrupts in the system;

- *<ISRStackSize>* specifies the size (in bytes) of the ISR stack. If the system property *UseMainStack* is set this parameter (and <*ISRStackAddress*> too) can be omitted;
- *<ISRStackAddress>* specifies the bottom of the ISR stack. This address is used explicitly to provide a way of optimizing stacks allocation. This parameter can be omitted.

Example:

```
DefineInterrupts( 0x8,0x0,0x0, 64);
```

Such statement defines masks for “I” bit of CPU08 and the ISR stack size. System Generator produces the code to allocate memory for the ISR stack.

12.4.3 DefineHooks

Syntax:

```
DefineHooks( <Error handler>, <PreTask handler>,
             <PostTask handler> )
```

Description:

This statement is used to define user hooks entry points. Only one such statement must be in the configuration file. Parameters are the following:

- *<Error handler>* is the entry point of the error handling hook routine;
- *<PreTask handler>* is the entry point of the hook routine executed before entering the tasks code;
- *<PostTask handler>* is the entry point of the hook routine executed after leaving the task code.

Example:

```
DefineHooks ( OSError, OSPreTask, OSPostTask);
```

The statement declares user’s hook routines. SG creates needed references. The routines should be provided by the user:

```
void OSError( StatusType Error )
{
...
}
void OSPreTask( TaskRefType TaskId, ... )
{
...
}
```

```
void OSPostTask( TaskRefType TaskId, ... )
{
...
}
```

12.4.4 DefineTask

Syntax:

```
DefineTask( <TaskName>, <TaskProperties>, <TaskPriority>,
            <EntryPoint> [,<TaskBank>] [,<InterruptMask>]
            [,<TaskStack>] [,<TaskStackSize] );
```

Description:

This statement is used to define task data. Several statements can be in the configuration file, each statement defines one task. The following parameters are used:

- *<TaskName>* specifies the symbolic task's name. This name is to be used by the user's code as symbolic reference to the task;
- *<TaskProperties>* specifies properties of the task. These properties are written via the OR operator. Task properties are described in table **Table 4–3**;
- *<TaskPriority>* specifies the task's priority. The highest priority has value 0;
- *<EntryPoint>* defines the starting address of the task. Typically this parameter is the name of the function, which is the task body;
- *<TaskBank>* defines the memory bank for the task code. This parameter may be omitted;
- *<InterruptMask>* defines the starting task's interrupt mask. If this parameter is omitted, then the default (middle) value defined in *DefineInterrupts* is used;
- *<TaskStack>* specifies task stack assignment. If the *NodeStack* task property is set, this parameter is ignored or should be omitted. If the *PoolStack* system property is set, then this parameter must be a symbolic name of the stack pool (see section **4.6.4 Task Stack**). If the *OwnStack* task property is set, then this parameter may contain the address of the bottom of the stack explicitly assigned for the task;
- *<TaskStackSize>* specifies the task stack size, if the *OwnStack* task property is set. Otherwise the parameter should be omitted or it will be ignored.

Example:

```
DefineTask(TASKA, EXTENDED|PREEMPT|ASSIGNNODE|POOLSTACK|ACTIVATE,
           3, TaskA, , , POOL2 );
DefineTask( TASKB, BASIC|ACTIVATE|OWNSTACK,2, TaskBStart,,,,64 );
```

SG produces the code to allocate memory for the stack of TASKB.

12.4.5 DefineStackPool

Syntax:

```
DefineStackPool( <StackPoolName>, <SizeOfStack>,
                 <NumberOfStacks> [,<AddressOfStackArea>] );
```

Description:

This statement is used to define stack pool data. Several statements can be in the configuration file, each statement defines one pool. The following parameters are used:

- *<StackPoolName>* specifies the stack pool name. This name is to be used during task generation to refer to the stack pool of the desired size;
- *<SizeOfStack>* specifies the size of stacks in the stack pool;
- *<NumberOfStack>* specifies the number of stack buffers;
- *<AddressOfStackArea>* specifies the name of the array where stacks are located. This optional parameter serves for explicit memory allocation for the task stack pool.

Example:

```
DefineStackPool(POOL1, 15, 24 );
```

This statement defines the pool with 24 stack buffers of 15 bytes size.

12.4.6 DefineResource

Syntax:

```
DefineResource( <ResourceName>, <ResourcePriority> );
```

Description:

This statement is used to define a resource. Several statements can be in the configuration file, each statement defines one resource. The following parameters are used:

- *<ResourceName>* specifies the resource name for references to it;
- *<ResourcePriority>* specifies the priority assigned to the resource which is used by resource management to support the priority ceiling protocol.

Example:

```
DefineResource( SCIRes, 1);
```

The name `SCIRes` have to be used to refer to this resource in the user’s code.

12.4.7 DefineSystemTimer

Syntax:

```
DefineSystemTimer( <CounterID>, <maxallowedvalue>,
                   <ticksperbase>, <tickduration> [,<mincycle>]
                  [, <HardwareType> [, <HardwareParams>]] );
```

Description:

This statement is used to declare the counter as a System Timer. An additional parameter - duration of a tick in nanoseconds - is defined for the system timer. The *OSMAXALLOWEDVALUE*, *OSTICKPERTIME*, *OSTICKDURATION*, *OSMINCYCLE* constants are defined by this statement. This statement always must be in the configuration file. The following parameters are defined for the system timer:

- *<CounterName>* specifies the system counter name for references to it;
- *<maxallowedvalue>* specifies the maximum allowed system timer value (in ticks);
- *<ticksperbase>* specifies the number of ticks required to reach 10 milliseconds. SG does not check correctness of this value;
- *<tickduration>* specifies duration of a system timer tick in nanoseconds. The user has to calculate this parameter according to the tick source frequency;
- <*mincycle*> specifies the minimum allowed number of ticks for a cyclic alarm. The parameter has a sense only for the system with the Extended Status since it is checked only in Extended Status systems. The parameter can be omitted;
- *<HardwareType>* is intended to select the hardware interrupt source for the system counter (among the accessible MCU devices). See **SECTION 14 Platform-Specific Features** for details about possible meanings of this parameters. This parameter can be omitted;
- *<HardwareParams>* is the list of parameters to tune the selected hardware interrupt source. One or more parameters can be here in accordance to the hardware features. For more details see **SECTION 14 Platform-Specific Features**. This parameter(s) can be omitted.

If hardware related parameters are defined, the code to initialize the system timer hardware and the interrupt handler are automatically provided for the user as a part of OSEK OS. In that case the user does not have to care about handling of this interrupt and he/she can not change the provided code. If the parameters are not defined the user has to provide the code to initialize the hardware and handle the interrupt.

Example:

```
DefineSystemTimer( Watch, 24, 1, 5, PIT, 4, 128 );
```

The name `Watch` have to be used to refer to the system counter in the user's code. The counter rolls over after it reaches value 24.

12.4.8 DefineCounter

Syntax:

```
DefineCounter( <CounterName>, <maxallowedvalue>, <ticksperbase>
              [,<mincycle>] );
```

Description:

This statement is used to define counter parameters. Several statements can be in the configuration file, each statement defines one counter. The following parameters are used:

- *<CounterName>* specifies the counter name for references to it;
- *<maxallowedvalue>* specifies the maximum allowed counter value;
- *<ticksperbase>* specifies the number of ticks required to reach a counter-specific unit. The *<tickperbase>* usage is up to the user;
- *<mincycle>* specifies the minimum allowed number of ticks for a cyclic alarm. If this parameter is omitted it has default value 0. The parameter has a sense only for the system with the Extended Status since it is checked only in Extended Status systems.

Example:

```
DefineCounter( ANGLE, 24, 1 );
```

The name `ANGLE` have to be used to refer to this counter in the user's code. The counter rolls over after it reaches value 24.

12.4.9 DefineAlarm

Syntax:

```
DefineAlarm( <AlarmName>, <CounterName>, <TaskName>
            [, <EventMask>]);
```

Description:

This statement is used to define alarm specific parameters. Several statements can be in the configuration file, each statement defines one alarm. The referenced counter and task must be already defined. The following parameters are used:

- *<AlarmName>* specifies the alarm name for references to it;

- *<CounterName>* specifies a reference to the assigned counter;
- *<TaskName>* specifies a reference to the task being notified when the alarm expires;
- *<EventMask>* specifies the event mask to be set when the alarm expires (only if events control is supported by the system). If the parameter is omitted then the specified task is activated when the alarm expires. Otherwise, the specified event is set and in this case the specified task must be Extended one.

Example:

```
DefineAlarm( Alarm1, Watch, TaskA, 0x01);
```

This statement defines that the alarm `Alarm1` is linked to the counter `Watch` and the task `TaskA` is notified by the specified event setting.

12.4.10 DefineStateMessage

Syntax:

```
DefineStateMessage( <MsgName>, <Type>, <Size>, <TStamp>
                    [,<DefaultValue>] );
```

Description:

This statement is used to define the state message. Several statements can be in the configuration file, each statement defines one State Message Object. The following parameters are used:

- *<MsgName>* specifies the symbolic name of the message. This name is to be used by the user's code as symbolic reference to the message;
- *<Type>* is the C data type of the message item. 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). 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;
- *<Size>* is the length of the message item in bytes. This value doesn't include the timestamp length;
- *<TStamp>* is a symbolic name for a timestamp qualifier. If the message includes a timestamp, then this parameter must have the value ***WithTimeStamp***, otherwise it must have the value ***WithoutTimeStamp***;
- *<DefaultValue>* is the address of the ROM-based variable, which holds the default value of the message. This variable must have the type *<Type>*, and

must be located in the user’s code. The parameter may be omitted.

NOTE:

If a message has a timestamp it always has the user defined data type since the message with timestamp is represented by a structure (struct). But the message size should reflect only the size of the message body (without timestamp).

Example:

```
DefineStateMessage( MsgA, char, sizeof(char), WithoutTimeStamp);
DefineStateMessage( MsgB, MSGTYPE, sizeof(int),
                   WithTimeStamp);
```

These statements defines two State Messages. The first one is the element of the `char` type and it has a timestamp. The second message has the user defined type and has no timestamp.

The second message `MsgB` has the following structure:

```
typedef struct tagMSG MSGTYPE;
struct tagMSG
{
 TickType timeStamp;
 int x;
};
MSGTYPE MsgB;
```

12.4.11 DefineEventMessage

Syntax:

```
DefineEventMessage( <MsgName>, <Type>, <Size>, <BufferSize>,
                    <NRec>, <OCheck>, <TStamp> );
```

Description:

This statement is used to define the event message. Several statements can be in the configuration file, each statement defines one State Message Object. The following parameters are used:

- *<Msg>* specifies the symbolic name of the message. This name is to be used by the user’s code as symbolic reference to the message;
- *<Type>* is the C data type of the message item. This parameter is identical to the corresponded parameter in *DefineStateMessage*, see **12.4.10**;
- *<Size>* is a length of the message item in bytes. This value doesn’t include the timestamp length;

- *<BufferSize>* is the capacity of the FIFO queue, i.e. the number of message items into the FIFO queue;
- *<NRec>* is the number of local task-receivers of the message. If it has a value greater than 1, then it is assumed, that several (*<NRec>*) tasks may receive a message item from the Event Message queue;
- *<OCheck>* is a symbolic name to indicate the FIFO overflow. If an overflow must be indicated, then this parameter must have the value ***WithOverwriteCheck***, otherwise it must have the value ***WithoutOverwriteCheck***;
- *<TStamp>* is a symbolic name for a timestamp qualifier. If the message includes a timestamp, then this parameter must have the value ***WithTimeStamp***, otherwise it must have the value ***WithoutTimeStamp***.

Example:

```
DefineEventMessage( MsgC, MSGCTYPE, sizeof(int),
                    6,3,WithOverwriteCheck, WithTimeStamp);
```

The statement defines the Event Message with 6 message items of the `MSGCTYPE` type, where the message item include the message body of the `int` type and the timestamp. The message is to be sent to 3 receivers with a timestamp and overwrite indication.

12.4.12 DefineMessageAlarm

Syntax:

```
DefineMessageAlarm( <MsgName>, <AlarmName>, <TimeOut>
                    [,<Start>] );
```

Description:

This statement is used to define the alarm to be restarted when a message arrives. The alarm is treated as the relative one and it will be restarted each time when message arrives. Several statements can be in the configuration file, one alarm per message. The following parameters are used:

- *<MsgName>* specifies a symbolic name of the message. This message must be previously defined via the *DefineStateMessage* or *DefineEventMessage* statements;
- *<AlarmName>* specifies the alarm attached to the message. This alarm must be previously defined via the *DefineAlarm* statement;
- *<TimeOut>* is a value of the time-out to be used when the alarm restarted. This parameter is specified in “ticks” of the counter linked with the alarm defined. The value is used to set the number of counter ticks after that the alarm expires (starting from the current counter value);

- *<Start>* specifies whether the alarm must be restarted during system start-up. This parameter may have the value ***Restart***, if the alarm must be restarted during start-up, or ***NoRestart***, if no. Restarting of the alarm may be used to trap the loss of the first message. If the parameter is omitted, then the ***NoRestart*** value is used by default.

Example:

```
DefineMessageAlarm( MsgC, Alarm1, 13 );
```

The statement attaches `Alarm1` to `MsgC`. The alarm is triggered after 13 ticks of the associated counter.

12.4.13 ActivateOnMessage

Syntax:

```
ActivateOnMessage( <MsgName>, <TaskName> );
```

Description:

This statement is used to define task activation when a message arrives. Several statements can be in the configuration file, one statement per message. The following parameters are used:

- *<MsgName>* specifies a symbolic name of the message. This message must be previously defined via the *DefineStateMessage* or *DefineEventMessage* statements;
- *<TaskName>* specifies the task attached to the message. This task is activated when the message arrives, and it must be previously defined via the *DefineTask* statement.

Example:

```
ActivateOnMessage( MsgA, TaskA);
```

12.4.14 SetEventOnMessage

Syntax:

```
SetEventOnMessage( <MsgName>, <TaskName>, <EventMask> );
```

Description:

This statement is used to define task event setting for a given task when a message arrives. Several statements can be in the configuration file, one statement per message. The following parameters are used:

- *<MsgName>* specifies a symbolic name of the message. This message must be previously defined via the *DefineStateMessage* or *DefineEventMessage* statements;

- *<TaskName>* specifies the task, attached to the message. This event for this task is set when a message arrives. The task shall be previously defined via the *DefineTask* statement;
- *<EventMask>* specifies the event mask to be set for the task when a message arrives.

Example:

```
SetEventOnMessage( MsgD, TaskA, 0x100 );
```

12.5 System Generator Warning and Error Messages

The system generator checks the compatibility of properties, parameters and limits and reports about possible errors via error messages. The error messages are divided into errors and warnings. The error code is presented by the following string ***LNN##***:

- ***L*** is equal to "**E**" for error and" **W**" for warning;
- ***NN*** presents the group number;
- *##* presents the error number inside the group.

An error message includes the file name, the line number, the error code and a short error (or warning) description. The error message has one of the following formats:

- `<filename>(<line number>) : error ENN## : <message>`
- `<filename>(<line number>) : warning WNN## : <message>`

The generated messages are presented in table **Table 12–4**:

Table 12–4 System Generator Error Messages

Code	Message
General	
E0001	syntax error
E0002	invalid token '<token>'
E0003	unexpected end of file found in comment
E0010	not enough memory
E0011	cannot open source file: <name>
E0011	cannot open include file: <name>

Table 12–4 System Generator Error Messages

Code	Message
E0011	cannot open output file: <name>
E0012	cannot read file: <name>
E0013	cannot write file: <name>
E0014	too many include files : depth = <level>
E0030	’identifier’ : system object redefinition
Command-line error	
W0020	ignoring unknown option <flag>
E0020	<flag> requires an argument
E0021	no input file specified
E0022	too many include paths specified
Property	
E0101	double property *<name>* definition (property *<name>* was defined in line ##)
E0102	undefined property definition
E0103	undefined value for property <name>
E0104	incorrect property <name> setting
E0105	method of task stack assignment is not defined
E0106	*SimpleScheduler* property can not be used with *Resources* property
Interrupt Management	
E0110	interrupt masks shall be defined
E0111	interrupt stack shall be defined
W0110	*UseMainStack* property is turned on, parameter ignored
Error Handling	
E0120	error hook shall be defined
E0121	context switch routines shall be defined
Scheduler	
E0201	the number of task control blocks shall be greater than 0
E0202	number of priorities shall be greater than 0

Table 12–4 System Generator Error Messages

Code	Message
E0203	size of scheduler stack shall be defined
E0204	size of task node stack shall be defined
E0205	number of task control blocks less than priority range
W0201	*NodeStack* property is turned off, parameter ignored
W0202	there are unused task control blocks
Task	
E0301	task may be either basic or extended (more than one task type attribute)
W0301	task type shall be defined
E0302	only basic tasks are allowed in basic conformance classes
W0302	task type set to basic by default
W0303	task preemptive property shall be defined
E0303	task may be either preempt or nonpreempt (more than one task preemptive property)
W0304	non-preemptive tasks are not supported by full-preemptive scheduler
W0304	preemptive tasks are not supported by non-preemptive scheduler
W0305	task preemptive property set to non-preempt
W0305	task preemptive property set to preempt
W0306	*PersistentNode* property is turned off, parameter ignored
W0307	*PersistentStack* property is turned off, parameter ignored
W0308	Persistent stack may be assigned only for task with persistent node, parameter ignored (*ASSIGNSTK* must be used with *ASSIGNNODE*)
W0309	persistent stack shall be assigned from stack pool, parameter ignored
W0310	*TaskOwnStack* property is turned off, parameter ignored
W0311	*StackPool* property is turned off, parameter ignored
E0304	task priority exceeds the maximum limit
E0305	the number of task nodes is not enough to allocate persistent nodes
E0306	the stack pool parameter shall be defined
E0307	the *<name>* stack pool is undefined

Table 12–4 System Generator Error Messages

Code	Message
E0308	the basic task cannot be notified by event setting
E0309	at least one task shall be defined
E0310	more than one task stack attachment method
E0311	task has no stack
E0312	the stack size parameter shall be defined
Resource	
E0401	resource priority exceeds the maximum limit
Counter	
E0501	system timer is already defined
E0502	system timer is undefined
Alarm	
E0601	assigned counter *<name>* for alarm *<name>* is undefined
E0602	assigned task *<name>* for alarm *<name>* is undefined
W0601	Events property is turned off, parameters ignored
Message	
W0701	*StateMsgDefaultValue* property is turned off, parameter ignored
W0702	*StateMsgTimeStamp* property is turned off, parameter ignored
W0703	*EventMsgTimeStamp* property is turned off, parameter ignored
W0704	*EventMsgOneToN* property is turned off, parameter ignored
W0705	*AlarmOnMsg* property is turned off, definition ignored
W0706	message shall be defined before its alarm
W0707	message shall be defined before its notification
W0708	*ActivateOnMsg* property is turned off, definition ignored
W0709	*SetEventOnMsg* property is turned off, definition ignored
E0701	assigned alarm *<name>* for message *<name>* is undefined
E0702	alarm for message *<name>* was defined in line ##
E0703	notified task *<name>* for message *<name>* is undefined

Table 12–4 System Generator Error Messages

Code	Message
E0704	notification for message *<name>* was defined in line ##
E0705	message <name> : parameters are undefined

More detailed description of System Generator error messages is provided in **APPENDIX D System Generation Error Messages**

SECTION 13
BUILDING OF APPLICATION

13.1 Application Structure

An application developed on the OSEK 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 OSEK OS to pass data and synchronize each other.

Tasks and ISRs are considered as system objects. Resources, State and Event messages, counters, alarms, hook routines are also considered as system objects, because they are controlled by the Operating System. An application typically has also configuration tables for different system objects, stack buffer pools and other entities. To create an application the user should develop the desired application structure with all needed 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 **SECTION 12 System Configuration** about this file. After processing files with system object descriptors are created automatically. These files provides 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.

13.2 Action Sequence to Build an Application

To build an application using the OSEK Operating System the user should perform a set of actions. These actions are relatively simple since the most important requirement is clearly understanding of application's algorithm. The actions includes creating of 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 OSEK OS code. This process is shown in **Figure 13–1 Application building process**.

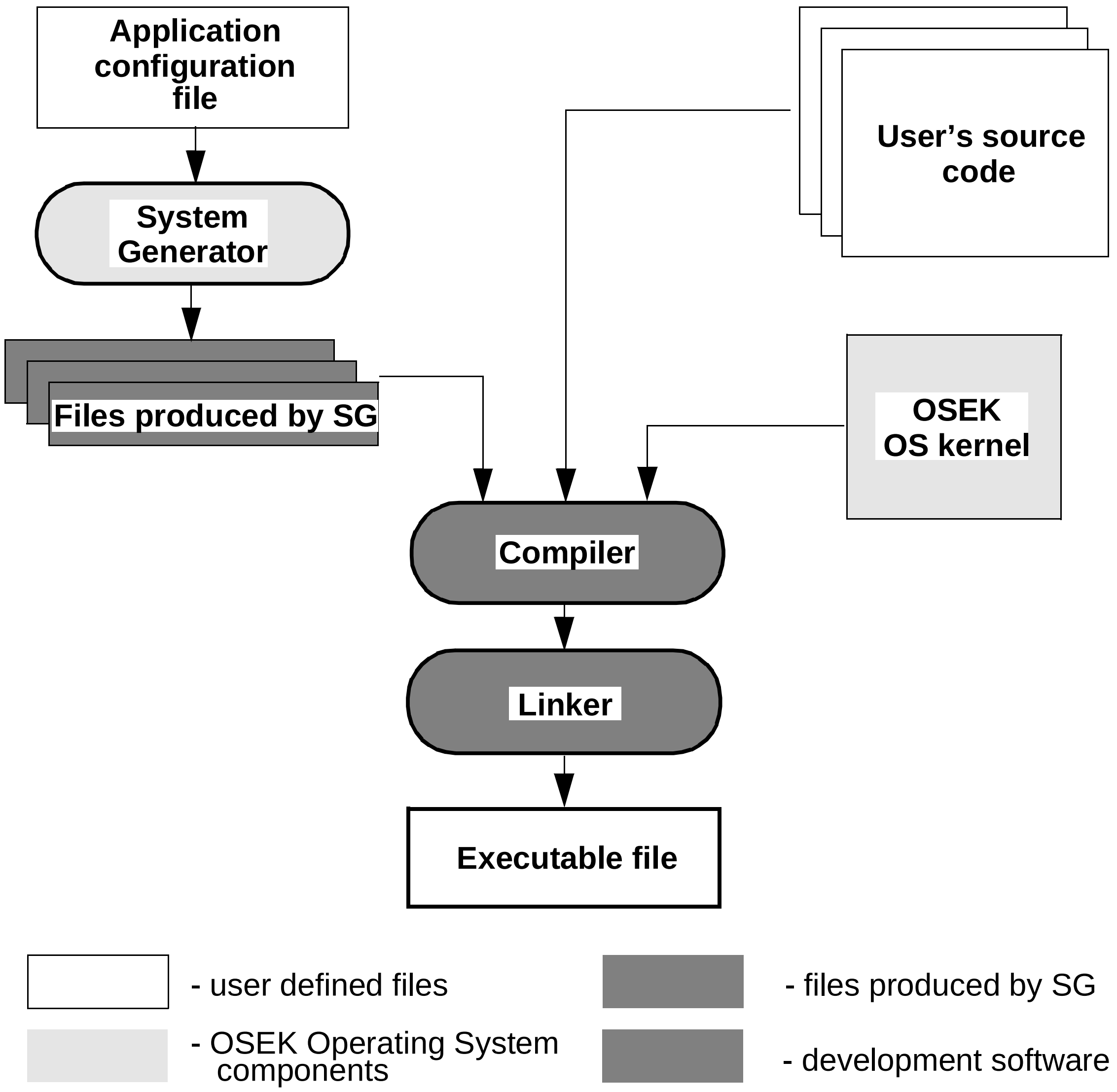

Figure 13–1 Application building process

13.2.1 Application Configuration

Applications built using OSEK OS are configured statically via the special configuration file. **SECTION 12 System Configuration** describes the structure of such file and all possible statements in detail. This configuration file defines system specific parameters as well as system objects. Such file can has any extension and the extension ".def" is suggested by default.

The configuration file has to be processed by the special utility named System Generator (SG). This utility is delivered as one of the part of the OSEK Operating

System. This tool runs as a simple MS-DOS application and produces header and source files. The command line options are described in **12.2 Application Configuration File**.

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 configuration of the operating system, in other word - system properties. 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 "**osprop.h**" but the user can assign another name (see **13.2.2 Source Files**).
2. The header file which contains definitions of data types and constants, external declarations of variables which needed to describe system objects. This file is used to compile application files. By default System Generator uses the input file name for this output file with ".h" extension.
3. The source file which contains initialized data, memory allocation for system objects. This file is compiled with "osprop.h" and other header files and then linked together with other application and OS files. The user can specify several files of this type to separately define different objects. By default System Generator uses the input file name for this output file with "c" extension.

The possibility to have separate C-language source files for different system objects is important for the user. These files are intended to allocate RAM areas and to initialize some parameters. Separate files allow the user to place different data in different memory areas by means of separate compilation.

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.

13.2.2 Source Files

OSEK Operating System 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 after OSEK OS installation, as it is described in OSEK OS Release Guide. 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 "osprop.h" describing system properties defines which functionality will have the OS kernel in run time. This file must be included in all user's and OS' source files. Since the user can specify another name for this file the special macro

OSPROPH is designed to substitute the name. The following code can be used in all user’s files (it is used in all OS source files):

```
#if !defined (OSPROPH)
#include  <osprop.h>
#else     /* !defined (OSPROPH) */
#include  OSPROPH
#endif    /* !defined (OSPROPH) */
```

The compiler command line (see **13.2.3 Compiling and Linking**) in this case should have the following option:

```
-dOSPROPH="<filename>".
```

`<filename>` is the name of the file with system properties definitions.

But the user is allowed to use some other method to include the property definition header file in his/her source code.

The files produced by SG are considered as application source files as well as files written by the user. The user can include produced files in his/her own source files. In case of user defined data types or variables which are used in system objects definitions such things have to be defined in user’s files before references to such entities. It may be done either via external declarations in header files or in source files. In the example below two variables and one data type are defined by the user which are referenced in files generated by SG. Variables are defined in the user’s file “user.c” and referenced in the produced file “task.c”. The data type is defined in the user’s file “user.h” and referenced in the produced file “msg.c”. The user’s code can 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 */
...
int varA = 22;         /* user defined variables */
int varB = 0;
...
```

```
#include "task.c"   /* references to variables are there */
#include "msg.c"    /* references to the data type are there */
```

Other variants are also possible. The main rule is:

RULE:

If the user specifies some user defined variables or data types in the application configuration file they are only referenced in the code produced by SG. So all definitions and external declarations have to be done by the user before these references.

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 OSEK 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 **SECTION 16 System Services**.

13.2.3 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. The current version of the OSEK Operating System uses the **Cosmic Software Version 4.0p for Motorola MC68HC08** as a development tool. The compiler options selected by the user should provide proper compiling. In the sample application the following command line for the Cosmic compiler is used:

```
cx6808 -l -m nowiden:nw -dOSPROPH=name.h -ao
```

If the user redefines the "osprop.h" name then the "**-d**" option is mandatory for the OSEK application, it defines macro *OSPROPH*. If the "osprop.h" name is used this option is not needed.

Since the System Generator allows definition of different objects in separate source files, during linking it is possible to allocate memory in the desired manner. Linking process is controlled by the typical linker directive file for the Cosmic linker.

13.3 Sample Application

In **APPENDIX A Sample Application** the code of an OSEK OS based application is provided. This code is simple demonstration of Operating System mechanisms. It also demonstrates how to write the configuration file and source code.

SECTION 14
PLATFORM-SPECIFIC FEATURES

14.1 HC08 features

14.1.1 Base Page Memory Usage

The system configuration option ***HCBasePage*** is of considerable importance for the OSEK Operating System for HC08 MCU. The average size of the OSEK OS kernel version with *HCBasePage* turned on is significantly smaller than the size of the version with extended addressing (*HCBasePage* is turned off). Exact characteristics are provided in **APPENDIX C Memory Requirements**. This feature can be important for applications with exacting requirements for memory and performance.

Also the task control blocks can be placed into the base page memory, if the system configuration option ***TaskBasePage*** is turned on. In this case the user is responsible for the needed amount of RAM in the base page for the desired number of task control blocks (the Cosmic linker option to control the segment size "**-m##**" can be used for this purpose). This system property increases the overall system performance.

14.1.2 Compiler Issues

The OSEK Operating System is designed to be used with the Cosmic Software Version 4.0p. In case of using other compiler versions some problems can arise. Contact with developers in this case to resolve them (see **2.5 Technical Support Information**). For instance, using of assembler optimization can lead to errors, since OSEK OS contains pieces of code written in assembler which cannot be optimized properly.

It is recommended to add environment variables and paths to access Cosmic compiler directories and OSEK directory. Installation procedure suggests the user to set these variables automatically. If they was not set during installation the user should do it manually. These variables are the following:

OSEKDIR = [path]\OSEK	- path to the OSEK directory
CXPATH = [path]\H6808	- path to the Cosmic header files directory
CXLIB = [path]\LIB	- path to the Cosmic library files directory

Also the path to the OSEK directory should be added into the PATH variable.

See the makefile in the \SAMPLE directory for additional information.

14.1.3 Interrupt Vector Table

The interrupt vector table is defined in the file “vector.c” which is delivered with the OSEK Operating System and located in the \SRC directory. This file is the example of the interrupt vector table coding (for two MCUs - MC68HC08AZ32 and MC68HC08XL36). The user should copy this file into the project directory and there modify it as needed. In addition, the file “vector.c” from the \SAMPLE directory may be used as a sample of the vector table.

14.1.4 Recommendations on System Properties

14.1.4.1 *UseMainStack* property

It is recommended to turn ON this property. It reduces the needed amount of RAM for stacks, since the same memory area is used during start-up (for `main()` program), for the interrupt stack and for the scheduler stack. Moreover, the size of the OS code is also reduced (ROM for the Operating System).

However, the user is responsible for allocation appropriate stack in the linker directive file.

14.1.4.2 *UseSameContext* property

It is recommended to turn ON this property to use only preemptive context for all tasks. It is important for CPU08 so the code to process the non-preemptive context is quite big, and it leads to additional ROM and RAM consuming and reduces system performance. When this property is ON, task switching is performed faster and the required ROM amount is reduced. Use of different contexts (preemptive and non-preemptive) have a sense for CPUs with many registers if some of them are not included into the non-preemptive context.

14.1.4.3 *InterruptMaskControl* property

It is better to keep this property turned OFF, if it is not absolutely necessary to control “I” bit of the Condition Code Register. Using of this property increases the amount of ROM for the OS code and requires additional time for all system services. Generally, this property is intended for CPUs with several interrupt priority levels, e.g. CPU16.

14.1.4.4 *CounterSize* property

CPU08 is a 8-bit one, and its best performance is achieved when operands are byte-sized. Therefore, it is recommended to use 8-bit counters (*CounterSize* = 8) to increase overall performance and reduce the needed amount of RAM. Of course, it is possible to use 16-bit and 32-bit counters, but it leads to additional RAM and time consuming, especially in case of 32-bit counters !

Moreover, the user is responsible to preserve the value of the compiler variable ”c_lreg” within the ISR, if a counter is triggered from the ISR. ???

14.1.4.5 Unused services

To reduce ROM consuming it is recommended to exclude OS services which are not used in the application from the OS code with the help of “excluding” system properties (see **Table 12–2 OSEK Operating System Properties**). It helps to save memory.

14.1.5 System Timer Hardware

In the system definition statement *DefineSystemTimer* it is required to define hardware interrupt source for this counter and its desired parameters. In the table below all possible values to define these parameters are listed. This table comprises HC08ABxx, HC08AZxx families and HC08XL36.

Table 14–1 Parameters to define System Timer hardware

HardwareType	HardwarePrescaler	HardwareModulo	Vector Address
HC08AZ0, HC08AB16, HC08AZ24, HC08AZ32			
TIMATOI	0...7	0...65535	$FFEC
TIMBTOI	0...7	0...65535	$FFE6
TIMAOC0	0...7	0...65535	$FFF4
TIMAOC1	0...7	0...65535	$FFF2
TIMAOC2	0...7	0...65535	$FFF0
TIMAOC3	0...7	0...65535	$FFEE
TIMBOC0	0...7	0...65535	$FFEA
TIMBOC1	0...7	0...65535	$FFE8
PIT	0...7	0...65535	$FFF6
HC08XL36			
TIMTOI	0...7	0...65535	$FFEC
TIMOC0	0...7	0...65535	$FFF4
TIMOC1	0...7	0...65535	$FFF2
TIMOC2	0...7	0...65535	$FFF0
TIMOC3	0...7	0...65535	$FFEE

Thus, the system definition statement *DefineSystemTimer* for the OSEK Operating System for HC08 MCU family has the following form:

```
DefineSystemTimer( <CounterID>, <maxallowedvalue>,
                   <ticksperbase>, <tickduration>, [<mincycle>,]
```

```
                    <HardwareType>, <HardwarePrescaler>,
                    <HardwareModulo> );
```

For example, the system timer is based on the Timer A Overflow interrupt, with Prescaler value equals 2 and Modulo Counter value equals 256. In this case the definition statement can be the following:

```
DefineSystemTimer( SYS_TIMER, 100, 40, 256000, , TIMATOI, 2, 256);
```

NOTE:

The system timer will not be triggered if the *EntryExitISR* property is turned OFF!

14.1.6 Scheduler Architecture

In an application does not use resources, it is strongly recommended to use simplified scheduler (the property *SimpleScheduler* must be turned ON). In this case all tasks running concurrently must have different priorities.That is, two tasks may have the same priority if they are never both in *ready* state and they maust have different priorities otherwise. The simple scheduler uses prioritized table of task nodes and works faster. It requires less amount of ROM and RAM than the standard scheduler.

In an application uses resources or claims to have more than one ready task per priority, the *SimpleScheduler* option should be turned off. In this case the POSIX-like scheduler is used. To reduce RAM consuming the number of priorities should be kept as small as possible.

SECTION 15
APPLICATION TROUBLESHOOTING

In this section some advises are given which may be useful for developers working with the OSEK Operating System.

15.1 System Generation

The System Generator is used to generate the code for the OSEK Operating System kernel and all application objects (tasks, messages, 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 **SECTION 12 System Configuration** and **SECTION 13 Building of Application** about system generation process.

If an undocumented problem arises please provide us with the detailed description of it and we will help to resolve the problem. See **2.5 Technical Support Information** for contact information.

15.2 Using OS Extended Status for Debugging

It is strongly recommended to use Operating System Extended Status when you develop an application to analyze return codes of system services. Such method is more memory and time consuming but it allows the user to save time for errors eliminating. Error codes returned by the OSEK 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_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!

When all errors in an application will be eliminated you may turn off the Extended Status and remove additional status checks from the application to get the reliable application of the smaller size.

15.3 Context Switch Routines

Breakpoints, traces and time stamps can be integrated individually into the application software with the help of context switch hook routines *OSPreTask* and *OSPostTask*.

Example: The user can set time stamps enabling him to trace the program execution 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.

The Operating System needs not include a time monitoring feature which ensures that each or only, e.g. the lowest-priority task has been activated in any case after a defined maximum time period.

The user can optionally use hook routines or establish a watchdog task that takes “one-shot displays” of the operating system status.

15.4 Stack Errors

Stack errors may be due to the stack pointer being incorrect or to stack contents being incorrect. Stack content problems are possible when using pointers into stack variables, but stack pointer problems seem to be more common. The symptom of either problem is usually a task or ISR executing normally, but then when a return is performed, the program executes at some incorrect address.

NOTE

Problems with a program running wild may sometimes be caught by always filling the unused program memory with $00, which the HC08 ‘`test`’ opcode does, and setting a breakpoint at the illegal opcode vector address.

ISRs using OS services or reenabling interrupts should begin with the *EnterISR* service and end via the *LeaveISR* service (see **SECTION 6 Interrupt Processing**). If the number of *EnterISR* and *LeaveISR* invocations do not match (e.g., in case of nested interrupts), the stack will be incorrect. See also **6.5 Local Variables Considerations** about using local variables in ISRs.

Tasks should have enough stack for their execution, therefore it is recommended to pay attention on task definition statements to provide each task with a needed amount of stack.

Generally, the recommended minimal task stack size equals:

- 24 bytes for extended tasks if no task activation, messages, and there are no interrupts in the system;
- 32 bytes for extended tasks if no task activation, messages and interrupts are supported in the system;
- 48 bytes for other cases.

15.5 Known Problems

None.

SECTION 16
SYSTEM SERVICES

16.1 General

This section provides detailed description of all OSEK 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:	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 lowest Conformance Class where the declaration element is provided.

Service description:

Syntax:	Interface in ANSI-C syntax. The return value of the service is always of data type StatusType.
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.
Standard:	List of return values provided in the operating system's standard version. Special case - service does not return.
Extended:	List of additional return values in the operating system's extended version.

Conformance: Specifies the lowest Conformance Class where the service is provided.

The specification of operating system services uses the following naming conventions for data types:

...Type:	describes the values of individual data.
...RefType:	describes the identifier referencing an object[1].

It is also possible to see all predefined OSEK OS data types in system header files.

[1] E.g., a pointer or an index

16.2 Task Management Services

16.2.1 Data types

The OSEK Operating System establishes the following data types for the task management:

• ***TaskRefType***	The abstract data type for task identification;
• ***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.

16.2.2 Constants

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

• ***RUNNING***	Constant of data type *TaskStateRefType* for task state *running*
• ***WAITING***	Constant of data type *TaskStateRefType* for task state *waiting*
• ***READY***	Constant of data type *TaskStateRefType* for task state *ready*
• ***SUSPENDED***	Constant of data type *TaskStateRefType* for task state *suspended*

16.2.3 Conventions

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

```
TASK task_start( void )
{
...
}
```

The keyword ***TASK*** is evaluated by the system generation to clearly distinguish between functions and tasks in the source code.

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

16.2.4 Task Declaration

To refer to a task the constructional element should be used to declare the task before references to it:

Syntax:

```
DeclareTask( TaskRefType <TaskName> )
```

Input:

- *<TaskName>* - a reference to the task

Description:

DeclareTask serves as an external declaration of a task. The function and use of this service are similar to that of the external declaration of variables.

Particularities: none

Conformance: BCC1

16.2.5 ActivateTask

Syntax:

```
StatusType ActivateTask( TaskRefType <TaskName> );
```

Input:

- *<TaskName>* - a reference to the task.

Output:

- None.

Description:

The specified task *TaskName* is transferred from the *suspended* state into the *ready* state. All needed actions for task initialization are accomplished by the system according to information from the task configuration table (e.g., dynamic stack and node allocation).

Particularities:

The service may be called both on the task level (from a task) and the interrupt level (from ISR).

In the case “call from ISR”, the operating system will reschedule tasks only after the ISR completion.

If the system configuration option *ActivareTask* is turned off the service is excluded from the OSEK OS code.

Status:

- Standard:

— *E_OK*	- no error;
— *E_STATE*	- the task has already been activated and multiple requests are not allowed.

- Extended:

— *E_ID*	- the task identifier *TaskName* is invalid;
— *E_LIMIT*	- too many task activations of the specified task or there is no enough resources to activate the task.

Conformance: BCC1

16.2.6 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 and releases all occupied system resources (that is, the stack, the task node, etc.).

Particularities:

The resources occupied by the task must have been released before.

If the call was successful, *TerminateTask* does not return to the call level and the status can not be evaluated. If the service *TerminateTask* is called successfully, it enforces a rescheduling.

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.

If the system configuration option *TerminateTask* is turned off the service is excluded from the OSEK OS code.

Status:

- Standard:
  - — No return to the caller.
- Extended:

— *E_RESOURCE*	- the task still occupies resources;
— *E_CALLEVEL*	- a call at the interrupt level.

Conformance: BCC1

16.2.7 ChainTask

Syntax:

```
StatusType ChainTask( TaskRefType <TaskName> );
```

Input:

• *<TaskName>*	- 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 *<TaskName>* is activated sequentially. Using this service, it ensures that the succeeding task only starts to run after the calling task has been terminated.

Particularities:

If the succeeding task is identical with the current task, this does not result in multiple requests. In this case the task will be terminated and after that activated again. It can be used, e.g. if multiple activation is disabled but it is needed to restart a task. If it is not needed to have such tricks the system property *ChainTaskItself* can be turned off to decrease the task control block size.

The resources occupied by the calling task must have been released before.

If called successfully, *ChainTask* does not return to the call level and the status can not be evaluated. In this case a rescheduling is enforced.

If the version with Extended Status is used, the service returns in case of error to the calling task, and provides a status which can be evaluated by the application.

If the system configuration option *ChainTask* is turned off the service is excluded from the OSEK OS code.

Status:

- Standard:
  - — No return to the caller.
- Extended:

— *E_ID*	- the task identifier is invalid;
— *E_STATE*	- the succeeding task has already been activated and multiple requests are not allowed;

— *E_RESOURCE*	- the calling task still occupies resources;
— *E_CALLEVEL*	- a call at the interrupt level;
— *E_LIMIT*	- too many activations of <*TaskName*> or there is no enough resources to activate the task.

Conformance: BCC1

16.2.8 Schedule

Syntax:

```
StatusType Schedule( void );
```

Input:

- None.

Output:

- None.

Description:

If a higher-priority task is *ready*, the current task is put into the *ready* state and a higher-priority task is executed. Otherwise the calling task is continued without delay. By means of using this service the task explicitly yields control to a higher-priority ready task (if any exists).

Particularities:

In not full-preemptive systems *Schedule* enables a processor assignment to other tasks in application-specific locations.

If the system configuration option *Schedule* is turned off the service is excluded from the OSEK OS code.

Status:

- Standard:
  — *E_OK* - no error.
- Extended:
  — *E_CALLEVEL* - a call at the interrupt level.

Conformance: BCC1

16.2.9 GetTaskId

Syntax:

```
StatusType GetTaskId( TaskRefType <TaskName> );
```

Input:

- None.

Output:

• *<TaskName>*	- a reference to the task which is currently active. The system saves the task reference into the variable <*TaskName*>.

Description:

This system service returns the name (<*TaskName*>) of the task which is currently active.

Particularities:

This service is useful, for instance, in the case if two or more tasks shares the same piece of code and in some point of the code the coming actions depend on which task is executed in the moment.

If the system configuration option *GetTaskId* is turned off the service is excluded from the OSEK OS code.

Status:

- Standard:

— *E_OK*	- no error.

- Extended:

— *E_CALLEVEL*	- a call at the interrupt level.

Conformance: BCC1

16.2.10 GetTaskState

Syntax:

```
StatusType GetTaskState( TaskRefType <TaskName>,
                         TaskStateRefType <State> );
```

Input:

• *<TaskName>*	- a reference to the task.

Output:

• *<State>*	- a reference to the state of task *<TaskName>*.

Description:

Returns the state of the specified task *<TaskName>* (*running*, *ready*, *waiting*, *suspended*) at the time of calling *GetTaskState*.

Particularities:

The service may be called both on the task level (from a task) and the interrupt level (from ISR).

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.

If the system configuration option *GetTaskState* is turned off the service is excluded from the OSEK OS code.

Status:

- Standard:

— *E_OK*	- no error.

- Extended:

— *E_ID*	- the task identifier is invalid.

Conformance: BCC1

16.2.11 Examples for Task Management Services

The example below assumes three tasks `TaskA`, `TaskB` and `TaskC`. These tasks use all OSEK OS task management services to coordinate each other.

The following definitions can be made in the definition file:

```
...
[Tasks]
DefineTask( TaskA, BASIC|ACTIVATE|OWNSTACK,1, task_a,,,,64 );
DefineTask( TaskB, EXTENDED|POOLSTACK,2, task_b,,,,POOL1 );
DefineTask( TaskC, EXTENDED|ACTIVATE|NODESTACK,3, task_c,,,,);
...
```

The C-language file:

```
DeclareTask( TaskA )
DeclareTask( TaskB )
DeclareTask( TaskC )

TASK   task_a( void )
{
TaskRefType  task;
...       /* any user’s code */

ActivateTask( TaskB );   /* activate TaskB */
Schedule();    /* yields CPU to a higher-priority task */
GetTaskId( task );
if( task == TaskA ) ActivateTask( TaskC );
else ChainTask( TaskB );
...       /* any user’s code */
TerminateTask();
}


TASK   task_b( void )
{
TaskStateType state;
EventMaskType cc = 0x4;
...       /* any user’s code */

GetTaskState( TaskC, &state );  /* check the state of TaskC */
switch( state )              /* and perform appropriate actions */
    {
    case READY:    break;
    case WAITING:  SetEvent( TaskC, cc );
                    break;
    case SUSPENDED: ChainTask( TaskC );
                    break;
    }
...       /* any user’s code */
}


TASK   task_c( void )
{
TaskStateType stateA, stateB;
...       /* any user’s code */

while( 1 )
{
   GetTaskState( TaskA, &stateA );
   GetTaskState( TaskB, &stateB );
   if( stateA == READY && stateB == SUSPENDED ) ChainTask( TaskB );
   if( stateB == READY && stateA == SUSPENDED ) ChainTask( TaskA );
  if( stateA == READY && stateB == READY ) Schedule();
   ...  /* any user’s code */
}
}
```

16.3 ISR Management Services

16.3.1 Data Types

The OSEK Operating System establishes the following data types for the task management:

• ***IntMaskType***	the data type for interrupt masks
• ***IntMaskRefType***	the data type for reference to interrupt mask.

16.3.2 Conventions

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

```
ISR IsrStart( void )
{
...
}
```

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

16.3.3 EnterISR

Syntax:

```
StatusType EnterISR( void );
```

Input:

- None.

Output:

- None.

Description:

EnterISR establishes conditions needed to request OS services within an interrupt service routine. Inside *EnterISR* the following activities are executed if needed:

- Registration of the switching to the interrupt level inside the operating system;
- A switch of the current context (to the ISR stack).

This function is called from the begin of the interrupt service routine, when hardware registers are pushed onto the current stack either by hardware or by compiler generated code. This way, the context of the running task, or ISR, or scheduler idle loop is known on the function entry. This context is named interrupt stack frame.

If the task is interrupted, then the pointer to the interrupt stack frame is stored in the stack pointer field of the running task. After that the top of the ISR stack is loaded into the CPU stack pointer, and value of the nested interrupts counter is incremented.

If the ISR is interrupted, then the value of the system counter of nested interrupts is advanced by one, and function returns to the caller.

If the scheduler idle loop is interrupted, then interrupt stack frame is ignored, because *LeaveISR* function returns directly to the scheduler idle loop. The top of the ISR stack is loaded into the CPU stack pointer, and value of the system counter of nested interrupts is changed to one.

Particularities:

EnterISR establishes the possibility to use operating system services in an ISR. It is necessary to place *EnterISR* before the first call of an operating system service.

See description of interrupt categories in section **6.3 ISR Categories**.

This service is not implemented if the system configuration option *EntryExitISR* or *EnterISR* are turned off in the configuration file.

Status:

- Standard:
  - — None.
- Extended:
  - — None.

Conformance: BCC1

16.3.4 LeaveISR

Syntax:

```
StatusType LeaveISR( void );
```

Input:

- None.

Output:

- None.

Description:

LeaveISR is the counterpart of *EnterISR* and resets the conditions to request operating system services in an ISR. Inside *LeaveISR* the following functions are executed if needed:

- Registration of the switching back from the interrupt level inside the operating system;
- A switch of the current context (for example a switch from the ISR stack back to the OS context or to the context of the task running before).

In case of an error the function returns to the call level.

This function is called in the end of the interrupt service routine.

If the ISR was interrupted by the corresponding *EnterISR*, then the value of the system counter of nested interrupts is decremented by one, and the function returns to the caller, because the current ISR is considered as nested one.

If the caller is the outermost ISR (task or scheduler idle loop was interrupted), then the function calls the scheduler to update the pointer to the running task. After this call the function loads the task stack pointer into the CPU stack pointer in assumption, that task stack pointer contains the address of valid interrupt stack frame. If there is no running task, then functions goes directly to the scheduler idle loop. In any case the value of the nested interrupts counter is decremented by one.

Particularities:

LeaveISR eliminates the possibility of using OS services in an ISR. It is highly recommended to place *LeaveISR* at the end of the ISR.

See description of interrupt categories in section **6.3 ISR Categories**.

This service is not implemented if the system configuration option *EntryExitISR* or *LeaveISR* are turned off in the configuration file.

Status:

- Standard:
  - — No return to call level.
- Extended:
  - — *E_CALLEVEL* - a call not at the interrupt level.

Conformance: BCC1

16.3.5 EnableInterrupt

Syntax:

```
StatusType EnableInterrupt( IntMaskType <Mask> );
```

Input:

• *<Mask>*	- a mask of interrupts to be enabled.

Output:

• None.

Description:

This service enables interrupts specified by *<Mask>*. In the case if several interrupts can be controlled simultaneously, this service allows enabling of several interrupts.

Particularities:

The service may be called from an ISR and from the task level.

For HC08 this service is intended to control only the “I” bit in the Condition Code Register.

This service should only be used with care. It destroys the contents of CCR according to C language conventions. In case of use it is highly recommended to know the reaction upon system behavior!

To save the current state of interrupts the application must use *GetInterruptMask* before.

This service is executed also in case of return of the state *E_NOFUNC*.

This service is not implemented if the system configuration option *EntryExitISR* or *EnableInterrupt* are turned off in the configuration file.

Status:

• Standard:

— *E_OK*	- no error.

• Extended:

— *E_NOFUNC*	- at least one of the interrupts was not disabled.

Conformance: BCC1

16.3.6 DisableInterrupt

Syntax:

```
StatusType DisableInterrupt( IntMaskType <Mask> );
```

Input:

• *<Mask>*	- a mask of interrupts to be disabled.

Output:

• None.

Description:

This service disables interrupts specified by *<Mask>*. In the case if several interrupts can be controlled simultaneously, this service allows disabling of several interrupts.

Particularities:

The service may be called from an ISR and from the task level.

For HC08 this service is intended to control only the “I” bit in the Condition Code Register.

This service should only be used with care. It destroys the contents of CCR according to C language conventions. In case of use it is highly recommended to know the reaction upon system behavior!

To save the current state of interrupts the application must use *GetInterruptMask* before.

This service is executed also in case of return of the state *E_NOFUNC*.

This service is not implemented if the system configuration option *EntryExitISR* or *DisableInterrupt* are turned off in the configuration file.

Status:

• Standard:

— *E_OK*	- no error.

• Extended:

— *E_NOFUNC*	- at least one of the interrupts was not enabled.

Conformance: BCC1

16.3.7 GetInterruptMask

Syntax:

```
StatusType GetInterruptMask( IntMaskRefType <Mask> );
```

Input:

- None.

Output:

- *<Mask>* - a reference to the interrupt mask to be filled.

Description:

Query of interrupt status and returns the current CPU interrupt mask.

Particularities:

The service may be called from an ISR and from the task level.

For HC08 this service is intended to get only the value of the "I" bit in the Condition Code Register.

This service is not implemented if the system configuration option *EntryExitISR* or *GetInterruptMask* are turned off in the configuration file.

Status:

- Standard:
  — *E_OK* - no error.
- Extended:
  — None.

Conformance: BCC1

16.3.8 Examples for Interrupt Management Services

Below examples for ISR category 1, 2 and 3 are presented.

The following definitions can be made in the definition file:

```
[Interrupt management]
DefineInterrupts( 0x8,0x0,0x0, 64);
...
[Tasks]
DefineTask( TaskB, EXTENDED|POOLSTACK,1, task_b,,,,POOL1 );
DefineTask( IndTask, BASIC|ACTIVATE|OWNSTACK,2, ind_tsk,,,,64 );
...
[Counters]
DefineCounter(Ctr1, 24, 1);
...
[State Messages]
DefineStateMessage( Temp, char, sizeof(char), WithoutTimeStamp);
[Event Messages]
DefineEventMessage( Wrn, MSGCTYPE, sizeof(int),
                    6,3,WithOverwriteCheck, WithoutTimeStamp);
```

The C-language code can be the following:

ISR category 1:

```
char CREG, DREG;
char data1;
...

ISR ISR_handler
{
if( CREG != 0xC0 ) CREG |= 0x40;
else CREG |= 0x03;
DREG = data1;
}
```

ISR category 2:

```
TaskStateType stateB;
DeclareCounter( Ctr1 )
DeclareTask( TaskB )
...
ISR ISR_handler
{
EnterISR();
CounterTrigger( Ctr1 );
GetTaskState( TaskB, &stateB );
if( stateB == SUSPENDED ) ActivateTask( TaskB );
LeaveISR();
}
```

ISR category 3:

```
DeclareTask( IndTask );
UsesStateMessage( Temp, Send, WithCopy)
UsesEventMessage( Wrn, Send );
int temp;

ISR ISR_handler
{
                    /* normal temperature, do nothing */
if( (temp >= LIMIT_L ) && (temp <= LIMIT_H) ) goto exit;
                   /* temperature is below critical value: */
if( LIMIT_L >= temp )
    { EnterISR();
      SendStateMessage( Temp, WithCopy ); /* send msg to notify */
      goto leave;
    }
                   /* temperature is higher critical value: */
if( (temp >= LIMIT_H) )
    { EnterISR();
      SendEventMessage( Wrn ); /* send alarm message */
      ActivateTask( IndTask );
leave: LeaveISR();
    }
exit: ;
}
```

16.4 Resource Management Services

16.4.1 Data types

The OSEK Operating System establishes the following data type for the resource management:

- ***ResourceRefType*** - the abstract data type for referencing a resource;

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

16.4.2 Resource Declaration

To refer to a resource the constructional element should be used to declare the resource before its using:

Syntax:

```
DeclareResource( ResourceRefType <ResName> )
```

Input:

- *<ResName>*   - a reference to the resource.

Description:

DeclareResource serves as an external declaration of a resource. The function and use of this service are similar to that of the external declaration of variables.

Particularities: none

Conformance: BCC1

16.4.3 GetResource

Syntax:

```
StatusType GetResource( ResourceRefType <ResName> );
```

Input:

- *<ResName>* - a reference to the resource.

Output:

- None.

Description:

This call serves to enter a critical section in the code that is assigned to the referenced resource. A critical section must always be left using *ReleaseResource*.

The *E_LIMIT* error code is not processed yet.

Particularities:

This function is fully supported only beginning from the Conformance Class BCC3. In Conformance Classes BCC1 and BCC2 only the standard resource scheduler can be occupied via the constant *RES_SCHEDULER*.

Corresponding calls to *GetResource* and *ReleaseResource* should appear within the same function on the same function level. Generally, critical sections should be short. Nested resource occupation is only allowed if the inner critical sections are executed completely within the surrounding critical section.

Regarding Extended Tasks, please note that *WaitEvent* within a critical section is prohibited.

This service is not implemented if the system configuration option *Resources* or *GetResource* are turned off in the configuration file.

Status:

- Standard:

— *E_OK*	- no error.

- Extended:

— *E_ID*	- the resource identifier is invalid;
— *E_ACCESS*	- the inadmissible access to resource;
— *E_CALLEVEL*	- a call at the interrupt level is not allowed;
— *E_LIMIT*	- too many resources are occupied in parallel.

Conformance: BCC1

16.4.4 ReleaseResource

Syntax:

```
StatusType ReleaseResource( ResourceRefType <ResName> );
```

Input:

- *<ResName>* - a reference to the resource.

Output:

- None.

Description:

This call serves to leave the critical section in the code that is assigned to the referenced resource. An *ReleaseResource* call is a counterpart of an *GetResource* service call.

The *E_ID* error code is generated in the following cases:

- the resource signature is invalid - this object is not resource;
- the resource is occupied by another task, the resource is not released in this case;
- the resource is not the last occupied resource (a nesting processing error), the resource is released (unlinked from the list) in this case.

Particularities:

This function is fully supported only beginning from the Conformance Class BCC3. In Conformance Classes BCC1 and BCC2 only the standard resource scheduler can be released via the constant *RES_SCHEDULER*.

For information on nesting conditions, see particularities of *GetResource*.
This service is not implemented if the system configuration option *Resources* or *ReleaseResource* are turned off in the configuration file.

Status:

- Standard:

— *E_OK*	- no error.

- Extended:

— *E_ID*	- the resource identifier is invalid;
— *E_NOFUNC*	- an attempt to release a resource which is not occupied;
— *E_CALLEVEL*	- a call at the interrupt level is not allowed.

Conformance: BCC1

16.4.5 Examples of using resources

The example below presents resource management directives.

The following definitions can be made in the definition file:

```
[Tasks]
DefineTask( TASK_A, EXTENDED|POOLSTACK,1, task_a,,,,POOL1 );
DefineTask( TASK_B, BASIC|ACTIVATE|OWNSTACK,2, task_b,,,,64 );
DefineTask( SCI_TASK, BASIC|ACTIVATE|NODESTACK,3, taskSCI,,,, );
[Resources]
DefineResource(SCI_res, 2);
...
```

The C-language code can be the following:

```
DeclareTask( SCI_TASK )
DeclareResource( SCI_res )

TASK taskSCI
{
...
GetResource( SCI_res );     /* occupy the SCI resource */
...                         /* user’s code */
ActivateTask( TASK_B );
GetResource( RES_SCHEDULER ); /* occupy the scheduler resource */
...                           /* user’s code */
ReleaseResource( RES_SCHEDULER );   /* release the scheduler */
ReleaseResource( SCI_res );        /* release the SCI resource */

TerminateTask();
}
```

16.5 Counters and Alarms Management Services

16.5.1 Data Types and Identifiers

The following data types are established by OSEK OS to work with counters and alarms:

- ***CtrRefType*** - the data type references a counter
- ***TickType*** - the data type represent count value in system ticks
- ***TickRefType*** - the data type references data corresponding to the data type *TickType*
- ***CtrInfoType*** - the data type represents a structure for storage of counter characteristics. This structure has the following elements:
    - ***maxallowedvalue*** - maximum possible allowed counter value;
    - ***tickperbase*** - number of ticks required to reach a counter-specific significant unit;
    - ***mincycle*** - minimum allowed number of ticks for a cyclic alarm (only for system with Extended Status).

  All elements have the data type *TickType*, and the structure looks like the following:

```
typedef CtrInfoType tagCIT;
struct tagCIT
{
    TickType maxallowedvalue;
    TickType tickperbase;
    TickType mincycle;
};
```

- ***CtrInfoRefType***- the data type references data corresponding to the data type *CtrInfoType*
- ***AlarmRefType***- the data type represents the reference to an alarm element.

16.5.2 Constants

For system counter, which is always a time counter, the special constants are provided by the operating system:

- ***OSMAXALLOWEDVALUE*** - maximum possible allowed value of the system timer in ticks;
- ***OSTICKSPERTIME*** - number of ticks that are required to reach 10 milliseconds in the system counter;
- ***OSTICKDURATION*** - duration of a tick of the system counter in nanoseconds;

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

16.5.3 Counter and Alarm Declaration

To refer to a counter or alarm the declaration statements should be used to declare the element (counter or alarm) before their using:

16.5.3.1 Counter Declaration

Syntax:

```
DeclareCounter( CtrRefType <CounterName> )
```

Input:

- *<CounterName>*- a reference to the counter

Description:

DeclareCounter serves as an external declaration of a counter. The function and use of this service are similar to that of the external declaration of variables.

Particularities: none

Conformance: BCC1

16.5.3.2 Alarm Declaration

Syntax:

```
DeclareAlarm( AlarmRefType <AlarmName> )
```

Input:

- *<AlarmName>* - a reference to the alarm

Description:

DeclareAlarm serves as an external declaration of an alarm. The function and use of this service are similar to that of the external declaration of variables.

Particularities: none

Conformance: BCC1

These declarations are equivalent to the external declaration of variables.

16.5.4 InitCounter

Syntax:

```
StatusType InitCounter( CtrRefType <CounterName>,
                        TickType <Ticks> );
```

Input:

• *<CounterName>*	- 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 *CounterTrigger*. If there are running attached alarms, then these alarms are checked whether they have expired at this tick value and the appropriate actions are performed, else their state stays unchanged.

Particularities:

This service is not implemented if the system configuration option *Counters* or *InitCounter* are turned off in the configuration file.

Status:

• Standard:	
— *E_OK*	- no error.
• Extended:	
— *E_ID*	- the counter identifier is invalid;
— *E_VALUE*	- the counter initialization value exceeds the maximum admissible value;
— *E_CALLEVEL*	- a call at interrupt level (not allowed).

Conformance: BCC1

16.5.5 CounterTrigger

Syntax:

```
StatusType CounterTrigger( CtrRefType <CounterName> );
```

Input:

- *<CounterName>*- a reference to the counter;

Output:

- None.

Description:

Increments the current value of the counter. If the counter reaches the value ***maxallowedvalue*** (see **16.5.1**), 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 event setting).

Particularities:

Call admissible both from interrupt and task levels.

Depending on the underlying hardware it is possible that parts of the functionality of *CounterTrigger* may by done by the hardware. In this case the remaining functionality of *CounterTrigger* has to be adapted.

This service is not implemented if the system configuration option *Counters* or *CounterTrigger* are turned off in the configuration file.

Status:

- Standard:
  - — *E_OK* - no error.
- Extended:
  - — *E_ID* - the counter identifier is invalid.

Conformance: BCC1

16.5.6 GetCounterValue

Syntax:

```
StatusType GetCounterValue( CtrRefType <CounterName>,
                            TickRefType <Ticks> );
```

Input:

- *<CounterName>*- a reference to the counter;

Output:

- *<Ticks>* - a counter value in ticks.

A reference to the return counter value in ticks.

Description:

The system service provides the current value of the counter *<CounterName>* in ticks.

Particularities:

This service is not implemented if the system configuration option *Counters* or *GetCounterValue* are turned off in the configuration file.

Status:

- Standard:
  - *E_OK* - no error.
- Extended:
  - *E_ID* - the counter identifier is invalid;
  - *E_CALLEVEL* - a call at interrupt level (not allowed).

Conformance: BCC1

16.5.7 GetCounterInfo

Syntax:

```
StatusType GetCounterInfo( CtrRefType <CounterName>,
                            CtrInfoRefType <Info> );
```

Input:

- *<CounterName>*- a reference to the counter;

Output:

- *<Info>* - a reference to the structure with constants of the counter.

Description:

Returns the counter characteristics into the *<Info>* structure. For a system counter special constants may be used instead of this service. The return value *<Info>* is a structure in which the information of data type *CtrInfoType* is stored.

Particularities:

The structure 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 the system configuration option *Counters* or *GetCounterInfo* are turned off in the configuration file.

Status:

- Standard:

— *E_OK*	- no error.

- Extended:

— *E_ID*	- the counter identifier is invalid;
— *E_CALLEVEL*	- a call at interrupt level (not allowed).

Conformance: BCC1

16.5.8 SetRelAlarm

Syntax:

```
StatusType SetRelAlarm( AlarmRefType <AlarmName>,
                        TickType <Increment>,
                        TickType <Cycle> );
```

Input:

• *<AlarmName>*	- a reference to the alarm;
• *<Increment>*	- a relative 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 *<AlarmName>* element. After this service call the counter will count *<Increment>* ticks starting from the current counter value at the moment of the call. After *<Increment>* ticks have elapsed from that moment, the task assigned to the alarm *<AlarmName>* is activated or the assigned event (only for Extended Tasks) is set.

If *<Cycle>* is unequal 0, the alarm element is logged on again immediately after expiry with the relative value *<Cycle>*. Otherwise, the alarm triggers only once.

Particularities:

If the relative value *<Increment>* is very small, the alarm may immediately expire, and the task may become *ready*. It is because the certain time is needed for system activities to return to the calling task after the *<Increment>* ticks for the counter have been set.

The alarm *<AlarmName>* must not already be in use. To change values of alarms already in use the alarm has to be cancelled first.

This service is not implemented if the system configuration option *Alarms* or *SetRelAlarm* are turned off in the configuration file.

Status:

• Standard:

— *E_OK*	- no error;
— *E_STATE*	- the alarm is already in use.

- Extended:

— *E_ID*	- the alarm identifier is invalid;
— *E_VALUE*	- the alarm initialization value or cycle value is greater than the maximum allowed value of the counter, or the cycle value is less than the minimum cycle value of the counter;
— *E_CALLEVEL*	- a call at the interrupt level is not allowed.

Conformance:

- BCC1;
- Event:ECC1.

16.5.9 SetAbsAlarm

Syntax:

```
StatusType SetAbsAlarm( AlarmRefType <AlarmName>,
                        TickType <Start>,
                        TickType <Cycle> );
```

Input:

- *<AlarmName>* - 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 *<AlarmName>* element. The counter will count *<Start>* ticks starting from zero counter value. When *<Start>* ticks are reached, the task assigned to the alarm *<AlarmName>* is activated or the assigned event (only for Extended Tasks) is set.

If *<Cycle>* is unequal 0, the alarm element is logged on again immediately after expiry with the relative value *<Cycle>*. Otherwise, the alarm triggers only once.

Particularities:

Since the current counter value at the moment of the service call is most likely not equal zero, there exist the some period of time while the counter reaches its maximum allowed value and will be logged on again (its value will become 0). Only starting from zero *<Start>* ticks will be counted.

If the absolute value *<Start>* is very close to the current counter value, the alarm may immediately expire, and the task may become *ready*. (E.g., the current counter value at the moment of service call equals 253, the maximum allowed counter value equals 255 and the *<Start>* value is 2.)

The alarm *<AlarmName>* must not already be in use.

To change values of alarms already in use the alarm has to be cancelled first.

This service is not implemented if the system configuration option *Alarms* or *SetAbsAlarm* are turned off in the configuration file.

Status:

- Standard:

— *E_OK*	- no error;
— *E_STATE*	- the alarm is already in use.

- Extended:

— *E_ID*	- the alarm identifier is invalid;
— *E_VALUE*	- the alarm initialization value or cycle value is greater than the maximum allowed value of the counter, or the cycle value is less than the minimum cycle value of the counter;
— *E_CALLEVEL*	- a call at the interrupt level is not allowed.

Conformance:

- BCC1;
- Event:ECC1.

16.5.10 CancelAlarm

Syntax:

```
StatusType CancelAlarm( AlarmRefType <AlarmName> );
```

Input:

- *<AlarmName>* - a reference to the Alarm.

Output:

None.

Description:

The service cancels the alarm (transfers it into the stop state).

Particularities:

This service is not implemented if the system configuration option *Alarms* or *CancelAlarm* are turned off in the configuration file.

Status:

• Standard:

— *E_OK*	- no error;
— *E_NOFUNC*	- the alarm is not in use.

• Extended:

— *E_ID*	- the alarm identifier is invalid;
— *E_CALLEVEL*	- a call at interrupt level (not allowed).

Conformance: BCC1

16.5.11 GetAlarm

Syntax:

```
StatusType GetAlarm( AlarmRefType <AlarmName>,
                     TickRefType <Ticks> );
```

Input:

- *<AlarmName>* - a reference to the Alarm;

Output

- *<Ticks>* - a relative value in ticks before the alarm expires.

Description:

This service calculates the time in ticks before the alarm expires. If the alarm is not started the *E_NOFUNC* error code is generated.

Particularities:

It is up to the application to decide whether for example an alarm may still be useful or not.

If *<AlarmName>* is not in use <*Ticks*> is 0.

This service is not implemented if the system configuration option *Alarms* or *GetAlarm* are turned off in the configuration file.

Status:

- Standard:
  - — *E_OK* - no error;
  - — *E_NOFUNC* - the alarm is not in use.
- Extended:
  - — *E_ID* - the alarm identifier is invalid;
  - — *E_CALLEVEL* - a call at interrupt level (not allowed).

Conformance: BCC1

16.5.12 Examples for Counter and Alarm Management

The example shows how counters and alarms can be used.

The following definitions are made in the definition file:

```
[Tasks]
DefineTask( TaskTime, EXTENDED|POOLSTACK,1, task_time,,,,POOL1 );
DefineTask( TASK_B, BASIC|ACTIVATE|OWNSTACK,2, task_b,,,,64 );
DefineTask( TASK_X, BASIC|ACTIVATE|NODESTACK,3, task_x,,,, );
...
[Counters]
DefineCounter(TimeCnt, 127, 1);
DefineCounter(DgrCnt, 36, 1);
[Alarms]
DefineAlarm( TimeAlm, TimeCnt, TASK_X);
DefineAlarm( DgrAlm, DgrCnt, TASK_B, 0x01);
...
[Event Messages]
DefineEventMessage( Norm, int, sizeof(int),
                    6,3,WithOverwriteCheck, WithoutTimeStamp);
```

The alarm `TimeAlm` activates the task `TASK_X` when the counter `TimeCnt` expires. The alarm `DgrAlm` sets the specified event for the task `TASK_B` when the counter `DgrCnt` expires.

The C-language code can be the following:

```
DeclareTask( TASK_B )
DeclareTask( TaskTime )
DeclareTask( TASK_X )
DeclareCounter( TimeCnt )
DeclareCounter( DgrCnt )
DeclareAlarm( TimeAlm )
DeclareAlarm( DgrAlm )
UsesEventMessage( Norm, SendReceive )

TASK task_time( void )
{
TickType curTime;
OSBYTE i=0;

InitCounter( TimeCnt, 0 );  /* init time counter with 0 value */
while (i != 1) {
    GetCounterValue( TimeCnt, &curTime ); /* read TimeCnt value */
    if( curTime == CONST )
```

```
        {           /* if desired value, activate TaskB */
          ActivateTask( TaskB );
          SetRelAlarm( TimeAlm, 10, 0 );
               /* activate TaskX when TimeCnt reaches 10 */
        }
    if( curTime > CONST )TerminateTask();
          /* if more than desired value, terminate the task */
    }
}

TASK task_b( void )
{
UsesEventMessage( Norm, SendReceive );
EventMaskType evMask;

evMask = 0x01;
InitCounter( DgrCnt, 0 );   /* init degree counter with 0 value */
SetAbsAlarm( DgrAlm, 75, 15 ); /* set cyclic alarm */
WaitEvent( evMask );
        /* wait for event which must be set by the alarm */
Norm = 1;  /* wake up and send the message that all is OK */
SendEventMesssage( Norm );
TerminateTask();
}

ISR Timer_Isr( void )
{
...          /* reset the hardware */
EnterISR();
CounterTrigger( TimeCnt); /* increment the counter */
LeaveISR();
}

ISR Dgr_Isr( void )
{
...          /* reset the hardware */
EnterISR();
CounterTrigger( DgrCnt); /* increment the counter */
LeaveISR();
}
```

16.6 Event Management Services

16.6.1 Data Types

The OSEK 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.

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

16.6.2 SetEvent

Syntax:

```
StatusType SetEvent( TaskRefType <TaskName>,
                     EventMaskType <Mask> );
```

Input:

• *<TaskName>*	- a reference to the task (the task’s name).
• *<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 may be referenced to set an event.

Particularities:

Any events not set in the event mask remain unchanged.

It is possible to set events for the running task (task-caller).

This service is not implemented if the system configuration option *Events* or *GetResource* is turned off in the configuration file.

Status:

• Standard:

— *E_OK*	- no error.

• Extended:

— *E_ID*	- the task identifier is invalid;
— *E_ACCESS*	- the referenced task is not an Extended Task;
— *E_STATE*	- the referenced task is in the suspended state;
— *E_CALLEVEL*	- a call at the interrupt level is not allowed.

Conformance: ECC1

16.6.3 ClearEvent

Syntax:

```
StatusType ClearEvent( EventMaskType <Mask> );
```

Input:

• *<Mask>*	- an event mask to be cleared.

Output:

- None.

Description:

The task which calls this service defines the event which has to be cleared.

Particularities:

The system service *ClearEvent* is restricted to Extended Tasks.

This service is not implemented if the system configuration option *Events* or *ClearEvent* are turned off in the configuration file.

Status:

- Standard:

— *E_OK*	- no error.

- Extended:

— *E_ACCESS*	- the calling task is not an Extended Task;
— *E_CALLEVEL*	- a call at the interrupt level is not allowed.

Conformance: ECC1

16.6.4 GetEvent

Syntax:

```
StatusType GetEvent( TaskRefType <TaskName>,
                     EventMaskRefType <Event> );
```

Input:

- *<TaskName>* - a reference to the task.

Output:

- *<Event>* - a pointer to the event mask to be filled.

Description:

The event mask which is referenced to in the call is filled according to the current state of the events of the desired task.

It is possible to get event mask of the running task (task-caller).

Particularities:

The referenced task must be an extended task.

This service is not implemented if the system configuration option *Events* or *GetEvent* are turned off in the configuration file.

Status:

- Standard:
  - — *E_OK* - no error.
- Extended:
  - — *E_ID* - the task identifier is invalid;
  - — *E_ACCESS* - the referenced task is not an Extended Task;
  - — *E_STATE* - the referenced task is in the suspended state;
  - — *E_CALLEVEL* - a call at the interrupt level is not allowed.

Conformance: ECC1

16.6.5 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.

Particularities:

This call enforces the rescheduling, if the wait condition occurs.

This service is not implemented if the system configuration option *Events* or *WaitEvent* are turned off in the configuration file.

Status:

• Standard:

— *E_OK*	- no error.

• Extended:

— *E_ACCESS*	- the calling task is not an Extended Task;
— *E_RESOURCE*	- the calling task occupies resources;
— *E_CALLEVEL*	- a call at the interrupt level is not allowed.

Conformance: ECC1

16.6.6 Examples of using events

The example below shows how events can be used in the OSEK Operating System.

The following definitions can be made in the definition file:

```
[Tasks]
DefineTask( TASK_A, EXTENDED|ACTIVATE|OWNSTACK,1, taskA_e,,,,64 );
DefineTask( TASK_B, EXTENDED|POOLSTACK,1, taskB_e,,,,POOL1 );
DefineTask( TASK_C, BASIC|ACTIVATE|NODESTACK,3, taskC_bas,,,,);
...
[Counters]
DefineCounter(DgrCnt, 150, 1);
[Alarms]
DefineAlarm( AWAKE, DgrCnt, TASK_B, 0x01);
```

The C-language file:

```
#define X_FLG  0x80    /* define masks for internal flags */
#define Y_FLG  0x40
#define Z1_FLG  0x20
#define Z2_FLG  0x10

DeclareTask( TASK_A )    /* the extended tasks */
DeclareTask( TASK_B )
DeclareTask( TASK_C )    /* the basic task */


TASK taskA_e (void)          /* Extended task TASK_A */
{
EventMaskType aa = 3;        /* ‘external’ events */
EventMaskType x, z1 = Z1_FLG,  /* ‘internal’ events (flags) */
                z2 = Z2_FLG;
int speed;

...
/* Check the variable and set internal flag if needed */
if (speed == LIMIT)
       {
         x = X_FLG;
         SetEvent( TASK_A, x );
       }
...

GetEventMask( TASK_A, &x );  /* check internal flag */
/* Perform some actions in accordance with internal flag status */
if ((x & X_FLG) != 0 ) ClearEvent( z1 );
else SetEvent( TASK_A, z2 );
if ((x & Y_FLG) == 0 ) ChainTask( TASK_C );
...

   WaitEvent( aa );  /* the task is stopped until one of three
                 'external' events is set by another task */

   ClearEvent( aa );  /* clear all 'external' events after
                          awakening  */
   ...
}


#define EVMASK1  0x01     /* event mask definitions */
#define EVMASK2  0x02
#define EVMASK3  0x04


TASK taskB_e (void)          /* Extended task TASK_B */
{
EventMaskType b_ev, a_ev;
b_ev = EVMASK1 | EVMASK3;
InitCounter( DgrCnt, 10 ); /* initialize the counter */
...
SetRelAlarm( AWAKE, 20 );  /* this alarm will awake the task */
WaitEvent( b_ev );    /* waiting for one of two events */

/* The task will be ready again when one of two events are set. One
of them - EVMASK1 will be set by the alarm AWAKE after 20 ticks of
the counter DgrCnt. Thus, the task can delay itself. */

ClearEvent( b_ev );   /* clear all events */
GetEvent( TASK_A, &a_ev );  /* get events of TASK_A */
if ( (a_ev & EVMASK2) == 0)
        {
          a_ev = EVMASK2;
          SetEvent( TASK_A, a_ev );
        }  /* set the event for TASK_A */
...
}
```

```
TASK taskC_bas              /* Basic task TASK_C */
{
EventMaskType bb, set;
set = EVMASK3;
...
GetEventMask( TASK_B, &bb );  /* if the event is clear, set it */
if ((bb & EVMASK3) == 0 ) SetEvent( TASK_B, set );
...
}
```

16.7 Communication Management Services

16.7.1 Data Types and Identifiers

The following names are used in the OSEK Operating System for work with messages:

• ***SymbolicName***	This is a unique name representing a message. It only can be used in conjunction with calls of the message service. A *SymbolicName* need not be a data type. Variables or constants of *SymbolicName* can be declared or used.
• ***AccessModeName***	This is a unique name defining access to a message object. Legal names are ***Send***, ***Receive*** and ***SendReceive***.
• ***CopyQualifierName***	This is a unique name defining whether local copies of a state message will be created. Legal names are ***WithCopy*** and ***WithoutCopy***.

16.7.2 Message Declaration

Messages must be declared before their using. The message can be declared either within the function (or task) where it is referenced, or outside the function where it is referenced (global message declarations). In the latter case the user is responsible for consistency of this message. It means that during operations with the global message special precautions should be taken so the task cannot be preempted by another task which can change the message data.

Declaration is made by the special statements placed in the beginning of the function.

16.7.2.1 State Message Declaration

Syntax:

```
UsesStateMessage[AsParameter]( SymbolicName <Msg>,
                               AccessModeName <Access>,
                               CopyQualifierName <Copy> );
```

Input:

• *<Msg>*	- symbolic name of a message
• *<Access>*	- symbolic name for access mode to the message object
• *<Copy>*	- symbolic name for copy qualifier

Description:

UsesStateMessage serves as an external declaration of a State Message. The function and use of this service are similar to that of the external declaration of variables. The qualifier *<Access>* specifies whether the function *Send*, *Receive* or *SendReceive* access to this message.

The parameter *<Copy>* indicates whether local copies of the message are created. This parameter must be consistent for the operations *SendStateMessage* and *ReceiveStateMessage*. The *<Copy>* and *<Access>* parameters can have different values for the same message in different declarations (e.g., one task declares a message for send only and the other task for receive only).

Particularities:

The using task or function can only call *SendStateMessage* and assign values to the variable *Msg* when *Send* or *SendReceive* access is declared. The using task or function can only call *ReceiveStateMessage* for the variable <*Msg*> when *Receive* or *SendReceive* access is declared.

Conformance: BCC1

16.7.2.2 Event Message Declaration

Syntax:

```
UsesEventMessage[AsParameter]( SymbolicName <Msg>,
                               AccessModeName <Access> );
```

Input:

• *<Msg>*	- symbolic name of a message
• *<Access>*	- symbolic name for access mode to the message object

Description:

UsesEventMessage serves as an external declaration of an Event Message. The function and use of this service are similar to that of the external declaration of variables. The qualifier *<Access>* specifies whether the function *Send*, *Receive* or *SendReceive* access to this message.

The *<Access>* parameters can have different values for the same message in different declarations (e.g., one task declares a message for send only and the other task for receive only).

Particularities:

The using task or function can only call *SendEventMessage* and assign values to the variable *<Msg>* when *Send* or *SendReceive* access is declared. The using task or function can only call *ReceiveEventMessage* for the variable *<Msg>* when *Receive* or *SendReceive* access is declared.

Conformance: BCC1

16.7.3 SendStateMessage

Syntax:

```
StatusType SendStateMessage( SymbolicName <Msg>,
                             CopyQualifierName <Copy> );
```

Input:

• *<Msg>*	- a symbolic name of the message;
• *<Copy>*	- a symbolic name for the copy qualifier.

Output:

- None.

Description:

Call admissible both from ISR and task levels. This service updates the message <*Msg*>. If the operation is performed with the ***WithCopy*** qualifier, then the contents of the message item is copied from the task data space into the message area. If ***WithoutCopy*** is specified, then no copying is performed, because the task data space is updated directly by the user code by means of de-referencing the pointer <*Msg*>.

In the Extended Status version the service indicates the receiver that the message is valid.

The operating system can restart an alarm, activate a task, or set an event for a task, if signalling is defined for the message. If any of these operations changes the state of the scheduler’s queues, then rescheduling is performed.

Particularities:

Multiple calls of *SendStateMessag*e are permissible. In such cases, the message data is updated more frequently.

If message alarm for the state message *<Msg>* is defined (as part of the system definition), each call of the *SendStateMessage* routine restarts the alarm.

Task activation or event settings is performed, if it is defined for the State Message *<Msg>* (see **12.4.14 SetEventOnMessage** and **12.4.13 ActivateOnMessage**).

This service is not implemented if the system configuration option *StateMessage* or *SendStateMessage* are turned off in the configuration file.

Status:

- Standard:

— E_OK	- no error.

- Extended:
  - E_ID - *<Msg>* is invalid.

Conformance:

- BCC1;
- Multiple activation: BCC3;
- Event setting: ECC1.

16.7.4 ReceiveStateMessage

Syntax:

```
StatusType ReceiveStateMessage( SymbolicName <Msg>,
                                CopyQualifierName <Copy> );
```

Input:

• *<Msg>*	- a symbolic name of the message;
• *<Copy>*	- a symbolic name for the copy qualifier.

Output:

• None.

Description:

This service reads the message *<Msg>*. If the operation is performed with the ***WithCopy*** qualifier, then the contents of the message item are copied into the task data space. If ***WithoutCopy*** is specified, then no copying is performed, because the task data space is updated directly by the user code by means of de-referencing the pointer *<Msg>*.

Before copying the message the service checks the status of the message (if the Extended Status is used). If the message has no value, *E_ID* is returned.

A *ReceiveStateMessage* call is normally located at the beginning of a task or a function, to ensure that *<Msg>* has a valid value before the first use.

Particularities:

This service is not implemented if the system configuration option *StateMessage* or *ReceiveStateMessage* are turned off in the configuration file.

Status:

• Standard:

— *E_OK*	- no error.

• Extended:

— *E_ID*	- *<Msg>* is invalid or hasn't value.

Conformance: BCC1

16.7.5 SendEventMessage

Syntax:

```
StatusType SendEventMessage( SymbolicName <Msg> );
```

Input:

- *<Msg>* - a symbolic name of the message

Output:

- None.

Description:

Call admissible both from ISR and task levels. This service sends the Event Message *<Msg>*, i. e. the local copy is stored in the message FIFO queue.

The sent message is copied from the task data space into the free location of the message FIFO area. If there is no free space left for the message, then an overflow is detected, and the oldest message item is overwritten by a new message item. In the extended status versions the overflow is indicated to the user by means of returning the *E_LIMIT* value.

The operating system can restart an alarm, activate a task, or set an event for the task, if signalling is defined for the message. If any of these operations changes the state of the scheduler's queues, then rescheduling is performed.

The operation is always performed with copy.

Particularities:

If the message alarm for the Event Message *<Msg>* is defined (as part of the system definition), each call of the *SendEventMessage* restarts the alarm.

Task activation or event setting is performed, if it is defined for the Event Message *Msg*.

In the case of a FIFO overflow, the oldest message is overwritten and in Extended Mode *E_LIMIT* is returned.

This service is not implemented if the system configuration option *EventMessage* or *SendEventMessage* are turned off in the configuration file.

Status:

- Standard:
  — *E_OK* - no error.

- Extended:

— *E_ID*	- *<Msg>* is invalid;
— *E_LIMIT*	- FIFO overflow.

Conformance:

- BCC1;
- Multiple activation: BCC3;
- Event:ECC1.

16.7.6 ReceiveEventMessage

Syntax:

```
StatusType ReceiveEventMessage( SymbolicName <Msg> );
```

Input:

- *<Msg>* - a symbolic name of the message

Output:

- None.

Description:

This service receives the message *<Msg>* from the message FIFO queue. The contents of the message item is copied from the message FIFO area into the task data space.

If there is no messages into the FIFO queue, then the *E_NOMSG* status is returned, otherwise the oldest message in the FIFO area is copied.

In the Extended Status version the overflow, occurred during the last send operation is indicated to the user by means of returning the *E_LIMIT* value.

If the message has only one receiver, then the message is consumed (removed from the FIFO queue) by this operation. If the message has N receivers, the message is consumed only when the last receiver gets the message. The receiver which has already got the message cannot read it twice - in case the second attempt to get the same message the *E_NOMSG* status is returned.

When the message is consumed (either by a single or the last receiver), the received message is removed from the FIFO area, and the allocated space may be used in a send operation.

The operation is always performed with copy.

The call to this service is normally located at the beginning of a task or a function, to ensure that *<Msg>* has a valid value before the first use.

Particularities:

This service is not implemented if the system configuration option *EventMessage* or *ReceiveEventMessage* are turned off in the configuration file.

Status:

- Standard:

— *E_OK*	- no error;
— *E_NOMSG*	- no message is available: FIFO empty, or the message was already received by the running task.

- Extended:

— *E_ID*	- *<Msg>* is invalid;
— *E_LIMIT*	- at least one message within the event message object has been overwritten during a *SendEventMessage* operation since the last call to *ReceiveEventMessage*.

Conformance: BCC1

16.7.7 Examples of using messages

Examples below present the usage of system services for communication. The State Message `MsgBAst` has a timestamp and it is defined as a message of the `MSGTS` type.

The following definitions can be made in the definition file:

```
[Tasks]
DefineTask( TASK_A, EXTENDED|POOLSTACK,1, task_a,,,,POOL1 );
DefineTask( TASK_B, BASIC|ACTIVATE|OWNSTACK,2, task_b,,,,64 );
DefineTask( TASK_X, BASIC|ACTIVATE|NODESTACK,3, task_x,,,, );
...
[Counters]
DefineCounter(Post, 24, 1);
...
[State Messages]
DefineStateMessage( msgAAst, int, sizeof(int), WithoutTimeStamp);
DefineStateMessage( msgBAst, MSGTS, sizeof(int), WithTimeStamp);
DefineStateMessage( msgCBst, int, sizeof(int), WithoutTimeStamp);
[Event Messages]
DefineEventMessage( msgDBev, int, sizeof(int),
                    6,3,WithOverwriteCheck, WithoutTimeStamp);
```

The C-language code can be the following:

```
DeclareTask( TASK_A )
DeclareTask( TASK_B )
DeclareTask( TASK_X )

UsesStateMessage( msgAAst, SendReceive,WithoutCopy )
UsesStateMessage( msgBAst, SendReceive, WithCopy )
UsesStateMessage( msgCBst, SendReceive, WithCopy )
UsesEventMessage( msgDBev, SendReceive )

DeclareCounter( Post );

typedef struct tagMSGTS MSGTS;
 struct tagMSGTS
  {
   TickType ts;
   int x;
  };

void Func( UsesEventMessageAsParameter( msgDBev, Receive));
{  /* The function uses the Event Message as a parameter -
       receives and processes it */
ReceiveEventMessage( msgDBev );
...
}


TASK task_a
{
...
ReceiveStateMessage( msgCBst, WithCopy );  /* get the message */
if( msgCBst == 2 ) *msgAAst = 1;
else *msgAAst = 33;
SendStateMessage( msgAAst, WithoutCopy );
...
msgBAst.x = 60;
GetCounterValue( Post, &(msgBAst.ts) );
SendStateMessage( msgBAst, WithCopy );
...

Func( msgDBev );
...
}


TASK task_b
{
TickType cur_time;
...
msgCBst = 15;
SendStateMessage( msgCBst, WithCopy );
...
ReceiveStateMessage( msgAAst, WithoutCopy );
if( *msgAAst == 1 ) ActivateTask( TASK_X );

ReceiveStateMessage( msgBAst, WithCopy );
msgBAst.ts = 60;
GetCounterValue( Post, &cur_time );
if( (cur_time - msgBAst.ts) > 15 ) SetEvent( TASK_X, 1 );
...

SendEventMessage( msgEBev );
...
}
```

16.8 Error Handling and Debugging Services

16.8.1 OSShutdown

Syntax:

```
void OSShutdown( StatusType <Error> );
```

Input:

- *<Error>* - a code of the error occurred.

Output:

- None.

Description:

The user can call this system service to abort the overall system (e.g. emergency off). The operating system also calls this function internally, if it has reached an undefined state and is no longer ready to run.

The meaning of *<Error>* if it is supplied by the application is user-defined. If the operating system calls *OSError* (see **16.8.2.1**) the error parameters listed in **11.2.1 Error Interface** may be transferred.

Particularities:

OSShutdown never returns to the location where it was called.

OSShutdown is application specific, since standardized error treatment is not possible.

OSShutdown runs in connection with the currently active context, which may be unknown to the user. Thus, no API functions are admitted within the *OSShutdown* routine.

This service is not implemented if the system configuration option *ErrorHandler* is turned off in the configuration file.

Status:

- Standard:
  — No return to call level.
- Extended:
  — No return to call level.

Conformance: BCC1

16.8.2 Hook Routines

16.8.2.1 OSError

Syntax:

```
void OSError( StatusType <Error>,...);
```

Input:

- *<Error>* - a code of the error occurred.

Output:

- None.

Description:

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*. It is called before returning to the call level.

For error parameters to be transferred see **16.8.1 OSShutdown**.

Particularities:

See **11.1 Hook Routines** for general description of hook routines.

This service is not implemented if the system configuration option *HookRoutines* is turned off in the configuration file.

Status:

- None.

Conformance: BCC1

16.8.2.2 OSPreTask

Syntax:

```
void OSPreTask( TaskRefType <TaskName>, ...);
```

Input:

- *<TaskName>* - a task identifier of the current task.

Output:

- None.

Description:

This hook is called before the operating system enters the context of the task. 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:

See **11.1 Hook Routines** for general description of hook routines.

This service is not implemented if the system configuration option *HookRoutines* is turned off in the configuration file.

Status:

None.

Conformance: BCC1

16.8.2.3 OSPostTask

Syntax:

```
void OSPostTask( TaskRefType <TaskName> );
```

Input:

- *<TaskName>* - a task identifier of the current task.

Output:

- None.

Description:

This hook is called after the operating system leaves the context of the task. 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.

Particularities:

See **11.1 Hook Routines** for general description of hook routines.

Status:

None.

Conformance: BCC1

APPENDIX A
SAMPLE APPLICATION

A.1 Description

The Sample application delivered with the OSEK Operating System should help to learn how to use OSEK OS. The Sample's source files are located in the SAMPLE directory - it contains all files needed to create an executable file.

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 OSEK OS mechanisms and allows the user to have the first look inside the OSEK OS.

The Sample consists of four tasks. It uses two counters (one of them is the standard OSEK OS timer), three alarms, one resource, one state and one event message. Timer interrupt is handled by the ISR.

Generally, Sample tasks are divided into two pairs. `TASKSND` and `TASKRCV` compose the first pair and `TASKPROD`, `TASKCONS` are the second pair. This two pairs interacts with the help of the event mechanism.

The task `TASKSND` is activated by the *StartUp* routine. It gets the resource to provide exclusive access to the state message `MsgA`, modifies the message and sends it to `TASKRCV`. After that the resource is released and the task terminates itself. `MsgA` consists of two parts (it has the user defined type) and only one part ('value' is changed by `TASKSND`). The message is used without copying. Arrival of the message activates the task `TASKRCV`.

`TASKRCV` receives the message `MsgA` and analyzes it. If the '`value`' is greater than the certain limit, the second part of `MsgA` is increased and the event is set for the task `TASKPROD` (producer). In the end the task releases the resource and terminates itself.

The task `TASKPROD` is activated by the *StartUp* routine. Just after the activation the task activates the system counter (timer) and the counter for messages and sets two alarms. To prevent rescheduling at this moment the scheduler is got by the tasks as a standard resource. One alarm is the relative cyclic alarm based on the system timer, it will be used to "awake" the task, and the second alarm is the absolute alarm based on the message counter. The message counter is intended to count the number of items in the event message `MsgB`. After all initialization activities `TASKPROD` calls the *WaitEvent* service and, thus, the task delays itself.

When `TASKPROD` becomes *running* again, it sends the event message `MsgB`. The message contents depends on the event which has caused task awakening. The message has a timestamp which is filled by the current value of the system timer. After sending `MsgB` the message counter is triggered.

The task `TASKCONS` (consumer) has the lowest priority. But this is non-preemptive task so it cannot be preempted by other tasks. `TASKCONS` is activated by the alarm when the message counter reaches the predefined number. The task reads (consumes) event message `MsgB` items and analyzes them. In case of the “`S_OK`” message the message timestamp is saved in the variable “`normal`”. In case of the “`S_LIMIT`” message the period of time elapsed between the “`S_OK`” and “`S_LIMIT`” messages is saved in the variable “`period`”.

The user can watch the “`normal`” and “`period`” variables, the message contents and so on.

A.2 Configuration File

The file contains all needed definitions for the Sample application. If some properties are missed it means that their default values are used by the System Generator (see **SECTION 12 System Configuration**).

```
[Property]
TargetMCU = HC08;
ConformanceClass =ECC1;
SimpleScheduler = OFF;
ExtendedStatus = ON;
HookRoutines = ON;
ErrorHandler = ON;
ContextSwitchRoutine = ON;
InterruptMaskControl = OFF;
TaskIndexMethod = OFF;
PersistentNode = OFF;
TaskOwnStack = ON;
TaskBasePage = ON;
StackPool = ON;
SchedulerPolicy = MIXPREEMPT;
CounterSize = 16;
Alarms = ON;
AlarmList = ON;
Resources = ON;
FastResource = OFF;
Events = ON;
StateMessage = ON;
StateMsgDefaultValue = ON;
StateMsgTimeStamp = ON;
```

```
EventMessage = ON;
EventMsgTimeStamp = ON;
EventMsgOneToN = OFF;
ActivateOnMsg = ON;      /* Message signalling mechanism */
AlarmOnMsg = ON;
SetEventOnMsg = ON;

[Scheduler]
DefineScheduler( 4, 5, 128, , 64 ) ;

[Interrupt management]
DefineInterrupts(  0x8, 0, 0, 64 );

[User’s hook]
DefineHooks ( OSError, OSPreTask, OSPostTask);

[Tasks]
DefineTask ( TASKSND, BASIC|PREEMPT|ACTIVATE, 3, TaskSND );
DefineTask ( TASKRCV, PREEMPT| BASIC|OWNSTACK, 1,
            TaskRCV,,, TaskStack, TASKSTACKSIZE );
DefineTask ( TASKPROD, PREEMPT| EXTENDED|ACTIVATE|POOLSTACK, 2,
            TaskProd,,, POOL );
DefineTask ( TASKCONS, NONPREEMPT|BASIC|POOLSTACK, 4,
             TaskCons,,, POOL );
DefineStackPool( POOL, 70, 2 );

[Resources]
DefineResource( MSGACCESS, 1);

[Counters]
DefineSystemTimer( SYSTEMTIMER, -1, 10, 1000000 );
DefineCounter( MSGCOUNTER, 6, 1 );

[Alarms]
DefineAlarm(  MSGALARM, SYSTEMTIMER, TASKSND );
DefineAlarm(  PRODALARM, SYSTEMTIMER, TASKPROD, TIMEVENT );
DefineAlarm(  EVMSGALARM, MSGCOUNTER, TASKCONS );

[State Messages]
DefineStateMessage( MsgA, MSGATYPE, sizeof(MSGATYPE),
                   WithoutTimeStamp );
ActivateOnMessage( MsgA, TASKRCV );
DefineMessageAlarm( MsgA, MSGALARM, 100 );
```

```
[Event Messages]
DefineEventMessage( MsgB, MSGBTYPE, sizeof( int), 5, 1,
                   WithOverwriteCheck, WithTimeStamp );
```

A.3 Source Files

Source files for the Sample application are the following:

- “ossample.c” - the application code
- “cfg.def” - the configuration file
- “makefile” - command files to build the executable file
- “vector.c” - interrupt vector table definition

To build the executable file the user should make sure that OSEK OS components are properly installed on the disk and pathes for the OSEK directory and Cosmic software are known. Run the MAKE utility for the makefile. The makefile was written for Microsoft Visual C++ 4.2. When all produced filed are ready, the executable file can be load into the MMDS08 and run.

Makefile uses the system environment variables CXPATH, CXLIB and OSEKDIR to get Cosmic and OSEK components. See **14.1.2 Compiler Issues**.

APPENDIX B
SYSTEM SERVICE TIMING

B.1 General Notes

The following notation is used to define four main states for run-time services:

- **Immediate Response (IR)** - The service completes immediately without any tasking changes.
- **Stop Task (ST)** - The service transfers the task from the *running* state into another state (*ready*, *waiting* or *suspended*). Task switching will take place.
- **Task Waiting (TW)** - The service called transfer another task into the *ready* state (from *waiting* or *suspended* states). This task has the same or lower priority than the task calling the service, or preemption is disabled, therefore task switching will not take place.
- **Task Waiting and Context Switching (TWCS)** - The service called transfer another task into the *running* state (from *ready*, *waiting* or *suspended* states). This task has the higher priority than the calling task, therefore task switching is performed.

Results in **Table B–1** below was got on the basis of the certain OS configuration. The list of systetm properties is shown below, and this configuration is called in the table as “Initial”. Properties that are not listed have their default values. In the column “Condition” the differences from the given list (“Initial“) are indicated. For each configuration the corresponded numbers are provided in the table.

```
[Property]
TargetMCU= HC08;
ConformanceClass = BCC3;
SchedulerPolicy = FULLPREEMPT;

/* Task properties */
TaskIndexMethod = OFF;
StackPool = OFF;
PersistentNode = ON;
PersistentStack = OFF;
TaskOwnStack = ON;
EntryExitISR = ON;
InterruptMaskControl = ON;
/* Debuging and error handling */
ExtendedStatus = ON;
```

```
HookRoutines = ON;
ErrorHandler = ON;
ContextSwitchRoutine = ON;
InternalErrorHandler = ON;
Resources = OFF;
FastResource = OFF;
Events = OFF;
StateMessage  = ON;
StateMsgDefaultValue = ON;
StateMsgTimeStamp  = ON;
EventMessage  = ON;
EventMsgTimeStamp  = ON;
EventMsgOneToN  = ON;
ActivateOnMsg  = OFF;
AlarmOnMsg = OFF;
SetEventOnMsg = OFF;
Counters = ON;
Alarms = ON;
AlarmList = ON;
CounterSize = 32;
```

All numbers in the table are the number of MCU system ticks counted by reading the free running counter. Therefore, to calculate these numbers in microseconds, these numbers must be multiplied by the period of MCU clock. For instance, for HC08AZ32 with 16Mhz oscillator, the bus frequency equals 4Mhz and the clock period equals 0.25 microseconds the duration of *GetResource* service (for “Initial” system configuration) is 39.75 microseconds (159 * $2.7*10^{-7}$ = $39.75*10^{-6}$).

It is possible that some real numbers can slightly differ from the presented values due to some last changes in OSEK OS.

B.2 Data Sheet

Table B–1 OSEK OS Run-time Services Timing Characteristics

Conditions	IR	ST	TW	TWCS
ActivateTask				
BCC1, NONPREEMPT, Extended-Status=OFF, Counters=OFF, Alarms=OFF, StateMessages=OFF, EventMessages=OFF, no hook routines, no interrupts	-	-	960	963
NONPREEMPT	-	-	1098	1101
Initial	-	-	1253	1687
TerminateTask				
BCC1, NONPREEMPT, Extended-Status=OFF, Counters=OFF, Alarms=OFF, StateMessages=OFF, EventMessages=OFF, no hook routines, no interrupts	-	577	-	-
NONPREEMPT	-	970	-	-
Initial	-	902	-	-
ChainTask				
BCC1, NONPREEMPT, Extended-Status=OFF, Counters=OFF, Alarms=OFF, StateMessages=OFF, EventMessages=OFF, no hook routines, no interrupts	-	-	-	1486
NONPREEMPT	-	-	-	1486 min 2010 max
Initial	-	-	-	1214 min 1964 max
Schedule				
BCC1, NONPREEMPT, Extended-Status=OFF, Counters=OFF, Alarms=OFF, StateMessages=OFF, EventMessages=OFF	116	-	-	297
NONPREEMPT	187	-	-	652
Initial	198	-	-	-
GetTaskId				
Initial	87	-	-	-

Table B–1 OSEK OS Run-time Services Timing Characteristics

Conditions	IR	ST	TW	TWCS
GetTaskState				
Initial: running task ready task suspended task	132 400 462	-	-	-
EnterISR				
Resources=ON, FastResource=ON	117	-	-	-
LeaveISR				
Resources=ON, FastResource=ON	186	-	-	-
EnableInterrupt				
Resources=ON, FastResource=ON	32	-	-	-
DisableInterrupt				
Resources=ON, FastResource=ON	34	-	-	-
GetInterruptMask				
Resources=ON, FastResource=ON	52	-	-	-
GetResource				
Resources=ON, FastResource=ON	159	-	-	-
ReleaseResource				
Resources=ON, FastResource=ON	239	-	-	884
SetEvent				
ECC1, NONPREEMPT	455	-	642	655
ECC1	455	-	720	1194
ClearEvent				
ECC1, NONPREEMPT	75	-	-	-
ECC1	75	-	-	-
GetEvent				
ECC1, NONPREEMPT	515	-	-	-
ECC1	512	-	-	-
WaitEvent				
ECC1, NONPREEMPT	158	996	-	-
ECC1	158	894	-	-

Table B–1 OSEK OS Run-time Services Timing Characteristics

Conditions	IR	ST	TW	TWCS
SendStateMessage				
BCC1, a message with copy, without timestamp	685	-	-	-
ReceiveStateMessage				
BCC1, a message with copy, without timestamp	680	-	-	-
SendEventMessage				
BCC1, 1 receiver, with overwrite check	839	-	-	-
ReceiveEventMessage				
BCC1, 1 receiver, with overwrite check	915	-	-	-
InitCounter				
Resources=ON, FastResource=ON	442	-	-	-
CounterTrigger				
Resources=ON, FastResource=ON	400	-	1871	2281
GetCounterValue				
Resources=ON, FastResource=ON	256	-	-	-
GetCounterInfo				
Resources=ON, FastResource=ON	807	-	-	-
SetRelAlarm				
Resources=ON, FastResource=ON	3463	-	4534	4941
SetAbsAlarm				
Resources=ON, FastResource=ON	712	-	1764	2171
CancelAlarm				
Resources=ON, FastResource=ON	229	-	-	-
GetAlarm				
Resources=ON, FastResource=ON	457	-	-	-

APPENDIX C
MEMORY REQUIREMENTS

C.1 Memory for the OSEK Operating System

The table below contains the data about ROM and RAM needed for the OSEK 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 reflects all possible configurations so the overall number of them is too big (more than 2000). Therefore, only some most important configurations are presented.

The following initial system property settings were used to determine memory requirements:

HCBasePage = ON;
HCLowPower = OFF;
HCBankCode = OFF;
SimpleScheduler = ON;
SchedulerPolicy = FULLPREEMPT;
ExtendedStatus = OFF;
UseMainStack = OFF;
UseSameContext = ON;
TaskBasePage = ON;
MultiplyActivation = OFF;
StackPool = OFF;
NodeStack = OFF;
PersistentNode = OFF;
PersistentStack = OFF;
TaskOwnStack = ON;
TaskIndexMethod = OFF;
EntryExitISR = OFF;
InterruptMaskControl = OFF;
ErrorHandler = OFF;
ContextSwitchRoutine = OFF;
InternalErrorHandler = OFF;
Resources = OFF;
FastResource = OFF;
Events = OFF;
Counters = OFF;
CounterSize = 8;
Alarms = OFF;

AlarmList = OFF;
StateMessage = OFF;
StateMsgDefaultValue = OFF;
StateMsgTimeStamp = OFF;
EventMessage = OFF;
EventMsgTimeStamp = OFF;
EventMsgOneToN = OFF;
ActivateOnMsg = OFF;
AlarmOnMsg = OFF;
SetEventOnMsg = OFF;

This initial property list was used for the first row in the table. It conforms to the BCC1 Conformance Class without any additional mechanisms and this is the minimal OSEK OS configuration. The rows below reflects memory requirements for the next Conformance Classes. System properties are shown in the rows which are turned on for the corresponded Conformance Class. For BCC2, BCC3, ECC1, ECC2 the scheduler policy is full-preemptive one.

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. Thus, the system functionality grows up.

Under the title "**Extensions**" the additions are shown for each additional system property (or group of them). These numbers are got on the base of ECC2 configuration. For example, the row "Counters = ON" presents the additional memory requirements for this mechanism. It allows the user to evaluate the amount of memory needed to support some particular mechanisms and features. Differences between the amount of memory required to support these features for various Conformance Classes are comparatively small. Therefore, the data is provided only for ECC2. Thus, since each next row includes all functionality of the previous ones, the last row presents the memory requirements for the system with full functionality.

Since each system object (a task, a message, 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.

NOTE ABOUT FORMULAS:

In the formulas in the table the following symbols are used:

• T_s	is the number of tasks in non-suspended state;
• T_t	is the total number of tasks;
• C	is the number of counters;
• A	is the number of alarms;
• S	is the number of state messages;
• E	is the number of event messages;
• P	is the number of tasks' priorities;
• R	is the number of resources;
• S_p	is the number of stack pools;
• S_b	is the number of stack buffers.

It is possible that some real numbers can slightly differ from the presented values due to some last changes in OSEK OS.

C.2 Data Sheet

Table C–1 OSEK OS Memory Requirements

System Properties (configuration)	Conformance Class	ROM	RAM	Base Page RAM
Initial	BCC1	1185+6*Tt	-	15+7*Ts
Initial SchedulerPolicy = NOPREEMPT		1169+6*Tt	-	15+9*Ts
Initial SchedulerPolicy= MIXPREEMPT		1185+6*Tt	-	15+7*Ts

Initial
SchedulerPolicy= MIXPREEMPT
UseSameContext = OFF
		1312+6*Tt	-	15+9*Ts

Initial
HCBasePage = OFF
TaskBasePage = OFF
		1520+6*Tt	12+7*Ts	7

Changed: ExtendedStatus = ON		1661+7*Tt	10+7*Ts	7
Changed: ErrorHandler = ON		1858+7*Tt	12+7*Ts	7

Changed:
ExtendedStatus = OFF
ErrorHandler = OFF
EntryExitISR = ON
		1741+6*Tt	15+7*Ts	7

Changed: InterruptMaskControl = ON		1918+7*Tt	17+7*Ts	7
Initial SimpleScheduler = OFF	BCC2	1298+6*Tt		18+9*Ts

Changed:
HCBasePage = OFF
TaskBasePage = OFF
		1789+6*Tt	17+10*Ts	7

Table C–1 OSEK OS Memory Requirements

System Properties (configuration)	Confor-mance Class	ROM	RAM	Base Page RAM

Initial
SimpleScheduler = OFF
MultiplyActivation = ON
	BCC3	1331+6*Tt	-	18+10*Ts

<u>Changed:</u>
HCBasePage = OFF
TaskBasePage = OFF
		1826+6*Tt	17+11*Ts	7

<u>Changed:</u> ExtendedStatus = ON		1963+7*Tt	15+11*Ts	7
<u>Changed:</u> ErrorHandler = ON		2156+7*Tt	17+11*Ts	7

Initial
SimpleScheduler = OFF
MultiplyActivation = ON
Events = ON
	ECC1	1537+6*Tt	-	18+12*Ts

<u>Changed:</u>
HCBasePage = OFF
TaskBasePage = OFF
		2105+6*Tt	17+13*Ts	7

<u>Changed:</u> ExtendedStatus = ON		2361+7*Tt	15+13*Ts	7
<u>Changed:</u> ErrorHandler = ON		2721+7*Tt	17+13*Ts	7

<u>Changed:</u>
ExtendedStatus = OFF
ErrorHandler = OFF
EntryExitISR = ON
		2335+6*Tt	20+13*Ts	7

<u>Changed:</u> InterruptMaskControl = ON		2557+7*Tt	22+13*Ts	7

Initial
SimpleScheduler = OFF
MultiplyActivation = ON
Events = ON
	ECC2	1508+6*Tt	-	18+12*Ts

Table C–1 OSEK OS Memory Requirements

System Properties (configuration)	Confor-mance Class	ROM	RAM	Base Page RAM
Extensions (based on the configuration for ECC2)				
Changed: Resources = ON	ECC2	1706+6*Tt+3*R	2*R	18+14*Ts
Changed: FastResource = ON		1672+6*Tt	-	20+12*Ts
Changed: Counters = ON		1863+6*Tt+4*C	1*C	20+12*Ts
Changed: CounterSize = 16		1914+6*Tt+6*C	2*C	20+12*Ts
Changed: CounterSize = 32		2183+6*Tt+ 10*C	4*C	20+12*Ts
Changed: Alarms = ON CounterSize = 8		2967+6*Tt+6*C +7*A	2*C+8*A	20+12*Ts
Changed: AlarmList = ON		3164+6*Tt+4*C +7*A	3*C+11*A	20+12*Ts
Changed: StateMessages = ON		3450+6*Tt+4*C +7*A+4*S	3*C+11*A+ 2*S	20+12*Ts
Changed: StateMsgDefaultValue = ON		3492+6*Tt+4*C +7*A+6*S	3*C+11*A+ 2*S	20+12*Ts
Changed: EventMessages = ON		4037+6*Tt+4*C +7*A+6*S+9*E	3*C+11*A+ 2*S+8*E	20+12*Ts
Changed: EventMsgOneToN = ON		4224+6*Tt+4*C +7*A+6*S+12*E	3*C+11*A+ 2*S+8*E	20+12*Ts
Changed: ActivateOnMessage = ON		4307+6*Tt+4*C +7*A+9*S+15*E	3*C+11*A+ 2*S+8*E	20+12*Ts
Changed: SetEventOnMessage = ON		4329+6*Tt+4*C +7*A+9*S+15*E	3*C+11*A+ 2*S+8*E	20+12*Ts
Changed: AlarmOnMessage = ON		4691+6*Tt+4*C +7*A+12*S+ 18*E	3*C+11*A+ 2*S+8*E	20+12*Ts
Changed: StackPool = ON		5072+6*Tt+7*S p+4*C+7*A+ 12*S+18*E	3*Sp+3*C+ 11*A+2*S+ 8*E	20+14*Ts

Table C–1 OSEK OS Memory Requirements

System Properties (configuration)	Confor-mance Class	ROM	RAM	Base Page RAM
Changed: NodeStack = ON		5157+6*Tt+7*Sp+4*C+7*A+12*S+18*E	3*Sp+3*C+11*A+2*S+8*E	20+16*Ts
Changed: PersistentNode = ON		5260+7*Tt+7*Sp+4*C+7*A+12*S+18*E	3*Sp+3*C+11*A+2*S+8*E	20+16*Ts
Changed: PersistentStack = ON	ECC2	5355+7*Tt+7*Sp+4*C+7*A+12*S+18*E	3*Sp+3*C+11*A+2*S+8*E	20+16*Ts
Changed: HCBasePage = OFF TaskBasePage = OFF		6070+8*Tt+7*Sp+4*C+7*A+12*S+18*E	19+17*Ts+3*Sp+3*C+11*A+2*S+8*E	7
Changed: ExtendedStatus = ON		6934+9*Tt+7*Sp+6*C+7*A+12*S+18*E	17+17*Ts+3*Sp+3*C+12*A+4*S+9*E	7
Changed: ErrorHandler = ON		7600+9*Tt+7*Sp+6*C+7*A+12*S+18*E	19+17*Ts++3*Sp+3*C+12*A+4*S+9*E	7
Changed: EntryExitISR = ON		8469+9*Tt+7*Sp+6*C+7*A+12*S+18*E	22+17*Ts++3*Sp+3*C+12*A+4*S+9*E	7
Changed: InterruptMaskControl = ON		8827+10*Tt+7*Sp+6*C+7*A+12*S+18*E	24+17*Ts++3*Sp+3*C+12*A+4*S+9*E	7

APPENDIX D
SYSTEM GENERATION ERROR MESSAGES

D.1 Error Message Description Format

Below all System Generator error messages are described in the following format (see **12.5 System Generator Warning and Error Messages** about the error message format):

message text

Description

The description includes possible reasons of an error and recommended methods to eliminate it.

D.2 Error Messages

E0001: syntax error

A syntax error was found in the input stream.

E0001: invalid token ’<token>’

An invalid (unrecognizable) token was found in the input stream.

E0003: unexpected end of file found in comment

The compiler found the end of a file while scanning a comment.

A comment cannot be split across source files.

This error can also be caused by a comment on the last line of a source file that is not followed by the ‘CR’ symbol as in the example:

```
int i; // error: last line not terminated by newline
```

To eliminate this error, go to the end of the line and add a new line.

E0010: not enough memory

There is not enough memory for source file processing.

E0011: cannot open source file: <name>
cannot open include file: <name>
cannot open output file: <name>)

This file either did not exist, could not be opened, or was not found.

E0012: cannot read file: <name>

System generator encountered an error when trying to read a file.

This error can be caused by a disk error or by a file-sharing conflict.

E0013: cannot write file: <name>

An error occurred while SG was trying to write into the file. One possible reason of this error is insufficient disk space.

E0014: too many include files: depth = <level>

The nesting depth of `#include` directives is too great.

W0020: ignoring unknown option <flag>

The <flag> in the command-line option is not valid; the option is ignored.

W0020: <flag> requires an argument

A command-line option requires an argument, but nothing is specified.

E0021: no input file specified

The input file is not specified in the command line.

E0021: too many include paths specified

The number of specified paths for include files in the command line is greater than a limit.

E0030: 'identifier': system object redefinition

The given identifier has already been used for a system object of the corresponded type.

E0101: double property <name> definition

The property *<name>* has been previously defined and cannot be re-defined.

W0101: undefined property definition

The specified property name is not supported by the OSEK System Generator. The statement which has caused this warning is ignored by SG.

E0103: undefined value for property <name>

The desired value for the property *<name>* is not supported by the OSEK System Generator.

E0102: incorrect property <name> setting

The setting of the property *<name>* is incompatible with other current settings.

For example, if *Events* property is turned off and *SetEventOnMsg* property is turned on, the *SetEventOnMsg* property setting is incorrect.

E0105: method of task stack assignment is not defined

At least one of task stack assignment methods shall be defined, either *NodeStack* or *StackPool* or *TaskOwnStack*.

E0106: SimpleScheduler property can not be used with Resources property

If *SimpleScheduler* is used then the resources can not be used.

E0110: interrupt masks shall be defined

The *InterruptMaskControl* property is turned on, but the interrupt masks are not defined.

E0111: interrupt stack shall be defined

The *EntryExitISR* property is turned on, but the interrupt stack is not defined.

E0120: error hook shall be defined

The *ErrorHandler* property is turned on, but the error hook is not defined.

E0121: context switch routines shall be defined

The *ContextSwitchRoutine* property is turned on, but the context switch routines are not defined.

E0201: the number of task control blocks shall be greater than 0

The *<NumberOfTasks>* parameter of the *DefineScheduler* statement equals 0.

E0202: number of priorities shall be greater than 0

The *<NumberOfPriorities>* parameter of the *DefineScheduler* statement equals 0.

E0203: size of scheduler stack shall be defined

The *UseMainStack* property is turned off, but the *<SchedulerStackAddress>* parameter is not defined in the *DefineScheduler* statement.

E0204: size of task node stack shall be defined

The *NodeStack* property is turned on, but the *<TaskNodeStackSize>* parameter is not defined in the *DefineScheduler* statement.

E0205: number of task control blocks less than priority range

The number of task control blocks are less than maximum of task priority, it is not enough in case of *SimpleScheduler*.

W0201: NodeStack property is turned off, parameter ignored

If the *NodeStack* property is turned off then the node stacks are not supported and parameters defining this stack are ignored.

W0202: there are unused task control blocks

The number of task control blocks are greater than maximum defined priority of a task, therefore in case of *SimpleScheduler* the unused control blocks exist.

E0301: task may be either basic or extended (more than one task type attribute)

More than one task type attribute (*BASIC* and *EXTENDED*) was applied to a task, as in the following example:

```
DefineTask( TaskA, BASIC|EXTENDED|NONPREEMPT, 3, u, tAStart,, );
```

W0301: task type shall be defined

The task type is not defined, neither *BASIC* nor *EXTENDED* flags are defined for a task.

E0302: only basic tasks are allowed in basic conformance classes

The OS Conformance Class is defined as one of the BCC classes, but a task has the *EXTENDED* type.

W0302: task type set to basic by default

Due to one of reasons (E0301, W0301, E0302) the task type shall be redefined.

W0303: task preemptive property shall be defined

Task preemptive property is not defined, neither *PREEMPT* nor *NONPREEMPT* flags are defined for a task.

E0303: task may be either preempt or nonpreempt (more than one task preemptive property)

Both *NONPREEMPT* and *PREEMPT* flags are defined for a task.

W0304: non-preemptive tasks are not supported by full-preemptive scheduler (pre-emptive tasks are not supported by non-preemptive scheduler)

The defined task preemptive property is incompatible with defined scheduling policy.

W0305: task preemptive property set to preempt (task preemptive property set to non-preempt)

Due to some reasons (W0303, E0303, W0304) task preemptive property shall be redefined.

W0306: PersistentNode property is turned off, parameter ignored

A task has the defined *ASSIGNNODE* flag, but the *PersistentNode* property is turned off.

W0307: PersistentStack property is turned off, parameter ignored

A task has the defined *ASSIGNSTACK* flag, but the *PersistentStack* property is turned off.

W0308: persistent stack may be assigned only for task with persistent node, parameter ignored (ASSIGNSTK must be used with ASSIGNNODE)

A task has the defined *ASSIGNSTACK* flag, but the *ASSIGNNODE* flag is not defined. The *ASSIGNSTACK* flag is ignored in this case.

W0309: persistent stack shall be assigned from stack pool, parameter ignored

A task has the defined *ASSIGNSTACK* flag, but the *POOLSTACK* flag is not defined. The *ASSIGNSTACK* flag is ignored in this case.

W0310: TaskOwnStack property is turned off, parameter ignored

A task has the defined *OWNSTACK* flag, but the *TaskOwnStack* property is turned off.

W0311: StackPool property is turned off, parameter ignored

A task has the defined *POOLSTACK* flag, but the *StackPool* property is turned off.

E0304: task priority exceeds the maximum limit

The *<TaskPriority>* parameter of the *DefineTask* statement is greater than *<NumberOfPriorities>* which has been defined by the *DefineScheduler* statement.

E0305: the number of task nodes is not enough to allocate persistent nodes

The number of tasks with persistent node are greater than the number of nodes. Also this error is caused when the number of tasks with persistent node is equal to number of nodes and there are at least one more tasks.

E0306: the stack pool parameter shall be defined

A task has the defined *POOLSTACK* flag, but the *<TaskStack>* parameter of the *DefineTask* statement has not been defined.

E0307: the <name> stack pool is undefined

The *<TaskStack>* parameter of the *DefineTask* statement contains the name of a stack pool which has not been defined.

E0308: the basic task cannot be notified by event setting

The basic task is referenced in *DefineAlarm* or *SetEventOnMsg* statements to be notified by event setting.

E0309: at least one task shall be defined

At least one task shall be defined in the system.

E0310: more than one task stack attachment method

Only one stack attachment method shall be defined for a task.

E0311: task has no stack

This error may arise due to one of the following reasons:

- an undefined stack attachment method is used;
- the *NodeStack* property turned off and it is not defined other method for a task.

E0312: the stack size parameter shall be defined

A task has the defined *OWNSTACK* flag, but the *<TaskStackSize>* parameter of the *DefineTask* statement has not been defined.

E0401: resource priority exceeds the maximum limit

The *<ResourcePriority>* parameter of the *DefineResource* statement is greater then *<NumberOfPriorities>* has been defined by the *DefineScheduler* statement.

E0501: system timer is already defined

The *DefineSystemTimer* statement shall be defined only once in the definition file.

E0502: system timer is undefined

The *DefineSystemTimer* statement shall always be in the configuration file.

E0601: assigned counter <name> for alarm <name> is undefined

The *<CounterName>* parameter of the *DefineAlarm* statement contains the name of a counter which has not been defined.

E0602: assigned task <name> for alarm <name> is undefined

The *<TaskName>* parameter of the *DefineAlarm* contains the name of a counter which has not been defined.

W0601: Events property is turned off, parameters ignored

The specified method of task notification is not supported.

W0701: StateMsgDefaultValue property is turned off, parameter ignored

The *<DefaulValue>* parameter of *DefineStateMessage* statement is defined, but *StateMsgDefaultValue* property is turned off.

W0701: StateMsgTimeStamp property is turned off, parameter ignored

The *<TStamp>* parameter of *DefineStateMessage* statement is defined as ***WithTimeStamp***, but *StateMsgTimeStamp* property is turned off.

W0703: EventMsgTimeStamp property is turned off, parameter ignored

The *<TStamp>* parameter of *DefineEventMessage* statement is defined as ***WithTimeStamp***, but *EventMsgTimeStamp* property is turned off.

W0704: EventMsgOneToN property is turned off, parameter ignored

The number of receivers is greater than one, but *EventMsgOneToN* is turned off.

W0705: AlarmOnMsg property is turned off, definition ignored

The *DefineMesssageAlarm* statement is detected, but *AlarmOnMsg* property is turned off.

W0706: message shall be defined before its alarm

The *DefineMessageAlarm* statement has been found before the corresponded *DefineStateMessage* or *DefineEventMessage* statement.

W0707: message shall be defined before its notification

Either *ActivateOnMessage* or *SetEventOnMessage* statement has been found before the corresponded *DefineStateMessage* or *DefineEventMessage* statement.

W0708: ActivateOnMsg property is turned off, definition ignored

The ActivateOnMessage statement has been found, while the *ActivateOnMsg* property is turned off.

W0709: SetEventOnMsg property is turned off, definition ignored

The SetEventOnMessage statement has been found, while the *SetEventOnMsg* property is turned off.

E0701: assigned alarm <name> for message <name> is undefined

The *<AlarmName>* parameter of the *DefineMessageAlarm* statement contains the name of alarm which has not been defined.

E0702: alarm for message <name> was defined in line ##

The message shall have only one assigned alarm.

E0703: notified task <name> for message <name> is undefined

The *<TaskName>* parameter of the *ActivateOnMessage* or *SetEventOnMessage* statement contains the name of a task which has not been defined.

E0704: notification for message <name> was defined in line ##

The message shall have only one notification method.

E0705: message <name>: parameters are undefined

Either *ActivateOnMessage*, or *SetEventOnMessage*, or *DefineMessageAlarm* statement has been processed without the corresponded *DefineStateMessage* or *DefineEventMessage* statement (see W0706 and W0707).

APPENDIX E
SYSTEM SERVICES QUICK REFERENCE

The brief list of all OSEK Operating System run-time services is provided here. Input and output parameters, syntax and ability to use in ISR are shown.

Service	Input	Output	ISR
ActivateTask	Task name	-	Yes
	syntax: *StatusType ActivateTask(TaskRefType <TaskName>);*		
TerminateTask	-	-	No
	syntax: *StatusType TerminateTask(void);*		
ChainTask	Task name	-	No
	syntax: *StatusType ChainTask(TaskRefType <TaskName>);*		
Schedule	-	-	No
	syntax: *StatusType Schedule(void);*		
GetTaskId	-	Task name	No
	syntax: *StatusType GetTaskId(TaskRefType <TaskName>);*		
GetTaskState	Task name	Task state	Yes
	syntax: *StatusType GetTaskState(TaskRefType <TaskName>, TaskStateRefType <State>);*		
EnterISR	-	-	Yes
	syntax: *StatusType EnterISR(void);*		
LeaveISR	-	-	Yes
	syntax: *StatusType LeaveISR(void);*		
EnableInterrupt	Interrupt mask	-	Yes
	syntax: *StatusType EnableInterrupt(IntMaskType <Mask>);*		
DisableInterrupt	Interrupt mask	-	Yes
	syntax: *StatusType DisableInterrupt(IntMaskType <Mask>);*		
GetInterruptMask	-	Interrupt mask	Yes
	syntax: *StatusType GetInterruptMask(IntMaskRefType <Mask>);*		
GetResource	Resource name	-	No
	syntax: *StatusType GetResource(ResourceRefType <ResName>);*		
ReleaseResource:	Resource name	-	No
	syntax: *StatusType ReleaseResource(ResourceRefType <ResName>);*		
SetEvent	Task name, Event mask	-	No
	syntax: *StatusType SetEvent (TaskRefType <TaskName>, EventMaskType <Mask>);*		
ClearEvent	Event mask	-	No
	syntax: *StatusType ClearEvent(EventMaskType <Mask>);*		

Service	Input	Output	ISR
GetEvent	Task name	Event mask	No
	syntax: *StatusType GetEvent(TaskRefType <TaskName>, EventMaskRefType <Mask>);*		
WaitEvent	Event mask	-	No
	syntax: *StatusType WaitEvent(EventMaskType <Mask>);*		
SendStateMessage	Message name, copy qualifier	-	Yes
	syntax: *StatusType SendStateMessage(<MessageName>, <CopyQualifier>);*		
ReceiveStateMessage	Message name, copy qualifier	-	No
	syntax: *StatusType ReceiveStateMessage(<MessageName>, <CopyQualifier>);*		
SendEventMessage	Message name	-	Yes
	syntax: *StatusType SendEventMessage(<MessageName>);*		
ReceiveEventMessage	Message name	-	No
	syntax: *StatusType ReceiveEventMessage(<MessageName>);*		
InitCounter	Counter name, initial value	-	No
	syntax: *StatusType InitCounter(CtrRefType <CtrName>, TickType <Ticks>);*		
CounterTrigger	Counter name	-	Yes
	syntax: *StatusType CounterTrigger(CtrRefType <CtrName>);*		
GetCounterValue	Counter name	Counter value	No
	syntax: *StatusType GetCounterValue(CtrRefType <CtrName>, TickRefType <Ticks>);*		
GetCounterInfo	Counter name	Counter constants	No
	syntax: *StatusType GetCounterInfo(CtrRefType <CtrName>, CtrInfoRefType <Info>);*		
SetRelAlarm	Alarm name, Counter relative value, Cycle value	-	No
	syntax: *StatusType SetRelAlarm (AlarmRefType <AlarmName>, TickType <Increment>, TickType <Cycle>);*		
SetAbsAlarm	Alarm name, Counter absolute value, Cycle value	-	No
	syntax: *StatusType SetAbsAlarm (AlarmRefType <AlarmName>, TickType <Start>, TickType <Cycle>);*		
CancelAlarm	Alarm name	-	No
	syntax: *StatusType CancelAlarm(AlarmRefType <AlarmName>);*		
GetAlarm	Alarm name	Relative value in ticks before the alarm expires	No
	syntax: *StatusType GetAlarm(AlarmRefType <AlarmName>, TickRefType <Ticks>);*		

Index

Index

Index Entry	Page Number	Index Entry	Page Number