To all our customers

Regarding the change of names mentioned in the document, such as Hitachi Electric and Hitachi XX, to Renesas Technology Corp.

The semiconductor operations of Mitsubishi Electric and Hitachi were transferred to Renesas Technology Corporation on April 1st 2003. These operations include microcomputer, logic, analog and discrete devices, and memory chips other than DRAMs (flash memory, SRAMs etc.) Accordingly, although Hitachi, Hitachi, Ltd., Hitachi Semiconductors, and other Hitachi brand names are mentioned in the document, these names have in fact all been changed to Renesas Technology Corp. Thank you for your understanding. Except for our corporate trademark, logo and corporate statement, no changes whatsoever have been made to the contents of the document, and these changes do not constitute any alteration to the contents of the document itself.

Renesas Technology Home Page: http://www.renesas.com

Renesas Technology Corp.
Customer Support Dept.
April 1, 2003

RENESAS
Renesas Technology Corp.

Cautions

Keep safety first in your circuit designs!

1. Renesas Technology Corporation puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage.
   Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corporation product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation or a third party.
2. Renesas Technology Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor for the latest product information before purchasing a product listed herein.
   The information described here may contain technical inaccuracies or typographical errors.
   Renesas Technology Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.
   Please also pay attention to information published by Renesas Technology Corporation by various means, including the Renesas Technology Corporation Semiconductor home page (http://www.renesas.com).
4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
5. Renesas Technology Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
6. The prior written approval of Renesas Technology Corporation is necessary to reprint or reproduce in whole or in part these materials.
7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.
   Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
8. Please contact Renesas Technology Corporation for further details on these materials or the products contained therein.

Ho7055 Operating System Manual

ADE-702-256

Cautions

PLEASE READ THE FOLLOWING CAREFULLY BEFORE YOU USE THIS PRODUCT.

1. If you use the enclosed software product and any related software products (hereafter referred to as "PRODUCT"), before exporting or taking such PRODUCT to other countries or states, you must comply with applicable export control laws and regulations of Japan and other countries with jurisdiction and the applicable states and provinces within Japan and such other countries.

2. Please be advised that Hitachi neither warrants nor grants licenses of any rights to the patents, copyrights, trademarks, or other intellectual property rights owned by Hitachi or any third party for the use of the PRODUCT, unless otherwise expressly granted to you by Hitachi in a contract or other document including without limitation any warranty or license included in the user's manual for the PRODUCT (hereinafter referred to as "CONTRACTS"). Please be further advised that Hitachi bears no responsibility for problems that may arise with third party's rights, including intellectual property rights, in connection with the use of the PRODUCT.

3. The PRODUCT, its specifications and/or its description in the user's manual are subject to change in the future without any prior notice. Confirm that you have received the latest standards and/or specification for the PRODUCT (including the user's manual) before you make your final design, purchase or use.

4. Please be advised that Hitachi will not have any liability whatsoever for damages, including indirect or consequential damages, arising out of your use of the PRODUCT (including the use based on the descriptions of the user's manual). Hitachi shall not be liable for any damages caused by any equipment or media used for delivery of the PRODUCT.

5. The PRODUCT is not designed for, and you may not use the PRODUCT for, applications that demand especially high quality and reliability, or where its failure or malfunction may directly threaten human life or cause risk of bodily injury, such as equipment used for aerospace, aeronautics, nuclear power, combustion control, transportation, traffic, safety equipment or medical equipment for life support. If you have any questions regarding whether or not your intended use of the PRODUCT is permitted by Hitachi,, please contact your local Hitachi's sales office.

6. At the time of designing or planning your system using the PRODUCT, please consider normally foreseeable failure rates or failure modes and employ sufficient systematic measures such as fail-safe systems so that the equipment incorporating the PRODUCT does not cause any accident or other consequential damage due to operation of the PRODUCT.

7. This manual and the PRODUCT are copyrighted by Hitachi. Under any circumstances, you may not copy, analyze, reverse engineer, and/or modify, in whole or in part, the PRODUCT, except to the extent expressly provided in the CONTRACTS.

8. You may not use or copy, in whole or in part, the user's manual for the PRODUCT without the prior written consent of Hitachi, except to the extent expressly provided in the CONTRACTS.

9. You may use the PRODUCT on just one (1) computer. You may not transfer, lease or otherwise assign the PRODUCT to any third party or parties, except to the extent expressly provided in the CONTRACTS.

10. Please contact your local Hitachi's sales office for any questions regarding the PRODUCT, any Hitachi semiconductor products or any related products.

Table of Contents

1 Overview

1.1 Introduction

This document defines the operation of an Operating System V2.1 (hereafter referred to as OS) which conforms to the OSEK/VDX (hereafter referred to as OSEK) open standard for operating systems, Specification Version 2.0 revision 1. **This document is described on the assumption that OSEK specification is understood.**
Read this document carefully and understand the contents of the following documents before using the OS.

- "OSEK/VDX Operating System", Version 2.0 revision 1 (OSEK/VDX steering committee)
- Release Notes of this product
- SuperH RISC engine C/C++ Compiler Package Manual
- Programming Manual and Hardware Manual of the target SH microprocessor

1.2 Features

- The OS is configured and scaled statically. The number of system objects (see Section 3.3), such as tasks and resources required is statically specified by the user.

- The OS supports portability of application by providing a standardised application program interface that is defined according to the ANSI C standard.

- The OS provides the following features for the real-time, multi-tasking operation of applications.

– **Task Management**
– **Scheduling Policies**
– **Interrupt Management**
– **Resource Management**
– **Event Management**
– **Counter and Alarm Management**
– **Error Handling**

2 OS Application Building

The following files are required for building an OS application.

- **Kernel library.** The kernel library is a set of object files implementing the functionality of the OS. The core functionality (task management, scheduler, etc.) is always included, but non-essential features (resource management, event management, etc.) are automatically combined by the linkage editor.

- **OS configuration files.** These files register information about system objects such as tasks and resources. The files are automatically generated by the OS Configurator.

- **Application program files.** The user application code is written in C language or assembly language.

Figure 2.1 shows the procedure for building an OS application.

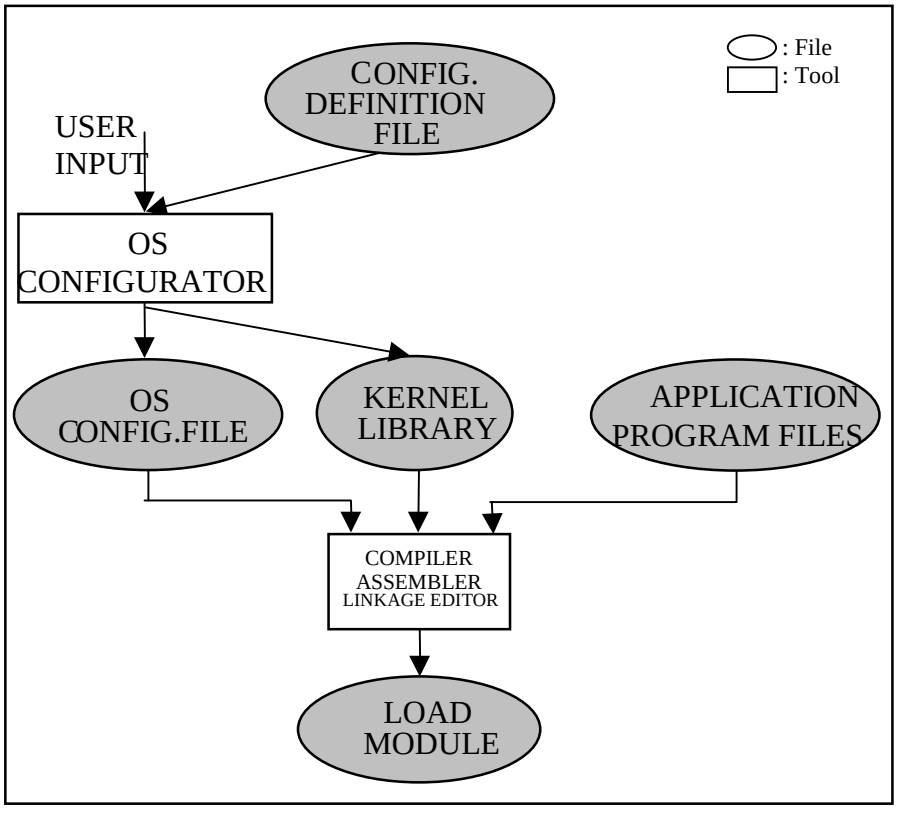

Figure 2.1 OS Application Building Flow

The configurator generates the OS configuration files and the kernel library files determined by the user input or configuration definition file. Along with the application program files, these files are compiled, assembled and linked to generate the load module. There are two formats for the configuration definition file: an OIL format which describes the application using the OSEK OIL format, and an OCF format which describes the application using an internal format.

The user is referred to the help file of OS Configurator for information on how to generate configuration files.

3 Operating System Function

3.1 Processing Levels

There are three types of processes:

- Interrupt service routines (hereafter referred to as ISRs)
- OS
- Tasks

The highest processing priority is assigned to the interrupt level, where ISRs are executed. This interrupt level also includes cases where the interrupt-mask level within a task is non-zero. This method of specification is an original feature of this system. The processing level of the operating system has a priority immediately below the interrupt level. The lowest is the task level on which the application is executed. This is illustrated in Figure 3.1 .

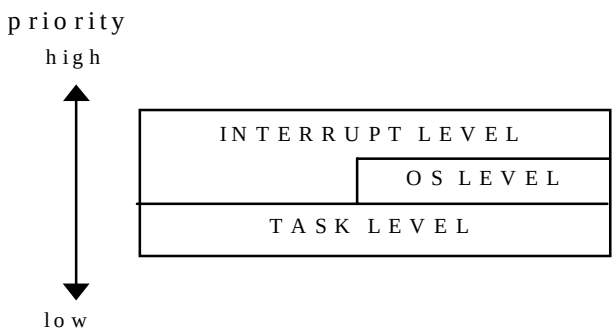

Figure 3.1 OSEK Processing Levels

3.2 Features

The OS provides the following features for use by the application.

- **Scheduling Kernel.** Provides a mechanism whereby tasks are allocated under the control of the CPU.

- **Task Management.** Provides system services for controlling *tasks*.

- **Interrupt Management.** Provides system services for controlling *interrupts*.

- **Resource Management.** Provides system services for controlling synchronisation of access to shared *resources* between tasks.

- **Event Management.** Provides system services for controlling *events*.

- **Counter Management.** Provides system services for controlling *counters*.

- **Alarm Management.** Provides system services for controlling *alarms*.

- **Error Handlers and Hook Routines.** Provides a mechanism for error handling and controlling hook routines called to specific timing.

3.3 Application Modes

Application modes are designed to allow the OS to start in different modes. This feature is intended for modes of operation that are totally mutually exclusive. For example, application modes may be used to distinguish between normal running mode, and service or maintenance mode.

Each of the following system objects can be selected whether it is valid or not in each application mode.

- **Task**
- **Resource**
- **Alarm**
- **Counter**
- **ISR**

Additionally, if a system object depends on another one, the user must ensure that both system objects are valid in all application modes that have been selected. For example, when a task uses a resource in a certain application mode, the resource should also be valid in the application mode. Note that objects which are defined in the application, but not used, will still be subject to object validation or error checking at the time of configuration file generation.
Once the operating system has started, it is not allowed to change the application mode.

3.4 Conformance Classes

Various requirements of the application software for the system and various capabilities of a specific system demand different features of the operating system. The OS provides a set of “conformance classes” which offer implementations with varying functionality.

Conformance classes are determined by the following attributes:

- number of times of queuing of task activation (multiple requesting)
- type of task (Basic or Extended)
- number of tasks per priority

The kind of conformance class is shown below.

- **BCC1** (only basic tasks, limited to one request per task and one task per priority, while all tasks have different priorities)

- **BCC2** (more than one task per priority and multiple requesting of task activation are added to BCC1)

- **ECC1** (extended tasks are added to BCC1)

- **ECC2** (extended tasks are added to BCC2)

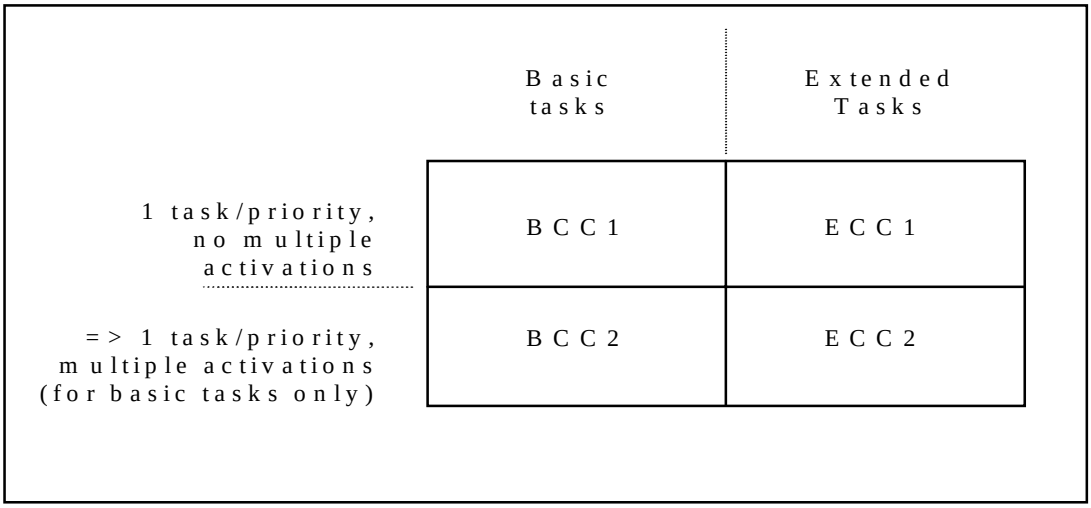

Figure 3.2 OS Conformance Classes

3.5 Maximum Parameters

Table 3.1 shows the maximum parameters for the OS.

Table 3.1 OS Maximum Parameters

Item	**Maximum number**			
	BCC1	BCC2	ECC1	ECC2
Number of tasks	1023			
Number of active tasks	1023 (max. 128 basic tasks)			
Number of priorities	128			
Number of tasks per priority	1	1023	1	1023
Upper limit for number of task activations (must be “1” for extended tasks)	1	127	1	127
Number of events per task	8			
Number of alarm objects	127			
Number of counter objects	127			
Number of resources*	1	127		
Number of application modes	8			
Number of interrupt service routines	256			

Note: Includes the RES_SCHEDULER resource (resource ID = 0) provided by OS.

4 Task Management

4.1 Task Concept

Complex control software can conveniently be subdivided into parts executed according to their real-time requirements. These parts can be implemented by means of tasks. The OS uses a task scheduler to provide concurrent and asynchronous execution of tasks.

A task can either be activated at OS start-up, or by other tasks. The task may then terminate, or run in an endless loop. Once terminated, the task will be re-started when activated by another task, or through an alarm expiry.

Two different task types are provided by the OS:

- Basic tasks
- Extended tasks

Basic tasks only release the processor, if

- They are being terminated,
- The OS is executing higher-priority tasks, or
- Interrupts occur which cause the processor to switch to an ISR.

In addition to the above, Extended tasks can release processor by waiting for an event. More than one Extended tasks cannot be activated at the same time.

4.2 Task State

4.2.1 Introduction

A task must be able to change between several states, as the processor can only execute one task at any time. Several tasks may be competing for the processor at the same time. The following sections explain the task states for basic and extended tasks.

4.2.2 Basic Tasks

Basic tasks have three task states:

- ***running***
  The processor 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 can be activated.

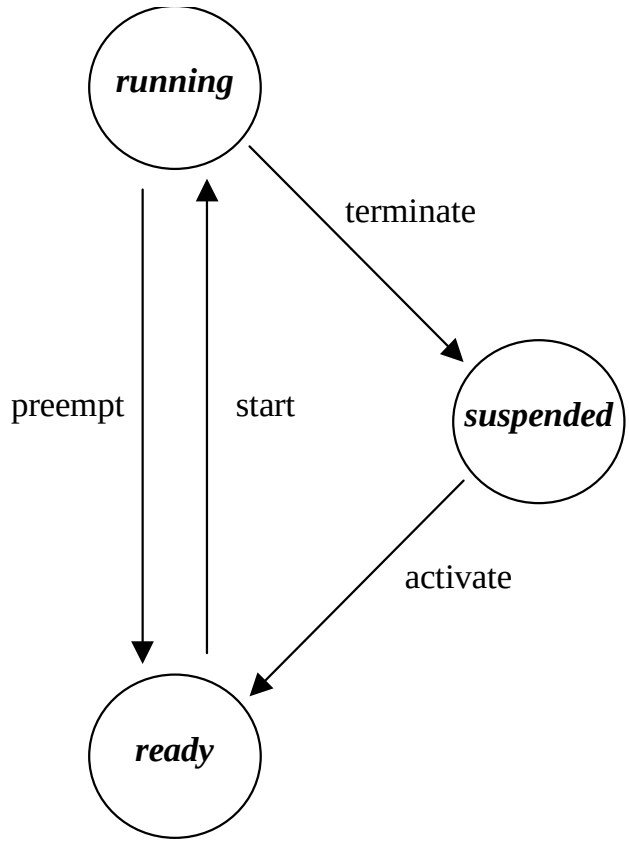

Figure 4.1 Basic Task State

Table 4.1 States and Transitions for Basic Tasks

Transition	Former state	New state	Description
activate	suspended	ready[1]	A new task is entered into the *ready* queue by a system service. The operating system ensures that the execution of the task will start with the first instruction.
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.

4.2.3 Extended Tasks

Extended tasks have four task states:

- ***running***
  The processor 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 other states can be adopted simultaneously be several tasks.

- ***ready***
  All functional prerequisites for a transition into the *running* state exist, and the task only waits for allocation for the processor. The scheduler decides which *ready* task is executed next.

- ***waiting***
  A task cannot continue execution because it has to *wait* for at least one event.

- ***suspended***
  In the *suspended* state the task is passive and can be activated.

[1] Activation of a task will not immediately change the state of the task in the case of multiple requesting. If the task is not *suspended*, the activation will only be recorded and performed later.

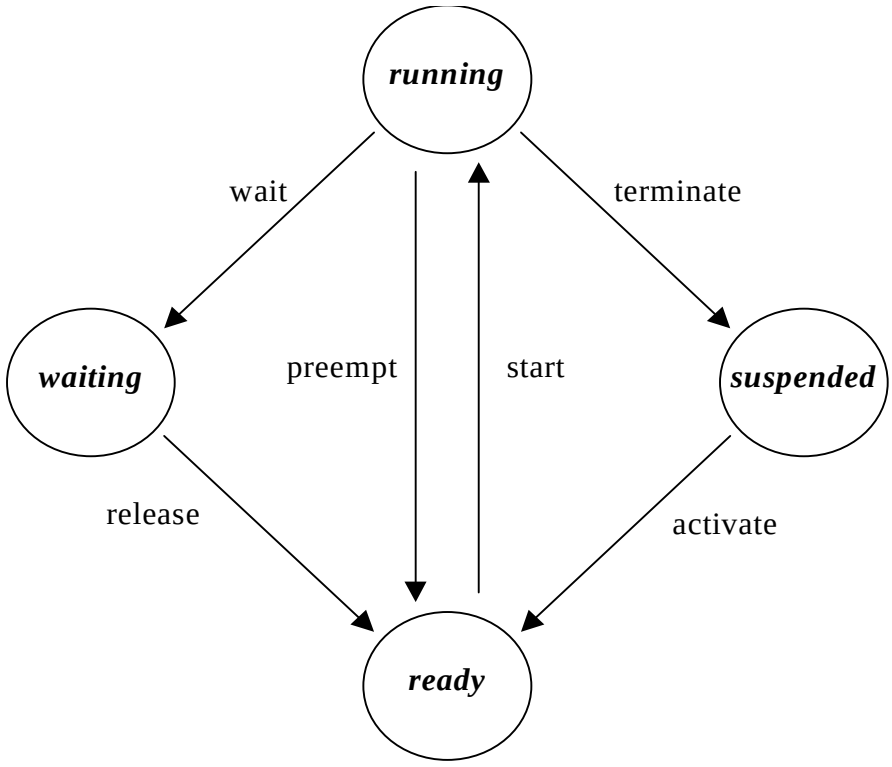

Figure 4.2 Extended Task State

Table 4.2 States and Transitions for Extended Tasks

Transition	**Former state**	**New state**	**Description**
activate	suspended	ready	A new task is entered into the *ready* queue by a system service. The operating system ensures that the execution of the task will start with the first instruction.
start	ready	running	A *ready* task selected by the scheduler is executed.
wait	running	waiting	The *running* task requires an event. It causes a transition into the *waiting* state by using a system service.
release	waiting	ready	At least one event has 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.

4.3 Comparison of the Task Types

Basic tasks have no *waiting* state, and thus only comprise synchronisation points at the beginning and the end of the task. Parts of the application with internal synchronisation points have to be implemented to synchronise more than one basic task.

Extended tasks can synchronise by using an event.

4.4 Task Activation and Termination

Task activation is performed using automatic activation at the time of OS initialisation, or by calling system services.

Depending on the conformance class, a basic task can be activated once or multiple times. The maximum number of multiple requests in parallel is defined at the time of system generation. If the maximum number of multiple requests has not been reached, the request of the basic task is queued in a First-In-First-Out (FIFO) queue per priority to preserve activation order.

A task can only terminate itself. The OS also provides a system service that can activate a task upon termination of the current task. When the task is terminated, be sure to issue the TerminateTask or ChainTask system service.

4.5 Task Priority

Each task has a priority, which is a number from 0 to 127; priority level 0 is the lowest and priority level 127 is the highest. This priority is assigned statically at the time of system generation, and is used by the scheduler to determine the next task to execute. Dynamic priority management scheme is not supported. However, in priority resource management function, the OS can treat a task with a defined higher priority. Please refer to Section 7.2 for more information.

4.6 Task Preemptability

Each task has the option of being **preemptive** or **non-preemptive**. A task that permits preemption of the CPU is a preemptive task. A task that does not permit preemption is a non-preemptive task.

If a task is preemptive, then if a higher priority task becomes *ready* to execute whilst the preemptive task is *running*, the scheduler will pass control to that task.

If a task is non-preemptive, then if a higher priority task becomes *ready* to execute whilst the preemptive task is *running*, task switching does not occurred. The scheduler

only passes control to that task if the current task is terminated, the current task explicitly calls the scheduler, or the current task enters the *waiting* state.

4.7 Task Stacks

Stacks are allocated to tasks depending on their types. The stack allocation method for each task type is described below.

- **Basic tasks**
  All basic tasks share a single stack. If a basic task is preempted by another basic task, the next task to execute will use the current stack pointer value of the preempted task. There will be no conflict with task stack overwriting, as all basic tasks of higher priority will execute before the preempted task will execute again.

  > *Note*: The basic task stack size depends on a function of the number of priorities. If the application uses less priorities, the basic task stack size will also be decreased.

- **Extended tasks**
  Each extended task executes in its own stack. They cannot share stacks as it is possible for them to enter the WAITING state (this is not possible with basic tasks).

4.8 Task System Services

Table 4.3 shows the system services supplied by the OS to manipulate tasks.

Table 4.3 Task System Services

System Service	Function
ActivateTask	Activates a task
TerminateTask	Terminates the current task
ChainTask	Activates a task and terminate the current task
Schedule	Calls the scheduler to switch to a higher priority task (if any exist)
GetTaskID	Gets the task ID of the current task
GetTaskState	Gets the task state of a task

5 Scheduler

5.1 Introduction

The OS provides task scheduling based on task priority and task preemption. This is in contrast to other operating systems that use time-sliced algorithms (round-robin scheduling, for example). The OS uses a set of ready queues, one for each task priority which exists in the application to keep track of all tasks which are in the *ready* and *running* states. Each ready queue is implemented as a First In First Out (FIFO) queue to preserve activation order. Figure 5.1 shows an example of using FIFO queues for each priority level.

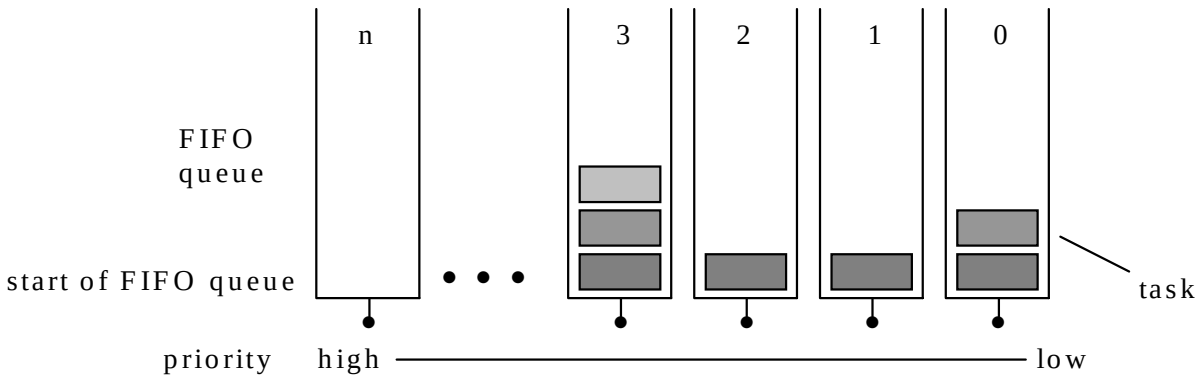

Figure 5.1 Ready Queues

The scheduler performs the following steps to determine the next task to be processed.

- From the set of tasks in the *ready* state, the scheduler determines the set of tasks with the highest priority.
- Within the set of tasks in the *ready* state and of highest priority, the scheduler finds the oldest task.

When there are no tasks in the *ready* queue, the OS enters IDLE mode.

A task, which is released from the waiting state, is treated as the newest task in the ready queue of its priority.

The OSEK OS provides three scheduling algorithms.

- Non-Preemptive scheduling
- Full Preemptive scheduling
- Mixed Preemptive scheduling

These are explained in the following sections.

5.2 Non-Preemptive Scheduling

If all tasks are non-preemptive, then the scheduling policy selected is non-preemptive scheduling. A non-preemptive task can block the execution of a higher-priority task which is *ready* to execute.

In the case of a non-preemptive task, rescheduling will take place exactly in the following cases:

- Successful termination of a task
- Explicit call of the scheduler (Schedule system service issue)
- A transition into the waiting state takes place

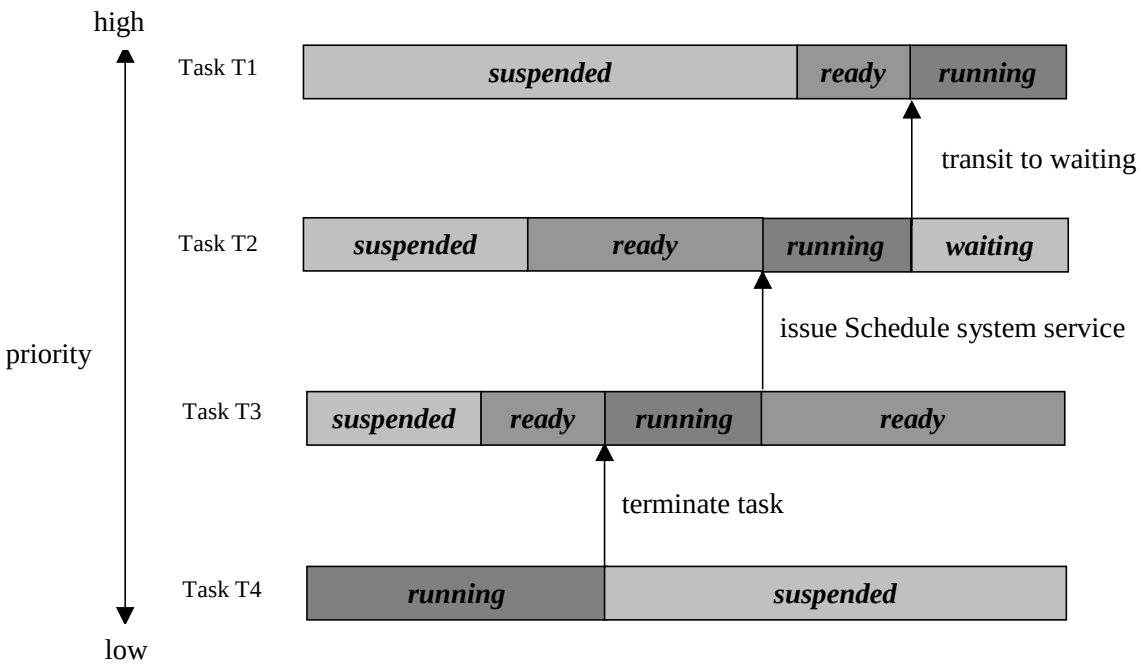

Figure 5.2 Non-Preemptive Scheduling

5.3 Full Preemptive Scheduling

Full preemptive scheduling is adopted when all tasks are preemptive. As soon as a higher priority task is *ready*, the current task will be swapped out. With full preemptive scheduling, the latency time is independent of the run time of lower priority tasks. As each task can be rescheduled at any location, access to data, which are used jointly with other tasks, must be synchronised.

In Figure 5.3, task T2 with the lower priority does not delay the scheduling of task T1 with higher priority.

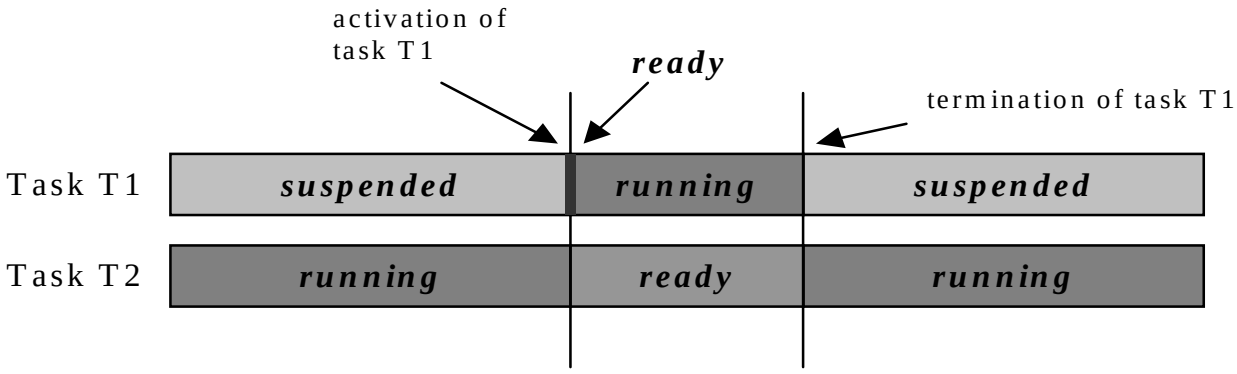

Figure 5.3 Full-Preemptive Scheduling

If a task fragment must not be preempted, this can be achieved by blocking the scheduler temporarily via the system service *GetResource*.

Summarised, rescheduling could be performed in all the following cases:

- successful termination of a task
- a transition into the *waiting* state takes place
- activating of a higher priority task
- setting an event to a *waiting* task
- alarm expiry, when higher priority task activation or event setting is defined for this alarm
- releasing a resource
- return from an ISR

5.4 Mixed Preemptive Scheduling

If both preemptive and non-preemptive tasks are used in the same system, then "mixed preemptive" scheduling is used. If a preemptive task is *running* then full preemptive scheduling is employed; and non-preemptive scheduling is used if a non-preemptive task is *running*.

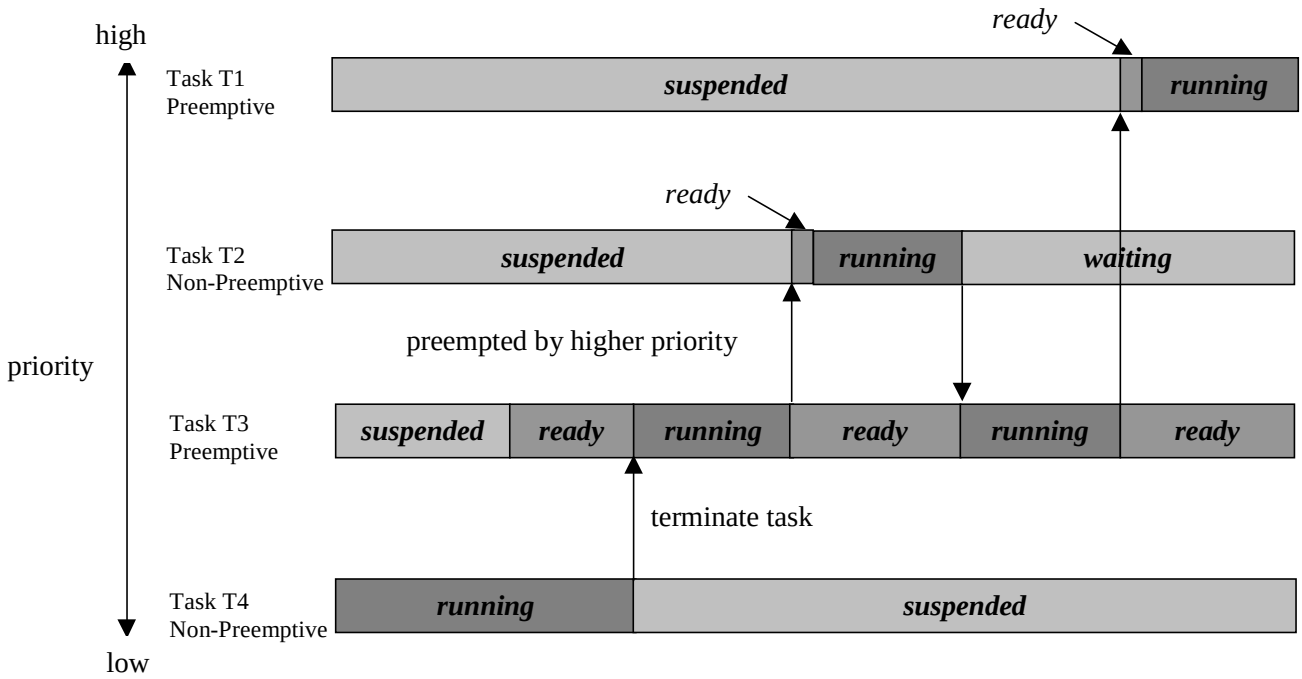

Figure 5.4 Mixed-Preemptive Scheduling

5.5 Interrupt Mask Level and Task Preemption

Preemption of task is independent of the interrupt mask level (i bit of CPU status register). When the interrupt-mask level is non-zero, task switching does not take place. When the interrupt-mask level is zero, preemption of that task by another task is possible.

6 Interrupt Management

6.1 Interrupt Categories

The functions for processing an interrupt are divided into three categories:

– *Interrupt Category 1*

The ISR for an interrupt of category 1 does not use system services. After the ISR has finished, processing continues exactly at the instruction where the interrupt has occurred. Category 1 interrupts do not execute under control of the OS. Therefore, if an interrupt occurs, program execution will branch to ISR immediately. Interrupts of this category are the fastest.

– *Interrupt Category 2*

The OS intervenes at the time of interrupt occurence. If an interrupt occurs, the processing level is switched to the interrupt processing level and the stack frame is changed. This mechanism is called the interrupt preamble. After it finishes processing, control then passes to the interrupt handler. Within the ISR, use of system services is restricted according to “Appendix A.3 System Service Calls”.

– *Interrupt Category 3*

These interrupts are same as interrupts of category 2, and they have the same structure and operate in the same way as those in category 2. They are used for compatibility with an earlier version of the OS. Although the EnterISR and LeaveISR system service are available, they are not processed or executed, so there is no point in calling these routines. For further information, see the description under interrupt category 2.

Note: This mode of operation is an original feature of this system.

Figure 6.1 shows the flow of interrupt category 1.

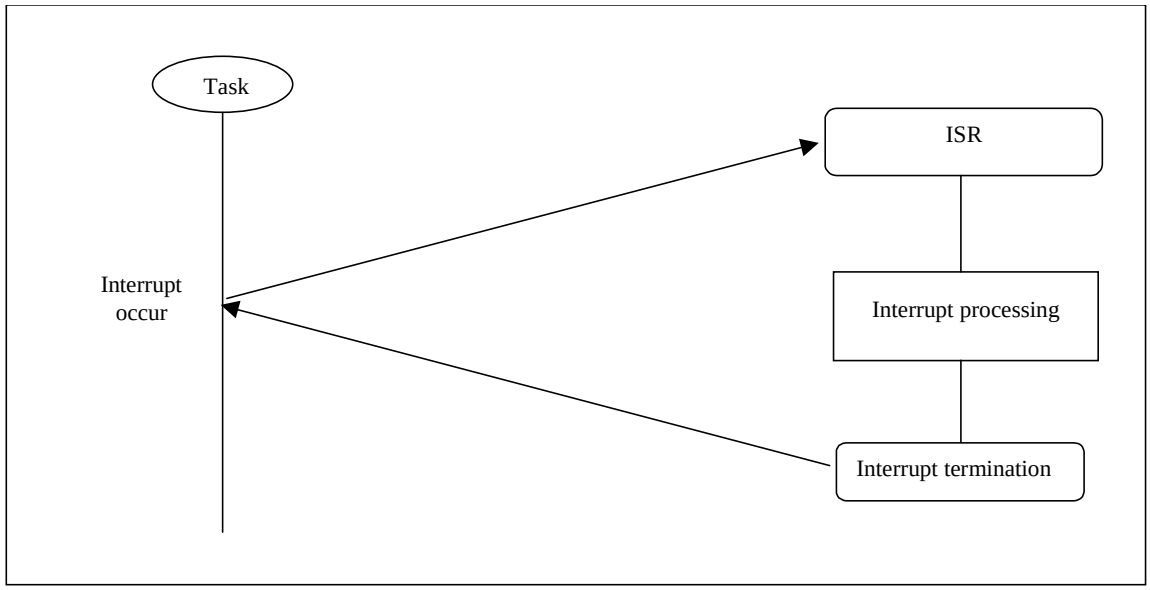

Figure 6.1 Flow of Interrupt Category 1

Figure 6.2 shows the flow of interrupt category 2.

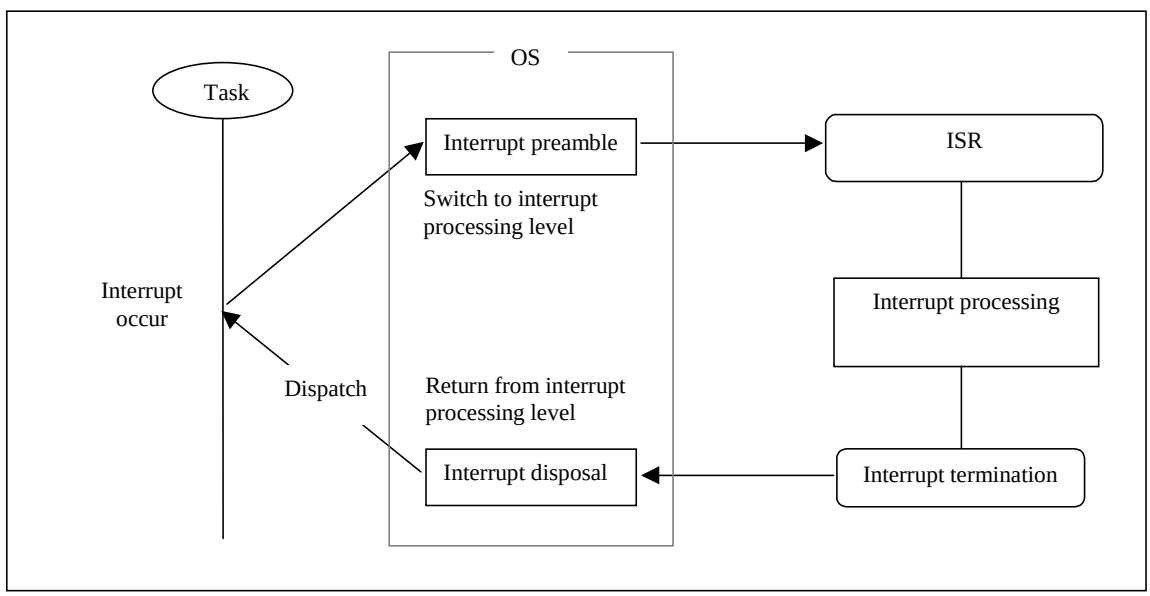

Figure 6.2 Flow of Interrupt Category 2

Figure 6.3 shows the contents of description of ISR for each category.

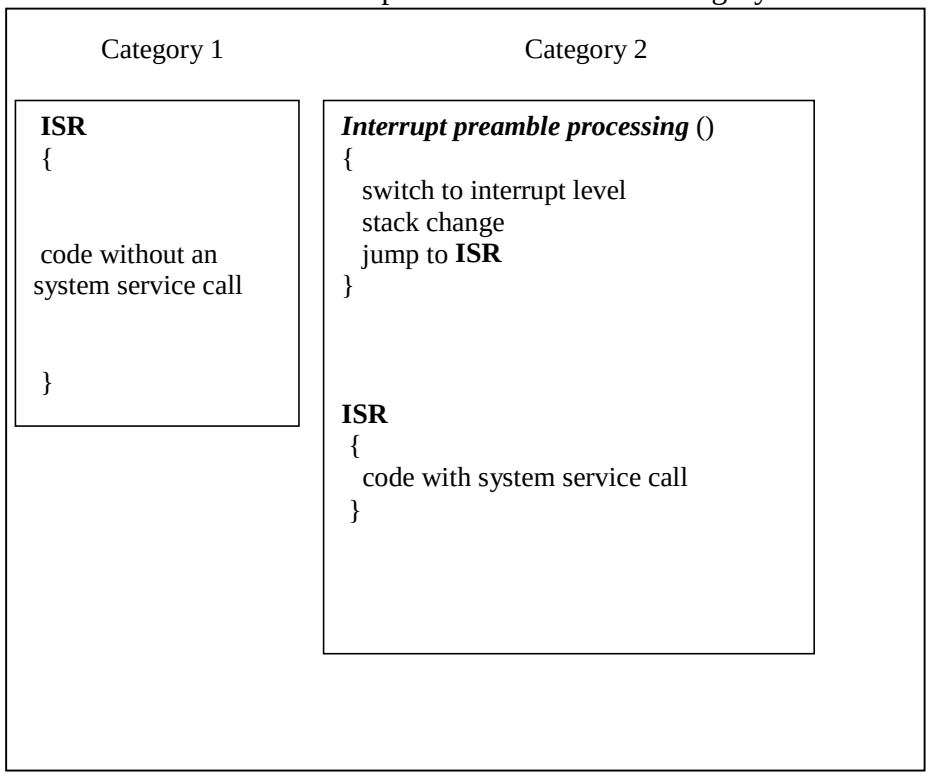

Figure 6.3 ISR Description in Each Category

Inside the ISR, no rescheduling may take place. Rescheduling may only take place on termination of the ISR of category 2 or 3 if a preemptive task has been interrupted, the scheduler has not been locked, and no other interrupt has been active.

6.2 Interrupt Control

Two system services *EnableInterrupt* and *DisableInterrupt* are provided to manipulate the servicing of interrupts. When the interrupt-mask level is selected as the interrupt control method, i.e., "Use mask level" is selected on the OS page of the OS configurator, these system services modify the interrupt mask bit (i) in the status register (SR), thus providing interrupt enabling and disabling based on interrupt priority level. If interrupts are disabled (other than zero), tasks are not switched. In this case, all interrupts are enabled by *EnableInterrupt* and preempted (zero).

When the interrupt source is selected as the interrupt control method, i.e., "Use source" is selected in the OS page of the OS configurator, these system services make the interrupt valid or invalid by changing the CPU's interrupt-request enable bit, or the values in the interrupt priority-level setting register (IPR).

6.2.1 Interrupt Mask Level Method

- *EnableInterrupt*(<**Descriptor**>) enables interrupts with priority level <Descriptor> and higher. The values of <Descriptor> range from H'00000000 to

H'000000f0 that are multiples of 16. When H'00000000 is specified, the OS performs rescheduling.

- *DisableInterrupt*(<**Descriptor**>) disables interrupts with priority level <Descriptor> and lower. The values of <Descriptor> range from H'00000000 to H'000000f0 that are multiples of 16. All interrupts are enabled when H'00000000 is specified. The interrupt mask bit of SR before the interrupt has been disabled is returned as a return value.

Note: These system services do not comply with the OSEK Operating System specification Version 2.0 revision 1. Processing is always performed and E_OK returns. The error code **E_OS_NOFUNC** will not be returned.

6.2.2 Interrupt Source Method

- *EnableInterrupt*(<**Descriptor**>) sets the interrupt-request enable bit to enable the interrupt source that corresponds to the ISR ID specified in <Descriptor> or makes the value in the interrupt priority-level setting register (IPR) valid.

- *DisableInterrupt*(<**Descriptor**>) sets the interrupt-request enable bit to disable the interrupt source that corresponds to the ISR ID specified in <Descriptor> or makes the value in the interrupt priority-level setting register (IPR) valid.

<Descriptor> (ISR ID) is specified by adding "_ID" to the end of the ISR name. The interrupt source for each interrupt is set in the configurator. For the values to be set to select interrupt sources that are provided as standard, refer to section 12.3.4. It is also possible to select an interrupt source defined by the user (CUSTOM0 to CUSTOM31). In this case, interrupt control is by using the user-defined source function (refer to section 12.3.5).

Note: These system services do not comply with Version 2.0, revision 1 of the OSEK Operating System Specification.

The interrupt mask level must not be lowered at the time of interrupt occurring level during execution of the ISR.
A task can change an interrupt mask level. When the interrupt-mask level is non-zero during the execution of a task, the task may increase their mask level during task execution to exceed the OS mask level, but it will be illegal to call system services.

The interrupt mask level of category 1 must have at least the same, or a higher interrupt level, as the OS mask level[2]. This is because interrupts of category 1 are outside the control of the operating system.

Figure 6.4 shows the interrupt level.

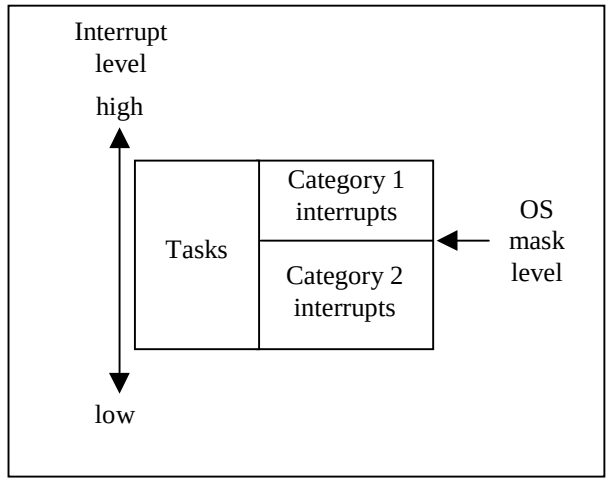

Figure 6.4 Interrupt Level

[2] The interrupt mask level of the OS is set to the highest interrupt level of any interrupt of category 2 or 3 by the configurator automatically.

6.3 Interrupt Source Classification

Interrupt sources are classified into two classes: NMI or *other* interrupts (the external interrupt and the internal peripheral module interrupt). The software trap is handled as an exception but not as an interrupt. The following sections explain the interrupt source classifications.

6.3.1 Non-Maskable Interrupt

The non-maskable interrupt (**NMI**) cannot be masked and is always accepted. It can only be of category 1.

6.3.2 Other Interrupts

Other interrupt causes are the internal peripheral module interrupt and the external interrupt other than the above.

6.4 Interrupt Stacks

Stacks are allocated to interrupts as described below. The stack size used by each interrupt is specified on the configurator.

6.4.1 Interrupts of Category 2

All ISRs of category 2 share the stacks in the same area as the basic task stack. The stack size of ISR specified on the configurator is reserved on the basic task stack. When an interrupt of this category is serviced, the OS swaps in the stack for the interrupt. If a nested interrupt of category 2 occurs, the OS will not change the stack.

6.4.2 Interrupts of Category 1

Each interrupt of category 1 will share a stack with those of the same interrupt mask level.

> *Note:* The OS has no control over the handling of interrupts of category 1, for performance reasons, thus the shared stack used by other interrupt categories cannot be used.

6.4.3 NMI Interrupts

ISR for NMI will use the same stack as the interrupted program.

6.5 Exceptions

The following exceptions in Table 6.1 are classified as fatal errors by the operating system, and thus are handled by the operating system.

Table 6.1 Exceptions

Exception	**Handler Name**
Unhandled exception	_OSEKUnhandledExceptionError
General illegal instruction	_OSEKIllegalInstructionError
Slot illegal instruction	_OSEKIllegalSlotInstructionError
CPU address error	_OSEKIllegalCPUAddressError

These exceptions can be inserted into any vector table entry that does not have a vector. If these exceptions occur, OS will perform shutdown processing. Exceptions other than the above can also be inserted into the vector table entry. In this case, the user must prepare the handler for an exception. This exception will use the same stack as the interrupted program.

6.6 Interrupt System Services

Table 6.2 shows the system services supplied by the OS to manipulate interrupts.

Table 6.2 Interrupt System Services

System Service	Function
EnableInterrupt or EnableInterruptMask	Enables interrupts (Interrupt Mask Level Method)
EnableInterrupt or EnableInterruptSource	Enables interrupts (Interrupt Source Method)
DisableInterrupt or DisableInterruptMask	Disables interrupts (Interrupt Mask Level Method)
DisableInterrupt or DisableInterruptSource	Disables interrupts (Interrupt Source Method)
GetInterruptDescriptor or GetInterruptDescriptorMask	Get the state of interrupts (Interrupt Mask Level Method)
GetInterruptDescriptor or GetInterruptDescriptorSource	Get the state of interrupts (Interrupt Source Method)

7 Resource Management

7.1 Introduction

Resource management is used to co-ordinate concurrent accesses of several preemptive tasks of different priorities to shared resources.

The priority of task becomes high temporarily during resource occupation. Two or more tasks cannot occupy the same resource at the same time.

7.2 Priority Resource Management

When a task requests a resource, the priority of the task is raised to the priority ceiling of the corresponding resource. This method excludes the possibility of two or more tasks occupying the same resource simultaneously. The resource priority is determined at system generation, and is calculated to be

1. Identical to or higher than the highest task priority with access to the resource, and
2. Lower than all other tasks with higher priority than the highest priority task accessing the resource.

Figure 7.1 illustrates the priority assignment. Task T0 has the highest priority, and task T4 the lowest. Tasks T1 and T4 share the same resource. When these tasks get the resource, the priority is raised to a higher value than or same value as task T1 and a lower value than or same value as task T0. Tasks T0, T2, and T3 do not share the resource.

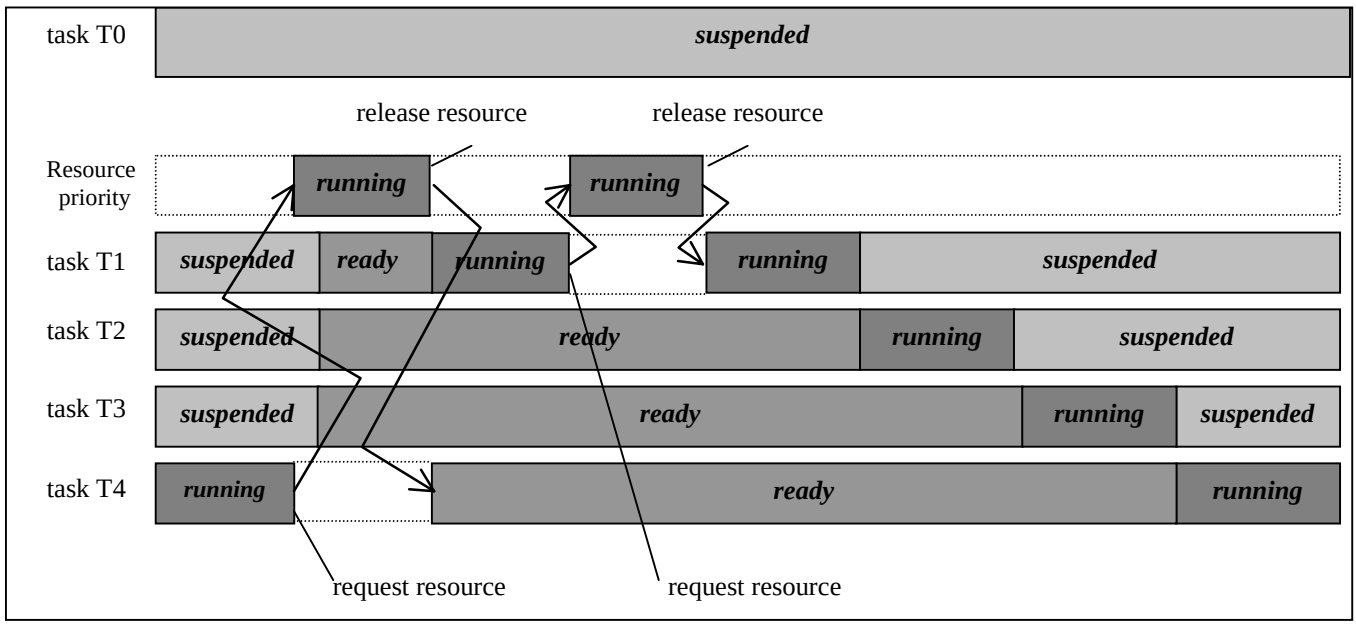

Figure 7.1 Resource Priority Assignment

7.3 Nested Resource Occupation

Nested resources must be occupied and released using the Last-In-First-Out (LIFO) principle.
When the inner resource priority that is lower than the current task priority is obtained, by OSEK specification, the task priority is lowered and E_OK returns. However, this version of the OS does not change the task priority. This provides a secure mechanism whereby deadlocks and priority inversion will never occur. An error will be returned by the system service in extended error status.

Note: This behavior deviates from the OSEK OS Specification.

7.4 Resource Occupation at Task Termination

Resources must be released before the task is terminated. In extended error status, the OS will return an error if resources are still occupied. In normal error status, the system operation cannot be guaranteed as there will be no error reporting.

7.5 Scheduler as a Resource

If a task is to prevent itself from being preempted during the execution of a critical section it may deactivate the scheduler. A standard resource with the defined name **RES_SCHEDULER** provided by the OS can be used by the task. The following points must be noted.

1. When a task requests the **RES_SCHEDULER** resource, the task priority is not changed.

2. The **RES_SCHEDULER** resource release timing is not influenced by other resources. Before terminating task, it can always release resource.

3. When a task requests the **RES_SCHEDULER** resource, the task is not changed even if the *Schedule* system service is issued.

7.6 Resource Priority Ceiling For ECC1 Conformance

In ECC1 conformance class, the configurator will automatically shift task priorities in order to accommodate resource priorities. **That is, the priority specified by the user may be changed.**
The priority of shifted task can be checked on the configurator. This function makes use of the feature of ECC1 conformance class, which has restriction of one task or one resource and single activation per one priority, and high performance is therefore drawn out.

7.7 Restrictions when Using Resources

- Neither the *waiting* state or task termination is admissible while a resource is occupied.
- A task does not switch to waiting state by resource acquisition. When resource cannot be obtained since other task is under resource occupancy, resource acquisition needs to be required again.
- The BCC1 conformance class can use only RES_SCHEDULER resource.

7.8 Resource System Services

Table 7.1 shows the system services supplied by the operating system to manipulate resources.

Table 7.1 Resource System Services

System Service	Function
GetResource	Occupies the resource
ReleaseResource	Releases the resource

8 Event Management

8.1 Introduction

Events are objects used by extended tasks. The OS can synchronise execution of extended tasks by means of “events”.

8.2 Event Operation

Several events can be assigned to an extended task. Any task can set an event for an extended task. Only the appropriate extended task is able to clear events and to *wait* for the setting of events. It can *wait* for more than one event.

An extended task in the *waiting* state is released to the *ready* state if at least one event for which the task is *waiting* for has occurred. If a *running* extended task tries to *wait* for an event and this event has already occurred, the task remains in the *running* state.

Events belonging to the extended task are cleared upon task activation.

Figure 8.1 and Figure 8.2 show the synchronisation of extended tasks by event setting.

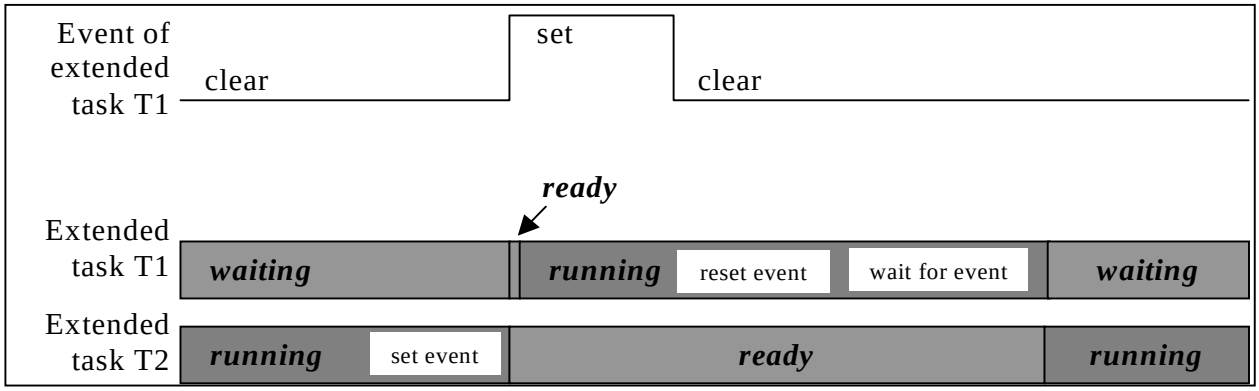

Figure 8.1 Synchronisation of Full Preemptive Extended Tasks

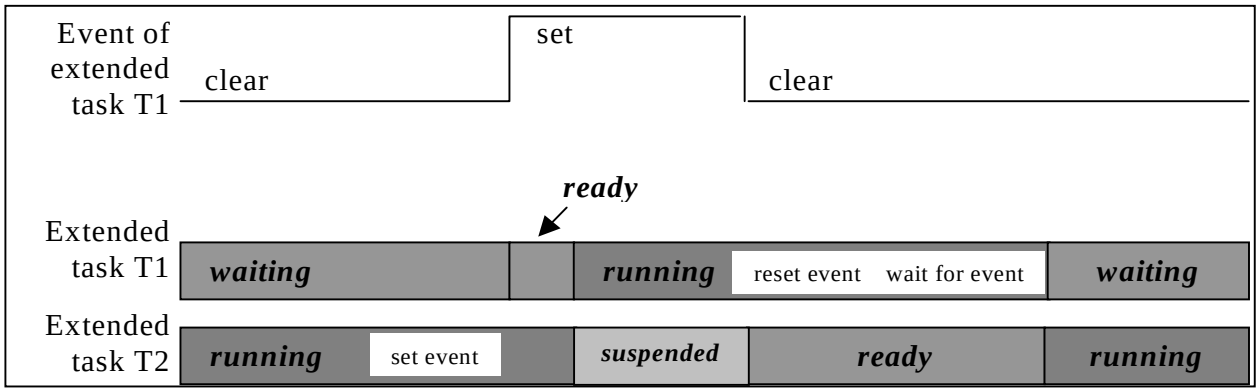

Figure 8.2 Synchronisation of Non-Preemptive Extended Tasks

In above example, extended task T1 has the highest priority. Task T1 *waits* for an event. Task T2 sets this event for T1. Subsequently, T1 is transferred from the *waiting* state into the *ready* state. In the case of full-preemptive scheduling in Figure 8.1, due to the highest priority of T1, this results in a task switch, and T2 is preempted by T1. Thereafter T1 *waits* for this event again, and T2 continues execution. In the case of non-preemptive scheduling in Figure 8.2, Task will not be swapped until the next scheduling point (until T2 enters the suspended state).

8.3 Event IDs

Each extended task may have eight events; such sets of events are represented in 8-bit units. Each bit represents the state of one event; '0' indicates that the event has not occurred or has been cleared, '1' indicates that the event has occurred. The events are given IDs based on the position of each event in the bit-mask, as shown in Figure 8.3.

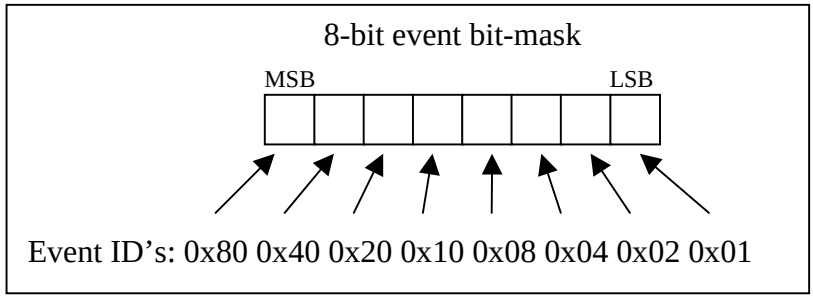

Figure 8.3 Event IDs

The event IDs are used to manipulate the event states; more than one event may be referenced by using the bit-wise OR operator at one time. For example, to clear event IDs 0x01 and 0x80, C*learEvent*(0x01 | 0x80) is executed.

The event ID is specified by the user or is automatically determined by configurator. The event used by each task by configurator must be defined. An event not defined by a task must not be used in that task.

8.4 Event System Services

Table 8.1 shows the system services supplied by the operating system to manipulate events.

Table 8.1 Event System Services

System Service	Function
SetEvent	Sets an event or events to the referenced task
WaitEvent	Makes the task *wait* for an event or events
ClearEvent	Clear an event or events
GetEvent	Returns the current state of the events of the referenced task

9 Alarm and Counter Management

9.1 Introduction

The OS provides services for processing recurring events. It provides a two-stage concept for alarms and counters, as shown in Figure 9.1.

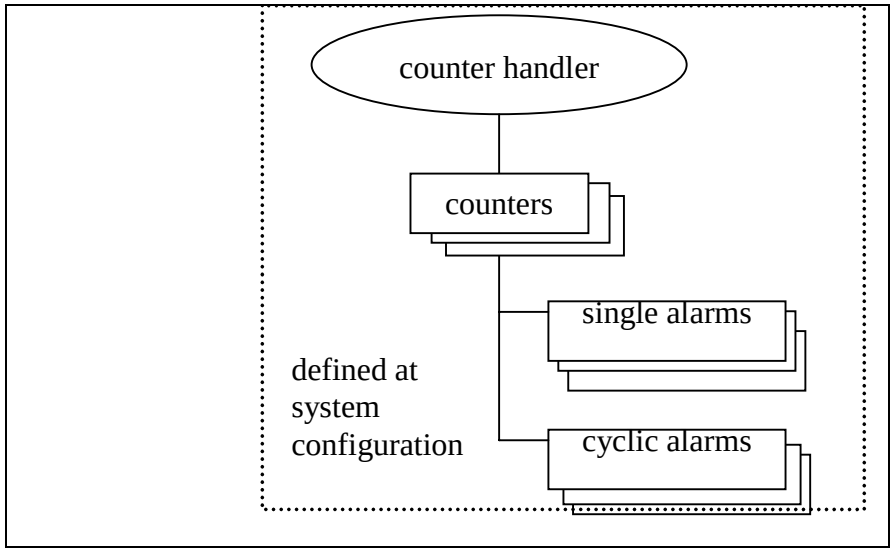

Figure 9.1 Layered Model of Alarm Management

Recurring events, such as timers that provide interrupts at regular intervals, are considered sources for counters. Each counter is represented by a counter value, measured in “ticks”. The count is incremented each time an event attached to that counter occurs by using the IncrementCounter system service. Based on counters, the OS offers alarm mechanisms to the application software. Alarms may be used, for example, to monitor the number of timer interrupts, or a specific angular position. Upon alarm expiry, tasks may be activated, or events set.

9.2 Counters

9.2.1 Counter Handler

Counter handlers are used to increment the counter for dynamic (variant) alarms. The application needs to increment the counter by calling the IncrementCounter system service. There is a timer interrupt service routine, etc. as an example of counter source. Please create these ISR by application.

As an example, an interrupt might be generated by a timer. The interrupt handler for the timer would then use the system service to increment the appropriate counter's count.

9.2.2 System Timer

The OS offers a system timer, called **SYSTEM_TIMER**, for use by the dynamic (variant) alarm.
The OS reserves the on-chip Compare Match Timer Channel 0 for use as the system timer. The ISR for the system timer, supplied by the OS, then has responsibility for incrementing the system timer.

Please refer to Section 11.9 for the ISR name used for the system timer.

9.2.3 Non-Variant Alarm Timer

The OS allows the user to define static (non-variant) alarms. These alarms may only use the non-variant alarm timer, called **NonVariantAlarmTimer**, provided by OS. The OS reserves the on-chip Compare Match Timer Channel 1 for use as the timer. The ISR for non-variant alarm timer, supplied by the OS, then has responsibility for incrementing the timer.

Please refer to Section 11.9 for the ISR name used for the non-variant alarm timer.

9.2.4 Counter Properties

Each counter has the following properties, set at the time of system generation:

- **maxallowedvalue**
- **tickperbase**
- **mincycle**

The following sections explain these properties.

9.2.4.1 maxallowedvalue

The **maxallowedvalue** property is used to determine the range of count values for the counter.

> **Note**: The **maxallowedvalue** count value cannot be reached. When a counter reaches a count value of (**maxallowedvalue** – 1), the next increment will cause the counter to roll over to a count value of 0.

If a counter has a **maxallowedvalue** of 8, the count values will range from 0 to 7, as shown in Figure 9.2.

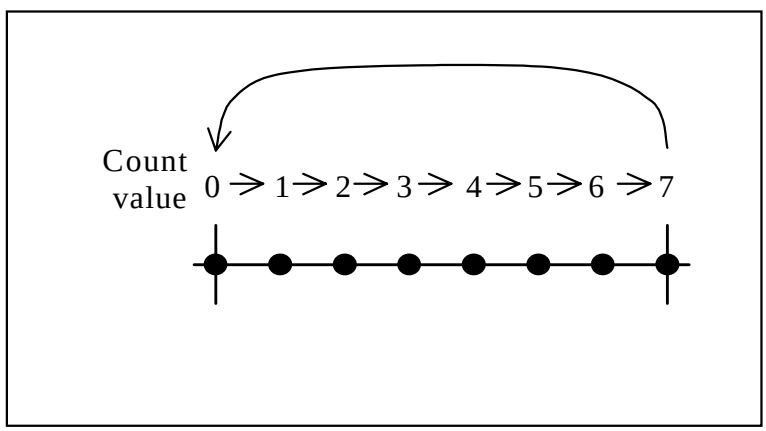

Figure 9.2 Range of Count Values for Counter with 'maxallowedvalue' of 8

When the counter reaches a count value of 7, the next increment will force to counter to roll over to 0. If a cyclic alarm is set with a cycle value of **maxallowedvalue**, the alarm will always expire at the same count value.

The count value is expressed as an unsigned 32 bit integer. Therefore, if the cycle of the counter is 100 microsecond, it will correspond to about 5 days.

9.2.4.2 tickspersbase

The **tickspersbase** property use of this property is user defined; it is not controlled (used) by the OS.

9.2.4.3 mincycle

The **mincycle** value is used when setting the minimum cycle of cyclic alarms. When a cyclic alarm is set, the cycle period is compared with the **mincycle** value to determine if it is within acceptable limits.

9.3 Alarms

9.3.1 Introduction

The OS provides services to start tasks or set events when an alarm expires. An alarm will expire when a predefined counter value is reached. Figure 9.3 shows the different kinds of alarms that can be used.

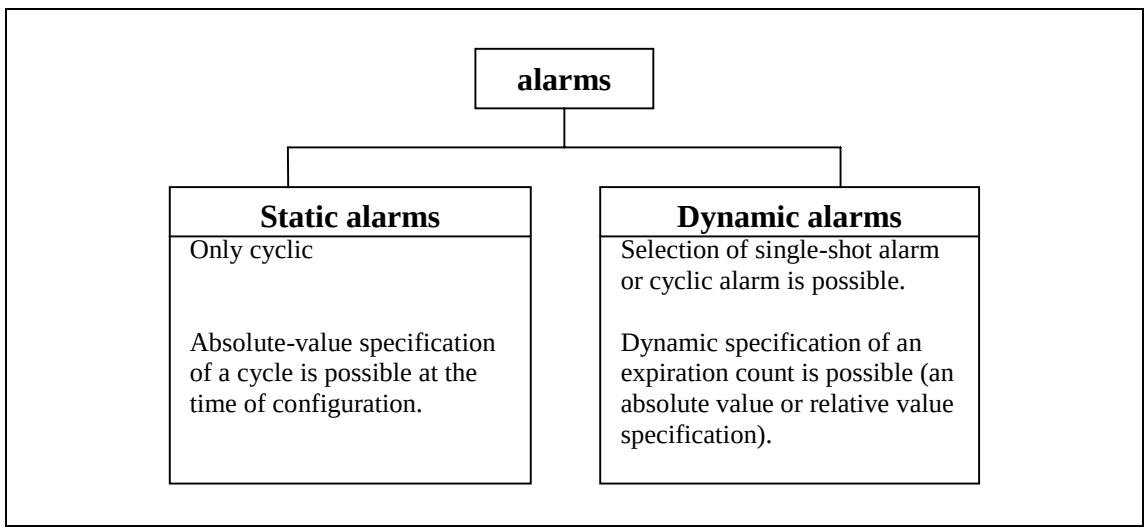

Figure 9.3 Alarm Classification

Alarms are classified into static alarm and dynamic alarm.

Dynamic alarms' parameters may be set during application run-time. The user is allowed to change the count value that an alarm expires at, and whether the alarm is cyclic, through a system service. Dynamic alarms may be connected to any counter. Dynamic alarm is active from **SetAbsAlarm** or **SetRelAlarm** system service to **CancelAlarm** system service.

Dynamic alarms have the following properties that can be set at run-time:

- Absolute count or relative count expiry
- Starting or canceling of the alarm

- Single or cyclic
- A system-timer counter or user-defined counter can be selected for use

Static (non-variant) alarms are used to activate periodic tasks, or to set events periodically. The period is set at the time of system generation, and cannot be altered. Static alarms cannot be cancelled. The attachment of alarms to counters is defined at the time of system generation. Static alarms are automatically started at the time of OS initialisation. Static alarms cannot be suspended.
Please note the following points.

- Static alarms can only be attached to the non-variant alarms timer counter (cannot be changed).
- The user cannot start and cancel static alarms.
- Static alarms can only be cyclic.
- The alarm expiry period is set at system generation time and cannot be altered.
- Static alarms carry a smaller overhead than dynamic alarms

9.3.2 Alarm Parameters

9.3.2.1 Expiry Count

Dynamic alarms may be set to expire at an absolute count value, or a count value relative to the current count value of the corresponding counter. This parameter may be set at run-time by using system services, thus allowing the user to decide whether an alarm should be absolute or relative at run-time.

When set to an absolute count value, the alarm will expire when the counter reaches that value.

When set as a relative count value, the alarm will expire at a count value relative to the counter's current count value.

The following points should be noted:

- If an absolute alarm is set in which the expiry count value is the same as the counter's current count value, the alarm will not expire immediately. Instead, the alarm will expire when the counter rolls over and reaches the expiry count.

- If a relative alarm is set with a relative count of 0, the alarm's expiry count will be set to the current count value of the counter. The alarm will expire when the counter rolls over and reaches the expiry count.

- If an absolute alarm is set with the expiry count value of **maxallowedvalue** for the counter, the expiry count will be set to 0.

- If a relative alarm is set with a relative count of **maxallowedvalue**, the expiry count will be set to the current count value of the counter.

9.3.2.2 Cyclic and Single Alarms

Alarms may be set to be either single alarms, or cyclic alarms at run-time via system services.

Single alarms expire at a particular count, and then stop. Cyclic alarms are restarted upon expiry with a count value relative to the current counter count. A system service is provided for cancelling single and cyclic alarms.

9.4 Example for Using Counter and Alarm

Figure 9.4 shows the example for using counter and alarm.

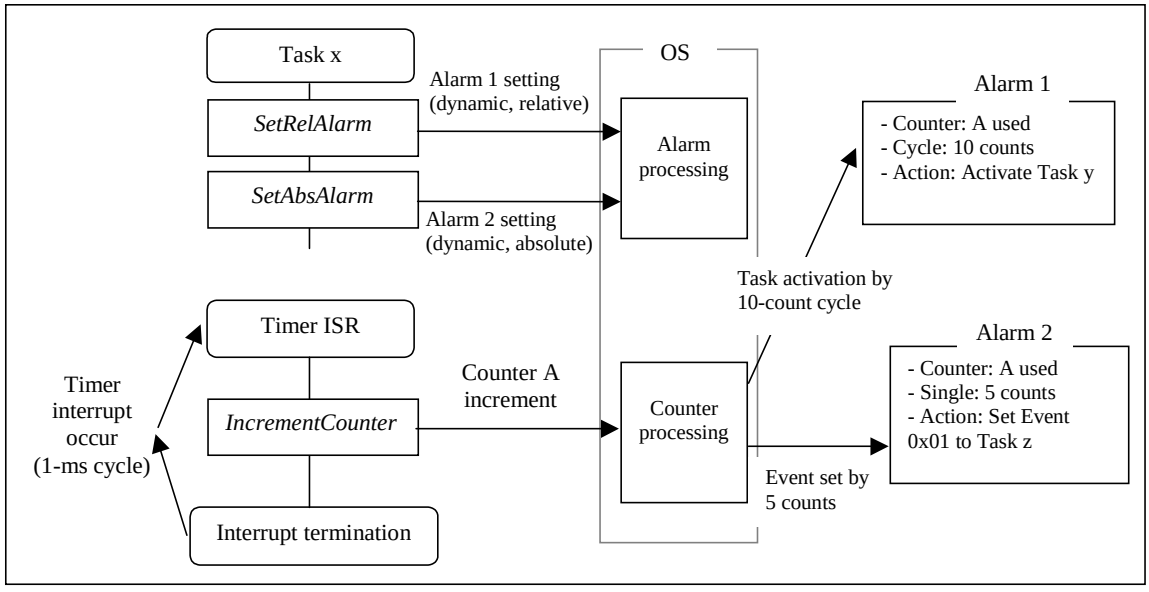

Figure 9.4 Example for Using Counter and Alarm

Alarm 1 and Alarm 2 share Counter A. Alarm 1 has been set up to be a cyclic alarm. Alarm 1 expires every 10 counts (10 ms) and activates Task y. Alarm 2 has been set up to be a single alarm. Alarm 2 expires after 5 counts (5 ms) and sets an event in Task z.

9.5 Counter and Alarm System Services

Table 9.1 shows the system services supplied by the OS to manipulate counters and alarms.

Table 9.1 Counter and Alarm System Services

System Service	Function
InitialiseCounter	Initialises a counter
IncrementCounter	Increments a counter
GetAlarmBase	Returns the attached counter's properties
GetAlarm	Returns the remaining counts to alarm expiry
SetRelAlarm	Sets a relative alarm
SetAbsAlarm	Sets an absolute alarm
CancelAlarm	Cancels an alarm

10 System Control

10.1 System Start-up

Figure 10.1 shows the start-up method for the OS.

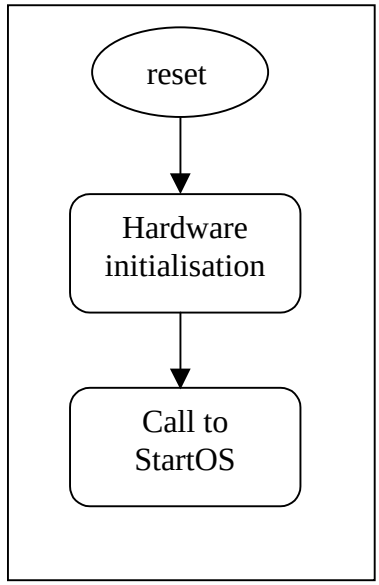

Figure 10.1 OS Start Up

The user has responsibility for initialising hardware if required.

The system service *StartOS* is provided to initialise and start the OS. The application mode must be specified as an argument. If this is not valid, the system service returns. If called successfully, the system service starts the OS. During the execution of *StartOS*, all interrupts except category 1 are masked; all interrupts are accepted when the scheduler is started. The scheduler then decides which task to execute from the tasks which have been defined to be auto-start tasks, and from the tasks activated from the StartupHook routine. The program that issues the *StartOS* system service must reserve the 8-byte stack area.

10.2 System Shutdown

The OS provides a system service, *ShutdownOS*, to provide shutdown facilities. This can either be called by the application, or requested by the OS due to a CPU exception. When *ShutdownOS* is called, the system will shutdown regardless of whether an incorrect error code was passed to the system service. During shutdown, all interrupts are masked. If shutdown processing is completed, it will return to the program that issued the *StartOS* system service. When returned, R15 and SR are recovered but other register values become undefined.

10.3 Hook Routines

The OS provides system-specific hook routines to allow user-defined actions within the OS internal processing. Hook routines may be used for:

- system startup.
  The corresponding hook routine (*StartupHook)* is called after the OS start-up and before the scheduler is called. This is called after automatic activation of selected tasks.

- system shutdown.
  The corresponding hook routine (*ShutdownHook)* is called at system shutdown. It does not necessarily need to return from the hook routine. For example, it is also possible to issue StartOS system service and to start OS again.

- task switching.
  Two hook routines (*PreTaskHook* and *PostTaskHook*) are called on task context switching. *PreTaskHook* is called after the selected task has made the transition to the *running* state, but before task switching occurs. *PostTaskHook* is called before the current task has exited the *running* state.

- error handling.
  The corresponding hook routine (*ErrorHook*) is called if a system service is not called correctly (and returns with an error code other than E_OK).

The following points must be noted:

1. Hook routines in the system are optional. The user has the choice of including some, and excluding others.

2. Hook routines may only use a sub-set of system services. These are shown in Appendix A.3 System Service Calls.

3. Only interrupts of category 1 are accepted when executing hook routines. Therefore, the execution times of these hooks should be as short as possible to reduce interrupt latency times.

4. The user should not lower the interrupt mask level when executing hook routines. **Operation of a system is not guaranteed when an interrupt mask level is lowered.**

10.4 Error Handling

10.4.1 Error Status

The OS can operate in one of two error modes. This is shown below.

Standard Error Status
- Minimal parameter checking
- Faster operation
- Less memory efficient

Extended Error Status
- More thorough parameter checking
- Slower operation
- More memory efficient

The operating system with extended error status is used in the development and debugging of applications; the operating system with standard error status is used in fully debugged systems. Appendix A.1 System Service Return Codes gives a summary of the error codes returned by system services.

10.4.2 Shutdown Errors

Table 10.1 shows the shutdown errors.

Table 10.1 Shutdown Errors

Error	Error Code
Illegal instruction executed	E_OS_SYS_ILLEGAL
Undefined exception	E_OS_SYS_EXCEPTION
Illegal slot instruction executed	E_OS_SYS_ILLEGAL_SLOT
CPU address error	E_OS_SYS_CPU_ADDRESS

If a shutdown error occurs, the *ShutdownOS* system service is executed. However, the error hook is not called in this case.

10.5 ErrorHook Re-Entry

If a system service is incorrectly called from the *ErrorHook* routine, the *ErrorHook* will not be called. In this case, *ErrorHook* routine is not called but processing is continued (an error code returns). The user must be aware that no centralised error handling for system service calls will occur at the *ErrorHook* level. This is the original function.

11 Programming

11.1 Registers

The initial value and the usage of register are shown in Table 11.1.

Table 11.1 Initial Value and Usage of Register

Register	Processing level	Contents
PC	Task	Starting address of task
	ISR	Starting address of ISR
	Hook routine	Starting address of hook routine
SR	Task	0. Can be used freely.
	ISR	It is set to the occurred interrupt level. Do not lower the interrupt level.
	Hook routine	It is set to the OS mask level. Do not lower the interrupt level.
R15	Task	It is set to the stack area for the task. Do not change to another stack area.
	ISR	It is set to the stack area for the ISR. Do not change to another stack area.
	Hook routine	It is set to the stack area for the OS. Do not change to another stack area.
R0-R7, FR0-FR11, FPUL, FPSCR[3]	Task	Unknown. Can be used freely.
	ISR	Unknown. Can be used freely.
	Hook routine	Unknown. Can be used freely.
R8-R14, MACH, MACL, PR, GBR, FR12-FR15	Task	Unknown. Can be used freely.
	ISR	Those are set to the value before interrupt occurring. The values in the registers that are used are saved and restored by code generated by the compiler.
	Hook routine	Unknown. The values in the registers that are used are saved and restored by code generated by the compiler.

The program which issues StartOS system service should specify #pragma noregalloc and not use gbr instrinsic function, #pragma gbr_base and #pragma gbr_base1.

The contents of Table 11.1 may be changed by version update etc. When they differ from the present condition, please design by present-condition specification priority.

11.2 Declaration of OSEK Processes

11.2.1 OS Initiation

Within the application, a program for activating OS is defined according to the following syntax:

[3] FR0-FR15, FPUL and FPSCR registers are only valid for the processor with Floating Point Unit (FPU).

```
#include "sysserv.h"

#pragma noregalloc (function name)
type-function name (argument)
{
   :
    StartOS (application mode ID);
   :
}
```

11.2.2Tasks

Within the application, a task is defined according to the following syntax:

```
#include "osekos.h"

TASK (Task name)
{
   :
   :
   TerminateTask(); or ChainTask (task name);
}
```

11.2.3 ISR

Within the application, an interrupt handler is defined according to the following syntax:

```
#include “osekos.h”
:
:
ISR (ISR name)
{
   :
   :
}
```

The macro definition for TASK and ISR is defined in the file “OSEKtype.h”

11.3 System Configuration Files

The following gives a list of the main header files generated by the configurator.

11.3.1 Header Files

Table 11.2 Header Files

File name	Comment
size.h	Machine word sizes
defsapp.h	OSEK state definitions
sysobjid.h	Gives the ID's of system objects
OSEKtype.h	Defines OSEK abstract type mapping
errcodes.h	Error code
intdecl.h	Interrupt declaration
sysserv.h	OSEK system service declarations
Api_id.h	System service ID
osekos.h	Header file that includes all of the above

A program, task, or ISR for activating OS may include "osekos.h".

11.4 Declaring System Objects Through System Services

The system services in Table 11.3 for system object declaration are not required to be used. The macro to define these system services is in the file "sysserv.h".

Table 11.3 Declaration of System Objects

System Service	Comment
DeclareTask	Declares a task object
DeclareResource	Declares a resource object
DeclareEvent	Declares an event object
DeclareAlarm	Declares an alarm object
EnterISR	Enters an interrupt service routine
LeaveISR	Resumes from an interrupt service routine

11.5 Referring to System Objects

System object ID's are declared in the file "sysobjid.h". This file declares enumerated type for system objects. IS system object IDs are the same as system object names. Note, however, that _ID is added to ISR ID names. They can be referred to by specifying an object name as follows:

```
#include "osekos.h"
        :
        :
TASK(Task2)
{
    ActivateTask(Task1);
```

```
    DisableInterrupt(Isr1_ID):
}
```

11.6 Calling a System Service From an Assembler Routine

System services may be called from assembler routines. In this case the application programmer must branch to the address of each system service by using the JSR instruction.
Follow the rules in Table 11.4. Refer to Table A.6 and C-language header file of configuration files for information on parameter types.

Table 11.4 Argument Convention

Register	Argument Number
R4	First argument
R5	Second argument
R6	Third argument
R7	Fourth argument

The general register R0 is used for the return value.

Registers R0 to R7, PR, FR0 to FR11, FPUL and FPSCR are not guaranteed before and after calling the system service. These registers must be saved before the call, and restored after the call. Do not use R8 to R14, MACH, MACL, GBR, FR12 to FR15, FPUL, and FPSCR for the program that issues the StartOS system service.

11.7 Assembler ISR

11.7.1 Interrupt Category 1

11.7.1.1 NMI ISR

Any registers that the ISR routine uses must be saved upon entry, and restored upon exit. Use the RTE instruction to exit the ISR.

11.7.1.2 Other Interrupts

11.7.1.2.1 Interrupt Stack Change

Change the stack at the beginning of the ISR. The corresponding interrupt stack for that interrupt level must be used.

11.7.1.2.2 ISR Return

The RTE instruction must be used to exit ISR.

11.7.1.2.3 Register Used

Any registers that the ISR routine uses must be saved upon entry, and restored upon exit.

11.7.1.2.4 Example of ISR

Figure 11.1 shows how the ISR, of interrupt level 4, should be implemented. In addition, in the case of the C language, since configurator generates it automatically, it does not need to be conscious by application.

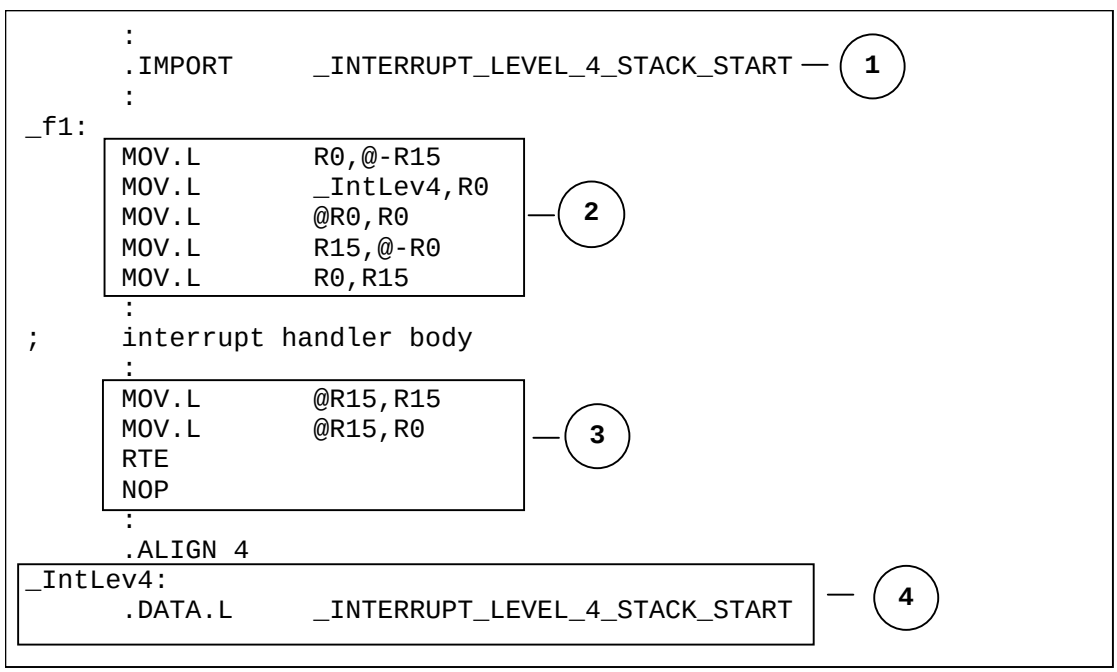

Figure 11.1 ISR Implementation in Assembler

Description:

1. In this case, the start of the interrupt stack for level 4 interrupts must be referenced. Fifteen symbols can be imported. These are:

Table 11.5 Stack Start Symbol of Interrupt Category 1

Symbol	Meaning
`_INTERRUPT_LEVEL_1_STACK_START`	Stack for interrupt level 1
`_INTERRUPT_LEVEL_2_STACK_START`	Stack for interrupt level 2
`_INTERRUPT_LEVEL_3_STACK_START`	Stack for interrupt level 3
`_INTERRUPT_LEVEL_4_STACK_START`	Stack for interrupt level 4
`_INTERRUPT_LEVEL_5_STACK_START`	Stack for interrupt level 5
`_INTERRUPT_LEVEL_6_STACK_START`	Stack for interrupt level 6
`_INTERRUPT_LEVEL_7_STACK_START`	Stack for interrupt level 7
`_INTERRUPT_LEVEL_8_STACK_START`	Stack for interrupt level 8
`_INTERRUPT_LEVEL_9_STACK_START`	Stack for interrupt level 9
`_INTERRUPT_LEVEL_10_STACK_START`	Stack for interrupt level 10
`_INTERRUPT_LEVEL_11_STACK_START`	Stack for interrupt level 11
`_INTERRUPT_LEVEL_12_STACK_START`	Stack for interrupt level 12
`_INTERRUPT_LEVEL_13_STACK_START`	Stack for interrupt level 13
`_INTERRUPT_LEVEL_14_STACK_START`	Stack for interrupt level 14
`_INTERRUPT_LEVEL_15_STACK_START`	Stack for interrupt level 15

2. These statements provide the stack change into the interrupt stack before execution of the ISR.

3. These statements provide the exit from the ISR, and stack change back to the interrupt process.

4. These provide references for the stack start address

11.7.2 Interrupt Category 2

Any registers that the ISR routine uses must be saved upon entry, and restored upon exit. Use the RTS instruction to exit the ISR. The stack is not needed to be changed in the ISR.

11.8 Registering ISRs

ISR must register their starting addresses to the vector table. Figure 11.2 shows the relationship between the vector table and the ISR of category 1.

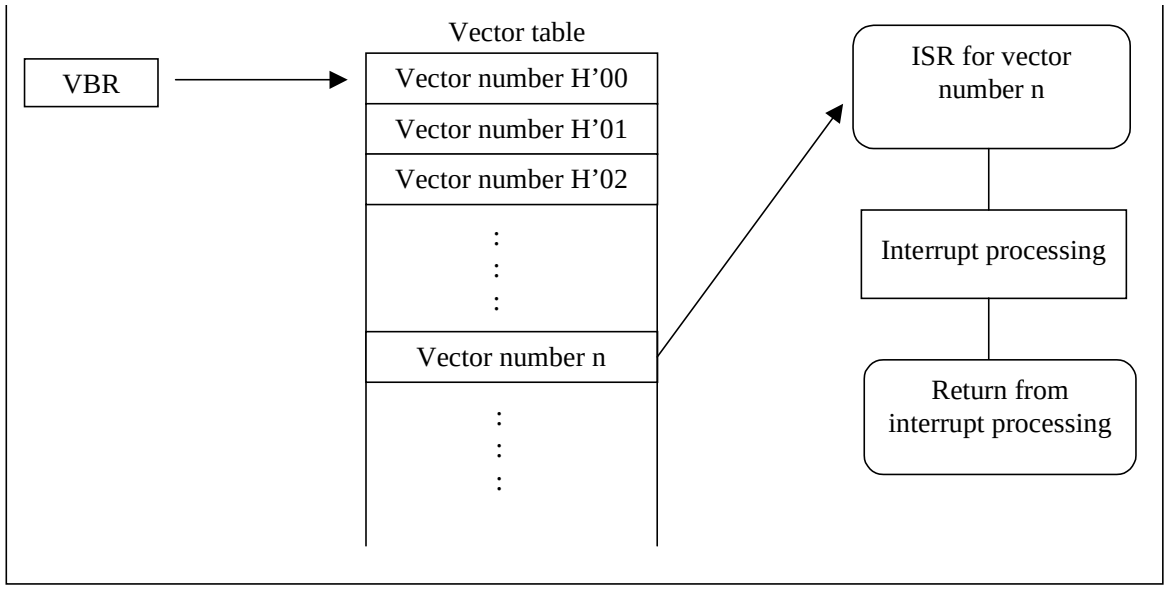

Figure 11.2 Relationship between Vector Table and ISR of Category 1

For interrupts of category 2, control must first be passed to the OS. Preambles are used to transfer control to the OS before calling the interrupt handler. Figure 11.3 shows the operation of the preamble.

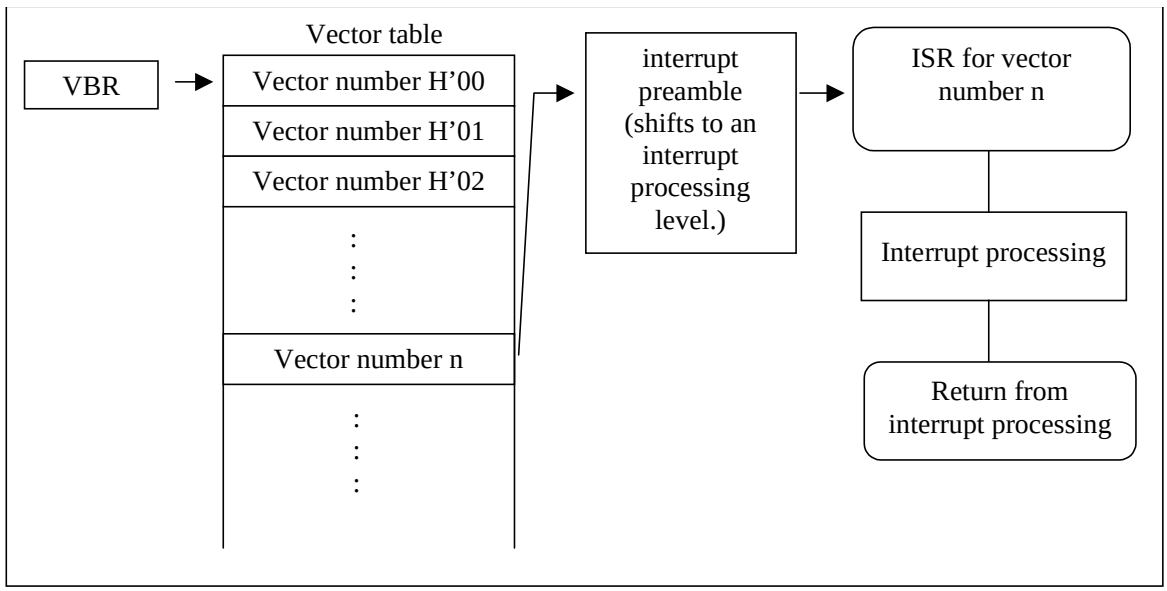

Figure 11.3 Relationship between Vector Table and ISR of Category 2

Each interrupt of category 2 has its own preamble which must be registered to the vector table. These preambles are automatically generated by the configurator. The preamble name is generated by adding “_ISR” to the name. Thus, for example, “ATUTimer1” becomes “ATUTimer1_ISR”.

Set an interrupt mask level same as a value defined by the OS configurator as interrupt priority level setting register (IPR) of the CPU. Then, don't change IPR.

11.9 OS Interrupts

The OS reserves the interrupts in Table 11.7:

Table 11.7 OS Interrupt

Interrupt	Vector number	Interrupt handler (Assembler name)
Compare Match Timer Interrupt 0	188	_SysTimer_ISR
Compare Match Timer Interrupt 1	192	_NonVarTimer_ISR

12 System Services

12.1 Introduction

The functional details of each system service will be described using the following fields.

System Service Description

Syntax	Interface
Parameter (In):	List of all input parameters.
Parameter (Out):	List of all output parameters.
Description:	Explanation of the functionality of the system service.
Particularities:	Detail explanation of the functionality of the system service.
Error Status: Standard Extended	List of return values. • List of error codes provided in the operating system's standard version. • List of additional error codes in the operating system's extended version
Conformance Class:	Specifies the conformance classes where the operating system 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 a pointer to the …Type.

12.2 Task Management Services

12.2.1 Data Types

- **StatusType**
  This data type is used for all status information returned by the operating system service.

- **TaskType**
  This data type identifies a task.

- **TaskRefType**
  This data type points to a variable of the data type TaskType.

- **TaskStateType**
  This data type identifies the state of a task.

- **TaskStateRefType**
  This data type points to a variable of the data type TaskStateType.

12.2.2.1 ActivateTask

Syntax	StatusType **ActivateTask**(TaskType <TaskID>)
Parameter (In): TaskID	 Activation task ID (or name)
Parameter (Out):	None
Description:	The task <TaskID> is transferred from the *suspended* state into the *ready* state[4]. The OS ensures that the task will start executing from the first statement of the task.
Particularities:	The service may be called from interrupt level, from task level and the hook routine *StartupHook*.
Error Status: Standard Extended	 • No error, **E_OK** • Too many activations of <TaskID>, **E_OS_LIMIT (original function)** • Called from invalid processing level, **E_OS_CALLEVEL** • <TaskID> is invalid, **E_OS_ID** • Too many multiple activations, **E_OS_LIMIT**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

[4] The ActivateTask system service will not immediately change the state of the task in the case of multiple requests. If the task is not *suspended*, the activation will only be recorded and performed later.

12.2.2.2 TerminateTask

Syntax	StatusType **TerminateTask**(void)
Parameter (In):	None
Parameter (Out):	None
Description:	This service causes the termination of the calling task. The calling task is transferred from the *running* state into the *suspended* state[5].
Particularities:	The resources occupied by the task must be released before the call to *TerminateTask*. Undefined behaviour will result if resources are still occupied at task termination in standard error status mode. If the version with standard error status is used, the system service does not return. If the version with extended status is used, the service returns in the case of an error. When there is no error, the system service does not return. This service may only be called from the task. When the task is terminated, be sure to issue this system service or ChainTask system service.
Error Status: Standard Extended	• No return to call level • Call at non-task level, **E_OS_CALLEVEL** • Task still occupies resources, **E_OS_RESOURCE**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

[5] In the case of tasks with multiple activations, the *suspended* state of the task is transferred to the *ready* state.

12.2.2.3 ChainTask

Syntax	StatusType **ChainTask**(TaskType <TaskID>)
Parameter(In): TaskID	Reference to the sequential succeeding task ID (or name) to be activated.
Parameter(Out):	None
Description:	This service causes the termination of the calling task. After termination, a succeeding task <TaskID> is activated sequentially.
Particularities:	If the succeeding task is identical to the current task, this situation will not result in multiple requests. It is activated after the task termination. The resources occupied by the task must be released before the call to *ChainTask*. Undefined behaviour will result if resources are still occupied at task termination in standard error status mode. If the version with standard error status is used, the system service does not return. If the version with extended status is used, the service returns in the case of an error. When it has no error, the system service does not return. This service may only be called from the task. When the task is terminated, be sure to issue this system service or TerminateTask system service.
Error Status: Standard Extended	• No return to call level • Too many activations of <TaskID>, **E_OS_LIMIT (original function)** • Call at non-task level, **E_OS_CALLEVEL** • <TaskID> is invalid, **E_OS_ID** • Task still occupies resources, **E_OS_RESOURCE** • Too many multiple activations, **E_OS_LIMIT**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.2.2.4 Schedule

Syntax	StatusType **Schedule**(void)
Parameter (In):	None
Parameter (Out):	None
Description:	If a higher-priority task is *ready*, the current task is put into the *ready* state, and the higher priority task is executed. Otherwise, the calling task continues execution.
Particularities:	This system service allows non-preemptive tasks to enforce a reschedule. If the scheduler is locked (RES_SCHEDULER is occupied), no reschedule will occur. This service may only be called from the task.
Error Status: Standard Extended	• No error, **E_OK** • Call at non-task level, **E_OS_CALLEVEL**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.2.2.5 GetTaskID

Syntax	StatusType **GetTaskID**(TaskRefType <TaskID>)
Parameter (In): TaskID	Task ID storage area and the addresses
Parameter (Out): *TaskID	Reference to the task ID which is currently active
Description:	This service returns the ID of the currently active task.
Particularities:	This service is allowed at task level, and from several hook routines. If no task is currently active, the task ID **ID_INVALID_TASK** is returned.
Error Status: Standard Extended	• No error, **E_OK** • Call from invalid processing level, **E_OS_CALLEVEL**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.2.2.6 GetTaskState

Syntax	StatusType **GetTaskState**(TaskType <TaskID>, TaskStateRefType <State>)
Parameter (In): TaskID State	 Task ID (or name) for reference State storage area and the addresses
Parameter (Out): *State	 Reference to the state of the task <TaskID>
Description:	This service returns the state of the task <TaskID> at the time of calling the system service.
Particularities:	This service may be called from task level, interrupt level, and some hook routines.
Error Status: Standard Extended	 • No error, **E_OK** • Call from invalid processing level, **E_OS_CALLEVEL** • <TaskID> is invalid, **E_OS_ID**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.2.3 Constants of data type TaskStateType

Constant	**Description**
SUSPENDED (D'0)	task state *suspended*
WAITING (D'1)	task state *waiting*
READY (D'2)	task state *ready*
RUNNING (D'3)	task state *running*

12.3 Interrupt Management Services

12.3.1 DataTypes

- **IntDescriptorType**
  Data type for logical interrupt masks.

12.3.2 System Services

12.3.2.1 EnableInterrupt

(a) By Interrupt-Mask Level (when "Use mask level" is selected in the OS page of the OS configurator)

Syntax	void **EnableInterrupt**(IntDescriptorType <Descriptor>) or void **EnableInterruptMask**(IntDescriptorType <Descriptor>)
Parameter (In): Descriptor	The status register value with an interrupt-mask level –1 to be made valid.
Parameter (Out):	None
Description:	This service enables interrupts at, and above, the interrupt level given in <Descriptor>.
Particularities:	<Descriptor> values must be multiples of 16 in the range from H'00000000 to H'000000f0. If some other value is specified, system operation cannot be guaranteed. When H'00000000 is specified, the OS performs rescheduling. This service can be called from task level and interrupt level. It is not allowed from hook routines, but no error will be returned for this error condition. In this case, operation is not guaranteed. **Note**: **The operation of this system service is original function.**
Error Status: Standard Extended	 None **(original function)** None **(original function)**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

(b) By Interrupt Source (when "Use source" is selected on the OS page of the OS configurator)

Syntax	StatusType **EnableInterrupt**(IntDescriptorType <Descriptor>) or StatusType **EnableInterruptSource**(IntDescriptorType <Descriptor>)
Parameter (In): Descriptor	The ISR ID (an ISR name ending with "_ID") for which interrupts are to be enabled.
Parameter (Out):	None
Description:	This service enables interrupts from the source that corresponds to the ISR ID given as <Descriptor>.
Particularities:	The interrupt is enabled by the setting of the CPU's interrupt-request enable bit or in the interrupt priority-level setting register (IPR). For information on the values to be set for each interrupt source, see Section 12.3.4. For the user-defined sources (CUSTOM0 to CUSTOM31), see Section 12.3.5. This service can be called from task level and interrupt level. It is not allowed from hook routines, but no error will be returned for this error condition. In this case, operation is not guaranteed. This service does not call the error-hook routine even when an error occurs. **Note**: **The operation of this system service is original function.**
Error Status: Standard Extended	 No error, **E_OK** The specified interrupt is enabled: **E_OS_NOFUNC**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.3.2.2 DisableInterrupt

(a) By Interrupt-Mask Level (when "Use mask level" is selected in the OS page of the OS configurator)

Syntax	IntDescriptorType **DisableInterrupt**(IntDescriptorType <Descriptor>) or IntDescriptorType **DisableInterruptMask**(IntDescriptorType <Descriptor>)
Parameter (In): Descriptor	The status register value with an interrupt-mask level to be made invalid.
Parameter (Out): R0	The value of the interrupt-mask level in the status register which before issued DisableInterrupt.
Description:	This service disables interrupts at, and below, the interrupt level given in <Descriptor>.
Particularities:	<Descriptor> values must be multiples of 16 in the range from H'00000000 to H'000000f0. If some other value is specified, system operation cannot be guaranteed. When H'00000000 is specified, all interrupts are enabled. If the specified interrupt-mask level has already been disabled (i.e., when the specified interrupt-mask level is lower than the level before invalid), the specified value is not set in the status register. This service can be called from task level and interrupt level. It is not allowed from hook routines, but no error will be returned for this error condition. In this case, operation is not guaranteed. **Note**: **The operation of this system service is original function.**
Error Status: Standard Extended	None **(original function)** None **(original function)**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

(b) By Interrupt Source (when "Use source" is selected in the OS page of the OS configurator)

Syntax	StatusType **DisableInterrupt**(IntDescriptorType <Descriptor>) or StatusType **DisableInterruptSource**(IntDescriptorType <Descriptor>)
Parameter (In): Descriptor	The ISR ID (an ISR name ending with "_ID") for which interrupts are to be disabled.
Parameter (Out):	None
Description:	This service disables interrupts from the source that corresponds to the ISR ID given as <Descriptor>.
Particularities:	The interrupt is disabled by the setting of the CPU's interrupt-request enable bit or in the interrupt priority-level setting register (IPR). For information on the values to be set for each interrupt source, see Section 12.3.4. For the user-defined sources (CUSTOM0 to CUSTOM31), see Section 12.3.5. This service can be called from task level and interrupt level. It is not allowed from hook routines, but no error will be returned for this error condition. In this case, operation is not guaranteed. This service does not call the error-hook routine even when an error occurs. **Note**: **The operation of this system service is original function.**
Error Status: Standard Extended	 No error, **E_OK** The specified interrupt has been disabled: **E_OS_NOFUNC**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.3.2.3 GetInterruptDescriptor

(a) By Interrupt-Mask Level (when "Use mask level" is selected in the OS page on the OS configurator)

Syntax	IntDescriptiorType **GetInterruptDescriptor**(void) or IntDescriptiorType **GetInterruptDescriptorMask**(void)
Parameter (In):	None
Parameter (Out): R0	Status register value with the current interrupt-mask level.
Description:	This service gets the current interrupt mask level.
Particularities:	This service can be called from task level, interrupt level, and some hook routines. Calls are not allowed from certain hook routines, but no error will be returned if such calls are made. In this case, operation is not guaranteed. **Note**: **The operation of this system service is original function.**
Error Status: Standard Extended	None **(original function)** None
Conformance Class:	BCC1, BCC2, ECC1, ECC2

(b) By Interrupt Source (when "Use source" is selected in the OS page on the OS configurator)

Syntax	StatusType **GetInterruptDescriptor**(IntDescriptorType <Descriptor>) or StatusType **GetInterruptDescriptorSource**(IntDescriptorType <Descriptor>)
Parameter (In): Descriptor	The ISR ID (an ISR name ending with "_ID") that acquires interrupt state.
Parameter (Out):	None
Description:	This service gets the state of the interrupt source that corresponds to the ISR ID given in <Descriptor>.
Particularities:	Refers to the CPU's interrupt-request enable bit or the value in the interrupt priority-level setting register (IPR). For a list of the values that enable and disable each interrupt source, see Section 12.3.4. In the interrupt source method to refer to IPR, only the value defined by the OS configurator is valid. For user-defined sources (CUSTOM0 to CUSTOM31), see Section 12.3.5. This service can be called from task level, interrupt level, and some hook routines. Calls are not allowed from certain hook routines, but no error will be returned if such calls are made. In this case, operation is not guaranteed. This service does not call the error hook routine even when an error occurs. **Note**: **The operation of this system service is original function.**
Error Status: Standard Extended	When interrupt is enabled: **1** When interrupt is disabled: **0**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.3.3 Constants of the IntDescriptorType Data Type

The constants of the IntDescriptorType data type are used when the interrupt mask level method is selected (i.e., when "Usr mask level" is selected on the OS page of the OS configurator).

Table 12.1 IntDescriptorType Data

Constant	Description
SR_IMS00 (H’00000000)	Interrupt mask level 0
SR_IMS01 (H’00000010)	Interrupt mask level 1
SR_IMS02 (H’00000020)	Interrupt mask level 2
SR_IMS03 (H’00000030)	Interrupt mask level 3
SR_IMS04 (H’00000040)	Interrupt mask level 4
SR_IMS05 (H’00000050)	Interrupt mask level 5
SR_IMS06 (H’00000060)	Interrupt mask level 6
SR_IMS07 (H’00000070)	Interrupt mask level 7
SR_IMS08 (H’00000080)	Interrupt mask level 8
SR_IMS09 (H’00000090)	Interrupt mask level 9
SR_IMS10 (H’000000a0)	Interrupt mask level 10
SR_IMS11 (H’000000b0)	Interrupt mask level 11
SR_IMS12 (H’000000c0)	Interrupt mask level 12
SR_IMS13 (H’000000d0)	Interrupt mask level 13
SR_IMS14 (H’000000e0)	Interrupt mask level 14
SR_IMS15 (H’000000f0)	Interrupt mask level 15

12.3.4 Interrupt Sources and Settings

These interrupt sources and settings are used when the interrupt source method is selected (i.e., when "User source" is selected on the OS page of the OS configurator).

Table 12.2 Interrupt Sources and Settings

No.	Interrupt cause	Vector No.	Register	Bit	Invalid value	Valid value
1	IRQ0	64	IPRA	Bits 15 to 12	0	1 to 15 *2
2	IRQ1	65	IPRA	Bits 11 to 8	0	1 to 15 *2
3	IRQ2	66	IPRA	Bits 7 to 4	0	1 to 15 *2
4	IRQ3	67	IPRA	Bits 3 to 0	0	1 to 15 *2
5	IRQ4 *1	68	IPRB	Bits 15 to 12	0	1 to 15 *2
6	IRQ5 *1	69	IPRB	Bits 11 to 8	0	1 to 15 *2
7	IRQ6 *1	70	IPRB	Bits 7 to 4	0	1 to 15 *2
8	IRQ7 *1	71	IPRB	Bits 3 to 0	0	1 to 15 *2
9	DMAC0 DEI0	72	CHCR0	IE (bit 2)	0	1
10	DMAC1 DEI1	74	CHCR1	IE (bit 2)	0	1
11	DMAC2 DEI2	76	CHCR2	IE (bit 2)	0	1
12	DMAC3 DEI3	78	CHCR3	IE (bit 2)	0	1
13	ATU0 ITV1(ITVE6)	80	ITVRR1	ITVE6 (bit 0)	0	1
14	ATU0 ITV1(ITVE7)	80	ITVRR1	ITVE7 (bit 1)	0	1
15	ATU0 ITV1(ITVE8)	80	ITVRR1	ITVE8 (bit 2)	0	1
16	ATU0 ITV1(ITVE9)	80	ITVRR1	ITVE9 (bit 3)	0	1
17	ATU0 ITV2A(ITVE10A)	80	ITVRR2A	ITVE10A (bit 0)	0	1
18	ATU0 ITV2A(ITVE11A)	80	ITVRR2A	ITVE11A (bit 1)	0	1
19	ATU0 ITV2A(ITVE12A)	80	ITVRR2A	ITVE12A (bit 2)	0	1
20	ATU0 ITV2A(ITVE13A)	80	ITVRR2A	ITVE13A (bit 3)	0	1
21	ATU0 ITV2B(ITVE10B)	80	ITVRR2B	ITVE10B (bit 0)	0	1
22	ATU0 ITV2B(ITVE11B)	80	ITVRR2B	ITVE11B (bit 1)	0	1
23	ATU0 ITV2B(ITVE12B)	80	ITVRR2B	ITVE12B (bit 2)	0	1
24	ATU0 ITV2B(ITVE13B)	80	ITVRR2B	ITVE13B (bit 3)	0	1
25	ATU0 ICI0A	84	TIER0	ICE0A (bit 0)	0	1
26	ATU0 ICI0B	86	TIER0	ICE0B (bit 1)	0	1
27	ATU0 ICI0C	88	TIER0	ICE0C (bit 2)	0	1
28	ATU0 ICI0D	90	TIER0	ICE0D (bit 3)	0	1
29	ATU0 OVI0	92	TIER0	OVE0 (bit 4)	0	1
30	ATU1 IMI1A	96	TIER1A	IME1A (bit 0)	0	1
31	ATU1 CMI1	96	TIER1B	CME1 (bit 0)	0	1
32	ATU1 IMI1B	97	TIER1A	IME1B (bit 1)	0	1
33	ATU1 IMI1C	98	TIER1A	IME1C (bit 2)	0	1
34	ATU1 IMI1D	99	TIER1A	IME1D (bit 3)	0	1
35	ATU1 IMI1E	100	TIER1A	IME1E (bit 4)	0	1
36	ATU1 IMI1F	101	TIER1A	IME1F (bit 5)	0	1
37	ATU1 IMI1G	102	TIER1A	IME1G (bit 6)	0	1
38	ATU1 IMI1H	103	TIER1A	IME1H (bit 7)	0	1

Table 12.2 Interrupt Sources and Settings (continue)

No.	Interrupt cause	Vector No.	Register	Bit	Invalid value	Valid value
39	ATU1 OVI1A	104	TIER1A	OVE1A (bit 8)	0	1
40	ATU1 OVI1B	104	TIER1B	OVE1B (bit8)	0	1
41	ATU2 IMI2A	108	TIER2A	IME2A (bit 0)	0	1
42	ATU2 CMI2A	108	TIER2B	CME2A (bit 0)	0	1
43	ATU2 IMI2B	109	TIER2A	IME2B (bit 1)	0	1
44	ATU2 CMI2B	109	TIER2B	CME2B (bit 1)	0	1
45	ATU2 IMI2C	110	TIER2A	IME2C (bit 2)	0	1
46	ATU2 CMI2C	110	TIER2B	CME2C (bit 2)	0	1
47	ATU2 IMI2D	111	TIER2A	IME2D (bit 3)	0	1
48	ATU2 CMI2D	111	TIER2B	CME2D (bit 3)	0	1
49	ATU2 IMI2E	112	TIER2A	IME2E (bit 4)	0	1
50	ATU2 CMI2E	112	TIER2B	CME2E (bit 4)	0	1
51	ATU2 IMI2F	113	TIER2A	IME2F (bit 5)	0	1
52	ATU2 CMI2F	113	TIER2B	CME2F (bit 5)	0	1
53	ATU2 IMI2G	114	TIER2A	IME2G (bit 6)	0	1
54	ATU2 CMI2G	114	TIER2B	CME2G (bit 6)	0	1
55	ATU2 IMI2H	115	TIER2A	IME2H (bit 7)	0	1
56	ATU2 CMI2H	115	TIER2B	CME2H (bit 7)	0	1
57	ATU2 OVI2A	116	TIER2A	OVE2A (bit 8)	0	1
58	ATU2 OVI2B	116	TIER2B	OVE2B (bit 8)	0	1
59	ATU3 IMI3A	120	TIER3	IME3A (bit 0)	0	1
60	ATU3 IMI3B	121	TIER3	IME3B (bit 1)	0	1
61	ATU3 IMI3C	122	TIER3	IME3C (bit 2)	0	1
62	ATU3 IMI3D	123	TIER3	IME3D (bit 3)	0	1
63	ATU3 OVI3	124	TIER3	OVE3 (bit 4)	0	1
64	ATU4 IMI4A	128	TIER3	IME4A (bit 5)	0	1
65	ATU4 IMI4B	129	TIER3	IME4B (bit 6)	0	1
66	ATU4 IMI4C	130	TIER3	IME4C (bit 7)	0	1
67	ATU4 IMI4D	131	TIER3	IME4D (bit 8)	0	1
68	ATU4 OVI4	132	TIER3	OVE4 (bit 9)	0	1
69	ATU5 IMI5A	136	TIER3	IME5A (bit 10)	0	1
70	ATU5 IMI5B	137	TIER3	IME5B (bit 11)	0	1
71	ATU5 IMI5C	138	TIER3	IME5C (bit 12)	0	1
72	ATU5 IMI5D	139	TIER3	IME5D (bit 13)	0	1
73	ATU5 OVI5	140	TIER3	OVE5 (bit 14)	0	1
74	ATU6 CMI6A	144	TIER6	CME6A (bit 0)	0	1
75	ATU6 CMI6B	145	TIER6	CME6B (bit 1)	0	1
76	ATU6 CMI6C	146	TIER6	CME6C (bit 2)	0	1
77	ATU6 CMI6D	147	TIER6	CME6D (bit 3)	0	1
78	ATU7 CMI7A	148	TIER7	CME7A (bit 0)	0	1
79	ATU7 CMI7B	149	TIER7	CME7B (bit 1)	0	1
80	ATU7 CMI7C	150	TIER7	CME7C (bit 2)	0	1
81	ATU7 CMI7D	151	TIER7	CME7D (bit 3)	0	1
82	ATU8 OSI8A	152	TIER8	OSE8A (bit 0)	0	1
83	ATU8 OSI8B	153	TIER8	OSE8B (bit 1)	0	1
84	ATU8 OSI8C	154	TIER8	OSE8C (bit 2)	0	1

Table 12.2 Interrupt Sources and Settings (continue)

No.	Interrupt cause	Vector No.	Register	Bit	Invalid value	Valid value
85	ATU8 OSI8D	155	TIER8	OSE8D (bit 3)	0	1
86	ATU8 OSI8E	156	TIER8	OSE8E (bit 4)	0	1
87	ATU8 OSI8F	157	TIER8	OSE8F (bit 5)	0	1
88	ATU8 OSI8G	158	TIER8	OSE8G (bit 6)	0	1
89	ATU8 OSI8H	159	TIER8	OSE8H (bit 7)	0	1
90	ATU8 OSI8I	160	TIER8	OSE8I (bit 8)	0	1
91	ATU8 OSI8J	161	TIER8	OSE8J (bit 9)	0	1
92	ATU8 OSI8K	162	TIER8	OSE8K (bit 10)	0	1
93	ATU8 OSI8L	163	TIER8	OSE8L (bit 11)	0	1
94	ATU8 OSI8M	164	TIER8	OSE8M (bit 12)	0	1
95	ATU8 OSI8N	165	TIER8	OSE8N (bit 13)	0	1
96	ATU8 OSI8O	166	TIER8	OSE8O (bit 14)	0	1
97	ATU8 OSI8P	167	TIER8	OSE8P (bit 15)	0	1
98	ATU9 CMI9A	168	TIER9	CME9A (bit 0)	0	1
99	ATU9 CMI9B	169	TIER9	CME9B (bit 1)	0	1
100	ATU9 CMI9C	170	TIER9	CME9C (bit 2)	0	1
101	ATU9 CMI9D	171	TIER9	CME9D (bit 3)	0	1
102	ATU9 CMI9E	172	TIER9	CME9E (bit 4)	0	1
103	ATU9 CMI9F	174	TIER9	CME9F (bit 5)	0	1
104	ATU10 CMI10A	176	TIER10	CME10A (bit 0)	0	1
105	ATU10 CMI10B	178	TIER10	CME10B (bit 2)	0	1
106	ATU10 ICI10A	180	TIER10	ICE10A (bit 1)	0	1
107	ATU10 CMI10G	180	TIER10	CME10G (bit 3)	0	1
108	ATU11 IMI11A	184	TIER11	IME11A (bit 0)	0	1
109	ATU11 IMI11B	186	TIER11	IME11B (bit 1)	0	1
110	ATU11 OVI11	187	TIER11	OVE11 (bit 8)	0	1
111	CMT0 CMTI0	188	CMCSR0	CMIE (bit 6)	0	1
112	A/D0 ADI0	190	ADCSR0	ADIE (bit 6)	0	1
113	CMT1 CMTI1	192	CMCSR1	CMIE (bit 6)	0	1
114	A/D1 ADI1	194	ADCSR1	ADIE (bit 6)	0	1
115	A/D2 ADI2 *1	196	ADCSR2	ADIE (bit 6)	0	1
116	SCI0 ERI0	200	SCR0	RIE (bit 6)	0	1
117	SCI0 RXI0	201	SCR0	RIE (bit 6)	0	1
118	SCI0 TXI0	202	SCR0	TIE (bit 7)	0	1
119	SCI0 TEI0	203	SCR0	TEIE (bit 2)	0	1
120	SCI1 ERI1	204	SCR1	RIE (bit 6)	0	1
121	SCI1 RXI1	205	SCR1	RIE (bit 6)	0	1
122	SCI1 TXI1	206	SCR1	TIE (bit 7)	0	1
123	SCI1 TEI1	207	SCR1	TEIE (bit 2)	0	1
124	SCI2 ERI2	208	SCR2	RIE (bit 6)	0	1
125	SCI2 RXI2	209	SCR2	RIE (bit 6)	0	1
126	SCI2 TXI2	210	SCR2	TIE (bit 7)	0	1
127	SCI2 TEI2	211	SCR2	TEIE (bit 2)	0	1
128	SCI3 ERI3	212	SCR3	RIE (bit 6)	0	1
129	SCI3 RXI3	213	SCR3	RIE (bit 6)	0	1
130	SCI3 TXI3	214	SCR3	TIE (bit 7)	0	1

Table 12.2 Interrupt Sources and Settings (continue)

No.	Interrupt cause	Vector No.	Register	Bit	Invalid value	Valid value
131	SCI3 TEI3	215	SCR3	TEIE (bit 2)	0	1
132	SCI4 ERI4	216	SCR4	RIE (bit 6)	0	1
133	SCI4 RXI4	217	SCR4	RIE (bit 6)	0	1
134	SCI4 TXI4	218	SCR4	TIE (bit 7)	0	1
135	SCI4 TEI4	219	SCR4	TEIE (bit 2)	0	1
136	HCAN0 ERS0(IRR5)	220	IMR0	IMR5 (bit 13)	1	0
137	HCAN0 ERS0(IRR6)	220	IMR0	IMR6 (bit 14)	1	0
138	HCAN0 OVR0(IRR2)	221	IMR0	IMR2 (bit 10)	1	0
139	HCAN0 OVR0(IRR3)	221	IMR0	IMR3 (bit 11)	1	0
140	HCAN0 OVR0(IRR4)	221	IMR0	IMR4 (bit 12)	1	0
141	HCAN0 OVR0(IRR7)	221	IMR0	IMR7 (bit 15)	1	0
142	HCAN0 OVR0(IRR9)	221	IMR0	IMR9 (bit 1)	1	0
143	HCAN0 OVR0(IRR12)	221	IMR0	IMR12 (bit 4)	1	0
144	HCAN0 RM0(IRR1)	222	IMR0	IMR1 (bit 9)	1	0
145	HCAN0 SLE0(IRR8)	223	IMR0	IMR8 (bit 0)	1	0
146	WDT ITI	224	TCSR	TME (bit 5)	0	1
147	HCAN1 ERS1(IRR5) *1	228	IMR1	IMR5 (bit 13)	1	0
148	HCAN1 ERS1(IRR6) *1	228	IMR1	IMR6 (bit 14)	1	0
149	HCAN1 OVR1(IRR2) *1	229	IMR1	IMR2 (bit 10)	1	0
150	HCAN1 OVR1(IRR3) *1	229	IMR1	IMR3 (bit 11)	1	0
151	HCAN1 OVR1(IRR4) *1	229	IMR1	IMR4 (bit 12)	1	0
152	HCAN1 OVR1(IRR7) *1	229	IMR1	IMR7 (bit 15)	1	0
153	HCAN1 OVR1(IRR9) *1	229	IMR1	IMR9 (bit 1)	1	0
154	HCAN1 OVR1(IRR12) *1	229	IMR1	IMR12 (bit 4)	1	0
155	HCAN1 RM1(IRR1) *1	230	IMR1	IMR1 (bit 9)	1	0
156	HCAN1 SLE1(IRR8) *1	231	IMR1	IMR8 (bit 0)	1	0

Notes:
1. Supported only by the SH7055 (not supported by the SH7052, SH7053, or the SH7054).
2. The interrupt level that is specified on the Isr page of the OS configurator.

12.3.5 Functions Used to Control Interrupts from User-Defined Sources

For user-defined source (CUSTOM0 to CUSTOM31), processing to enable or disable interrupts and to get the interrupt status is not provided normally. Therefore, when the user-defined source is used, add the processing to enable interrupts for the EnableInterruptSource (or EnableInterrupt) system service to the _OSEKInterrupt_CUSTOMxx function interrupt in the Example\OSEKisc.c file. Also, add the processing to disable interrupts for the DisableInterruptSource (or DisableInterrupt) system service and the processing to get the interrupt status for the GetInterruptDescriptorSource (or GetInterruptDescriptor) system service.
An input parameter for the user-defined source function is passed by the OS. On the other hand, an output parameter needs to be set by the user. Set an error code (E_OK or E_OS_NOFUNC) for the EnableInterruptSource (or EnableInterrupt) and DisableInterruptSource (or DisableInterrupt) system services, and an interrupt status value (1 or 0) for the GetInterruptDescriptorSource (or GetInterruptDescriptor) system service. The output parameter that has been set is returned to the system-service calling program as the return code of each system service: EnableInterruptSource (or EnableInterrupt), DisableInterruptSource (or DisableInterrupt), and GetInterruptDescriptorSource (or GetInterruptDescriptor).

Table 12.3 Functions Used to Control Interrupts from User-Defined Source

Cause name	Function name
CUSTOM0	_OSEKInterrupt_CUSTOM0
CUSTOM1	_OSEKInterrupt_CUSTOM1
CUSTOM2	_OSEKInterrupt_CUSTOM2
CUSTOM3	_OSEKInterrupt_CUSTOM3
CUSTOM4	_OSEKInterrupt_CUSTOM4
CUSTOM5	_OSEKInterrupt_CUSTOM5
CUSTOM6	_OSEKInterrupt_CUSTOM6
CUSTOM7	_OSEKInterrupt_CUSTOM7
CUSTOM8	_OSEKInterrupt_CUSTOM8
CUSTOM9	_OSEKInterrupt_CUSTOM9
CUSTOM10	_OSEKInterrupt_CUSTOM10
CUSTOM11	_OSEKInterrupt_CUSTOM11
CUSTOM12	_OSEKInterrupt_CUSTOM12
CUSTOM13	_OSEKInterrupt_CUSTOM13
CUSTOM14	_OSEKInterrupt_CUSTOM14
CUSTOM15	_OSEKInterrupt_CUSTOM15
CUSTOM16	_OSEKInterrupt_CUSTOM16
CUSTOM17	_OSEKInterrupt_CUSTOM17
CUSTOM18	_OSEKInterrupt_CUSTOM18
CUSTOM19	_OSEKInterrupt_CUSTOM19
CUSTOM20	_OSEKInterrupt_CUSTOM20
CUSTOM21	_OSEKInterrupt_CUSTOM21
CUSTOM22	_OSEKInterrupt_CUSTOM22
CUSTOM23	_OSEKInterrupt_CUSTOM23
CUSTOM24	_OSEKInterrupt_CUSTOM24
CUSTOM25	_OSEKInterrupt_CUSTOM25
CUSTOM26	_OSEKInterrupt_CUSTOM26
CUSTOM27	_OSEKInterrupt_CUSTOM27
CUSTOM28	_OSEKInterrupt_CUSTOM28
CUSTOM29	_OSEKInterrupt_CUSTOM29
CUSTOM30	_OSEKInterrupt_CUSTOM30
CUSTOM31	_OSEKInterrupt_CUSTOM31

The specifications for a user-defined source function are as listed below.

Syntax	StatusType **OSEKInterrupt_CUSTOM*xx***(IntDescriptorType <Descriptor>, DWORD<Service>)
Parameter (In*): Descriptor Service	 The ISR ID specified for this interrupt service System service ID (EnableInterruptSource_ID, DisableInterruptSource_ID, or GetInterruptDescriptorSource_ID)
Parameter (Out):	None
Description:	Carries out processing of various kinds for user-defined interrupt sources, such as enabling or disabling the interrupts, and acquiring their status.
Particularities:	This service is called when the ISR ID for the user-defined source is specified in the interrupt-system service call by the interrupt source method. Do not issue a system service from this routine. Only category 1 interrupts are accepted during the execution of this routine. Therefore, its execution time should be kept as short as possible to reduce the period of interrupt masking. Do not reduce the interrupt-mask level during the execution of this routine. **When an interrupt system-service is issued with an ISR ID that corresponds to a given user-defined source, the stack size used by this routine should be added to the stack size of the calling program (task, ISR, or hook routine).**
Error Status: Standard Extended	No error, **E_OK (EnableInterruptSource_ID or DisableInterruptSource_ID)** When the interrupt is enabled: **1** When the interrupt is disabled: **0** **(GetInterruptDescriptorSource_ID)** When the specified interrupt is enabled or disabled: **E_OS_NOFUNC** (EnableInterruptSource_ID or DisableInterruptSource_ID)
Conformance Class:	BCC1, BCC2, ECC1, ECC2

Note: *: The OS sets up the input parameters.

12.4 Resource Management Services

12.4.1 Data Types

- **ResourceType**
  Resource ID

12.4.2 System Services

12.4.2.1 GetResource

Syntax	StatusType **GetResource**(ResourceType <ResID>)
Parameter (In): ResID	 Resource ID (or name)
Parameter (Out):	None
Description:	This service gets <ResID> and increases the priority of task.
Particularities:	Services, which put the running task into the suspended or waiting state, must not be used while the resource is occupied (i.e. *TerminateTask, ChainTask,* and *WaitEvent*). This service is only allowed on task level. Nested resource occupation must follow the LIFO principle. **Note: If a nested resource has a lower resource priority than the outer resource, an error will be returned (except in the case of the RES_SCHEDULER resource). This behaviour is original function.** The system service is still executed in this case.
Error Status: Standard Extended	• No error, **E_OK** • Call from non-task level, **E_OS_CALLEVEL** • <ResID> is invalid, **E_OS_ID** • Under <ResID> use or <ResID> is using definition outside, **E_OS_ACCESS** • Incorrect nesting of resource priority, **E_OS_INVALID_NEST (original function)**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.4.2.2 ReleaseResource

Syntax	StatusType **ReleaseResource**(ResourceType <ResID>)
Parameter (In): ResID	 Resource ID (or name)
Parameter (Out):	None
Description:	<ResID> is released and returned to the original priority.
Particularities:	For information on nesting conditions, see particularities of *GetResource*. This service is only allowed on task level.
Error Status: Standard Extended	 • No error, **E_OK** • Call from non-task level, **E_OS_CALLEVEL** • <ResID> is invalid, **E_OS_ID** • Attempt to release a resource which is not occupied, or another resource has to be released first, **E_OS_NOFUNC**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.5 Event Management Services

12.5.1 Data Types

- **EventMaskType**
  Event mask value.

- **EventMaskRefType**
  Pointer for the data type EventMaskType.

12.5.2.1 SetEvent

Syntax	StatusType **SetEvent**(TaskType <TaskID>, EventMaskType <Mask>)
Parameter (In): TaskID Mask	 Task ID (or name) for setting one or several events. Mask value (or name) of the events to be set.
Parameter (Out):	None
Description:	An event is set to <TaskID>. The <TaskID> is transferred to the *ready* state if it was *waiting* for at least one event specified in <Mask>. The task can set more than one event. For example, if the task wishes to set event ID's 0x01 and 0x80, *SetEvent*(<TaskID>, 0x01 | 0x80) is executed.
Particularities:	No events set in the event mask remain unchanged. The referenced task <TaskID> must be an extended task. The service can be called from tasks or interrupts.
Error Status: Standard Extended	• No error, **E_OK** • Call from invalid processing level, **E_OS_CALLEVEL** • <TaskID> is invalid, **E_OS_ID** • <TaskID> is not an extended task, **E_OS_ACCESS** • <TaskID> is in the *suspended* state, **E_OS_STATE**
Conformance Class:	ECC1, ECC2

12.5.2.2 ClearEvent

Syntax	StatusType **ClearEvent**(EventMaskType <Mask>)
Parameter (In): Mask	 Mask value (or name) of the events to be cleared.
Parameter (Out):	None
Description:	The events of the extended task calling *ClearEvent* are cleared according to the event mask <Mask>. The task can clear more than one event. For example, if the task wishes to clear event ID's 0x01 and 0x80, *ClearEvent*(0x01 | 0x80) is executed.
Particularities:	The system service *ClearEvent* is restricted to extended tasks. The service can only be called at task level.
Error Status: Standard Extended	 • No error, **E_OK**. • Call from non-task level, **E_OS_CALLEVEL**. • Call from basic task, **E_OS_ACCESS**
Conformance Class:	ECC1, ECC2

12.5.2.3 GetEvent

Syntax	StatusType **GetEvent**(TaskType <TaskID>, EventMaskRefType <Mask>)
Parameter (In): TaskID Mask	 Task ID (or name) for event reference. Event mask storage area and the addresses
Parameter (Out): *Mask	 Current event mask value.
Description:	This service gets the current event mask value of the task <TaskID>.
Particularities:	This service can be called from task level, interrupt level, and some hook routines. The referenced task <TaskID> must be an extended task.
Error Status: Standard Extended	 • No error, **E_OK**. • Call from invalid processing level, **E_OS_CALLEVEL** • <TaskID> is invalid, **E_OS_ID** • <TaskID> is not an extended task, **E_OS_ACCESS** • <TaskID> is in the *suspended* state, **E_OS_STATE**
Conformance Class:	ECC1, ECC2

12.5.2.4 WaitEvent

Syntax	StatusType **WaitEvent**(EventMaskType <Mask>)
Parameter (In): Mask	 Mask value (or name) of the events *waited* for.
Parameter (Out):	None
Description:	The state of the calling task is set to *waiting*, unless at least one of the events specified in <Mask> has already been set. If the event is already set, task state does not change. The task can wait for more than one event. For example, if the task is *waiting* for event ID's 0x01 and 0x80 to occur, *WaitEvent*(0x01 | 0x80) is executed.
Particularities:	Rescheduling occurs if the task enters the *waiting* state. When task occupies resource, resource needs to be released before calling WaitEvent system service. In standard error status, when waiting for an event during resource occupancy, operation is not guaranteed. This service can only be called by extended tasks.
Error Status: Standard Extended	 • No error, **E_OK**. • Call from non-task level, **E_OS_CALLEVEL** • Call from basic task, **E_OS_ACCESS** • Occupying resource, **E_OS_RESOURCE**
Conformance Class:	ECC1, ECC2

12.6 Counter and Alarm Management Services

12.6.1 Data Types

- **TickType**
  Counter value.

- **TickRefType**
  Pointer for data type TickType.

- **AlarmBaseType**
  Elements of counter characteristics. The individual elements of the structure are:

  – **TickType maxallowedvalue**: maximum count value
  – **TickType ticksperbase**: number of counts required to reach a counter-specific unit (Not used by OS)
  – **TickType mincycle**: minimum allowed number of counts for a cyclic alarm

- **AlarmBaseRefType**
  Pointer for data type AlarmBaseType.

- **AlarmType**
  Alarm ID.

12.6.2.1 GetAlarmBase

Syntax	StatusType **GetAlarmBase**(AlarmType <AlarmID>, AlarmBaseRefType <Info>)
Parameter (In): AlarmID Info	 Reference alarm ID (or name) Counter characteristics storage area and the addresses
Parameter (Out): *Info	 Counter characteristics
Description:	The service gets counter characteristics. The information of data type AlarmBaseType (maximum count value, number of counts required to reach a counter-specific unit, and minimum allowed number of counts for a cyclic alarm) is stored.
Particularities:	Allowed on task level, interrupt level, and some hook routines. Not allowed for non-variant alarms.
Error Status: Standard Extended	 • No error, **E_OK**. • Call from invalid processing level, **E_OS_CALLEVEL** • <AlarmID> is invalid, **E_OS_ID**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.6.2.2 GetAlarm

Syntax	StatusType **GetAlarm**(AlarmType <AlarmID>, TickRefType <Tick>)
Parameter (In): AlarmID Tick	 Reference alarm ID (or name) Count value storage area and the addresses
Parameter (Out): *Tick	 Relative value in counts before the alarm <AlarmID> expires.
Description:	This service gets the relative value in counts before the alarm <AlarmID> expires.
Particularities:	If <AlarmID> is not in use, <Tick> is not defined. Allowed on task level, interrupt level, and some hook routines. Not allowed for non-variant alarms.
Error Status: Standard Extended	 • No error, **E_OK**. • <AlarmID> is not used, **E_OS_NOFUNC** • Call from invalid processing level, **E_OS_CALLEVEL** • <AlarmID> is invalid, **E_OS_ID**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.6.2.3 SetRelAlarm

Syntax	StatusType **SetRelAlarm**(AlarmType <AlarmID>, TickType <increment>, TickType <cycle>)
Parameter (In): AlarmID increment cycle	 Alarm ID (or name) Relative count value in counts for alarm expiration at the first time Cycle value in the case of cyclic alarm setting
Parameter (Out):	None
Description:	This service activates the alarm <AlarmID>. After <increment> counts have elapsed, the associated action (task activation or event setting) is executed.
Particularities:	If a single alarm is to be set, the <cycle> value must be zero (0). When <increment> is 0, the expiration count is set to the present count value. Alarm expiry will not occur immediately; the alarm will expire once the counter has rolled-over and reached the present counter value. If a cyclic alarm is to be set (by setting the <cycle> value to some non-zero value), the alarm is re-activated after expiry with the relative value <cycle>. If the alarm is already in use, the call will be ignored and the error **E_OS_STATE** is returned. Allowed on task level and interrupt level. Not allowed for non-variant alarms.
Error Status: Standard Extended	 • No error, **E_OK**. • <AlarmID> is already in use, **E_OS_STATE** • Call from invalid processing level, **E_OS_CALLEVEL** • <AlarmID> is invalid, **E_OS_ID** • Value of <increment> is illegal (lower than zero, or greater than **maxallowedvalue**), **E_OS_VALUE** • Value of <cycle> is illegal (lower than **mincycle** or greater than **maxallowedvalue**), **E_OS_VALUE**
Conformance Class:	BCC1, BCC2, ECC1, ECC2 (When alarm sets an event, only ECC1 or ECC2 can be used.)

12.6.2.4 SetAbsAlarm

Syntax	StatusType **SetAbsAlarm**(AlarmType <AlarmID>, TickType <start>, TickType <cycle>)
Parameter(In): AlarmID start cycle	 Alarm ID (or name) Absolute count value in counts for alarm expiration at the first time Cycle value in the case of cyclic alarm setting
Parameter(Out):	None
Description:	This service activates the alarm <AlarmID>. When <start> counts are reached, the associated action (task activation or event setting) is executed.
Particularities:	If a single alarm is to be set, the <cycle> value must be zero (0). If a cyclic alarm is to be set (by setting the <cycle> value to some non-zero value), the alarm is re-activated after expiry with the relative value <cycle>. If the alarm is already in use, the call will be ignored and the error **E_OS_STATE** is returned. Allowed on task level and interrupt level. Not allowed for non-variant alarms.
Error Status: Standard Extended	 • No error, **E_OK**. • <AlarmID> is already in use, **E_OS_STATE** • Call from invalid processing level, **E_OS_CALLEVEL** • <AlarmID> is invalid, **E_OS_ID** • Value of <start> is illegal (lower than zero, or greater than **maxallowedvalue**), **E_OS_VALUE** • Value of <cycle> is illegal (lower than **mincycle** or greater than **maxallowedvalue**), **E_OS_VALUE**
Conformance Class:	BCC1, BCC2, ECC1, ECC2 (When alarm sets an event, only ECC1 or ECC2 can be used.)

12.6.2.5 CancelAlarm

Syntax	StatusType **CancelAlarm**(AlarmType <AlarmID>)
Parameter (In): AlarmID	 Alarm ID (or name)
Parameter (Out):	None
Description:	This service cancels the alarm <AlarmID>
Particularities:	Allowed on task level and interrupt level. Not allowed for non-variant alarms.
Error Status: Standard Extended	 • No error, **E_OK**. • <AlarmID> is not used, **E_OS_NOFUNC** • Call from invalid processing level, **E_OS_CALLEVEL** • <AlarmID> is invalid, **E_OS_ID**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.6.2.6 InitialiseCounter

Syntax	StatusType **InitialiseCounter**(CounterType <CounterID>, TickType <tick>)
Parameter (In): CounterID tick	 Counter ID (or name) Counter initial value
Parameter (Out):	None
Description:	This service initialises the counter with the <tick> value.
Particularities:	Allowed on task level and interrupt level. If not initialised, the count will not be incremented and alarm does not operate. The counter can be re-initialised. Not allowed for non-variant alarm timer counter. **Note: This system service is original function.**
Error Status: Standard Extended	 • No error, **E_OK**. • Call from invalid processing level, **E_OS_CALLEVEL** • <CounterID> is invalid, **E_OS_ID** • <tick> is invalid (lower than zero or greater than **maxallowedvalue**), **E_OS_VALUE**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.6.2.7 IncrementCounter

Syntax	StatusType **IncrementCounter**(CounterType <CounterID>)
Parameter (In): CounterID	Counter ID (or name)
Parameter (Out):	None
Description:	This service advances the count of counter <CounterID> by 1. If alarms are due to expire at the count, the actions associated with the expiring alarm will be performed.
Particularities:	Allowed on task level and interrupt level. Do not call this parameter from other levels. In extended error status, if multiple errors occur, **E_OS_MULTIPLE** error returns. If an error is detected while performing the associated actions of the expiring alarms, the service will cancel the current action, but will continue service execution. If the counter is not initialised, then the count value will not be incremented. Not allowed for non-variant alarm timer counter. Do not issue this system service to the same counter ID from the task and the ISR. Do not issue this system service for the same counter ID from the different interrupt mask level. The system operation cannot be guaranteed. **Note: This system service is original function.**
Error Status: Standard Extended	• No error, **E_OK**. • Multiple task activations, **E_OS_LIMIT** • **E_OS_STATE** or **E_OS_LIMIT** occurs two or more times, **E_OS_MULTIPLE** • Call from invalid processing level, **E_OS_CALLEVEL** • <CounterID> is invalid, **E_OS_ID** • Events cannot be set as the task associated with the expiring alarm is in the *suspended* state, **E_OS_STATE**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.7 Operating System Execution Control

12.7.1Data Types

- **AppModeType**
  Application mode ID.

12.7.2 System Services

12.7.2.1 GetActiveApplicationMode

Syntax	AppModeType **GetActiveApplicationMode**(void)
Parameter (In):	None
Parameter (Out): Return value	Application mode ID
Description:	This service gets the current application mode ID.
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.7.2.2 StartOS

Syntax	void **StartOS**(AppModeType <Mode>)
Parameter (In): Mode	Application mode ID (or name)
Parameter (Out):	None
Description:	This service starts the OS in a specific application mode.
Particularities:	If this service is called with an illegal application mode, it will cancel OS initialisation and return to the original call. Call this system service after initialising CPU, etc.
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.7.2.3 ShutdownOS

Syntax	StatusType **ShutdownOS**(StatusType <Error>)
Parameter (In): Error	 Error classification
Parameter (Out):	None
Description:	This service shuts down the OS.
Particularities:	Returns to StartOS call origin. This service can only be called from the task level.
Error Status: Standard Extended	 • No return to original call • Call from invalid processing level, **E_OS_CALLEVEL (original function)**
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.8 Hook Routines

12.8.1 System Services

12.8.1.1 ErrorHook

Syntax	void **ErrorHook** (StatusType <Error>, APICall <APICallId>, ContextType <ContextId>)
Parameter (In*): Error APICall ContextID	 Error code that occurred System service ID which returned error **(original function; refer to Table A.3)** Context ID of calling program **(original function; refer to Table A.4)**
Parameter (Out):	None
Description:	If a system service error occurs, this hook routine is called by the OS at the end of a system service. It is called before returning to the calling program.
Particularities:	See Section 10.3 for a general description of hook routines.
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.8.1.2 PreTaskHook

Syntax	void **PreTaskHook** (void)
Parameter (In*):	None
Parameter (Out):	None
Description:	This hook routine is called by the OS before task switch to *running*. The task is in the *running* state when this hook routine is called (to get task ID by *GetTaskID*).
Particularities:	See Section 10.3 for a general description of hook routines.
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.8.1.3 PostTaskHook

Syntax	void **PostTaskHook** (TaskStateType <State>)
Parameter (In*): State	 Reference to the task state **(original function; refer to section 12.2.3)**
Parameter (Out):	None
Description:	This hook routine is called by the OS after task switch to *ready*, *suspended* or *waiting*. The task is in the *running* state when this hook routine is called (to get task ID by *GetTaskID*).
Particularities:	See Section 10.3 for a general description of hook routines.
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.8.1.4 StartupHook

Syntax	void **StartupHook** (void)
Parameter (In*):	None
Parameter (Out):	None
Description:	This hook routine is called by the OS at the end of the OS initialisation and before the scheduler is running. At this time the application can start tasks, alarms, initialise device drivers, etc.
Particularities:	See Section 10.3 for a general description of hook routines.
Conformance Class:	BCC1, BCC2, ECC1, ECC2

12.8.1.5 ShutdownHook

Syntax	void **ShutdownHook** (StatusType <Error>)
Parameter (In*): Error	Error classification
Parameter (Out):	None
Description:	This hook routine is called by the OS when the system service *ShutdownOS* has been called or the OS has shutdown.
Particularities:	See Section 10.3.
Conformance Class:	BCC1, BCC2, ECC1, ECC2

Note: *: The OS sets up the input parameters.

A Appendix

A.1 System Service Return Codes

Table A.1 System Service Return Codes

System Service	Standard Error Status	Additional Codes in Extended Error
ActivateTask	E_OK, E_OS_LIMIT*	E_OS_ID,E_OS_CALLEVEL
TerminateTask	---	E_OS_RESOURCE, E_OS_CALLEVEL
ChainTask	E_OS_LIMIT*	E_OS_ID, E_OS_RESOURCE, E_OS_CALLEVEL
Schedule	E_OK	E_OS_CALLEVEL
GetTaskID	E_OK	E_OS_CALLEVEL
GetTaskState	E_OK	E_OS_ID, E_OS_CALLEVEL
EnterISR	---	---
LeaveISR	---	---
EnableInterrupt or EnableInterruptMask (Interrupt Mask Level Method)*	---	---
DisableInterrupt or DsiableInterruptMask (Interrupt Mask Level Method)*	---	---
GetInterruptDescriptor or GetIterruptDescriptorMask (Interrupt Mask Level Method)*	---	---
EnableInterrupt or EnableInterruptSource (Interrupt Source Method)*	E_OK	E_OS_NOFUNC
DisableInterrupt or DsiableInterruptSource (Interrupt Source Method)*	E_OK	E_OS_NOFUNC
GetInterruptDescriptor or GetInterruptDescriptorSource (Interrupt Source Method)*	---	---
GetResource	E_OK	E_OS_ID, E_OS_ACCESS, E_OS_CALLEVEL, E_OS_INVALID_NEST*
ReleaseResource	E_OK	E_OS_ID, E_OS_NOFUNC, E_OS_CALLEVEL,
SetEvent	E_OK	E_OS_ID, E_OS_ACCESS, E_OS_STATE, E_OS_CALLEVEL
ClearEvent	E_OK	E_OS_ACCESS, E_OS_CALLEVEL
GetEvent	E_OK	E_OS_ID, E_OS_ACCESS, E_OS_STATE, E_OS_CALLEVEL

Table A.1 System Service Return Codes (continue)

System Service	Standard Error Status	Additional Codes in Extended Error
WaitEvent	E_OK	E_OS_ACCESS, E_OS_RESOURCE, E_OS_CALLEVEL
GetAlarmBase	E_OK	E_OS_ID, E_OS_CALLEVEL
GetAlarm	E_OK, E_OS_NOFUNC	E_OS_ID, E_OS_CALLEVEL
SetRelAlarm	E_OK, E_OS_STATE	E_OS_ID, E_OS_VALUE, E_OS_CALLEVEL
SetAbsAlarm	E_OK, E_OS_STATE	E_OS_ID, E_OS_VALUE, E_OS_CALLEVEL
CancelAlarm	E_OK, E_OS_NOFUNC	E_OS_ID, E_OS_CALLEVEL
InitialiseCounter*	E_OK	E_OS_ID, E_OS_CALLEVEL, E_OS_VALUE
IncrementCounter*	E_OK, E_OS_LIMIT, E_OS_MULTIPLE	E_OS_ID, E_OS_STATE, E_OS_CALLEVEL
GetActiveApplicationMode	---	---
StartOS	---	---
ShutdownOS	---	E_OS_CALLEVEL*

Note: *: Original function.

A.2 IDs

A.2.1 Return Code IDs

Table A.2 Return Codes

Error	**Description**	**ID**
E_OK	No error	0 (H'0)
E_OS_ACCESS	- Inadmissible access to resource - Referenced task is not an extended task - Call from basic task	1 (H'1)
E_OS_CALLEVEL	Call from invalid level	2 (H'2)
E_OS_ID	ID is invalid	3 (H'3)
E_OS_LIMIT*	Multiple task activations	4 (H'4)
E_OS_NOFUNC	- Object not in use - No resource has been got or illegal resource release order (another resource has to be released first) - The interrupt is already in whichever state was requested (enabled or disabled)	5 (H'5)
E_OS_RESOURCE	Task still occupies resources	6 (H'6)
E_OS_STATE	- Task is in the suspended state - Alarm is in use	7 (H'7)
E_OS_VALUE	- Illegal count value (lower than zero, or greater than **maxallowedvalue**) - Illegal cycle (lower than **mincycle** or greater than **maxallowedvalue**)	8 (H'8)
E_OS_INVALID_NEST*	Illegal resource nest	9 (H'9)
E_OS_MULTIPLE*	**E_OS_STATE** or **E_OS_LIMIT** occurs two or more times	10 (H'a)
E_OS_SYS_EXCEPTION*	Undefined exception	100 (H'64)
E_OS_SYS_ILLEGAL*	Illegal general instruction	101 (H'65)
E_OS_SYS_ILLEGAL_SLOT*	Illegal slot instruction	102 (H'66)
E_OS_SYS_CPU_ADDRESS*	CPU address error	103 (H'67)

Note: *: Original function.

A.2.2 System Service IDs

Table A.3 System Service IDs

System Service	ID
DeclareTask	0 (H'0)
ActivateTask	1 (H'1)
TerminateTask	2 (H'2)
ChainTask	3 (H'3)
Schedule	4 (H'4)
GetTaskID	5 (H'5)
GetTaskState	6 (H'6)
EnterISR	7 (H'7)
LeaveISR	8 (H'8)
EnableInterrupt	9 (H'9)
DisableInterrupt	10 (H'a)
GetInterruptDescriptor	11 (H'b)
DeclareResource	12 (H'c)
GetResource	13 (H'd)
ReleaseResource	14 (H'e)
DeclareEvent	15 (H'f)
SetEvent	16 (H'10)
ClearEvent	17 (H'11)
GetEvent	18 (H'12)
WaitEvent	19 (H'13)
DeclareAlarm	20 (H'14)
GetAlarmBase	21 (H'15)
GetAlarm	22 (H'16)
SetRelAlarm	23 (H'17)
SetAbsAlarm	24 (H'18)
CancelAlarm	25 (H'19)
GetActiveApplicationMode	26 (H'1a)
StartOS	27 (H'1b)
ShutdownOS	28 (H'1c)
InitialiseCounter*	29 (H'1d)
IncrementCounter*	30 (H'1e)
NonVariant (Non-variant alarm timer counter)	31 (H'1f)
EnableInterruptMask*	32 (H'20)
DisableInterruptMask*	33 (H'21)
GetInterruptDescriptorMask*	34 (H'22)
EnableInterruptSource*	35 (H'23)
DisableInterruptSource*	36 (H'24)
GetInterruptDescriptorSource*	37 (H'25)

Note: *: Original function.

A.2.3 Context IDs

Table A.4 Context IDs

Context	**ID**
OS	0 (H'0)
OS_IDLE	1 (H'1)
TASK	2 (H'2)
ISR	3 (H'3)
ERRORHOOK	4 (H'4)
PRETASKHOOK	5 (H'5)
POSTTASKHOOK	6 (H'6)
SHUTDOWNHOOK	7 (H'7)
STARTUPHOOK	8 (H'8)

A.3 System Service Calls

System service that can be issued is shown by ✔in task, ISR, and hook routine.

Table A.5 System Service Calls

Service	Task Level	ISR Level	Error Hook	PreTask Hook	PostTask Hook	Startup Hook	Shutdown Hook	OS calling program
ActivateTask	✔	✔	---	---	---	✔	---	---
TerminateTask	✔	---	---	---	---	---	---	---
ChainTask	✔	---	---	---	---	---	---	---
Schedule	✔	---	---	---	---	---	---	---
GetTaskID	✔	---	✔	✔	✔	---	---	---
GetTaskState	✔	✔	✔	✔	✔	---	---	---
EnableInterrupt	✔	✔	---	---	---	---	---	---
DisableInterrupt	✔	✔	---	---	---	---	---	---
GetInterruptDescriptor	✔	✔	✔	✔	✔	---	---	---
GetResource	✔	---	---	---	---	---	---	---
ReleaseResource	✔	---	---	---	---	---	---	---
SetEvent	✔	✔	---	---	---	---	---	---
ClearEvent	✔	---	---	---	---	---	---	---
GetEvent	✔	✔	✔	✔	✔	---	---	---
WaitEvent	✔	---	---	---	---	---	---	---
GetAlarmBase	✔	✔	✔	✔	✔	---	---	---
GetAlarm	✔	✔	✔	✔	✔	---	---	---
SetRelAlarm	✔	✔	---	---	---	---	---	---
SetAbsAlarm	✔	✔	---	---	---	---	---	---
CancelAlarm	✔	✔	---	---	---	---	---	---
InitialiseCounter	✔	✔	---	---	---	---	---	---
IncrementCounter	✔	✔	---	---	---	---	---	---
GetActiveApplicationMode	✔	✔	✔	✔	✔	✔	✔	---
StartOS	---	---	---	---	---	---	---	✔
ShutdownOS	✔	---	---	---	---	---	---	---
EnableInterruptMask	✔	✔	---	---	---	---	---	---
DisableInterruptMask	✔	✔	---	---	---	---	---	---
GetInterruptDescriptorMask	✔	✔	✔	✔	✔	---	---	---
EnableInterruptSource	✔	✔	---	---	---	---	---	---
DsiableInterruptSource	✔	✔	---	---	---	---	---	---
GetInterruptDescriptorSource	✔	✔	✔	✔	✔	---	---	---

A.4 Data Types

Table A.6 Data Types

Name	Data Type (size)	Description
StatusType	unsigned long (4 bytes)	Return value
TaskType	signed short (2 bytes)	Task ID
TaskRefType	TaskType * (4 bytes)	Pointer to TaskType
TaskStateType	signed char (1 byte)	Task status
TaskStateRefType	TaskStateType * (4 bytes)	Pointer to TaskStateType
IntDescriptorType	unsigned long (4 bytes)	Status register value
IntDescriptorRefType	IntDescriptorType * (4 bytes)	Pointer to IntDescriptorType
ResourceType	signed char (1 byte)	Resource ID
EventMaskType	signed char (1 byte)	Event mask value
EventMaskRefType	EventMaskType * (4 bytes)	Pointer to EventMaskType
TickType	unsigned long (4 bytes)	Counter value
TickRefType	TickType * (4 bytes)	Pointer to TickType
AlarmBaseType	struct AlarmBase (12 bytes)	Element of the counter characteristics
AlarmBaseRefType	AlarmBaseType * (4 bytes)	Pointer to AlarmBaseType
AlarmType	signed char (1 byte)	Alarm ID
AppModeType	signed char (1 byte)	Application mode ID
CounterType	signed char (1 byte)	Counter ID
APICall	unsigned long (4 bytes)	System service ID
ContextType	unsigned long (4 bytes)	Context ID

Ho7055 Operating System Manual

Publication date: 1st Edition, May 2001