Detailed Design for

AUTOSAR OS

Issue 4.0.10

The Management team listed agrees to supply the necessary capital and engineering resources to support the project requirements described in this document		
Printed Name	**Signature**	**Sign-off Date**
Project Manager - Irina Bratanova	/signed/	17-Jan-2013
RTOS SW Architect - Boris Guzhov	/signed/	17-Jan-2013

This Document was prepared by:
Name: Boris Guzhov
Title: Senior Engineer
Location: St.Peterburg, Russia
Phone: +7 (812) 324-7222
Email: Boris.Guzhov@freescale.com

Date: 29 December 2012
Printed from: DetailedDesign.pdf
Security Classification: Freescale Confidential

Distribution List:

- Andrei Kovalev
- Boris Guzhov
- Rakhim Mayorov
- German Semenov
- Arthur Lebedev
- Oleg Oreshko
- Konstantin Zemskov
- Alexey Artamonov
- Marina Gorshkova

Change History

Issue	Description	Date	Author
3.0.1	First version	30 Jul 2010	A.Troshikhin B.Guzhov M. Gorshkova A.Perch R. Mayorov
3.0.2	Chapter IOC added; references tag added	20 Jan 2011	M. Gorshkova S. Mozhaev
4.0.3	Updated for AUTOSAR 4.0	11 Apr 2011	R. Mayorov
3.0.4	Schedule Table chapter updated	12 Aug 2011	A.Perch
4.0.5	MPU Descriptors tables corrected	25 Oct 2011	M. Gorshkova
3.0.6	Misprints corrected	09 Dec 2011	M. Gorshkova
4.0.7	Trusted function description updated	02 Apr 2012	R. Mayorov M. Gorshkova
4.0.8	Updated to new codebase	12 Oct 2012	M. Gorshkova
4.0.10	Updated for OS/MPC56xxL v.4.0	29 Dec 2012	M. Gorshkova

Contents

Introduction

Purpose

{DD_286}
This document describes the software architecture, design features, functions, and interfaces of Freescale AUTOSAR OS, an implementation of the real-time operating system compliant with:

[2], (AUTOSAR Requirements on Operating System v.5.0.0 (AUTOSAR SRS OS), R.4.0, rev 3).

Operating System should have ORTI implementation compliant with

[7], (OSEK/VDX OSEK Run Time Interface (ORTI) Part A: Language Specification Version: 2.2, 14 Nov 2005)

and

[8], (OSEK/VDX OSEK Run Time Interface (ORTI) Part B: OSEK Objects and Attributes, Version 2.2, 25 Nov 2005).

System generator should be compliant with: {DD_287}
[6], (OSEK/VDX System Generation OIL: OSEK Implementation Language, Version 2.5, July 1, 2004).

The document is primarily intended to be used by the software engineering team designing this product.The document includes a detailed description of the Freescale AUTOSAR OS.

Definitions and Acronyms

API	Application Program Interface
AUTOSAR	Automotive Open System Architecture
ISR	Interrupt Service Routine
MCU	MicroController Unit
OIL	OSEK Implementation Language
OS	Operating System, product implementing OSEK OS specification
OSEK	Open systems and their corresponding interfaces for automotive electronics (in German)
OSEK/VDX	OSEK (See above) and Vehicle Distributed eXecutive
SysGen	System Generation utility

Document navigation

-» .....Cross-Reference; to navigate inside the document click on this symbol

References

The following documents are referenced in this specification:

[1] AUTOSAR Requirements on Operating System (AUTOSAR SRS OS)

[2] AUTOSAR Requirements on Operating System v.5.0.0 (AUTOSAR SRS OS), R.4.0, rev 3

[3] Statement of Work for AUTOSAR OS, issue 1.0.0

[4] American National Standard for Information System - Programming Language C, X3. 159-1989

[5] OSEK/VDX Operating System, Version 2.2.3, February 17, 2005

[6] OSEK/VDX System Generation OIL: OSEK Implementation Language, Version 2.5, July 1, 2004

[7] OSEK/VDX OSEK Run Time Interface (ORTI) Part A: Language Specification Version: 2.2, 14 Nov 2005

[8] OSEK/VDX OSEK Run Time Interface (ORTI) Part B: OSEK Objects and Attributes, Version 2.2, 25 Nov 2005

[9] Freescale AUTOSAR Common PRD, issue 1.9

[10] AUTOSAR BSW Makefile Interface v.0.3, 03 June 2005

Design Goals

Quality

The OS shall conform to AUTOSAR standard.

Criteria

The main criterion of developing is performace of the OS with reasonable consumption of RAM and ROM.

The second criterion is RAM consumption by the OS data.

The third criterion is RAM consumption by the OS code.

The OS shall be written in correspondence with MISRA C Guidelines, all deviations from MISRA shall be documented.

Task Management

AUTOSAR OS provides a set of services for the user to manage tasks. The tasks are activated before their execution - they are transferred to ready state. Memory resources needed for a task are allocated at system start-up. Tasks may be terminated after needed actions were performed - tasks are transferred to suspend state. After a task is activated it may run and terminate or it may be implemented in an endless loop with at least one synchronizing system call. It is not possible to run several parallel endless loops without switching mechanism (scheduler). Tasks can be switched during their execution from one to another, or may be interrupted by ISR. If no task is active, only the scheduler idle loop runs. Two different task concepts are provided by the AUTOSAR OS:

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

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

Extended Tasks

{DD_001}

Extended Tasks have four task states:

Running

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

Ready

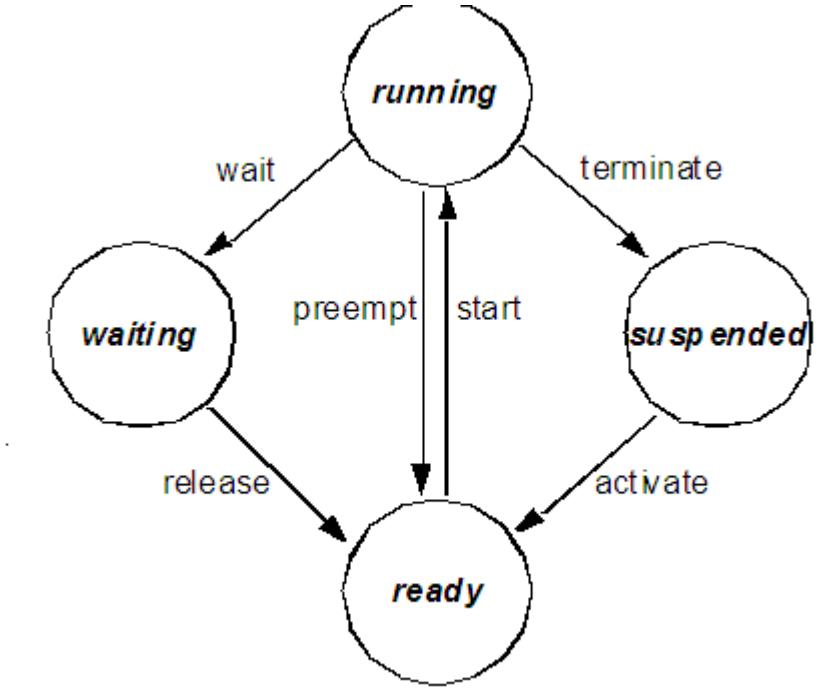
All functional prerequisites for a transition into the running state exist, and the task only waits for allocation of the processor. The scheduler decides which ready task is executed next.

Waiting

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

Suspended

In the suspended state, the task is passive and does not take a part in any OS activities. In this state task can be activated.



Termination of tasks is only possible if the task terminates itself ("self-termination"). This restriction is to avoid complex book-keeping of resources dynamically allocated to the task. There is no provision for a direct transition from the suspended state into the waiting state. This transition is redundant and would add to the complexity of the scheduler. The waiting state is not directly entered from the suspended state, as the task starts and explicitly enters the waiting state on its own.

Basic Tasks

{DD_002}

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

Running

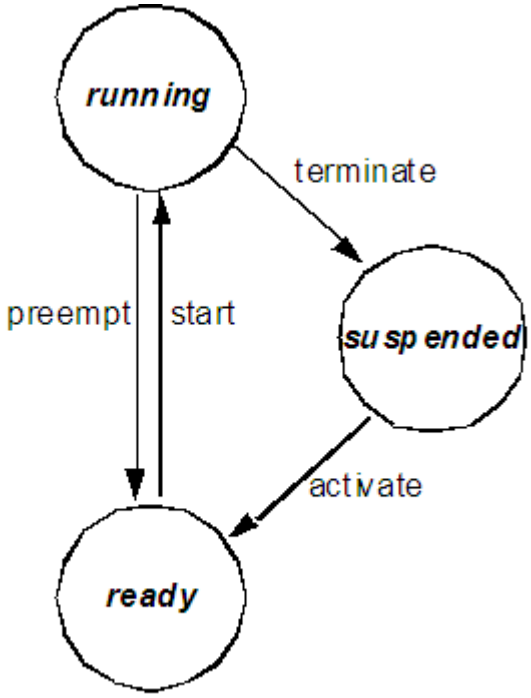
In the running state, the CPU is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any point in time, while several tasks can be in any of the other states simultaneously.

Ready

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

Suspended

In the suspended state, the task is passive and does not take a part in any OS activities. In this state task can be activated.



Data Types

{DD_003}

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

- -»**TaskType** The abstract data type for task identification;
- -»**TaskRefType** The data type to refer the variables of the TaskType data type. Reference to the TaskType variable can be used instead of the TaskRefType variable;
- -»**TaskStateType** The data type for the variables for storage of the task state;
- -»**TaskStateRefType** The data type to refer the variables of the TaskStateType data type. Reference to the TaskStateType variable can be used instead of the TaskStateRefType variable.

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

The following data types are used by the OS internally:

- -»TagOSTSK - task configuration table structure
- -»TagOSTSKCB - task control block

The OS uses following global objects to handle tasks:

If the usage of Task Control Block is allowed (fast termination is not configured or OS Resources are configured in the OS):

- `const -»OSTSK -»OsTaskCfgTable[-»OSNTSKS]` - the task configuration table. It's generated by the SysGen.
- `-»OSTSKCB -»OsTaskTable[-»OSNTSKS+1]` - the task control blocks table.

If the usage of Task Control Block is not allowed:

- `const -»OSBYTE -»OsTaskProperty[-»OSNTSKS]` - Tasks' properties. It's generated by the SysGen.
- `-»OSBYTE -»OsTaskStatus[-»OSNTSKS]` - Tasks status.

The macro **OSNTSKS** is used by the OS to access to the task configuration table. The macro is defined as the number of TASK objects in the OS configuration.

Constants

{DD_004}

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

- **RUNNING** The constant of data type TaskStateType for task state running
- **WAITING** The constant of data type TaskStateType for task state waiting
- **READY** The constant of data type TaskStateType for task state ready
- **SUSPENDED** The constant of data type TaskStateType for task state suspended These constants can be used for the variables of TaskStateType.

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

- **INVALID_TASK**

  The constant of data type Task Type for an undefined task

Task Management functions

Within the application a task is defined according to the following pattern: {DD_288}

TASK (<name of task>)

{

}

DeclareTask

{DD_289}

void DeclareTask(TaskID)

A dummy declaration, intended for compatibility with other OSEK versions

OSInitTasks

{DD_005}

void OSInitTask (-»AppModeType *mode*)

for OSNAPPMODES > 1

else

void OSInitTask()

Initializes tasks

Parameters:

[in] *mode* The OS-Application mode (if defined **OSNAPPMODES** > 1)

Returns:

none

Note:

none

Implementation

- If the OS uses Task Control Block init task's control blocks

  - If OS Resources are configured

    – Clear Priority Link table -»OsPrioLink

  - Fill the task control blocks table -»OsTaskTable using the task configuration table -»OsTaskCfgTable generated by the SysGen:

    – If the memory protection is configured

      - Copy application identification mask value

      - Copy application identification value

      - Copy application identification value

    – Copy entry point of task

    – Copy task identifier

    – If OS internal Resources are configured

      - Copy running priority

    – If OS Resources or OS internal Resources are configured

      - Init Priority Link table -»OsPrioLink

    – Init task status (’first run of task’ | ’task status from the task configuration table’)

    – Init the top and the bottom of task stack

    – Clear set_event field

    – If OS Resources are configured

      - Clear the head of the occupied resources list

    – Activate auto-startup task: set task status as non-suspended and set bits in -»OsScheduleVector(s) to dispatch the Scheduler

    – Initialize control block for NULLTASK (background task when no activated tasks)

- Else initialize -»OsTaskStatus from -»OsTaskProperty
- Initialize -»OsRunning to NULLTASK (no running task yet)

Used data

-»OsTaskTable

-»OsTaskCfgTable

-»OsPrioLink

-»OsRunning

-»OsTaskStatus

-»OsTaskProperty

OSSCActivateTask

OSINLINE **void OS_ActivateTask (-»OSWORD *taskInd*)**

activates the Task, make it RUNNING if conditions are met

Parameters:

[in] *taskInd* - a task index in OsTaskTable array

Returns:

none.

Implementation

- If the OS uses Task Control Block or there are Internal Resources
  - Clear field '*set_event*' in the task control block (ECC and number of extended TASK objects > 0)
  - Set the corresponding status in the Task Control Block (the first run of task and the task is not in suspended state)
- Set bits in -»OsScheduleVector(s) to dispatch the Scheduler

Used data

-»OsTaskTable

-»OsSchedulerVector1

-»OsSchedulerVector2

OSMCActivateTask

{DD_316}

-»StatusType OSMCActivateTask (-»OSWORD *taskInd*, *OSAPPLICATIONTYPE appId*)

activates the Task, make it RUNNING if conditions are met

Parameters:

[in] *taskInd* - a task index in OsTaskTable array

[in] *appId* - an ID of calling application

Returns:

Standard:
E_OK no error.
E_OS_LIMIT too many task activations of the specified task.

E_OS_ACCESS insufficient access rights

Implementation

- Check access rights of calling application to givent task
- If activated task is already activated return error code E_OS_LIMIT
- Call OSSCActivateTask() function
- Return E_OK

Used data

-»OsTaskTable

-»OsScheduleVector1

-»OsScheduleVector2

OS_ActivateTask

{DD_006}

-»StatusType OS_ActivateTask (-»TaskType *taskId*)

activates the Task, make it RUNNING if conditions are met

Parameters:

[in] *taskId* - a reference to the task

Returns:

Standard:
E_OK no error.
E_OS_LIMIT too many task activations of the specified task.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the task identifier is invalid.

E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

The specified task <TaskID> is transferred from the suspended state into the ready state.

Particularities: The service may be called both on the task level (from a task) and the interrupt level (from ISR). In the case of calling from ISR, the operating system will reschedule tasks only after the ISR completion.

Implementation

- Check context of service call and given task index
- For Extended status:
  - Get the task index in Tasks tables
  - Return error code E_OS_ID if the task has the invalid ID
- If given task belong to the another core
  - perform remote call by calling OSRemoteCall1 macro
  - Call -»DISPATCH function
  - Leave OS critical section via macro *OSRI(),* return status and leave service
- else (given task belong to the same core)
  - Enter OS critical section via *OSDIS()*
  - If access rights of current OS-Application to given task are insufficient or state of OS-Application to which belong given task is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- If activated task is already activated
  - Leave the OS critical section via the macro *OSRI()*
  - Return error code E_OS_LIMIT
- Call -»OSSCActivateTask() function
- Call -»OSDISPATCH() function
- Leave the OS critical section via the macro *OSRI()*
- Return E_OK

Used data

-»OsTaskTable

-»OsIsrLevel

OS_TerminateTask

{DD_007}

-»StatusType OS_TerminateTask (void)

Terminate the running task.

Returns:

Extended:
E_OS_RESOURCE the task still occupies resources.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_SPINLOCK unreleased spinlock.

Note:

Do not return on success

This service causes the termination of the calling task. The calling task is transferred from the running state into the suspended state.
Particularities: The resources occupied by the task shall be released before the call to TerminateTask service. If the call was successful, TerminateTask does not return to the call level and enforces a rescheduling. Ending a task function without strictly forbidden. If the system with extended status is used, the service returns in case of error, and provides a status which can be evaluated in the application. There are the following limitations for BCC1 class if FastTerminate is set to TRUE: TerminateTask service shall be called in task function body from the function level; in STANDARD status this service does not return a status and can not be used in expressions. The service call is allowed on task level only.

Implementation

- For extended status:
  - check if the current running task still occupies resources; return *E_OS_RESOURCE*
  - check if the current running task still occupies spinlocks; return *E_OS_SPINLOCK*
- Enter the OS critical section via the macro *OSDI()*
- Call -»OSTaskTerminateDispatch (force dispatcher for terminate case)
- Never return from this function

Used data

-»OsRunning

OSMCChainTask

{DD_317}

-»StatusType OS_ChainTask (-»OSWORD *taskInd*, -»OSAPPLICATIONTYPE *appId*)

Start the given task.

Parameters:

[in] *taskInd* - a task index in OsTaskTable array
[in] *appId* - an ID of calling application

Returns:

Standard:
E_OK no error.
E_OS_LIMIT too many task activations of the specified task.

E_OS_ACCESS insufficient access rights

Implementation

- Check access rights of calling application to givent task
- If activated task is already activated return error code E_OS_LIMIT
- Clear field '*set_event*' in the task control block (if the number of extended TASK objects > 0)
- Set the corresponding status in the Task Control Block (the first run of task and the task is not in suspended state)
- Set bits in -»OsScheduleVector(s) to dispatch the Scheduler
- Return E_OK

Used data

-»OsTaskTable

-»OsScheduleVector1

-»OsScheduleVector2

OS_ChainTask

{DD_008}

-»StatusType OS_ChainTask (-»TaskType *taskId*)

Terminate the running task and start the given task.

Parameters:

[in] *taskId* - a reference to the sequential succeeding task to be activated.

Returns:

Standard:
E_OK no error.
E_OS_LIMIT too many task activations of the specified task.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the task identifier is invalid.
E_OS_RESOURCE the task still occupies resources.

E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_SPINLOCK unreleased spinlock.

Note:

Do not return on success

This service causes the termination of the calling task. After termination of the calling task a succeeding task <TaskID> is transferred from the suspended state into the ready state. Using this service ensures that the succeeding task only starts to run after the calling task has been terminated.
Particularities: The resources occupied by the calling task shall be released before the call to ChainTask service. If the call was successful, ChainTask does not return to the call level and enforces a rescheduling. Ending a task function without calling TerminateTask or ChainTask service is strictly forbidden. If the succeeding task is identical to the current task, this does not result in multiple requests. The service returns in case of error and provides a status which can be evaluated by the application. There are the following limitations for BCC1 class if FastTerminate is set to TRUE: ChainTask service shall be called in task function body from the function level; in STANDARD status this service does not return a status and can not be used in expressions. The service call is allowed on task level only.

Implementation

- Check context of service call and given task index
- For Extended status:
  - Get the succeeding task index in Tasks tables
  - check if the current running task still occupies resources; return *E_OS_RESOURCE*
  - check if the current running task still occupies spinlocks; return *E_OS_SPINLOCK*
  - Return error code E_OS_ID if the succeeding task has the invalid ID
- If given task belong to the another core
  - perform remote call by calling OSChainTaskRC macro
  - Leave OS critical section via macro *OSRI()*, return status and leave service
- else (given task belong to the same core)
  - Enter OS critical section via *OSDI()*
  - If access rights of current OS-Application to given task are insufficient or state of OS-Application to which belong given task is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSEI()* and return E_OS_ACCESS error code
- If succeeding task is already activated
  - Leave the OS critical section via the macro *OSEI()*
  - Return error code E_OS_LIMIT
- Clear field '*set_event*' in the task control block (if the number of extended TASK objects > 0)
- Set the corresponding status in the Task Control Block (the first run of task and the task is not in suspended state)
- Set bits in -»OsScheduleVector(s) to dispatch the Scheduler
- Call -»OSTaskTerminateDispatch (force dispatcher for terminate case)
- Never return from this function

Used data

-»OsIsrLevel

-»OsTaskTable

-»OsRunning

-»OsSchedulerVector1

-»OsSchedulerVector2

OS_GetTaskID

{DD_009}

-»StatusType OS_GetTaskID (-»TaskRefType *taskIdRef*)

Return the index, corresponding to running task.

Parameters:

[in] *taskIdRef* - a pointer to the variable contained reference to the task which is currently running

Returns:

Standard:
E_OK no error.
Extended:
E_OS_ILLEGAL_ADDRESS illegal <taskIdRef> (only for SC3, SC4)
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

This service returns reference to the task which is currently running. If there is no task in the running state, the service returns INVALID_TASK into the variable.
Particularities: The service call is allowed on task level, ISR level and in ErrorHook, PreTaskHook and PostTaskHook hook routines.

Implementation

- Check context of service call and given task index
- Check the possibility to write into given pointer 'taskIdRef'
- Get the index corresponding to running task to the pointer 'taskIdRef'

Used data

-»OsRunning

OSSCGetTaskState

-»void OSSCGetTaskState (-»OSWORD *taskInd*, -»TaskStateRefType *stateRef*)

Return the status of the given task.

Parameters:

[in] *taskInd* - a tak index in OSTaskTable array
[in] *stateRef* - a pointer to the state of task

Returns:

none

Implementation

- If the index of the current running task (-»OsRunning) is equal the index of the specified task <TaskID>, the state -»`RUNNING` is saved by the pointer 'stateRef'
- else if -»OsTaskTable[<TaskID>].status is "waiting", the state -»`WAITING` is saved by the pointer 'stateRef' (only if ECC is defined and the number of extended TASK objects > 0)
- Else
  - If -»OsTaskTable[<TaskID>].status is "no suspended", the state -»`READY` saved by the pointer 'stateRef'
  - Else the state -»`SUSPENDED` is saved by the pointer 'stateRef'

OS_GetTaskState

{DD_010}

-»StatusType OS_GetTaskState (-»TaskType *taskId*, -»TaskStateRefType *stateRef*)

Return the status of the given task.

Parameters:

[in] *taskIdRef* - a reference to the task
[in] *stateRef* - a pointer to the state of task

Returns:

Standard:
E_OK no error.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.

Extended:
E_OS_ID the task identifier is invalid.
E_OS_ILLEGAL_ADDRESS illegal <stateRef> (only for SC3, SC4).
E_OS_CALLEVEL call at not allowed context.

E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

The service returns the state of the specified task <TaskID> (running, ready, waiting, suspended) at the time of calling GetTaskState.
Particularities: The service may be called both on the task level (from a task) and the interrupt level (from ISR). This service may be called from ErrorHook, PreTaskHook, PostTaskHook hook routines. Within a full-preemptive system, calling this operating system service only provides a meaningful result if the task runs in an interrupt disabling state at the time of calling. When a call is made from a task in a full-preemptive system, the result may already be incorrect at the time of evaluation. When the service is called for a task, which is multiply activated, the state is set to running if any instance of the task is running.

Implementation

- Check context of service call and given task index
- Check access rights and the possibility to write into given pointer 'stateRef'
- If given task belong to the another core, perform remote call by calling OSRemoteCall2 macro
- else (given task belong to the same core)
- Enter OS critical section via *OSDIS()*
- Call OSSCGetTaskState function
- Leave the OS critical section via the macro *OSRI()*
- Return error code E_OK.

Used data

-»OsRunning

-»OsTaskTable

OSCheckStack

{DD_011}

void OSCheckStack (void)

Check task stack overflow for preempted Task.

Returns:

none

Note:

none

Particularities: The service defined only if the OS checks stacks (no User check functions)

Implementation if stack defined by -»TagOSTSK

- Check whether the index of the running task is not equal 0
- Get the bottom of the stack of the running task from the structure -»TagOSTSK and check whether the stack is filled with the correct pattern.
- If the stack is invalid

  - If memory protection feature is configured

  – If hook protection is defined

  - save violator Ids by special global variables (-»OsViolatorAppId, -»OsViolatorTaskId, -»OsViolatorISRId)

  - Call ProtectionHook with the parameter *E_OS_STACKFAULT*

  - Against protection return type the following actions are executed:

  - *PRO_KILLTASKISR*: -»OSKillTask called

  - *PRO_KILLAPPL:* call -»OSTerminateApplication with -»NO_RESTART parameter;

  - *PRO_KILLAPPL_RESTART*: call -»OSTerminateApplication with -»RESTART parameter;

  If -»OSTerminateApplication returns *OSFALSE*, call -»OSShutdownOS with parameter *E_OS_STACKFAULT*

– Else (hook protection is not defined) call -»OSShutdownOS with parameter *E_OS_STACKFAULT*

- Else (memory protection is not configured) call -»OSShutdownOS with parameter *E_OS_STACKFAULT*

Used data

-»OsRunning

-»OsTaskTable

-»OsViolatorAppId

-»OsViolatorTaskId

-»OsViolatorISRId

Implementation if stack defined by -»OSSTACKBOTTOM

- Check whether the stack defined in -»OSSTACKBOTTOM is filled with the correct pattern.
- If stack is invalid call -»OSShutdownOS with parameter *E_OS_STACKFAULT*

OSCheckIsrStack

{DD_011}

void OSCheckIsrStack (void)

Check ISR stack overflow.

Returns:

none

Note:

For SC2,3,4 check is done in Interrupt dispatcher

Particularities: The service defined only if the OS checks stacks (no User check functions). If there are ISR category 2 or Systimer then common ISR stack is used in SC2 and SC1, ECC1. This service is defined in this case.

Implementation

- Check whether the stack defined in OSISRSTKEND (-»OsISRStack) is filled with the correct pattern.
- If stack is invalid call -»OSShutdownOS with parameter *E_OS_STACKFAULT*

OSGetUsedBytes

{DD_012}

-»OSWORD OSGetUsedBytes (const -»OSDWORD * *_start*, -»OSWORD *num*)

Calculates number of used bytes.

Parameters:

[in] *_start* The start stack pointer
[in] *num* The size in long words

Returns:

Number of bytes

Note:

none

Implementation

- Calculate the number of double words (-»OSDWORD) of the stack filled with the correct pattern
- Return the number of bytes

GetRunningStackUsage

{DD_013}

-»OSWORD GetRunningStackUsage (void)

Calculates running task stack usage.

Returns:

stack usage in bytes;
OSSTKNOASSIGNED if Task has no own stack

Note:

none

Implementation

- If the number of extended Tasks is equal 0, returns *OSSTKNOASSIGNED*
- Returns *OSSTKNOASSIGNED* if there is no running task
- Returns *OSSTKNOASSIGNED* if there the running task is BASIC task (it has single stack)

- If the memory protection feature is defined and -»OsIsrLevel is used for context check, returns *OSSTKNOASSIGNED* if ErrorHook flag is set for ISR level
- Call -»OSGetUsedBytes to calculate the number of used bytes in stack for the current running task

Used data

-»OsRunning

GetStackUsage

{DD_014}

-»OSWORD GetStackUsage (-»TaskType *taskId*)

Calculates task stack usage.

Parameters:

[in] *taskId* - a reference to the task

Returns:

stack usage in bytes;
OSSTKNOASSIGNED if Task has no own stack

Note:

none

Particularities: The service defined only if the memory protection feature is not configured**.**

Implementation

- Returns *OSSTKNOASSIGNED* in case of invalid Task Id
- If the number of extended Tasks is equal 0, returns *OSSTKNOASSIGNED*
- Returns *OSSTKNOASSIGNED* if there the running task is BASIC task (it has single stack)
- call -»OSGetUsedBytes to calculate the number of used bytes in stack for the task with Id <taskId>

Used data

-»OsTaskTable

OSKillAppTasks

{DD_015}

void OSKillAppTasks (void)

Kills all tasks which belong to current application.

Returns:

none

Note:

none

Particularities: The service defined only if OS-Applications are configured.

Implementation

- Get all tasks of the application using -»OsAppTable.tasks (-»OsAppTable.tasks2). It is the scheduler vector
- Get task Ids from the scheduler vector to dispatch the Scheduler
- If the task is non-suspended state, call -»OSKillTask

Used data

-»OsAppTable

-»OsTaskTable

OSReleaseTaskResource

{DD_016}

OSINLINE void OSReleaseTaskResource (const -»OSTSKCBPTR *taskPtr*)

Releases resources taken by the task.

Parameters:

[in] *taskPtr* The reference to the Task

Returns:

none

Note:

Used for TerminateApplication service

Particularities: The service defined only for Extended status.

Implementation

- Get the pioneer to resource object (-»TagOSRESCB) occupied of the task
- While the task occupies the resource

  - If ISR resource are configured

    – If the resource priority < 0 (it is the ISR resource)

      - Decrement -»OsISRResourceCounter (ISR resource counter)

      - If it is equal 0 (it is the first ISR resource), unblock scheduler (set the corresponding bit in -»OsIsrLevel)

  - Else (if ISR resource are not configured or the resource priority >= 0, i.e. it is the task's resource) get the task Id referenced to this resource

  - If the task is non-suspended, clear bit in -»OsScheduleVector(s) to dispatch the Scheduler

  - Clear flag -»TagOSRESCB.isUsed (the resource is not used)

  - Get the next item from the resource list (set the head of the occupied resources list of the task -»TagOSTSKCB.resources to the next resource -»TagOSRESCB.nextRes)

Used data

-»OsISRResourceCounter

-»OsIsrLevel

-»OsTaskTable

OSKillRunningTask

{DD_017}

void OSKillRunningTask (void)

Kills running Task and performs appropriate cleanup.

Returns:

none

Note:

Shall be called with TP interrupts disabled intended for use in the task dispatcher, shall not be used by protection handlers.

Implementation

- Clear status flags for the current running task
- Set flag that task has no valid context yet
- If the memory protection feature is configured
  - Restore appId in case it was changed (from -»OsTaskCfgTable)
- If OS Resources are configured
  - Call -»OSReleaseTaskResource with parameter -»OsRunning
- Clears bit in scheduler vector (to dispatch the Scheduler) if there is no ready task of running priority, resets Task prio to assigned; clear bit in -»OsScheduleVector(s)
- If system ISRs are used for Remote Calls

  Clear ISR level and context bits: set -»OsIsrLevel to 0

  Reset timing protection for OS interrupt lock time

  Clear level of the nested Suspend / Resume pairs: set -»OsSuspendLevel to 0
- Else

  Clear all context bits (-»OsIsrLevel)
- Clear level of the nested Suspend / ResumeAll pairs: set -»OsSuspendLevelAll to 0
- Set MSR for Task, EE = 1

Used data

-»OsRunning

-»OsTaskCfgTable

-»OsIsrLevel

-»OsSuspendLevel

-»OsSuspendLevelAll

OSKillTask

{DD_018}

void OSKillTask (-»OSTSKCBPTR *taskPtr*)

kills any task (running or ready) and performs appropriate cleanup

Parameters:

[in] *taskPtr* The reference to the Task

Returns:

none

Note:

Called with OS and TP interrupts disabled

Particularities: The service defined only if OS feature to terminate runables via TerminateApplication().

Implementation

- Clear status flags for the task referenced by <taskPtr>
- If the memory protection feature is configured
  - Restore appId in case it was changed (from -»OsTaskCfgTable)
- If the task is the current running task (-»OsRunning)

  - If OsIsrLevel is used for context check and the execution is on Task level
    - Clear ISR level and context bits: set -»OsIsrLevel to 0
  - Reset execution budget for the task
  - If system ISRs are used for Remote Calls
    - Reset timing protection for OS interrupt lock time
    - Clear level of the nested Suspend / Resume pairs: set -»OsSuspendLevel to 0
  - Clear level of the nested Suspend / ResumeAll pairs: set -»OsSuspendLevelAll to 0
  - Restore stack pad of running task
  - Enter OS critical section via *OSDI*()
  - Set MSR for Task, EE = 1
  - Clear all context bits

- Clear flags that Task was killed (-»OsKilled)

- Else (it is not running task)

  - Reset task timing protection (the budget and / or the resource locking time) for task in ready state (not running)

  - Remove the context of killed preempted basic task from the context list

  - Set flag that task has no valid context yet

  - If OS Resources are configured

    – Call -»OSReleaseTaskResource

  - If OS Resources are configured or there are internal Resources;

    – If this Task static prio points to this Task

      - Clear bit in -»OsScheduleVector(s) to dispatch the Scheduler

  - Else

    – Clear bit in -»OsScheduleVector(s) to dispatch the Scheduler

  - If there are internal Resources

    – If there is no ready task of this priority

      Clear bit in -»OsScheduleVector(s) to dispatch the Scheduler

    – If this priority was occupied by this Task and it is not it's own priority

      Updated Priority Link table -»OsPrioLink

Used data

-»OsTaskCfgTable

-»OsRunning

-»OsIsrLevel

-»OsSuspendLevel

-»OsSuspendLevelAll

-»OsPrioLink

OSResetInternalPrio

{DD_019}

OSINLINE void OSResetInternalPrio (void)

Clears bit in scheduler vector to dispatch the Scheduler if there is no OTHER ready task of running priority, resets Task prio to assigned.

Returns:

none

Note:

none

Particularities: The service defined only if there are internal Resources, otherwise empty function is defined

Implementation

- If there is no ready task of -»OsRunning priority then clear bit in OsScheduler (-»OsScheduleVector(s)) to dispatch the Scheduler
- Reset Priority Link table -»OsPrioLink

Used data

-»OsRunning

-»OsPrioLink

OSResetInternalPrio2

{DD_020}

OSINLINE void OSResetInternalPrio2 (void)

Clears bit in scheduler vector to dispatch the Scheduler if there is no ready task of running priority, resets Task prio to assigned.

Returns:

none

Note:

none

Particularities: The service defined only if there are internal Resources, otherwise empty function is defined

Implementation

- If there is no ready task of -»OsRunning priority then clear bit in OsScheduler (-»OsScheduleVector(s)) to dispatch the Scheduler
- Reset Priority Link table -»OsPrioLink

Used data

-»OsRunning

-»OsPrioLink

Data Structures

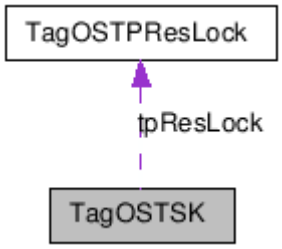
TagOSTSK Struct Reference

{DD_021}

```
#include <Os_task_internal_types.h>
```

Task configuration table

Collaboration diagram for TagOSTSK:



Data Fields

- ->OSDWORD appMask
  Application identification mask value
- ->OSTASKENTRY entry
  Entry point of task
- ->OSDWORD autostart
  Whether the task auto-started
- ->OSDWORD * tos
  Top of task stack
- ->OSDWORD * bos
  Bottom of task stack
- ->OSTPTICKTYPE tpExecBudget
  Execution budget for TASK
- ->OSTPTICKTYPE tpTimeFrame
  Time frame for task
- ->OSTPTICKTYPE tpIntLockTime

Interrupt locking time

- const ->OSTPRESLOCK * tpResLock

  Pointer to array with resource locking time configurations

- ->OSWORD tpNumberResLock

  Number of resource with locking time for this task

- ->OSBYTE runprio

  Running prio (internal resource)

- ->OSBYTE property

  Properties of task OSTSKACTIVATE, OSTSKEXTENDED, OSTSKNONPREMP

- ->TaskType taskId

  Task id (task number in the task table)

- ->OSPRIOTYPE prio

  Task priority

- ->ApplicationType appId

  Application identification value

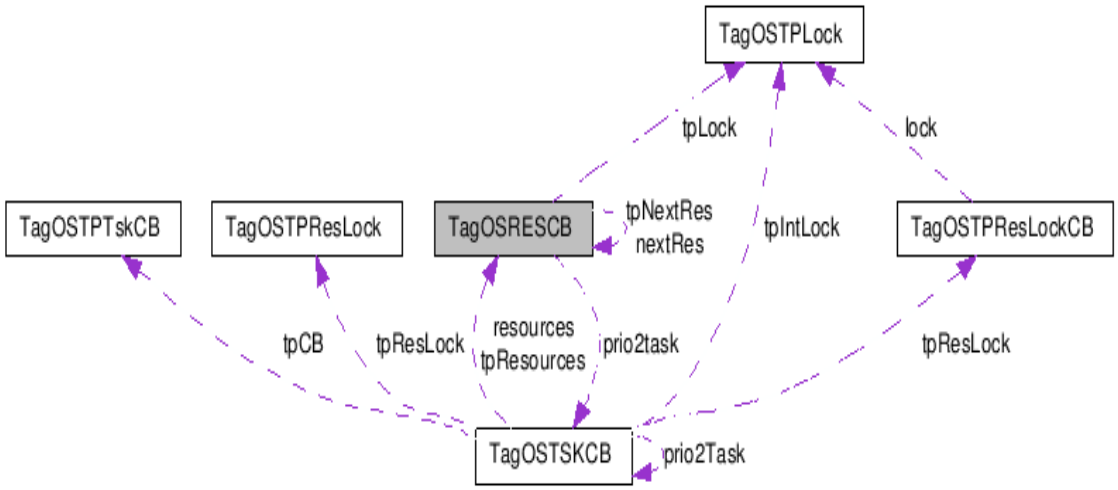
TagOSTSKCB Struct Reference

{DD_022}

```
#include <Os_task_internal_types.h>
```

Task control block structure.

Collaboration diagram for TagOSTSKCB:



Data Fields

- ->OSDWORD appMask
  Application identification mask value
- ->OSTASKENTRY entry
  Entry point of task
- ->OSDWORD * tos
  Top of task stack
- ->OSDWORD * bos
  Bottom of task stack
- ->EventMaskType wait_event
  Mask of waited events
- ->EventMaskType set_event
  Mask of set events

- ->OSJMP_BUF context
  Task context (stored by setjmp function)
- ->OSResType resources
  The head of the occupied resources list
- ->OSPRIOTYPE curPrio
  Current task priority
- ->OSBYTE runprio
  Running prio (internal resource)
- ->OSTSKCBPTR prio2Task
  References from priority to task
- ->OSBYTE status
  Task status (initial equal propery)
- ->TaskType taskId
  Task id (task number in task table)
- ->ApplicationType appId
  Application identification value
- ->OSTPTSKCB * tpCB
  TP control block
- ->OSTPLOCK * tpIntLock
  TP interrupt lock configuration
- ->OSTPRESLOCKCB * tpResLock
  TP resource lock configurations
- ->OSTPTICKTYPE tpExecBudget
  The execution budget
- ->OSTPTICKTYPE tpRemained
  Remained time on the budget
- ->OSTPTICKTYPE tpTimeFrame
  Time frame
- ->OSSIGNEDQWORD tpLast
  63-bit time of last successful transition to 'ready' state
- ->OSTPTICKTYPE tpIntLockTime

Interrupt locking time

- ->OSTPRESLOCK * tpResLock

  Pointer to array with resource locking time configurations

- ->OSResType tpResources

  The head of the list of occupied resources with started TP

- ->OSWORD tpNumberResLock

  Number of resource with locking time

- ->SpinlockIDType spinId

  Id of occupied spinlock

- ->OSPRIOTYPE prio

  Task ID in the core context

TagOSTPResLock Struct Reference

{DD_023}

```
#include <Os_tp_internal_types_v3.h>
```

TP configuration / control block of resource locking time

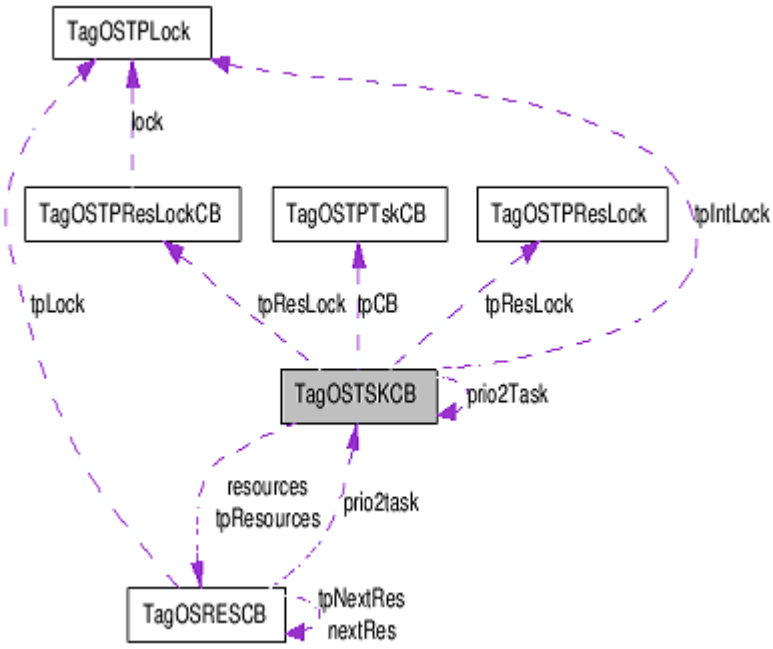
Data Fields

- -»OSTPTICKTYPE lockTime
  Resource locking time
- -»OSWORD resId
  Resource id

TagOSRESCB Struct Reference

{DD_024}

`#include <s_resource_internal_types.h>`
Collaboration diagram for TagOSRESCB:



Data Fields

- ->OSDWORD appMask
  Application identification mask value
- ->OSPRIOTYPE prio
  Resource priority for task resources
- ->OSTSKCBPTR prio2task
  To save the OsPrioLink[]/ISR last resource previous value
- ->OSINTMASKTYPE savedIsrMask
  Previous Interrupt mask
- ->OSResType nextRes
  Link to next item in the resource list
- ->OSTPLOCK * tpLock

Current TP resource configuration

- ->OSTPTICKTYPE tpRemained

  Remained time on resource locking time

- ->OSResType tpNextRes

  Link to next item in the list of resources with started TP

- ->OSBYTE isUsed

  Flag that the resource is used

- ->OSPRIOTYPE curPrio

  Current task priority

- ->ApplicationType appId

  Application identification value

TagOSTPTskCB Struct Reference

{DD_025}

```
#include <Os_tp_internal_types.h>
```

TP control block of Task execution budget

Data Fields

- ->OSObjectType object
  Object type&id: TASK
- ->OSTPTICKTYPE execBudget
  Execution budget for TASK
- ->OSTPTICKTYPE timeFrame
  Time frame for task
- ->OSTPTICKTYPE spent
  Spent time of budget
- ->OSQWORD bound
  Current time frame bound

TagOSTPLock Struct Reference

{DD_026}

```
#include <Os_tp_internal_types.h>
```

TP control block for resource or interrupt locking time

Data Fields

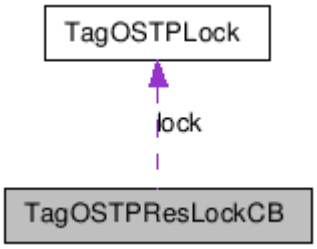
- ->OSObjectType object
  Resource type&id or OBJECT_OS_INTERRUPT_LOCK
- ->OSTPTICKTYPE lockTime
  Interrupt or resource locking time
- ->OSObjectType owner
  Owner object type&id: TASK or ISR2
- ->OSWORD index
  Owner object type&id: TASK or ISR2

TagOSTPResLockCB Struct Reference

{DD_027}

```
#include <Os_tp_internal_types.h>
```

TP control block of resource locking time
Collaboration diagram for TagOSTPResLockCB:



Data Fields

- -»OSDWORD num

  Number of resource lock configurations
- -»OSTPLOCK * lock

  Pointer to array of resource lock configurations in RAM

Alarm Management

The alarm management is built on top of the counter management. The alarm management allows the user to link task activation or event setting or a call to callback function to a certain counter value. These alarms can be defined as either single (one-shoot) or cyclic alarms. The AUTOSAR OS allows the user to set alarms (relative or absolute), cancel alarms and read information out of alarms by means of system services. Alarm is referenced via its symbolic name which is assigned to the alarm statically at the configuration stage.

Alarm Subsystem Design

{DD_028}

Alarm management is based on Counter management. The OS supports two types of Counters - Software and Hardware Counters [see Counters chapter].

All active Alarms are kept in the double-linked lists attached to each Counter. Alarms are presented via Alarm Control Blocks (see Alarm Data Types).

OS handles two internal types of Alarms - usual standard user's Alarms and internal' scheduletable Alarms'. All Alarm Control Blocks are grouped in two user's and scheduletable's arrays (see Alarm Data Types).

Alarms processing significantly depends on the type of underlining Counter.

- For Alarms on SW counters has a straight-forward design based on the fact that the underlining Counter can not change its value while being processing. All Alarms in the attached linked list are checked on each increment of the Counter. If there are alarms with expire time equal to Counter value then they are proceeded according to configured action.
- Alarms on HW Counters have more sophisticated because the HW counter continues to run regardless of interrupt disabling. A special checks at each point were decision based on counter value shall be taken to avoid race conditions when counter has been increased while processed. See descriptions of functions -»OSCounterNotify, -»OSInsertAlarm, -»OSKillAlarm.

Alarm Data Types

{DD_029}

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

- -»TickType - data type represents count values in ticks;
- -»TickRefType - the data type of a pointer to the variable of data type TickType;
- -»AlarmBaseType - the data type represents a structure for storage of counter characteristics;
- -»AlarmBaseRefType - the data type references data corresponding to the data type -»AlarmBaseType;
- **-»**AlarmType - the data type represents an alarm object.

The following data types are used by the OS internally:

- -»TagOSREFALM - common alarm structure, it is used only as a reference type during the processing alarm queue. Note that user's and schedule table's alarm control block structures (-»TagOSALMCB) have the same items at the beginning
- -»OSALLALARMS - the structure includes two alarm arrays (for user's alarms and for schedule table's alarms).
- -»tagALMAUTOTYPE - the structure of autostarted alarm
- -»TagOSALM - structure of user's alarm configuration.
- -»TagOSALMACT - structure of the alarm action
- -»TagOSALMCB - structure of user's alarm control block

There is basic type -»TagOSREFALM. There aren't any OS objects with this type. The pointer to this type (-»OSAlmType) is used in all places where alarm lists are handled (where it is not important - is the alarm user's or schedule table's) to cast user's alarm pointer or schedule table's alarm pointer to the type -»OSAlmType.

The structure -»OSALLALARMS is intended to ensure that both user alarm's and schedule table alarm's arrays were located in the memory directly one after the other. The order of arrays is important because alarm processing code depends on this.

The OS uses following global objects to handle alarms:

- *const ->OSALM ->OsAlarmsCfg[->OSNUSERALMS]* - the array of all user's alarm configurations. It's generated by the SysGen.
- *const ->OSALMAUTOTYPE ->OsAutoAlarms[->OSNAUTOALMS]* - the array of configurations for all user autostarted alarms. It's generated by the SysGen.
- *->OSALLALARMS ->OsAllAlarms* - this object includes two alarm arrays - array of control blocks for user's alarms and array of control blocks for schedule table's alarms. It declared only if both user's alarm(s) and schedule table(s) are configured in the system.
- *->OSALMCB ->OsAlarms[->OSNUSERALMS]* - the array of control blocks for user's alarms. It is declared only user's alarms are configured in the system. There is not any configured schedule table(s).
- *->OSSCTALMCB ->OsSCTAlarms[->OSNSCTALMS]* - the array of control blocks for schedule table's alarms. It is declared when schedule table(s) are configured but there is not any user's alarm(s) in the system.

The macro **OSALMARRAY** is used by the OS to access to user's alarm control blocks. The macro is defined as ->OsAllAlarms.OsAlarms or as OsAlarms depending on the OS configuration.

The macro **OSALMSCTARRAY** is used by the OS to access to schedule table's alarm control blocks. The macro is defined as OsAllAlarms.OsSCTAlarms or as OsSCTAlarms depending on the OS configuration.

The macro **OSNUSERALMS** is generated by the SysGen and it is equal to the number of configured user's alarms.

The macro **OSNSCTALMS** is generated by the SysGen and it is equal to the number of configured schedule tables.

The macro **OSNAUTOALMS** is generated by the SysGen and it is equal to the number of configured autostarted alarms.

ALARM management functions

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

DeclareAlarm

{DD_298}

void DeclareAlarm(Alarm)

A dummy declaration, intended for compatibility with other OSEK versions.

OSInitAlarms

{DD_030}

void OSInitAlarms (void)

Initialize the user alarms.

Parameters:

none

Returns:

none

Note:

used in StartOS

Implementation

Fill control block for each user alarm (see the type **-»OSALMCB**) using alarm constant configuration generated by the SysGen.

Used data

-»OSALMARRAY

-»OsAlarmsCfg

OSKillAlarm

{DD_031}

void OSKillAlarm (-»OSAlmType *alarmPtr*)

Stop the alarm without cooking (unlike -»OS_CancelAlarm).

Parameters:

[in] *alarmPtr* The pointer to Alarm

Returns:

none

Note:

used in TerminateApplication

Implementation

- Get Counter associated with the alarm
- Remove the alarm from the list on the Counter
- If Counter is "HW":
  - mark the alarm as not active
  - if the list on the Counter is empty, disable interrupts from Counter timer
  - if the list is not empty
  - find the alarm with minimal value
  - if no other alarm armed with the minimal value, arm timer with this value and call internal OS service *OSCounterNotyfy()* if armed alarm has already triggered.
- If Counter is "SW"
  - mark the alarm as not active

Used data

-»OsTimVal

OSNotifyAlarmAction

{DD_032}

void OSNotifyAlarmAction (-»OSALMACT * *action*))

Action to notify alarm.

Parameters:

[in] *action* The pointer to Alarm action

Returns:

none

Note:

none

Implementation for conformance class BCC1

- If activated task is already activated call internal OS service *OSErrorHook_noPar()* with error code *E_OS_LIMIT*;
- Else activate the task

Implementation for conformance class ECC1

- If the action is' set event'
  - If the task is suspended, call internal OS service *OSErrorHook_noPar()* with error code *E_OS_STATE*
  - Else set event for the task
- Else (the action is 'activate task')
  - If activated task is already activated call internal OS service *OSErrorHook_noPar()* with error code *E_OS_LIMIT*;
  - Else clear field '*set_event*' in the task control block and activate the task

Used data

-»OsTaskTable

OSCounterNotify

{DD_033}

void OSCounterNotify (-»OSWORD *ctrInd*)

Check for expired alarms and perform appropriate actions

Parameters:

[in] *ctrInd* The index of the related Counter

Returns:

none

Note:

1. called only if at least one alarm expired
2. this is intended for alarms on "HW" counters only

Implementation

The function performs an iterative loop until all the expired alarms will be processed. The function is called from System / Second Timer

interrupt handler or may be called from places where alarms are queued / dequeued (-»OSInsertAlarm and -»OSKillAlarm) if it is determined that there is an alarm to be expired. The function performs on each iteration:

- Fix the Counter time when an alarm expired

  (to the variable -»*OsTimVal*).
- Get the pointer to the alarm list on Counter.
- Check all alarms in the list for expiring until the end of the list
  - If an alarm is expired (<alarm value> is equal to -»*OsTimVal*):

    - if alarm is schedule table's (almId >= &OSALMSCTARRAY[0]), call internal OS service *OSProcessScheduleTable()* to handle active expiry point.

    - else if user's alarm has alarm callback, call the alarm callback in necessary context.

    - else if user's alarm has action 'increment counter', increment appropriate SW counter and possible other SW counters until all bound 'increment counter will be triggered

    - else call OS internal service -»OSNotifyAlarmAction to perform appropriate action.

    - if the alarm is not cyclic, remove the alarm from the list and mark the alarm as not active. Else recalculate alarm value as <alarm value> + <alarm cycle> and recalculate minimal value of all alarms in the list.

    - if there is an active alarm set by *OSProcessScheduleTable()*, add the alarm to the list and go to the label 'almAdded' to start the list checking from the beginning.
  - Else recalculate minimal value of all alarms in the list.
- Then the function arms the timer HW if necessary:
  - If the list is not empty

    - arm the timer on minimal alarm value in the list

    - if alarm already expired, disable the timer and go to the label 'again' to renew checking loop.

- Else disable the timer.

Used data

-»OSALMSCTARRAY

-»OsCounters

-»OsIsrLevel

-»OsService

-»OsCtrIncCounter

-»OsCtrIncValue

OSCheckAlarms

{DD_034}

void OSCheckAlarms (-»OSWORD *ctrInd*)

Check linked alarms.

Parameters:

[in] *ctrInd* The index of the related Counter

Returns:

none

Note:

for SW counters

Implementation

The function performs an iterative checking loop until all the expired alarms bound with SW Counter will be processed. The function is called from System / Second SW Timer interrupt handler or may be called from places where alarms are queued or the alarm action 'increment counter' is performed (-»OSInsertAlarm, -»OSCounterNotify) if it is determined that there is an alarm to be expired.

The function performs on each iteration for each alarm beginning at head of the list:

- Keep pointer to the field 'value' of the alarm control block to variable' delta'
- If the alarm expired (*delta is equal the Counter value

- If the alarm is schedule table's, call the OS internal service *OSProcessScheduleTable()* for this alarm.
- Else if the user's alarm has a alarm callback, call the callback under appropriate context.
- Else if user's alarm has action 'increment counter', increment appropriate increment counter value and common counter of increments.
- Else call the OS internal service -»OSNotifyAlarmAction with pointer to user's alarm action structure as parameter.
- If the alarm is not cyclic, remove alarm from the list.
- Else reinsert the alarm on the time equal to (<alarm value> + <alarm cycle>) by the Counter modulo kept the time to *delta.
- If the service *OSProcessScheduleTable()* returned non-zero pointer to the schedule table's alarm to be queued, insert the alarm to the list and go to the label 'almAdded' to renew checking loop.

• Get the next item of the alarm list.

Used data

-»OSALMSCTARRAY

-»OsCounters

-»OsCountersCfg

-»OsIsrLevel

-»OsService

-»OsCtrIncCounter

-»OsCtrIncValue

OSInsertAlarm

{DD_035}

void OSInsertAlarm (-»OSAlmType *almId*)

Insert alarm into list attached to counter, arm HW counter if it is the first alarm.

Parameters:

[in] *almId* The alarm identifier

Returns:

none

Note:

called with interrupts disabled

Implementation

- Get associated Counter index
- Insert this alarm on the head of Counter's alarm list
- If Counter is "HW"
  - If the alarm is first to expire then arm HW for its value
  - If armed time already expired call the OS internal service OSCounterNotify.
- Else (Counter is SW)
  - If the alarm is already expired then call the OS internal service

    -»OSCheckAlarms.

Used data

-»OsCounters

OSKillAppAlarms

{DD_036}

void OSKillAppAlarms (void)

Kills all alarms of given application.

Returns:

none

Note:

none

Implementation

Check all the user alarms (alarm id from 0. to -»OSNUSERALMS-1). If an alarm belongs to killed OS-Application and the alarm is active, call internal OS service -»OSKillAlarm for given alarm.

Used data

-»OSALMARRAY

OSSetAlarm

{DD_037}

void OSSetAlarm (-»OSALMCB * *almId*, -»TickType *start*, -»TickType *cycle*)

Insert alarm into list attached to counter.

Parameters:

[in] *almId* - a reference to the alarm
[in] *start* The alarm initialization value in ticks
[in] *cycle* The alarm cycle value in ticks in case of cyclic alarm

Returns:

none

Note:

none

Implementation

- Set value and cycle fields for given alarm (almId).
- Call -»OSInsertAlarm for the alarm.
- Clear "error" parameters
- Call the OS dispatcher if necessary.

Used data

-»OsRunning

OS_GetAlarmBase

{DD_038}

-»StatusType OS_GetAlarmBase (-»AlarmType *almId*, -»AlarmBaseRefType *info*)

Get the alarm base characteristics.

Parameters:

[in] *almId* - a reference to the alarm
[out] *info* The pointer to the structure <info> with returned values of the alarm base

Returns:

Standard:
E_OK no error.
E_OS_ACCESS insufficient access rights.

Extended:
E_OS_ID <almId> is not valid.

E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_ILLEGAL_ADDRESS illegal <info> (only for SC3, SC4).

Note: The service returns the alarm base characteristics into the structure, referenced by <info>.

Particularities: The service call is allowed on task level, ISR cat.2 level and in ErrorHook, PreTaskHook and PostTaskHook hook routines.

Implementation

- Common error check
- Specific error check
- Get the Counter index for given alarm
- Fill fields of the info structure using fields from OsCountersCfg[<the Counter index>].

Used data

-»OSALMARRAY

-»OsCountersCfg

-»OsIsrLevel

OSSCGetAlarm

-»void OSSCGetAlarm (-»OSALMCB* *almRef*, -»TickRefType *tickRef*)

Get the number of tick before alarm expires.

Parameters:

[in] *almRef*- a reference to the alarm control block
[in] *tickRef* - a pointer to a variable which gets a relative value in ticks before the alarm expires.

Returns:

none.

Implementation

- If the Counter is SW, get the Counter value via macro *OSSWCtrValue()*, else - via macro *OSHWCtrValue()*
- Calculate the relative value before the alarm will expire and save it by the pointer 'tickRef'.

Used data

-»OSALMARRAY

-»OsCounters

-»OsContersCfg

OSMCGetAlarm

{DD_319}

-»StatusType OSMCGetAlarm (-»OSALMCB* *almRef*, -»TickRefType *tickRef*, -»OSAPPLICATIONTYPE *appId*)

Get the number of tick before alarm expires.

Parameters:

[in] *almRef* - a pointer to the alarm control block
[in] *tickRef* - a pointer to a variable which gets a relative value in ticks before the alarm expires.
[in] *appId* - an ID of calling application

Returns:

Standard:
E_OK no error.
E_OS_NOFUNC alarm <almId> is not used.
Extended:
E_OS_ACCESS insufficient access rights.

Implementation

- Check access rights of calling application to given alarm
- If the alarm is not active, return error code E_OS_FUNC
- Call OSSCGetAlarm() function
- Return E_OK

Used data

-»OSALMARRAY

OS_GetAlarm

{DD_039}

-»StatusType OS_GetAlarm (-»AlarmType *almId*, -»TickRefType *tickRef*)

Get the number of tick before alarm expires.

Parameters:

[in] *almId* - a reference to the alarm
[in] *tickRef* - a pointer to a variable which gets a relative value in ticks before the alarm expires.

Returns:

Standard:
E_OK no error.
E_OS_NOFUNC alarm <almId> is not used.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <almId> is not valid.
.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_ILLEGAL_ADDRESS illegal <info> (only for SC3, SC4).

Note:

The service returns the relative value in ticks before the alarm expires into variable referenced by <tickRef>. If <almId> is not in use, returned value is not defined.

Particularities: The service call is allowed on task level, ISR cat.2 level and in ErrorHook, PreTaskHook and PostTaskHook hook routines.

Implementation

- Check context of service call and given alarm index
- Get the Counter index for the alarm
- If the alarm is not active, return error code E_OS_FUNC
- Check possibility to write into given pointer 'tickRef'
- If given alarm belong to the another core
  - perform remote call by calling OSRemoteCall2 macro
  - Call -»DISPATCH function
  - Leave OS critical section via macro *OSRI()*, return status and leave service
- else (given alarm belong to the same core)
  - Enter OS critical section via *OSDIS()*
  - If access rights of current OS-Application to given alarm are insufficient or state of OS-Application to which belong given alrm is not APPLICATION_ACCESSIBLE,

leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code

- Call OSSCGetAlarm() function.
- Leave the OS critical section via the macro *OSRI()*
- Return error code E_OK.

Used data

-»OSALMARRAY

-»OsIsrLevel

-»OsAppTable

OSInitAutoAlarms

{DD_040}

void OSInitAutoAlarms (-»AppModeType *mode*)

for OSNAPPMODES > 1
else

void OSInitAutoAlarms ()

Initialize autostarted alarms.

Parameters:

[in] *mode* The OS-Application mode (if defined **OSNAPPMODES** > 1)

Returns:

none

Note:

none

Implementation

The function produces the cycle for all configured autostarted alarms and for each alarm , if the alarm is autostarted for current application mode, performs following:

- Get the Counter index, pointer to the control block of the alarm and the alarm time.
- If the alarm type is relative, calculate the alarm value via the internal OS service *OSAbsTickValue()* with parameters - the Counter index and the alarm time. If Counter is HW, the service also fixes current counter value to the variable OsTimaVal.

- Else call the internal OS service -»OSSetTimVal to fix current counter value to the variable *OsTimaVal* (only if the Counter is HW).
- Fill the alarm control block with calculated alarm value and the alarm period value and insert the alarm into the list via the OS internal service -»OSInsertAlarm.

Used data

-»OsAutoAlarms[]

OSMCSetRelAlarm

{DD_320}

-»StatusType OSMCSetRelAlarm (-»OSALMCB* *almRef*, -»TickType *increment*, -»TickType *cycle*, -»OSAPPLICATIONTYPE *appId*)

Start the relative alarm.

Parameters:

[in] *almRef* - a pointer to the alarm control block
[in] *increment* The alarm initialization value in ticks.
[in] *cycle* The alarm cycle value in ticks in case of cyclic alarm.
[in] *appId* - an ID of calling application.

Returns:

Standard:
E_OK no error.
E_OS_STATE alarm already in use.
Extended:
E_OS_ACCESS insufficient access rights.

Implementation

- Check access rights of calling application to given alarm
- Check that the alarm is not active
- Calculate absolute alarm value
- Call the internal OS service -»OSInsertAlarm
- Return E_OK

Used data

-»OSALMARRAY

OS_SetRelAlarm

{DD_041}

-»StatusType OS_SetRelAlarm (-»AlarmType *almId*, -»TickType *increment*, -»TickType *cycle*)

Start the relative alarm.

Parameters:

[in] *almId* - a reference to the alarm
[in] *increment* The alarm initialization value in ticks
[in] *cycle* The alarm cycle value in ticks in case of cyclic alarm

Returns:

Standard:
E_OK no error.
E_OS_NOFUNC alarm <almId> is not used.
E_OS_VALUE Value of <increment> is equal to 0.
E_OS_STATE alarm already in use.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <almId> is not valid.

E_OS_VALUE Value of <increment> is outside of the admissible limits (greater than maxallowedvalue or 0) or value of <cycle> unequal to 0 and outside of the admissible counter limits (less than mincycle or greater than maxallowedvalue);
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

The service sets the alarm <almId> to the specified time. After <increment> ticks have elapsed, the task assigned to the alarm <almId> is activated or the assigned event is set or callback is called. If <cycle> is unequal "zero", the alarm element is logged on again immediately after expiry with the relative value <cycle>. If alarm is already in use, the service does not change it's state.

Particularities: Allowed on task level and ISR cat.2 level, but not in hook routines.

Implementation

- Check the context of the service call.
- Check that given relative value > 0.

- Check that the alarm id is valid
- Get the pointer to the alarm control block.
- Get the Counter index from the alarm control block.
- Check given relative value and cycle value.
- Check access rights of current OS-Application to the alarm.
- If given alarm belong to the another core
  - perform remote call by calling OSRemoteCall3 macro
  - Call -»DISPATCH function
  - Leave OS critical section via macro *OSRI(),* return status and leave service
- else (given alarm belong to the same core)
  - Enter OS critical section via *OSDIS()*
  - If access rights of current OS-Application to given alarm are insufficient or state of OS-Application to which belong given alrm is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Check that the alarm is not active
- Calculate absolute alarm value
- Call the internal OS service -»OSInsertAlarm
- Call OSDISPATCH() function
- Leave the OS critical section via the macro *OSRI*
- Return the code E_OK.

Used data

-»OSALMARRAY

-»OsIsrLevel

-»OsAppTable

OSMCSetAbsAlarm

{DD_321}

-»StatusType OSMCSetAbsAlarm (-»OSALMCB* *almRef*, -»TickType *start*, -»TickType *cycle*, -»OSAPPLICATIONTYPE *appId*)

Start the alarm.

Parameters:

[in] *almRef* - a pointer to the alarm control block
[in] *start* The alarm initialization value in ticks.
[in] *cycle* The alarm cycle value in ticks in case of cyclic alarm.
[in] *appId* - an ID of calling application.

Returns:

Standard:
E_OK no error.
E_OS_STATE alrm already in use.
Extended:
E_OS_ACCESS insufficient access rights.

Implementation

- Check access rights of calling application to given alarm
- Check that the alarm is not active
- Fix the current Counter value to the variable -»*OsTimVal* via the internal OS service -»OSSetTimVal (only if the Counter is HW)
- Call the internal OS service -»OSInsertAlarm with the pointer to the alarm control block, the alarm value and cycle value to insert the alarm into the Counter list.
- Return E_OK

Used data

-»OSALMARRAY

OS_SetAbsAlarm

{DD_042}

-»StatusType OS_SetAbsAlarm (-»AlarmType *almId*, -»TickType *start*, -»TickType *cycle*)

Start the alarm.

Parameters:

[in] *almId* - a reference to the alarm
[in] *start* The alarm initialization value in ticks
[in] *cycle* The alarm cycle value in ticks in case of cyclic alarm

Returns:

Standard:
E_OK no error.
E_OS_STATE the alarm is already in use.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <almId> is not valid.

E_OS_VALUE Value of <start> is outside of the admissible limits (greater than maxallowedvalue) or value of <cycle> unequal to 0 and outside of the admissible counter limits (less than mincycle or greater than maxallowedvalue)
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

The service sets the alarm <almId> to the specified time. When <Start> ticks are reached, the task assigned to the alarm <almId> is activated or the assigned event is set or callback is called. If <Cycle> is unequal "zero", the alarm element is logged on again immediately after expiry with the relative value <cycle>. If the absolute value <start> is very close to the current counter value, the alarm may expire and assigned task may become ready or alarm callback may be called before the system service returns. If the absolute value <start> already was reached before the service call, the alarm will only expire when <start> value will be reached again. If alarm is already in use, the service does not change it's state.
Particularities: Allowed on task level and ISR cat.2 level, but not in hook routines.

Implementation

- Check the context of the service call.
- Check that given alarm id is valid.
- Get the pointer to the alarm control block.
- Get the Counter index from the alarm control block.
- Check given alarm value and cycle value.
- If given alarm belong to the another core

  - perform remote call by calling OSRemoteCall3 macro
  - Call -»DISPATCH function
  - Leave OS critical section via macro *OSRI()*, return status and leave service
- else (given alarm belong to the same core)
  - Enter OS critical section via *OSDIS()*
  - If access rights of current OS-Application to given alarm are insufficient or state of OS-Application to which belong given alrm is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Check that alarm is not active.
- Fix the current Counter value to the variable -»*OsTimVal* via the internal OS service -»OSSetTimVal (only if the Counter is HW)
- Call the internal OS service -»OSInsertAlarm with the pointer to the alarm control block, the alarm value and cycle value to insert the alarm into the Counter list.
- Call OSDISPATCH() function
- Leave the OS critical section via the macro *OSRI*.
- Return the error code E_OK.

Used data

-»OSALMARRAY

-»OsIsrLevel

-»OsCountersCfg

OSMCCancelAlarm

{DD_322}

-»StatusType OSMCCancelAlarm (-»OSALMCB* *almRef*, -»OSAPPLICATIONTYPE *appId*)

Stop the alarm.

Parameters:

[in] *almRef* - a pointer to the alarm control block
[in] *appId* - an ID of calling application.

Returns:

Standard:
E_OK no error.
E_OS_NOFUNC the alarm is not in use.
Extended:
E_OS_ACCESS insufficient access rights.

Implementation

- Check access rights of calling application to given alarm
- Check that the alarm is active
- Call the internal OS service -»OSKillAlarm with the pointer to the alarm control block to delete the alarm from the Counter list and disable the timer if necessary.
- Return E_OK

Used data

-»OSALMARRAY

OS_CancelAlarm

{DD_043}

-»StatusType OS_CancelAlarm (-»AlarmType *almId*)

Stop the alarm.

Parameters:

[in] *almId* - a reference to the alarm

Returns:

Standard:
E_OK no error.
E_OS_NOFUNC the alarm is not in use.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <almId> is not valid.

E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

The service cancels the alarm <almId>. This service is not implemented if no alarms are defined in the configuration file.
Particularities: Allowed on task level and in ISR, but not in hook routines.

Implementation

- Check the context of the service call.
- Check that given alarm id is valid.
- Get the pointer to the alarm control block.
- If given alarm belong to the another core
  - perform remote call by calling OSRemoteCall1 macro
  - Call -»DISPATCH function
  - Leave OS critical section via macro *OSRI()*, return status and leave service
- else (given alarm belong to the same core)
  - Enter OS critical section via *OSDIS()*
  - If access rights of current OS-Application to given alarm are insufficient or state of OS-Application to which belong given alrm is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Check that alarm is active.
- Call the internal OS service -»OSKillAlarm with the pointer to the alarm control block to delete the alarm from the Counter list and disable the timer if necessary.
- clear stored parameters.
- If a running task is changed, call the OS task dispatcher
- Leave the OS critical section via the macro *OSRI*.
- Return the error code E_OK.

Used data

-»OsIsrLevel

-»OSALMARRAY

-»OsRunning

OSAbsTickValue

{DD_044}

OSINLINE -»TickType OSAbsTickValue (-»OSWORD *ctrInd*, -»TickType *rel*)

Calculate absolute tick value for given relative tick value; fix current Counter value into -»OsTimVal.

Parameters:

[in] *ctrInd* The counter Id
[in] *rel* The number of ticks

Returns:

Absolute tick value

Note:

none

Implementation

- If the Counter is 'SW' calculate absolute value taking into account maximal allowed value of the Counter
- Else
  - Fix the current Counter value in the variable -»*OsTimVal*
  - If there is an alarm in the list on the Counter and interrupt from the Counter timer is raised, fix -»*OsTimVal* as the Counter value when interrupt raised (timer OC register value) then re-read the counter value (it's necessary to re-read the Counter value here to avoid race conditions). Calculate absolute value as relative value plus Counter value.
  - Else calculate absolute value as relative value plus -»*OsTimVal*
  - Cut calculated absolute value to the timer capacity

Used data

-»OsTimVal

OSSetTimVal

{DD_045}

OSINLINE void OSSetTimVal (-»OSWORD *ctrInd*)

Set OsTimValue to the current counter value.

Parameters:

[in] *ctrInd* The counter Id

Returns:

none

Note:

This is intended for "HW" counters only

Implementation

If the Counter is HW:

- Fix the current Counter value to the variable *OsTimaVal*
- If the Counter list is not empty and there is raised interrupt from the Counter timer, refix the current Counter value as the value of the timer compare register to remember the Counter value when the timer interrupt occurred.

Used data

-»OsTimVal

Data Structures

{DD_046}

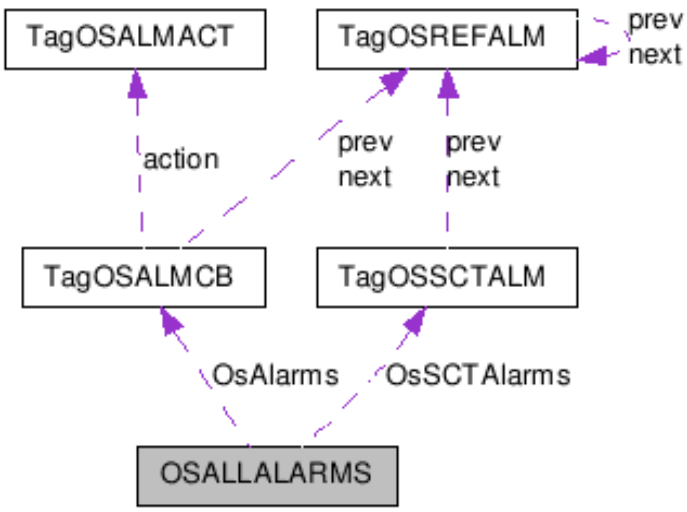
OSALLALARMS Struct Reference

`#include <Os_types_common_alm_sct.h>`

OSALLALARMS structure is intended to ensure that both OsAlarms and OsSCTAlarms arrays were located in the memory directly one after another.

The order of items is important because alarm processing code depends on this.

Collaboration diagram for OSALLALARMS:



Data Fields

- -»OSALMCB OsAlarms [-»OSNUSERALMS]
- -»OSSCTALMCB OsSCTAlarms [-»OSNSCTALMS]

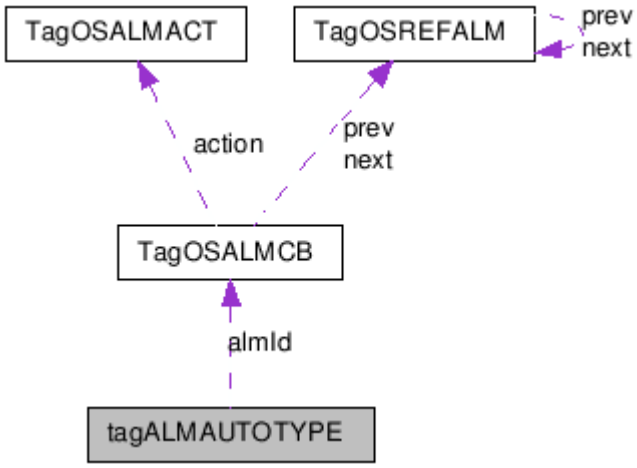
tagALMAUTOTYPE Struct Reference

{DD_047}

```
#include <Os_alarm_config.h>
```

Structure of autostarted alarm.

Collaboration diagram for tagALMAUTOTYPE:



Data Fields

- ->OSALMCB * almId
  Reference to alarm
- ->TickType time
  Time to start (relative)
- ->TickType cycle
  Alarm cycle, 0 for non-cycled
- ->OSDWORD autostart
  Bit mask of modes for start
- ->OSBYTE type
  Type of autostart alarm

TagOSALM Struct Reference

{DD_048}

```
#include <Os_alarm_config.h>
```

Structure of the alarm configuration table.

Data Fields

- ->OSDWORD appMask

  Application access mask

- ->OSWORD taskIndex

  Task to start or to set Event

- ->EventMaskType event

  If event == 0 alarm activates task else it sets event for this task

- ->OSWORD ctrIndex

  Attached Counter ID

- ->OSCALLBACK callBack

  Alarms' hook entry

- ->OSWORD ctrIndexInc

  Alarms' increment counter

- ->ApplicationType appId

  Application identification value

TagOSALMACT Struct Reference

{DD_049}

```
#include <Os_types_common_internal.h>
```

Structure of the alarm action. It is used in Alarms and Schedule Tables for Task Activation and Events only.

Data Fields

- -»OSWORD taskIndex

  Task to start or to set Event

- -»EventMaskType event

  If event==0 alarm activates task else it sets event for this task

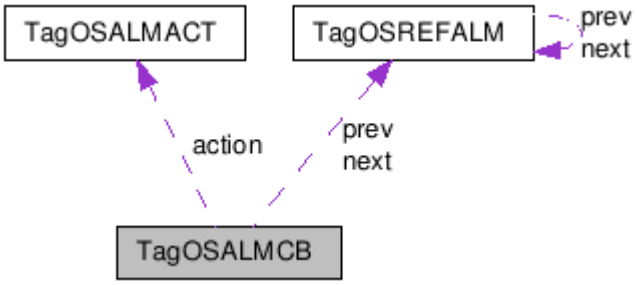
TagOSALMCB Struct Reference

{DD_050}

```
#include <Os_alarm_internal_types.h>
```

Structure of the alarm control block.

Collaboration diagram for TagOSALMCB:



Data Fields

- ->OSAlmType next
  Next alarm in the list
- ->OSAlmType prev
  Previous alarm in the list
- ->TickType value
  Alarms' expiration value
- ->TickType cycle
  Period value for cyclic alarm
- ->OSWORD ctrIndex
  Attached Counter ID
- ->OSWORD ctrIndexInc
  Alarms' increment counter
- ->OSDWORD appMask
  Application access mask
- ->OSALMACT action
  Alarm action
- ->OSCALLBACK callBack

Alarms' hook entry

TagOSREFALM Struct Reference

{DD_051}

```
#include <Os_types_common_internal.h>
```

Common alarm structure. It is used only as a reference type (**OsAlmType**) during the processing alarm queue.

User and schedule table alarms must have the same items at the beginning

Collaboration diagram for TagOSREFALM:

Data Fields

- -»OSAlmType next

  Attached Counter ID
- -»OSAlmType prev

  Previous alarm in the list
- -»TickType value

  Alarms' expiration value
- -»TickType cycle

  Alarms' expiration value
- -»OSWORD ctrIndex

  Period value for cyclic alarm

TagOSTPAlm Struct Reference

{DD_053}

```
#include <Os_tp_internal_types.h>
```

Collaboration diagram for TagOSTPAlm:

Data Fields

- -»OSTPALMCB * next
- -»OSTPALMCB * prev
- -»OSTPTICKTYPE value
- void * tpData

Counter Management

AUTOSAR OS supports two types of Counters: Hardware (represented in the HW timer register) and Software. A counter is represented by a current counter value and some counter specific parameters. These parameters are the counter initial value, the conversion constant and the maximum allowed counter value. They are defined by the user. The latter two parameters are constants and they are defined at system generation time. The counter initial value is the dynamic parameter. The user can initialize the counter with this value and thereafter on task or on interrupt level advance it using the system service IncrementCounter. The maximum allowed counter value specifies the number after that the counter rolls over. After a counter reaches its maximum allowed possible value (or rolls over the predefined size - byte etc.), it starts counting again from zero.

System(Second) Timer

{DD_054}

If the user needs to have system timer in the application, at least one counter shall exist in the system. This counter is used as a system timer (the internal system clock). Freescale AUTOSAR OS supports two system timers, named "System" and "Second". The system(second) timer is a standard counter with the following additions:

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

In the system definition statement for the system(second) timer the user should define one of possible hardware interrupt sources. Parameters to tune the hardware can be also defined by the user in this statement. This possibility allows the user to exactly tune the system. If hardware related parameters are defined, the code to initialize the system(second) timers hardware and the interrupt handler are automatically provided for the user as a part of the OS. In that case the user does not have to care about handling of this

interrupt and he/she can not change the provided code. If the parameters are not defined the user has to provide the code to initialize the hardware and handle the interrupt. Freescale AUTOSAR OS supports two types of system(second) timers:

- Software triggered;
- Hardware triggered.

Software triggered system(second) timer is based on periodic interrupt. In this case in ISR for the specified interrupt the services for triggering counter and notifying alarms must be used. In contrast, hardware triggered system(second) timer will generate interrupt only when an alarm expires.

Data Types

{DD_056}

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

- -»TickType - the data type represent count value in ticks
- -»TickRefType - the data type of a pointer to the variable of data type TickType
- -»CounterType the data type references a counter
- -»CtrInfoRefType the data type of a pointer to the structure of data type CtrInfoType
- -»CtrInfoType the data type represents a structure for storage of counter characteristics.

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

Counter Management functions

DeclareCounter

{DD_297}

void DeclareCounter(Counter)

A dummy declaration, intended for compatibility with other OSEK versions

OSInitCounters

{DD_057}

void OSInitCounters (void)

Initialize counters.

Parameters:

none

Returns:

none

Note:

used in StartOS

Implementation

- Fill control block for each counter (see the type **-»OSCTRCB**) using counter constant configuration generated by the SysGen.
- Clear -»*OSCtrIncValue* and -»*OSCtrIncCounter* are used for counter increments performed by alarms with INCREMENTCOUNTER action.

Used data

-»OsCounters

-»OsCountersCfg

-»OSCtrIncValue

-»OSCtrIncCounter

OSSysTimerCancel

{DD_058}

void OSSysTimerCancel (-»OSWORD *ctrInd*)

disable interrupts from output compare channel

Parameters:

[in] *ctrInd* - a pointer to Counter (if defined **OSSYSTIMER** or **OS2SYSTIMER**)

Returns:

None

Note:

for HW counters

Implementation

- If SystemTimer and SecondTimer are configured in OS:
  - If *ctrInd* is belong to SystemTimer, disable interrupt from the SytemTimer hardware and clear interrupt flag.
  - Else disable interrupt from the SecondTimer hardware and clear interrupt flag.
- If only SystemTimer is configured in OS,disable interrupt from the SytemTimer hardware and clear interrupt flag.

OSISRSystemTimer

{DD_059}

void OSISRSystemTimer (void)

ISR for system timer.

Returns:

none

Note:

called only if timer is armed - for HW counter or
SW counter is used

Implementation

- If SystemTimer is configured in OS:
  - Return if SystemTimer is HW and if interrupt flag of SystemTimer is not set
  - In SC3,4 set current application as invalid because of OS ISR does not belongs to any OS-Application.
  - Call internal OS macro *OSSystemEnterISR2()*.
  - Clear interrupt flag of SystemTimer compare channel.
  - Enter OS critical section via macro *OSDI*.
  - If SystemTimer is HW timer and OS has OS Alarms, call internal OS service *OSCounterNotify()*.
  - If SystemTimer is STM SW timer:

    Get new compare channel value, set it to HW register and call internal OS service *OSAlmCounterTrigger()* or *OSCounterTrigger()* in depending on whether increment counter Alarms in OS or not.

If time to a next SW tick has been expired, repeat it with interrupt flag clearing.

– Call internal OS service *OSSystemLeaveISR2()*.

OSISRSecondTimer

{DD_060}

void OSISRSecondTimer (void)

ISR for second timer.

Returns:

none

Note:

SW counter used

Implementation

- If SecondTimer is configured in OS:
  - Return if SystemTimer is HW and if interrupt flag of SecondTimer is not set
  - In SC3,4 set current application as invalid because of OS ISR does not belongs to any OS-Application.
  - Call internal OS macro *OSSystemEnterISR2()*.
  - Clear interrupt flag of SecondTimer compare channel.
  - Enter OS critical section via macro *OSDI*.
  - If SecondTimer is HW timer and OS has OS Alarms, call internal OS service *OSCounterNotify()*.
  - If SecondTimer is STM SW timer:

    Get new compare channel value, set it to HW register and call internal OS service *OSAlmCounterTrigger()* or *OSCounterTrigger()* in depending on whether increment counter Alarms in OS or not.

    If time to a next SW tick has been expired, repeat it with interrupt flag clearing.
- Call internal OS service *OSSystemLeaveISR2()*.

OSShutdownSystemTimer

{DD_061}

void OSShutdownSystemTimer (void)

disable all timer(s)

Returns:

none

Note:

only interrupts are disabled, timers are not stopped

Implementation

- If SystemTimer is configured in OS, call internal OS service *OSSysTimerCancel()* with SystemTimer identifier and then stop it.
- If SecondTimer is configured in OS, call internal OS service *OSSysTimerCancel()* with SecondTimer identifier and then stop it.

OSInitializeSystemTimer

{DD_062}

void OSInitializeSystemTimer (void)

initialize HW and start timer(s)

Returns:

none

Note:

1. MPC55xx/56xx HW specific,
2. called with interrupts disabled

Implementation

- If SystemTimer is configured in OS, initialize SystemTimer hardware, start timer, clear interrupt flag of compare channel and enable interrupt from this channel if the timer has SW type.
- If SecondTimer is configured in OS, initialize SecondTimer hardware, start the timer, clear interrupt flag of compare channel and enable interrupt from this channel if the timer has SW type.

OS_InitCounter

{DD_063}

-»StatusType OS_InitCounter (-»CounterType *ctrId*, -»TickType *value*)

Set initial counter value.

Parameters:

[in] *ctrInd* - a pointer to Counter
[in] *value* a counter initialization value in ticks

Returns:

Standard:
E_OK no error.
E_OS_CORE counter is bound to another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the counter identifier is invalid.

E_OS_VALUE the counter initialization value exceeds the maximum admissible value.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

If alarms are linked to this counter, then its state unchanged.

Sets the initial value of the counter with the value <value>. After this call the counter will advance this initial value by one via the following call of IncrementCounter. If there are running attached alarms, then their state stays unchanged, but the expiration time becomes indeterminate.
Particularities: The service call is allowed on task level only. This service is not implemented if no counters are defined in the configuration file.

Implementation

- Check the context of the service call.
- Check that given a Counter id is valid.
- Check given Counter initialization value.
- Check access rights of current OS-Application to the Counter.
- If type of the Counter is SW, fix the initialization value to the variable *OsCounters* via the macro *OSSWCtrInit()*.
- If type of the Counter is HW, fix the initialization value to the counter register of System or Second timer via the macro *OSHWCtrInit()*.
- Return the error code E_OK.

Used data

-»OsCounters

-»OsCountersCfg

GetElapsedCounterValue

{DD_064}

-»StatusType OS_GetElapsedCounterValue (-»CounterType *ctrId*,
-»TickRefType *valueRef*, -»TickRefType *tickRef*)

The service returns in the variable referenced by <tickRef> the number of elapsed ticks since the given <previousValue> value and updates <v> with new Counter value.

Parameters:

[in] *ctrInd* - a pointer to Counter
[in/out]valueRef - a reference to the previously read value
[out] *tickRef* - a reference to the difference with previous read value in ticks

Returns:

Standard:
E_OK no error.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the counter identifier is invalid.

E_OS_CALLEVEL call at not allowed context.
E_OS_VALUE given <previousValue> is not valid.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_ILLEGAL_ADDRESS illegal reference to
(only for SC3, SC4).

Note:

The service returns in the variable referenced by <tickRef> the number of elapsed ticks since the given <valueRef> value and updates <valueRef> with new counter value.
Particularities: The service call is allowed on task and ISR level. If the timer already passed the <valueRef> value a second time, the returned result is invalid. This service is not implemented if no counters are defined in the configuration file.

Implementation

- Check the context of the service call.
- Check that given a Counter id is valid.
- Check given elapsed value.
- Check access rights of current OS-Application to the Counter.

- Check write access rights to variable referenced by *tickRef*.
- Check write access rights to variable referenced by *valueRef*.
- If type of the Counter is SW, get Counter value via the macro *OSSWCtrValue()*.
- If type of the Counter is HW, get Counter value via the macro *OSHWCtrValue()*.
- Get elapsed Counter value since the given *valueRef* value and fix it in the variable referenced by *tickRef*.
- Fix Counter value in the variable referenced by *valueRef*.
- Return the error code E_OK.

Used data

-»OsCounters

OSAlmCounterTrigger

{DD_065}

void OSAlmCounterTrigger (-»OSWORD *ctrInd*)

Increment counter value and another counters values if expired alarm(s) has "increment counter" action.

Parameters:

[in] *ctrInd* - a pointer to Counter

Returns:

none

Note:

none

Implementation

- Increment Counter value via internal OS service *OSCounterTrigger()*.
- Increment all other SW Counters that have alarm(s) with "increment counter" action until all bound 'increment counter' will be triggered.

Used data

-»OsCtrIncCounter

-»OsCtrIncValue

OS_IncrementCounter

{DD_066}

-»StatusType OS_IncrementCounter (-»CounterType *ctrId*)

Increment counter value and check attached alarms.

Parameters:

[in] *ctrInd* - a pointer to Counter

Returns:

Standard:
E_OK no error.
E_OS_CORE counter belongs to another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the counter identifier is invalid.

E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

The service increments the current value of the counter. If the counter value was equal to maxallowedvalue it is reset to zero. If alarms are linked to the counter, the OS checks whether they expired after this tick and performs appropriate actions (task activation and/or event setting).
Particularities: The service call is allowed on task level and ISR level, but not in hook routines. The service shall not be used for counters assigned to System and Second timers. This service is not implemented if no counters are defined in the configuration file.

Implementation

- Check the context of the service call.
- Check that given a Counter id is valid and the Counter has SW type.
- Check that counter belongs to calling core.
- Enter OS critical section via macro *OSDIS*.
- If access rights of current OS-Application to given counter are insufficient or state of OS-Application to which belong given counter is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Save service parameters via the macro *OSPUTPARAM*.

- If there are alarms are linked to the Counter with "increment counter" action, that increment the Counter value and another counters values with expired alarm(s) via internal OS service *OSAlmCounterTrigger()*.
- Else increment the Counter value via internal OS service *OSCounterTrigger()*.
- Clear service parameters via the macro *OSCLEARPARAM*.
- Leave OS critical section via macro *OSRI*.
- Do rescheduling if it is necessary.
- Return the error code E_OK.

Used data

-»OsCounters

-»OsRunning

-»OsIsrLevel

OS_GetCounterInfo

{DD_067}

-»StatusType OS_GetCounterInfo (-»CounterType *ctrId*, -»CtrInfoRefType *info*)

Fill counter info structure.

Parameters:

[in] *ctrInd* - a pointer to Counter
[out] *info* - a pointer to the structure of CtrInfoType data type

Returns:

Standard:
E_OK no error.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the counter identifier is invalid.

E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_ILLEGAL_ADDRESS illegal <info> (only for SC3, SC4).

Note:

The service provides the counter characteristics into the structure

referenced by <info>. For a system counter special constants may be used instead of this service.
Particularities: The service call is allowed on task level, ISR level and in ErrorHook, PreTaskHook and PostTaskHook hook routines. The structure referenced by <info> consists of two elements in case of the "Standard Status", and of three elements in case of the "Extended Status". This service is not implemented if no counters are defined in the configuration file.

Implementation

- Check the context of the service call.
- Check that given a Counter id is valid.
- Check given elapsed value.
- Check access rights of current OS-Application to the Counter.
- Check write access rights to variable referenced by *info*.
- Fix reference to Counter structure in the variable referenced by *info*.
- Return the error code E_OK.

Used data

-»OsCounters

-»OsCountersCfg

OSSysTimerArm

{DD_068}

void OSSysTimerArm (-»OSHWTickType *ticks*)

Set interruption point and enable interrupts.

Parameters:

[in] *ctrInd* - a pointer to Counter (defined if **OSNHWCTRS** > 1)
[in] *ticks* The number of ticks

Returns:

none

Note:

none

Implementation

This internal OS service is implemented for HW timers only.

- If SystemTimer is configured in OS, set next point to compare channel register, clear interrupt flag and enable interrupt from this channel.
- If SecondTimer is configured in OS, set next point to compare channel register, clear interrupt flag and enable interrupt from this channel.

OS_GetCounterValue

{DD_069}

-»StatusType OS_GetCounterValue (-»CounterType *ctrId*, -»TickRefType *tickRef*)

Return counter value.

Parameters:

[in] *ctrInd* - a pointer to Counter
[in] *tickRef* - a pointer to counter value in ticks

Returns:

Standard:
E_OK no error.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the counter identifier is invalid.

E_OS_CALLEVEL a call at interrupt level (not allowed).
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_ILLEGAL_ADDRESS illegal reference to
(only for SC3, SC4).

Note:

none

Implementation

- Check the context of the service call.
- Check that given a Counter id is valid.
- Check given Counter initialization value.
- Check access rights of current OS-Application to the Counter.

- If type of the Counter is SW, fix the initialization value to the variable *OsCounters* via the macro *OSSWCtrInit()*.
- If type of the Counter is HW, fix the initialization value to the counter register of System or Second timer via the macro *OSHWCtrInit()*.
- Return the error code E_OK.

Data Structures

TagOSCTR Struct Reference

{DD_070}

```
#include <Os_counter_config.h>
```

Data Fields

- ->OSDWORD appMask
- ->TickType maxallowedvalue
- ->TickType ticksperbase
- ->TickType mincycle
- ->OSWORD coreId
- ->ApplicationType appId

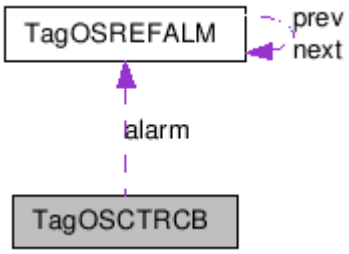
TagOSCTRCB Struct Reference

{DD_071}

```
#include <Os_counter_internal_types.h>
```

Counter control block

Collaboration diagram for TagOSCTRCB:



Data Fields

- -»OSAlmType alarm

  Pointer to assigned running alarms
- -»OSDWORD appMask

  Application identification mask value
- -»TickType value

  Current value of counter
- -»ApplicationType appId

  Application identification value

Event Management

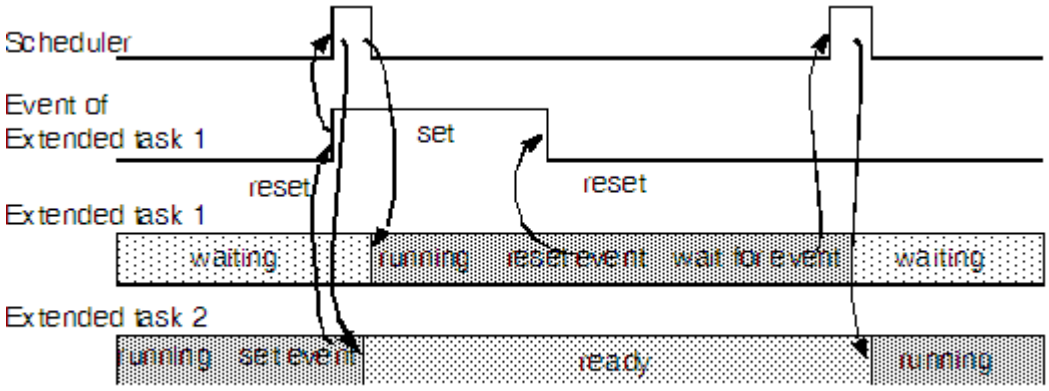
Within the AUTOSAR operating system tasks can be synchronized by means of the event mechanism, which is provided for Extended Tasks only. An event is an entity managed by the AUTOSAR Operating System, which is able to store binary data. The interpretation of the event is up to the user. Examples are: the signalling of a timer's expiry, the changes of a resource status, the receipt of a message, etc.

Events and Scheduling

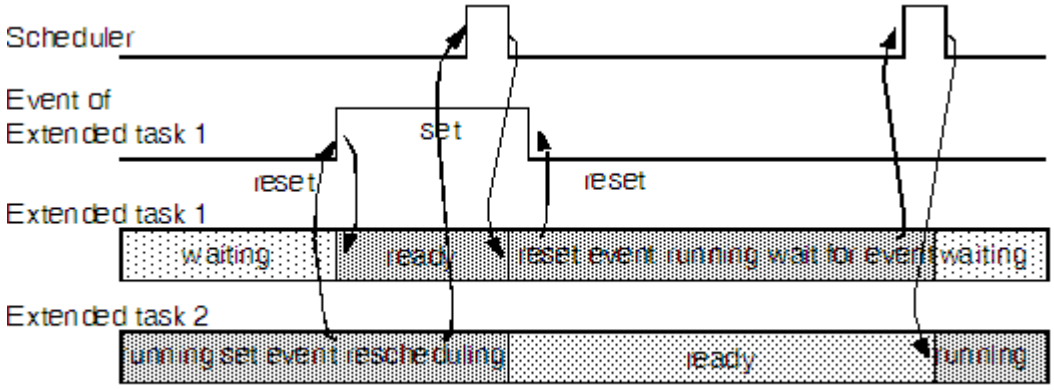
{DD_072}

An event is an exclusive signal which is assigned to an Extended Task. For the scheduler, events are the criteria for the transition of Extended Tasks from the waiting state into the ready state. The operating system provides services for setting, clearing and interrogation of events, and for waiting for events to occur. Extended Tasks are in the waiting state, if an event for which the task is waiting has not occurred. If an Extended Tasks tries to wait for an event and this event has already occurred, the task remains in the running state. Figure 4 illustrates the procedures which are effected by setting an event: Extended Task 1 (with higher priority) waits for an event. Extended Task 2 sets this event for Extended Task 1. The scheduler is activated. Subsequently, Task 1 is transferred from the waiting state into the ready state. Due to the higher priority of Tasks 1 this results in a task switch, Task 2 being preempted by Task 1. Task 1 resets the event. Thereafter Task 1 waits for this event again and the scheduler continues execution of Task 2.

Figure 0.1S



ation by Events for Full-preemptive Scheduling



If non-preemptive scheduling is supposed, rescheduling does not take place immediately after the event has been set as it is shown in Figure 5. Extended Task 1 became ready after event was set, but Extended Task 2 remains running until the point of rescheduling (for example, the Schedule service call).

Figure 0.2 Synchronization by Events for Non-preemptive Scheduling

The number of events per task is defined during compile time, and corresponding data type (typedef) is created. The size of this data type may be 8, 16, or 32 bits.

Data Types and Identifiers

{DD_073}

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

- -»**EventMaskType** - the data type of the event mask;
- -»**EventMaskRefType** - the data type to refer to an event mask. Reference to EventMaskType

EVENT management functions

DeclareEvent

{DD_295}

void DeclareEvent(Event)

A dummy declaration, intended for compatibility with other OSEK versions.

OSSetEvent

{DD_074}

void OSSetEvent (-»OSTSKCBPTR *task*, -»EventMaskType *mask*)

Set event and moves Task to Ready state.

Parameters:

[in] *task* - a reference to the task for which one or several events are to be set
[in] *mask* - an event mask to be set

Returns:

none

Note:

Set event and moves Task to Ready state

Implementation

- Set 'set_event' field in the task control block (-»TagOSTSKCB) to the event mask to be set
- If the task is in -»`WAITING` state and the mask of the waited event ('wait_event' field in the task control block) is equal the mask event from 'set_event' field (any event is occured), the task is transformed to -»`READY` state. Set bits in -»OsScheduleVector(s) to dispatch the Scheduler.

OSSetEventInAlm

{DD_075}

void OSSetEventInAlm (-»OSTSKCBPTR *task*, -»EventMaskType *mask*)

Set event and moves Task to *READY* state.

Parameters:

[in] *task* - a reference to the task for which one or several events are to be set
[in] *mask* - an event mask to be set

Returns:

none

Note:

none

Particularities: The service called only from alarm/schedule table event action.

Implementation

- Set 'set_event' field in the task control block (-»TagOSTSKCB) to the event mask to be set
- If the task is in -»WAITING state and the mask of the waited event ('wait_event' field in the task control block) is equal the mask event from 'set_event' field (any event is occured), the task is transformed to -»*READY* state. Set bits in -»OsScheduleVector(s) to allow scheduling of the Task.

OS_SetEvent

{DD_076}

-»StatusType OS_SetEvent (-»TaskType *taskId*, -»EventMaskType *mask*)

Performs standard service ***SetEvent***.

Set the event and perform dispatching if necessary

Parameters:

[in] *taskId* - a reference to the task for which one or several events are to be set
[in] *mask* - an event mask to be set

Returns:

Standard:
E_OK no error.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the task identifier is invalid.
E_OS_ACCESS the referenced task is not an Extended Task.

E_OS_STATE the referenced task is in the suspended state.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

This service is used to set one or several events of the desired task according to the event mask. If the task was waiting for at least one of the specified events, then it is transferred into the ready state. The events not specified by the mask remain unchanged. Only an extended task which is not suspended may be referenced to set an event.

Particularities: It is possible to set events for the running task (task-caller). The service call is allowed on task level and ISR level, but not in hook routines. This service is not implemented if no events are defined in the configuration file.

Implementation

- Check context of service call
- If given task belong to the another core
  - perform remote call by calling OSRemoteCall2 macro
  - Call -»DISPATCH function
  - Leave OS critical section via macro *OSRI()*, return status and leave service
- else (given task belong to the same core)
  - Enter OS critical section via *OSDIS()*
  - If access rights of current OS-Application to given task are insufficient or state of OS-Application to which belong given task is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Call the internal OS service -»OSSetEvent to set the event and move the task to -»READY state
- Call -»OSDISPATCH() function
- Leave the OS critical section via the macro *OSRI()*
- Return E_OK

Used data

-»OsTaskTable

-»OsIsrLevel

-»OsRunning

OS_ClearEvent

{DD_077}

-»StatusType OS_ClearEvent (-»EventMaskType *mask*)

Performs standard service ***ClearEvent***.

The task which calls this service defines the event which has to be cleared.

Parameters:

[in] *mask* - an event mask to be cleared

Returns:

E_OK no error.
Extended:
E_OS_ACCESS the referenced task is not an Extended Task.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

The task which calls this service defines the event which has to be cleared.

Particularities: The system service -»ClearEvent can be called from extended tasks which own an event only. This service is not implemented if no events are defined in the configuration file.

Implementation

- Check the context of the service call
- Check if the running task is an extended task
- Enter OS critical section via *OSDIS()*
- Clear bits in 'set_event' field, corresponding the mask of the cleared event, in the task control block (-»TagOSTSKCB) of the running task.
- Leave the OS critical section via the macro *OSRI()*
- Return error code E_OK.

Used data

-»OsRunning

OS_WaitEvent

{DD_078}

-»StatusType OS_WaitEvent (-»EventMaskType *mask*)

Performe standard service ***WaitEvent***

The calling task is transferred into the waiting state until at least one of the events specified by the mask is set. The task is kept the running state if any of the specified events is set at the time of the service call.

Parameters:

[in] *mask* - an event mask to wait for

Returns:

E_OK no error.

Extended:

E_OS_ACCESS the referenced task is not an Extended Task.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_RESOURCE the calling task occupies resources.
E_OS_SPINLOCK unreleased spinlock.

Note:

The calling task is transferred into the waiting state until at least one of the events specified by the mask is set. The task is kept the running state if any of the specified events is set at the time of the service call.

Particularities: This call enforces the rescheduling, if the wait condition occurs. All resources occupied by the task must be released before WaitEvent service call. The service can be called from extended tasks which own an event only. This service is not implemented if no events are defined in the configuration file.

Implementation

- Check the context of the service call
- Check if the running task is not extended task (returns *E_OS_ACCESS*)
- Check if a resource is occupied (returns *E_OS_RESOURCE*)
- Check if a spinlock is occupied (returns *E_OS_SPINLOCK)*
- Enter OS critical section via *OSDI()*: it is not necessary to save / restore interrupt mask because this service shall be called with enabled interrupts only
- Set 'status' field in the task control block (-»TagOSTSKCB) of the running task to the state "Task is inside WaitEvent function"
- If one of the events specified by the mask is set, set the task to `-»WAITING` state, call -»OSTaskForceDispatch function
- Else if the timing protection feature is configured in the OS
  - If protection hook is configured in the OS and there are tasks which have the length of timeframe for inter-arrival rate protection > 0

- Call -»OSTPStartTaskFrame function (start control inter-arrival a task is activated or released)

If the function returns TRUE (if it's possible to activate / release the task) call -»OSTPRestartTaskFrame to restart task budget for the tasks

Else (the function returns FALSE - there is task arrival rate violation)

- Check -»OsTPHookAction. If the protection hook returned *PRO_IGNORE* (the OS shall do nothing), set the task to -»`WAITING` state, call -»OSTaskForceDispatch function

- Else call -»OSShutdownOS with parameter *E_OS_PROTECTION_ARRIVAL*

– If there are tasks which have the task execution budget > 0

- Call -»OSTPStartTaskFrame function (start control inter-arrival a task is activated or released)

- Call -»OSTPRestartTaskBudget to restart the execution budget for the tasks

- Clear 'status' field in the task control block (-»TagOSTSKCB) of the running task from the state "Task is inside WaitEvent function"
- Leave the OS critical section via the macro *OSEI()*
- Return error code E_OK

Used data

-»OsRunning

-»OsTPHookAction

OS_GetEvent

{DD_079}

-»StatusType OS_GetEvent (-»TaskType *taskId*, -»EventMaskRefType *maskRef*)

Performs standard service ***ClearEvent***.

The event mask which is referenced to in the call is filled according to the state of the events of the desired task. Current state of events is returned but not the mask of events that task is waiting for.

Parameters:

[in] *taskId* - a reference to the task whose event mask is to be returned
[in] *maskRef* - a pointer to the variable of the return state of events

Returns:

Standard:
E_OK no error.
E_OS_ACCESS insufficient access rights
Extended:
E_OS_ID the task identifier is invalid.
E_OS_ACCESS the referenced task is not an Extended Task.
E_OS_STATE the referenced task is in the suspended state.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

without critical section

The event mask which is referenced to in the call is filled according to the state of the events of the desired task. Current state of events is returned but not the mask of events that task is waiting for. It is possible to get event mask of the running task (task-caller).

Particularities: The referenced task must be an extended task and it can not be in suspended state. The service call is allowed on task level, ISR level and in ErrorHook, PreTaskHook and PostTaskHook hook routines. This service is not implemented if no events are defined in the configuration file

Implementation

- Check the context of the service call
- Check if the task has the invalid identifier (returns *E_OS_ID*)
- Check if the task is not extended task (returns *E_OS_ACCESS*)
- Check if the task is in suspended state (returns *E_OS_STATE*)
- Check access rights and the possibility to write into given pointer 'maskRef'
- Copy the event mask from the 'set_event' field in the task control block (-»TagOSTSKCB) to variablr pointed by *maskRef*.
- Return code E_OK

Interrupt Processing

Interrupt processing is an important part of any real-time operating system. An Interrupt Service Routine (ISR) is a routine which is invoked from an interrupt source, such as a timer or an external hardware event. ISRs have higher priority than all tasks and the scheduler. In the Freescale AUTOSAR in SC3 and SC4 classes all ISRs of category 2 use the configured stacks. In SC1 with ECC1 class and in SC2 all ISRs of category 2 use the "ISR stack" which is used only by ISRs during their execution. The size of the ISR stack(s) is defined by the user. In the classes with memory protection (SC3, SC4) all ISRs of category 2 with the same PRIORITY value, including System and Second Timer ISRs, shares the stack memory. The OS protection handlers has their own stacks, not configurable by the user. According to the OSEK/VDX Operating System specification there are no services for manipulation of CPU and/or OS interrupt levels directly. Therefore nested interrupts with the same hardware level can not occur. For CPU with one hardware level nested interrupts are forbidden. For multilevel modes nested interrupts with different priority are allowed. Application is not allowed to manipulate interrupt enabling bits in the CPU state registers.

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

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

Interrupts cannot use any OS services except those which are specially allowed to be used within ISRs.

ISR Categories

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

Category 1

{DD_080}

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

Category 2

{DD_081}

In ISR category 2 the Operating System provides an automatic switch to the ISR stack (except in BCC1, SC1) and enters OS ISR execution context. After that, any user's code can be executed, including allowed OS calls (to activate a task, send a message or increment a counter). At the end of the ISR, the System automatically switches back to the stack of interrupted Task / ISR and restores context. Inside the ISR, no rescheduling will take place. Rescheduling may only take place after termination of the ISR of category 2 if a preemptable task has been interrupted.

Interrupt processing functions

Interrupts are enabled during task execution if there are any ISR or any Timer is configured, otherwise OS dispatcher does not control interrupt levels of MCU. Interrupts can be disabled via disable / enable interrupt API functions or by using resource mechanism.

AUTOSAR OS provides services to control the CPU interrupt recognition status.

Within the application an interrupt service routine is defined according to the following pattern: {DD_290}

ISR (<name of ISR>)

{

}

DeclareISR

{DD_291}

void DeclareISR(ISR)

DeclareISR is a ISR declaration, may be used for Interrupt Service Routine declaration in the “vector.c” file.

OSISRException

{DD_082}

void OSISRException (void)

ISR for catching unregistered interrupts.

Returns:

none

Note:

Called only if unregistered interrupt occurred

ISR for catching unregistered interrupts. Called only if unregistered interrupt occured.

Implementation

- Call OSShutdownOS(E_OS_SYS_FATAL); never returns.

OSKillRunningISR

{DD_083}

void OSKillRunningISR (void)

Kills ISR2 that caused ProtectionHook.

Returns:

none

Note:

Called from protection handler only in case PRO_KILLTASKISR.

Implementation

- Cancel the ISR budget.
- Release this ISR resources, if any.
- Mark the ISR as killed’
- Restore patterns on the ISR stack for stack checking.

- Cancel interrupt lock time.
- Clear OsSuspendLevelAll and OsSuspendLevel variables.
- Clear context bits in OsIsrLevel
- Return.

OSKillISR

{DD_084}

void OSKillISR (-»OS_ISRTYPE * *isr*)

Kills given ISR2.

Parameters:

[in] *isr* The a reference to ISR

Returns:

none

Note:

-»OsIsrLevel is not changed

Called from protection handler only.

Implementation

- Restore this ISR appId.
- IF this ISR is running (
  - Cancel the ISR budget.
  - Restore patterns on the ISR stack for stack checking.
  - Cancel interrupt lock time.
  - Clear OsSuspendLevelAll and OsSuspendLevel variables.
  - Clear context bits in OsIsrLevel
- Release this ISR resources, if any.
- Mark the ISR as killed'
- Return.

OSInitIVORS

{DD_085}

OSASM void OSInitIVORS (void)

Initializes all IVORS to point to vector table.

Returns:

none

Note:

Initializes all IVORS to point to vector table

Implementation

- Write into IVORs 0..15 and 32..34 consequential values beginning with 0 with step 0x10.

OSInitializeISR

{DD_086}

void OSInitializeISR (void)

Initializes ISRs.

Returns:

none

Note:

none

Implementation

- For each configured ISR
  - Copy values from configuration table into ISR control block
- Clear OsISRResourceCounter, OsIsrLevel and OsSuspendLevel variables.
- Set IVPR register with vector table address.
- Call OSInitIVORS.
- Set ME bit in MSR.
- Set configured priority for all ISRs, set 0 for non-configured channels in INTC.
- Configure INTC for SW mode.
- Enable ISRs (set INTC prio level to 0)

OSLeaveISR

{DD_087}

void OSLeaveISR (void)

Leave ISR and switch tasks' context, if needed.

Returns:

none

Note:

Different implementations for different SC.

Implementation

- If there is a valid running Task and dispatching is not allowed
  - Resume Task budget
  - Allow access to task stack area by MPU
  - Return
- Set MSR for Tasks
- IF there is a valid running Task
  - Set current Application
  - Call PostTaskHook
- Calculate new OsRunning.
- Save old Task context via OSSetJmp
- IF OSSetJmp returns non-zero value
  (it is return via OSLongJmp)
  - Set running priority for this Task
  - Call PreTaskHook
  - Disable all interrupts (EI=0)
  - return
- IF the old Task is *BASIC*
  - Save TCB address in linked list
- Clear OsKilled.
- Call OSTaskInternalDispatch (never returns)

OSTPLeaveISR

{DD_088}

void OSTPLeaveISR (void)

Performs dispatch before return from TP ISR if needed.

Returns:

none

Note:

Called only if ISR and/or Task was killed, under *OSDHI()*

Implementation

- IF returning to task level
  - IF running Task is killed

    - Calculate new OsRunning

    - Clear OsKilled

    - Set MSR for Tasks

    - Call OSTaskInternalDispatch - it never returns.
- ELSE
  - IF ISR is killed

    - Decrease OsIsrLevel and return into OSInterruptDispatcher via LongJmp

OSNonTrustedISR2

{DD_089}

OSINLINE void OSNonTrustedISR2 (-»OSVOIDFUNCVOID *userISR*)

This is the wrapper to call a user's non trusted ISR of category 2.

Returns:

none

Note:

called from OSInterruptDispatcher only.

Implementation

- Switch to User mode.
- Call *userISR*.

- Switch to Supervisor mode via special system call.
- Disable OS interrupts.
- Disable all interrupts.

OSTrustedISR2

{DD_090}

OSINLINE void OSTrustedISR2 (-»OS_ISRTYPE * *ptr*)

This is the wrapper to call a user's trusted ISR of category 2 or ISR of category 2 for SC1 or SC2 configuration.

Returns:

none

Note:

called from OSInterruptDispatcher only.

Implementation

- Switch to common ISR stack if required.
- Increment OsIsrLevel.
- Call PreIsrHook
- Start this ISR budget.
- Enable all interrupts (EI=1).
- Call user ISR.
- Disable OS interrupts.
- Disable all interrupts.
- Reset ISR budget.
- Call PostIsrHook.
- Decrement OsIsrLevel.
- Switch to Task stack if required.

OSInterruptDispatcher1

{DD_091}

void OSInterruptDispatcher1 (void)

This is a Dispatcher for External Interrupts.

Returns:

none

Note:

none

Implementation

- Save current IPL from INTC (workaround for INTC bug)
- Calculate pointer to current ISR CB from read IACKR value
- IF this is ISR of category 1
  - save OsIsrLevel value
  - Mark in OsIsrLevel that it is ISR cat.1
  - Enable interrupts (EI=1)
  - call user ISR.
  - Disable interrupts (EI=0)
  - restore saved OsIsrLevel.
  - Restore IPL via writing to EOIR.
  - return
- IF current IPL is less or equal to saved (workaround for INTC bug)
  - Restore IPL via writing to EOIR.
  - return
- IF this is TP ISR "type 1"
  - call the ISR function.
  - Restore IPL via writing to EOIR.
  - return
- Save current application
- Write pointer to this ISR CB into OsIsrArray indexed by OsIsrLevel
- Check Task Stack if required.
- IF it is a System Timer ISR
  - Set PID0 for OS process to access all stacks
  - Load SP

  - Call the ISR function
  - Restore SP
- ELSE / / it is a User ISR cat.2
  - Save old context via OSSetJmp
  - IF OSSetJmp returns zero value

    (it is for new ISR)

    - Set current application fro this ISR

    - Check arrival rate.

    - Open by MPU stack for this ISR

    - Load SP

    - IF this ISR is non-trusted

    - - Open by MPU data access for this application

    - - Increment OsIsrLevel.

    - - Call PreIsrHook

    - - Start this ISR budget.

    - - Call OSNonTrustedISR2 for this ISR.

    - - Reset ISR budget.

    - - Call PostIsrHook.

    - - Decrement OsIsrLevel.

    - - Restore SP form OsIsBufs[]

    - ELSE / / ISR is trusted

    - - Call OSTrustedISR2 for this ISR.

    - - Restore SP form OsIsBufs[]
  - Set PID0 for OS process to access all stacks

  (this is a return point for all ISRs except cat.1)
- IF it is a User ISR
  - Check ISR stack
  - IF ISR is not killed and there is a stack violation

    - Set OsViolator*** variables

    - Call ProtectionHook

- Proceed ProtectionHook return code:
- Restore stack guard area

– Clear this ISR "isKilled" field

– IF there are unclosed Suspend[OS/All}Interrupts

- Clear OSSuspend(All)Level variables and flags in OsIsrLevel
- Kill TP lock time

– IF there are unreleased Resources - release them.

- IF the preempted ISR is killed
  – Decrease OsIsrLevel
  – Perform OSLongJmp for preempted ISR
- IF Task dispatch might be required
  – IF running Task is killed
    - Enable interrupts (EI=1)
    - Calculate new OsRunning.
    - Cleare OsKilled.
    - Call OSTaskInternalDispatch (never returns)
  – Call OSLeaveISR / / - it does dispatch if necessary
  – Restore IPL with saved value (workaround for INTC bug)
- ELSE / / no Task dispatch
  – IF it is return to Task level
    - Resume Task budget, if any
    - Allow access to task stack area by MPU
  – Restore IPL with saved value (workaround for INTC bug)
- Set current application from saved
- Open by MPU data access for this application
- return

OSInterruptDispatcher

{DD_092}

void OSINTERRUPTNEST OSInterruptDispatcher (void)

This is a wrapper for Interrupt Dispatcher.

Returns:

none

Note:

written in assm for Z), Z1, Z3 and Z4 cores to get optimized code.

Implementation

- Call OSInteeruotDispatcher1

OS_DisableAllInterrupts

{DD_093}

void OS_DisableAllInterrupts (void)

This service saves the current interrupts state and disables all hardware interrupts. This service is intended to start a critical section of the code.

Returns:

none

Note:

The service disables recognition of all interrupts, i.e. no CPU interruptions will occur. The current recognition status (before call) is saved for the -»EnableAllInterrupts call.
This service saves current interrupt state and disables all hardware interrupts. This service is intended to start a critical section of the code. This section must be finished by calling the -»EnableAllInterrupts service. No API service calls are allowed within this critical section. -»DisableAllInterrupts/-»EnableAllInterrupts section does not support nesting.
Particularities: The service call is allowed in any context except inside -»DisableAllInterrupts/-»EnableAllInterrupts pair.

Implementation

- IF this function was not called before
  - Save current MSR in OsOldMsr and disable interrupts (EI=0)
  - Set context bit in OsIsrLevel I
  - Increment OSSuspendLevelAll

OS_EnableAllInterrupts

{DD_094}

void OS_EnableAllInterrupts (void)

This service restores the interrupts state saved by -»DisableAllInterrupts service. It can be called after -»DisableAllInterrupts only. This service is a counterpart of -»DisableAllInterrupts service, and its aim is the completion of the critical section of code. No API service calls are allowed within this critical section.

Returns:

none

Note:

Shall be called only after -»DisableAllInterrupts

The service restores recognition status of interrupts. These services does not support nesting. No AUTOSAR OS services may be called between -»DisableAllInterrupts and -»EnableAllInterrupts pairs.
This service restores the interrupts state saved by -»DisableAllInterrupts service. This service is a counterpart of -»DisableAllInterrupts service. It can be called after -»DisableAllInterrupts only, otherwise the service does nothing.
Particularities: The service call is allowed in any context.

Implementation

- IF there was a call to DisableAllInterrupts
  - Decrement OSSupendLevelAll
  - Clear context bit
  - Restore MSR from saved in OsOldMsr.

OS_SuspendAllInterrupts

{DD_095}

void OS_SuspendAllInterrupts (void)

Save interrupt state and disable all interrupts.

Returns:

none

Note:

This function is MD.

The service saves the recognition status of all interrupts and disables all interrupts for which the hardware supports disabling.
-»ResumeAllInterrupts restores recognition status saved by -»SuspendAllInterrupts call. These services support nesting. The only services that can be called between the pair -»SuspendOSInterrupts/-»ResumeOSInterrupts are -»SuspendAllInterrupts/-»ResumeAllInterrupts and -»SuspendOSInterrupts/-»ResumeOSInterrupts.
This service saves current interrupt state and disables all interrupts. This service is intended to start a critical section of the code. This section must be finished by calling the -»ResumeAllInterrupts service. No API service calls beside -»SuspendAllInterrupts/-»ResumeAllInterrupts and -»SuspendOSInterrupts/-»ResumeOSInterrupts pairs are allowed within this critical section. In case of nesting pairs of the calls -»SuspendAllInterrupts and -»ResumeAllInterrupts the interrupt status saved by the first call of -»SuspendAllInterrupts is restored by the last call of the -»ResumeAllInterrupts service.
Particularities: The service call is allowed in any context except inside -»DisableAllInterrupts/-»EnableAllInterrupts pair.

Implementation

- IF the function is called in allowed context
  - IF OSSupendLevelAll is zero

    - Save current MSR in OsOldIntMaskAll and disable interrupts (EI=0)

    - Set context bit in OsIsrLevel I
  - Increment OSSuspendLevelAll

OS_ResumeAllInterrupts

{DD_096}

void OS_ResumeAllInterrupts (void)

Restore interrupt state saved by -»SuspendAllInterrupts service.

Returns:

none

Note:

This function is MD.

This service restores the interrupts state saved by -»SuspendAllInterrupts

service. This service is a counterpart of -»SuspendAllInterrupts service. It can be called after -»SuspendAllInterrupts only. If STATUS is set to EXTENDED then the function shall check correspondence of Suspend/ Resume pairs and ignore the call in case of error. If the function is called without previous call to -»SuspendAllInterrupts the call is ignored.
Particularities: The service call is allowed in any context except inside -»DisableAllInterrupts/-»EnableAllInterrupts pair.

Implementation

- IF the function is called in allowed context
  - IF OSSupendLevelAll is zero - return
  - Decrement OSSuspendLevelAll
  - IF OSSupendLevelAll is zero

    - Clear context bit

    - Restore MSR from saved in OsOldIntMaskAll.

OS_SuspendOSInterrupts

{DD_097}

void OS_SuspendOSInterrupts (void)

Save interrupt state and disable all OS interrupts.

Returns:

none

Note:

The service disables recognition of interrupts serviced by ISRs categories 2. The current recognition status (before call) is saved for the -»ResumeOSInterrupts call.
This service saves current interrupt state and disables all interrupts category 2. This service is intended to start a critical section of the code. This section must be finished by calling the -»ResumeOSInterrupts service. No API service calls beside -»SuspendAllInterrupts/-»ResumeAllInterrupts and -»SuspendOSInterrupts/-»ResumeOSInterrupts pairs are allowed within this critical section. In case of nesting pairs of the calls -»SuspendOSInterrupts and -»ResumeOSInterrupts the interrupt status saved by the first call of -»SuspendOSInterrupts is restored by the last call of the -»ResumeOSInterrupts service.

Particularities: The service call is allowed in any context except inside -»DisableAllInterrupts/-»EnableAllInterrupts pair.

Implementation

- IF the function is called in allowed context
  - IF OSSupendLevel is zero
    - Save current IPL in OsOldInterruptMask
    - set IPL to OSHIGHISRPRIO
    - Start TP Lock Time
    - Set context bit in OsIsrLevel I
  - Increment OSSuspendLevel

OS_ResumeOSInterrupts

{DD_098}

void OS_ResumeOSInterrupts (void)

Restore interrupt state saved by -»SuspendOSInterrupts service.

Returns:

none

Note:

Allows nesting

The service restores recognition status. These services support nesting. The only services that can be called between the pair -»SuspendOSInterrupts/-»ResumeOSInterrupts are -»SuspendAllInterrupts/-»ResumeAllInterrupts and -»SuspendOSInterrupts/-»ResumeOSInterrupts.
This service restores the interrupts state saved by -»SuspendOSInterrupts service. This service is a counterpart of -»SuspendOSInterrupts service. It can be called after -»SuspendOSInterrupts only. If STATUS is set to EXTENDED then the function shall check correspondence of Suspend/Resume pairs and ignore the call in case of error. If the function is called without previous call to -»SuspendOSInterrupts the call is ignored.
Particularities: The service call is allowed in any context except inside -»DisableAllInterrupts/-»EnableAllInterrupts pair.

Implementation

- IF the function is called in allowed context
  - IF OSSupendLevel is zero - return

- IF OSSupendLevel == 1
  - Kill TP Interrupt LockTime
  - Decrement OSSuspendLevel
  - Set IPL to saved in OsOldInterruptMask
- ELSE - decrement OSSuspendLevel.

OS_GetISRID

{DD_099}

-»ISRType OS_GetISRID (void)

Shows which ISR is running.

Returns:

current ISR Id or
INVALID_ISR

Note:

This service returns the identifier of the currently executed ISR, if called from interrupt level, otherwise service return *INVALID_ISR*.
Particularities: The service call is allowed on task level and ISR cat.2 level, in ErrorHook and ProtectionHook.

Implementation

- IF the function is called in not allowed context
  - Call ErrorHook and return appropriate error code
- IF there is no running ISR
  - return INVALID_ISR
- return Id of running ISR

DisableInterruptSource

{DD_100}

-»StatusType OS_DisableInterruptSource (-»ISRType *isrId*)

disables interrupt source in INTC

Parameters:

[in] *isrId* identifier of ISR

Returns:

Standard:
E_OK no error.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the ISR identifier is invalid.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

This service is not defined in AUTOSAR 3.0, it is retained for compatibility with AUTOSAR 2.1

Implementation

- IF the function is called in not allowed context or argument is invalid
  - Call ErrorHook and return appropriate error code
- If ISR is bound to another core return E_OS_CORE error code
- Enter OS critical section via the macro *OSDIS*.
- If access rights of current OS-Application to given ISR are insufficient, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Disable ISR via INTC
- Leave the OS critical section via the macro *OSRI*.
- Return E_OK.

EnableInterruptSource

{DD_101}

-»StatusType OS_EnableInterruptSource (-»ISRType *isrId*)

Enables interrupt source in INTC.

Parameters:

[in] *isrId* identifier of ISR

Returns:

Standard:
E_OK no error.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.

Extended:
E_OS_ID the ISR identifier is invalid.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

This service is not defined in AUTOSAR 3.0, it is retained for compatibility with AUTOSAR 2.1

Implementation

- IF the function is called in not allowed context or argument is invalid
  - Call ErrorHook and return appropriate error code
- If ISR is bound to another core return E_OS_CORE error code
- Enter OS critical section via the macro *OSDIS*.
- If access rights of current OS-Application to given ISR are insufficient, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Enable ISR via INTC (restore its priority)
- Leave the OS critical section via the macro *OSRI*.
- Return E_OK.

OSKillAppISRs

{DD_102}

void OSKillAppISRs (void)

Kills all ISRs belonging to given application.

Returns:

none

Note:

none

Implementation

- FOR all configured ISRs
  - IF ISR belongs to current application

    - Restore appId in OsIsrTable
- FOR all active ISRs

– IF ISR belongs to current application

- Call OSKillISR for it

OSSuspendInterrupts

{DD_103}

OSINLINE void OSSuspendInterrupts (void)

Save interrupt mask and disable ISRs cat.2.

Returns:

none

Note:

Called from ErrorHook only if OsSuspendLevel == 0

Implementation

- IF current IPL is less than OSHIGHISRPRIO
  - – - Save current IPL in OsOldInterruptMask
  - – - set IPL to OSHIGHISRPRIO
- Increment OsSupendLevel

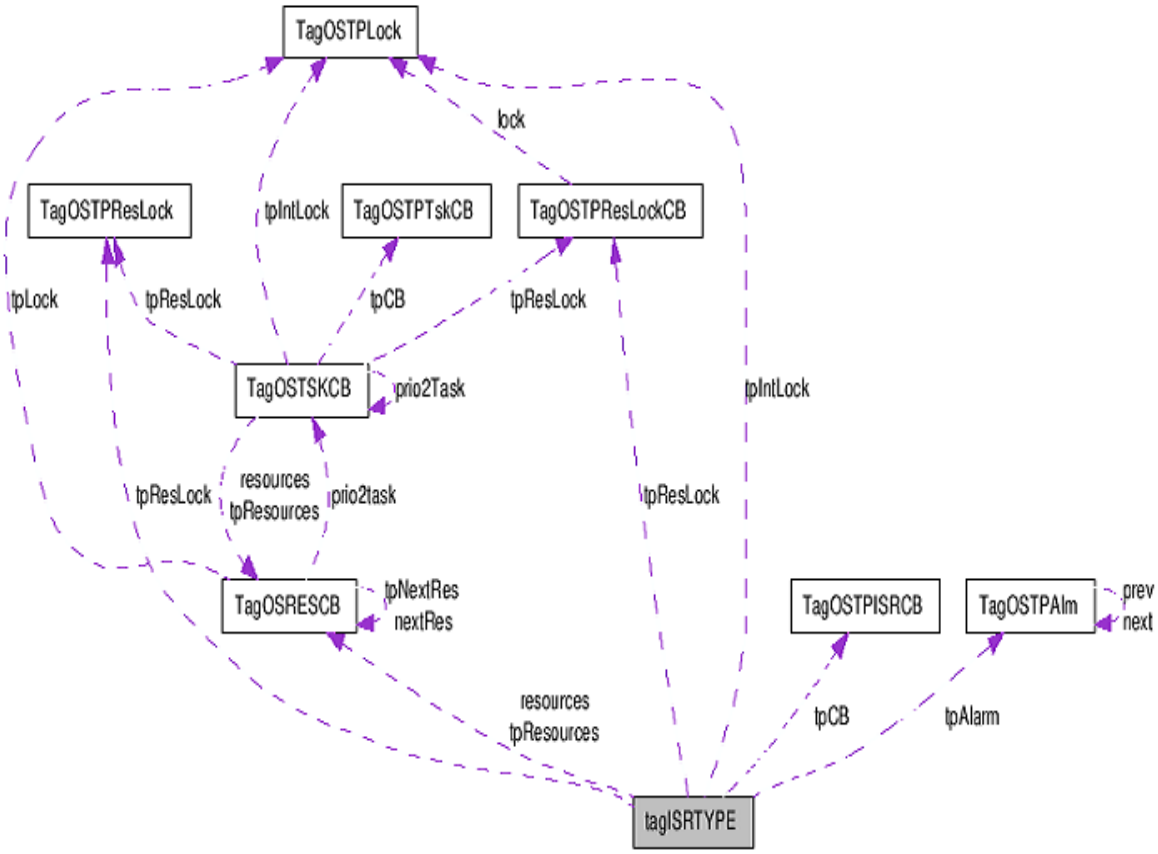
Data Structures

tagISRTYPE Struct Reference

{DD_104}

```
#include <s_isr_internal_types.h>
```

Collaboration diagram for tagISRTYPE:



Data Fields

- ->OSDWORD appMask
- ->OSDWORD * stackPtr
- ->OSDWORD * isrBos
- ->OSBYTE isKilled

- ->OSResType resources
- ->OSVOIDFUNCVOID userISR
- ->OSISRTYPE type
- ->OSWORD index
- ->OSBYTE prio
- ->OSTPISRCB * tpCB
- ->OSTPLOCK * tpIntLock
- ->OSTPRESLOCKCB * tpResLock
- ->OSTPTICKTYPE tpTimeFrame
- ->OSWORD tpCountLimit
- ->OSTPALMCB * tpAlarm
- ->OSQWORD tpBound
- ->OSWORD tpCount
- ->OSBYTE tpEnIntSrcFlag
- ->OSTPTICKTYPE tpExecBudget
- ->OSTPTICKTYPE tpRemained
- ->OSSIGNEDQWORD tpLast
- ->OSTPTICKTYPE tpIntLockTime
- ->OSTPRESLOCK * tpResLock
- ->OSResType tpResources
- ->OSWORD tpNumberResLock
- ->SpinlockIDType spinId
- ->OSWORD coreId
- ->ApplicationType appId

TagOSTPISRCB Struct Reference

{DD_105}

```
#include <Os_tp_internal_types.h>
```

Data Fields

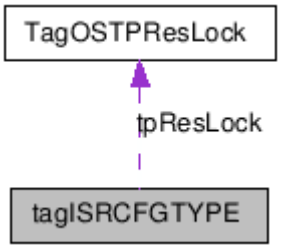
- -»OSObjectType object
- -»OSTPTICKTYPE execTime
- -»OSTPTICKTYPE remained

tagISRCFGTYPE Struct Reference

{DD_106}

```
#include <Os_isr_config.h>
```

Collaboration diagram for tagISRCFGTYPE:



Data Fields

- ->OSDWORD appMask
- ->OSDWORD * stackPtr
- ->OSDWORD * isrBos
- ->OSVOIDFUNCVOID userISR
- ->OSISRTYPE type
- ->OSWORD index
- ->OSBYTE prio
- ->OSTPTICKTYPE tpTimeFrame
- ->OSWORD tpCountLimit
- ->OSWORD tpAlarmId
- ->OSTPTICKTYPE tpExecBudget
- ->OSTPTICKTYPE tpIntLockTime
- const ->OSTPRESLOCK * tpResLock
- ->OSWORD tpNumberResLock
- ->OSWORD coreId
- ->ApplicationType appId

TagOSTPLock Struct Reference

{DD_107}

```
#include <Os_tp_internal_types.h>
```

TP control block for resource or interrupt locking time

Data Fields

- ->OSObjectType object
  Resource type&id or OBJECT_OS_INTERRUPT_LOCK
- ->OSTPTICKTYPE lockTime
  Interrupt or resource locking time
- ->OSObjectType owner
  Owner object type&id: TASK or ISR2
- ->OSWORD index
  Owner object type&id: TASK or ISR2

IOC management

IOC stands for Inter OS-Application Communicator.

IOC is responsible for the communication between OS-Applications and in particular for the communication crossing core and / or memory protection boundaries. IOC provides communication buffers, queues and functions / macros for protected access to these buffers / queues that can be used from configured applications.

IOC supports 1:1 (one sender, one receiver) and N:1 (N senders, one receiver) communications. IOC is configured and provides generated APIs for each IOC communication object. In case of N:1 communication, each sender has a separate API.

User does not have direct access to OS functions implementing IOC functionality. IOC APIs that are presented to the user, are in fact macros generated by SysGen. For example:

```
#define IocWrite_comm01(DataRef) OSIocWriteRef( 0,
(OSBYTEPTR) DataRef)
```

User calls API in the following form: IocWrite_<IocId>(<DataRef>) and it is expanded to the call of internal OS function that have communication id as a parameter. This correspondence of user symbolic identifiers to internal numeric communication identifiers is maintained by SysGen and is not visible to the user.

There are two types of IOC communications: unqueued (Last-is-Best, data semanctics) and queued (First-In-First-Out, event semantics).

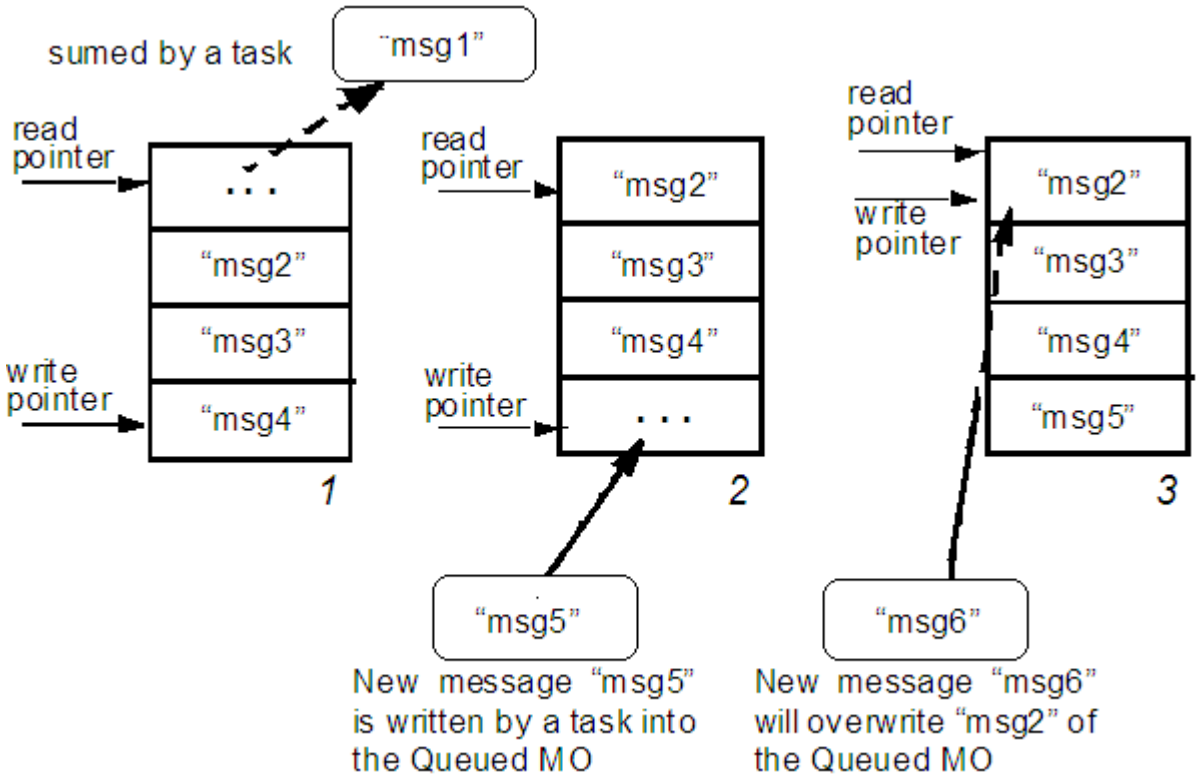
Last-is-best (unqueued) communications

Last-is-Best IOC communication has a single internal buffer to store data. Senders write to this buffer and it always stores a value written by the last sender. Whenever the receiver will invoke receive function it will get this last written value, hence the name of the communication, "last-is-best". It can be thought that such communication object respresents the current value of a state variable. Specification defines the following API functions for last-is-best IOC management: IocWrite, IocRead. They are implemented

by the following OS functions: OS_OSIocWriteRef, OS_OSIocWriteAcrossRef, OS_OSIocRead, OS_OSIocReadAcross.

Queued communications

Queued IOC communication maintains an internal queue of data items. The queue has a fixed (pre-configured) length. It has a head (first element) and a tail (last element). When sender puts new data it is put to the tail of the queue. When receiver reads data an item is read from the head of the queue. The items are maintained in FIFO



order, hence they are received in the same order they were sent. Specification defines the following API functions for queued IOC management: IocSend, IocReceive, IocEmptyQueue. They are implemented by the following OS functions: OS_OSIocSendRef, OS_OSIocSendAcrossRef, OS_OSIocReceive, OS_OSIocReceiveAcross.

IOC notifications

IOC may notify the receiver as soon as the transferred data is available for access on receiver side. There are three options for notification:

- Callback function
- Task activation
- Event setting

Task activation and event setting work just the same way as in alarms. Callback function is implemented as a separate ISR. When Write / Send function is called first data is copied to internal buffer / queue, and then if callback configured for this IOC communication, a dedicated interrupt is invoked. Then, user callback function is called from interrupt dispatcher like a usual user ISR2. Callback function has a right to call all OS services that can be called from ISR2. Priority of this IOC callback ISR is the highest ISR2 priority (`OSHIGHISRPRIO`).

For cross-core communications IOC services use OSHWSEM_RC hardware semaphore to protect against concurrent access to common data from different cores. This semaphore is also used in services that do not cross core boundaries, but when OSIOCCROSSCORECALLBACK is defined. In this case semaphore protects OsIOCCallbackCall structure that holds a communication id to invoke a callback for. Semaphore is needed because both current and remote core can request to invoke a callback on the current core and will write to this structure. The semaphore is locked at one time with interrupt disabling (via macro OSHWSemWaitDIS()).

IOC data types

{DD_348}

AUTOSAR OS establishes the following data types to operate with IOC:

- Std_ReturnType - a return type for all IOC functions defined in specification.
- OSWORD - used for internal numeric communication identifier.
- OSBYTEPTR - used for references to user data.
- tagIOCBUFFERINFO - structure of IOC last-is-best communication configuration.
- tagIOCBUFFERCOMMCB - structure of IOC last-is-best communication control block.
- tagIOCQUEUEINFO - structure of IOC queued communication configuration.
- tagIOCQUEUECOMMCB - structure of IOC queued communication control block.

Macro **OSNIOCBUFFERS** is generated by SysGen and is equal to the number of configured IOC last-is-best communications.

Macro **OSNIOCQUEUES** is generated by SysGen and is equal to the number of configured IOC queued communications.

Macro **OSIOCACTION** is generated by SysGen if there is IOC object(s) with configured action (ACTIVATETASK or SETEVENT).

Macro **OSIOCEVENT** is generated by SysGen if there is IOC object(s) with SETEVENT action.

Macro **OSIOCCALLBACK** is generated by SysGen if there is IOC object(s) with configured callback.

Macro **OSIOCCROSSCORECALLBACK** is generated by SysGen if there is cross-core IOC object(s) with configured callback.

IOC Management functions

OSInitIOC

{DD_303}

void OSInitIOC (void)

Initialization of IOC internal structures.

Returns:

none

Note:

Should be called from StartOS()

Implementation

- Fill control blocks for IOC buffer and queues from configuration structures generated by SysGen.
- If there are IOC last-is-best communications configured: call `OSIOC_Init()` - SysGen generated function used to initialise IOC buffer objects.

OS_OSIocWriteRef

{DD_304}

Std_ReturnType OS_OSIocWriteRef (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data transmission for "LastIsBest" (data semantic) communication. Used for unidirectional 1:1 and N:1 communication between OS-Applications located on the same core. Data is passed by reference.

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* A reference to data value to be sent

Returns:

IOC_E_OK no errors
E_OS_DISABLEDINT call while interrupts are disabled by OS services
E_OS_CALLEVEL service call from wrong context
E_OS_ACCESS access to the object denied for this application

Note:

Allowed at task and ISR2 call level.

Implementation

- Check context of service call
- Check access rights of current OS-Application to given IOC communication
- Enter OS critical section:
  - via macro *OSDIS()* if no cross-core callback(s) configured;

  - via macro *OSHWSemWaitDIS()* if cross-core callbacks are configured. HW semaphore OSHWSEM_RC is used in this case.
- Copy given data item to buffer of given IOC communication.
- If callback is configured for given IOC communication:
  - Check that the state of the receiver application is APPLICATION_ACCESSIBLE. If not, unlock semaphore OSHWSEM_RC (if cross-core callbacks are configured), leave critical section via macro *OSRI()* and return error code E_OS_ACCESS.
  - Invoke IOC callback interrupt on this core - place given communication id in OsIOCCallbackCall and set corresponding interrupt flag. Interrupt will not be executed until service leaves OS critical section.
- else (if callback is not configured for this IOC communication):
  - Unlock semaphore OSHWSEM_RC (if cross-core callbacks are configured)
  - Call OSIocAction() function to invoke action configured for this IOC communication
  - Call OS dispatcher via macro *OSDISPATCH()*. Dispatching is necessary here because IOC action could place some high priority task in ready state.
- Leave OS critical section via macro *OSRI()*
- Return error code IOC_E_OK

Used data

- OsIocBufferCommCB
- OsIocCallbackCall

OS_OSIocWriteAcrossRef

{DD_305}

Std_ReturnType OS_OSIocWriteAcrossRef (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data transmission for "LastIsBest" (data semantic) communication. Used for unidirectional 1:1 and N:1 communication between OS-Applications located on different cores. Data is passed by reference.

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* A reference to data value to be sent

Returns:

IOC_E_OK no errors
E_OS_DISABLEDINT call while interrupts are disabled by OS services
E_OS_CALLEVEL service call from wrong context
E_OS_ACCESS access to the object denied for this application

Note:

Allowed at task and ISR2 call level.

Implementation

- Check context of service call
- Check access rights of current OS-Application to given IOC communication
- Enter OS critical section via macro *OSHWSemWaitDIS()*. HW semaphore OSHWSEM_RC is used to protect critical section from the other core.
- Copy given data item to buffer of given IOC communication.
- If callback is configured for given IOC communication:
  - If receiver application is on the same core:
    - Check that the state of the receiver application is APPLICATION_ACCESSIBLE. If not, unlock semaphore OSHWSEM_RC, leave critical section via macro *OSRI()* and return error code E_OS_ACCESS.
    - Invoke IOC callback interrupt on this core - place given communication id in OsIOCCallbackCall and set corresponding interrupt flag. Interrupt will not be executed until service leaves OS critical section. Semaphore OSHWSEM_RC will be unlocked as a part of interrupt handling in this case.
  - If receiver application is on another core:
    - Invoke IOC callback interrupt on another core - place given

communication id in OsIOCCallbackCall, set OSRC_WAIT state in corresponding OsRC structure and set corresponding interrupt flag for another core. This is similar to remote call invocation with the exception that another interrupt is used for this.

• Call -»OSWaitRC() function to wait for remote call completion on remote core

• Unlock semaphore OSHWSEM_RC

• Call OS dispatcher via macro *OSDISPATCH()*. Dispatching is necessary here because while waiting for remote call to finish on another core this core could have been forced to perform some service by the means of oncoming remote call from another core. This service could place some high priority task in ready state.

- else (if callback is not configured for this IOC communication):
  - Unlock semaphore OSHWSEM_RC
  - Call OSIocAction() function to invoke action configured for this IOC communication
  - Call OS dispatcher via macro *OSDISPATCH()*. Dispatching is necessary here because IOC action could place some high priority task in ready state.
- Leave OS critical section via macro *OSRI()*
- Return error code IOC_E_OK

Used data

- OsIocBufferCommCB
- OsIocCallbackCall

OS_OSIocRead

{DD_306}

Std_ReturnType OS_OSIocRead (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data reception for "LastIsBest" (data semantic) communication. Used for unidirectional communication between OS-Applications located on the same core.

Parameters:

[in] *commId* Numeric IOC communication identifier
[out] *dataRef* Data reference to be filled with the received data

Returns:

IOC_E_OK no errors
E_OS_DISABLEDINT call while interrupts are disabled by OS services
E_OS_CALLEVEL service call from wrong context
E_OS_ACCESS access to the object denied for this application
E_OS_ILLEGAL_ADDRESS an invalid address is given as a parameter to the service

Note:

Allowed at task and ISR2 call level.

Implementation

- Check context of service call
- Check access rights of current OS-Application to given IOC communication
- Check write access of current OS-Application to given pointer *dataRef*.
- Enter OS critical section via macro *OSDIS()*.
- Copy data item from buffer of given IOC communication to given pointer *dataRef*.
- Leave OS critical section via macro *OSRI()*
- Return error code IOC_E_OK

Used data

- OsIocBufferCommCB

OS_OSIocReadAcross

{DD_307}

Std_ReturnType OS_OSIocReadAcross (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data reception for "LastIsBest" (data semantic) communication. Used for unidirectional communication between OS-Applications located on different cores.

Parameters:

[in] *commId* Numeric IOC communication identifier
[out] *dataRef* Data reference to be filled with the received data

Returns:

IOC_E_OK no errors
E_OS_DISABLEDINT call while interrupts are disabled by OS services

E_OS_CALLEVEL service call from wrong context
E_OS_ACCESS access to the object denied for this application
E_OS_ILLEGAL_ADDRESS an invalid address is given as a parameter to the service

Note:

Allowed at task and ISR2 call level.

Implementation

- Check context of service call
- Check access rights of current OS-Application to given IOC communication
- Check write access of current OS-Application to given pointer *dataRef*.
- Enter OS critical section via macro *OSHWSemWaitDIS()*. HW semaphore OSHWSEM_RC is used to protect critical section from the other core.
- Copy data item from buffer of given IOC communication to given pointer *dataRef*.
- Unlock semaphore OSHWSEM_RC
- Leave OS critical section via macro *OSRI()*
- Return error code IOC_E_OK

Used data

- - OsIocBufferCommCB

OS_OSIocSendRef

{DD_308}

Std_ReturnType OS_OSIocSendRef (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data transmission for "Queued" (event semantic) communication. Used for unidirectional 1:1 and N:1 communication between OS-Applications located on the same core. Data is passed by reference.

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* A reference to data value to be sent

Returns:

IOC_E_OK no errors
IOC_E_LIMIT internal communication queue is full

E_OS_DISABLEDINT call while interrupts are disabled by OS services
E_OS_CALLEVEL service call from wrong context
E_OS_ACCESS access to the object denied for this application

Note:

Allowed at task and ISR2 call level.

Implementation

- Check context of service call
- Check access rights of current OS-Application to given IOC communication
- Enter OS critical section:
  - via macro *OSDIS()* if no cross-core callback(s) configured;
  - via macro *OSHWSemWaitDIS()* if cross-core callbacks are configured. HW semaphore OSHWSEM_RC is used in this case.
- If queue head and tail pointers are equal:
  - If number of items in the queue is equal to the queue size, then queue is full - set queue overflow flag, unlock semaphore OSHWSEM_RC (if cross-core callbacks are configured), leave critical section via macro *OSRI()* and return error code IOC_E_LIMIT.
- Copy given data item to buffer of given IOC communication.
- Increment number of items in the queue. Advance queue head pointer, if it reaches queue size, wrap it (assign zero).
- If callback is configured for given IOC communication:
  - Check that the state of the receiver application is APPLICATION_ACCESSIBLE. If not, unlock semaphore OSHWSEM_RC (if cross-core callbacks are configured), leave critical section via macro *OSRI()* and return error code E_OS_ACCESS.
  - Invoke IOC callback interrupt on this core - place given communication id in OsIOCCallbackCall and set corresponding interrupt flag. Interrupt will not be executed until service leaves OS critical section.
- else (if callback is not configured for this IOC communication):
  - Unlock semaphore OSHWSEM_RC (if cross-core callbacks are configured)

  - Call OSIocAction() function to invoke action configured for this IOC communication
  - Call OS dispatcher via macro *OSDISPATCH()*. Dispatching is necessary here because IOC action could place some high priority task in ready state.
- Leave OS critical section via macro *OSRI()*
- Return error code IOC_E_OK

Used data

- OsIocQueueCommCB
- OsIocCallbackCall

OS_OSIocSendAcrossRef

{DD_309}

Std_ReturnType OS_OSIocSendAcrossRef (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data transmission for "Queued" (event semantic) communication. Used for unidirectional 1:1 and N:1 communication between OS-Applications located on different cores. Data is passed by reference.

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* A reference to data value to be sent

Returns:

IOC_E_OK no errors
IOC_E_LIMIT internal communication queue is full
E_OS_DISABLEDINT call while interrupts are disabled by OS services
E_OS_CALLEVEL service call from wrong context
E_OS_ACCESS access to the object denied for this application

Note:

Allowed at task and ISR2 call level.

Implementation

- Check context of service call
- Check access rights of current OS-Application to given IOC communication
- Enter OS critical section via macro *OSHWSemWaitDIS()*. HW semaphore OSHWSEM_RC is used to protect critical section from the other core.

- If queue head and tail pointers are equal:
  - If number of items in the queue is equal to the queue size, then queue is full - set queue overflow flag, unlock semaphore OSHWSEM_RC, leave critical section via macro *OSRI()* and return error code IOC_E_LIMIT.
- Copy given data item to buffer of given IOC communication.
- Increment number of items in the queue. Advance queue head pointer, if it reaches queue size, wrap it (assign zero).
- If callback is configured for given IOC communication:
  - If receiver application is on the same core:
    • Check that the state of the receiver application is APPLICATION_ACCESSIBLE. If not, unlock semaphore OSHWSEM_RC, leave critical section via macro *OSRI()* and return error code E_OS_ACCESS.

    • Invoke IOC callback interrupt on this core - place given communication id in OsIOCCallbackCall and set corresponding interrupt flag. Interrupt will not be executed until service leaves OS critical section. Semaphore OSHWSEM_RC will be unlocked as a part of interrupt handling in this case.
  - If receiver application is on another core:
    • Invoke IOC callback interrupt on another core - place given communication id in OsIOCCallbackCall, set OSRC_WAIT state in corresponding OsRC structure and set corresponding interrupt flag for another core. This is similar to remote call invocation with the exception that another interrupt is used for this.

    • Call -»OSWaitRC() function to wait for remote call completion on remote core

    • Unlock semaphore OSHWSEM_RC

    • Call OS dispatcher via macro *OSDISPATCH()*. Dispatching is necessary here because while waiting for remote call to finish on another core this core could have been forced to perform some service by the means of oncoming remote call from another core. This service could place some high priority task in ready state.
- else (if callback is not configured for this IOC communication):
  - Unlock semaphore OSHWSEM_RC

  - Call OSIocAction() function to invoke action configured for this IOC communication
  - Call OS dispatcher via macro *OSDISPATCH()*. Dispatching is necessary here because IOC action could place some high priority task in ready state.
- Leave OS critical section via macro *OSRI()*
- Return error code IOC_E_OK

Used data

- OsIocQueueCommCB
- OsIocCallbackCall

OS_OSIocReceive

{DD_310}

Std_ReturnType OS_OSIocReceive (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data reception for "Queued" (event semantic) communication. Used for unidirectional communication between OS-Applications located on the same core.

Parameters:

[in] *commId* Numeric IOC communication identifier
[out] *dataRef* Data reference to be filled with the received data

Returns:

IOC_E_OK no errors
IOC_E_NO_DATA no data is available for reception
IOC_E_LOST_DATA internal queue overflow happened during last OS_OSIocSend and data passed with it was lost. There is no error in data returned by OS_OSIocReceive however
E_OS_DISABLEDINT call while interrupts are disabled by OS services
E_OS_CALLEVEL service call from wrong context
E_OS_ACCESS access to the object denied for this application
E_OS_ILLEGAL_ADDRESS an invalid address is given as a parameter to the service

Note:

Allowed at task and ISR2 call level.

Implementation

- Check context of service call
- Check access rights of current OS-Application to given IOC communication

- Check write access of current OS-Application to given pointer *dataRef*.
- Enter OS critical section via macro *OSDIS()*
- If there are items in the queue:
  - Copy data item from queue tail of given IOC communication to given pointer *dataRef*.
  - Decrement number of items in the queue
  - Increment tail pointer or wrap it to zero if it reached queue size
  - If overflow flag is not set: leave OS critical section via macro *OSRI()* and return error code IOC_E_OK.
  - If overflow flag is set: clear flag, leave OS critical section via macro *OSRI()* and return error code IOC_E_LOST_DATA.
- If there are no items in the queue:
  - Leave OS critical section via macro *OSRI()*
  - Return error code IOC_E_NO_DATA

Used data

- OsIocQueueCommCB

OS_OSIocReceiveAcross

{DD_311}

Std_ReturnType OS_OSIocReceiveAcross (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data reception for "Queued" (event semantic) communication. Used for unidirectional communication between OS-Applications located on different cores.

Parameters:

[in] *commId* Numeric IOC communication identifier
[out] *dataRef* Data reference to be filled with the received data

Returns:

IOC_E_OK no errors
IOC_E_NO_DATA no data is available for reception
IOC_E_LOST_DATA internal queue overflow happened during last OS_OSIocSend and data passed with it was lost. There is no error in data returned by OS_OSIocReceive however

E_OS_DISABLEDINT call while interrupts are disabled by OS services
E_OS_CALLEVEL service call from wrong context
E_OS_ACCESS access to the object denied for this application
E_OS_ILLEGAL_ADDRESS an invalid address is given as a parameter to the service

Note:

Allowed at task and ISR2 call level.

Implementation

- Check context of service call
- Check access rights of current OS-Application to given IOC communication
- Check write access of current OS-Application to given pointer *dataRef*.
- Enter OS critical section via macro *OSHWSemWaitDIS()*. HW semaphore OSHWSEM_RC is used to protect critical section from the other core.
- If there are items in the queue:
  - Copy data item from queue tail of given IOC communication to given pointer *dataRef*.
  - Decrement number of items in the queue
  - Increment tail pointer or wrap it to zero if it reached queue size
  - If overflow flag is not set: unlock semaphore OSHWSEM_RC, leave OS critical section via macro *OSRI()* and return error code IOC_E_OK.
  - If overflow flag is set: clear flag, unlock semaphore OSHWSEM_RC, leave OS critical section via macro *OSRI()* and return error code IOC_E_LOST_DATA.
- If there are no items in the queue:
  - Unlock semaphore OSHWSEM_RC
  - Leave OS critical section via macro *OSRI()*
  - Return error code IOC_E_NO_DATA

Used data

- OsIocQueueCommCB

OS_OSIocEmptyQueue

{DD_312}

Std_ReturnType OS_OSIocEmptyQueue (OSWORD *commId*)

Deletes the content of the IOC internal communication queue.

Parameters:

[in] *commId* Numeric IOC communication identifier

Returns:

IOC_E_OK no errors
E_OS_DISABLEDINT call while interrupts are disabled by OS services
E_OS_CALLEVEL service call from wrong context
E_OS_ACCESS access to the object denied for this application

Note:

Allowed at task and ISR2 call level.

Implementation

- Check context of service call
- Check access rights of current OS-Application to given IOC communication
- Enter OS critical section via macro *OSDIS()*
- Clear contents of given IOC communication queue:
  - Fill all data items with zeros
  - Set to zero head, tail and number of items
  - Clear overflow flag
- Leave OS critical section via macro *OSRI()*
- Return error code IOC_E_OK

Used data

- OsIocQueueCommCB

OS_OSIocEmptyQueueAcross

{DD_313}

Std_ReturnType OS_OSIocEmptyQueueAcross (OSWORD *commId*)

Deletes the content of the IOC internal communication queue. Used for cross-core communications.

Parameters:

[in] *commId* Numeric IOC communication identifier

Returns:

IOC_E_OK no errors
E_OS_DISABLEDINT call while interrupts are disabled by OS services
E_OS_CALLEVEL service call from wrong context
E_OS_ACCESS access to the object denied for this application

Note:

Allowed at task and ISR2 call level.

Implementation

- Check context of service call
- Check access rights of current OS-Application to given IOC communication
- Enter OS critical section via macro *OSHWSemWaitDIS()*. HW semaphore OSHWSEM_RC is used to protect critical section from the other core.
- Clear contents of given IOC communication queue:
  - Fill all data items with zeros
  - Set to zero head, tail and number of items
  - Clear overflow flag
- Unlock semaphore OSHWSEM_RC
- Leave OS critical section via macro *OSRI()*
- Return error code IOC_E_OK

Used data

- OsIocQueueCommCB

OSIocAction

{DD_314}

void OSIocAction (OSIOCACT * *action*)

Invoking of configured IOC action

Parameters:

[in] *action* Reference to IOC action to be invoked

Returns:

none

Note:

This function is used internally in OS. It is called from IOC Write/Send services.

Implementation

- Check that task index in the action is not invalid. Return from the function if it is invalid.
- If task index corresponds to task on another core, call function *OSRemoteNotifyAction*(). It will make a remote call to another core.
- If event in the action is not empty:
  - If the task is not suspended, call internal OS service *OSErrorHook_noPar()* with error code *E_OS_STATE;*
  - Else set event for the task
- else (if action is 'activate task'):
  - If task is already activated call internal OS service *OSErrorHook_noPar()* with error code *E_OS_LIMIT*;
  - Else clear field '*set_event*' in the task control block and activate the task

Used data

- OSTaskTable

OSIocPrepareCallbackISR

{DD_315}

void OSIocPrepareCallbackISR (OS_ISRTYPE* *isrPtr*)

This function is called from interrupt dispatcher. It changes isrPtr given as a parameter so that an IOC callback can be invoked like a usual user ISR2.

Parameters:

[in] *istPtr* A pointer to ISR description structure that shall be changed

Returns:

OSTRUE - if interrupt flag is set and callback shall be called
OSFALSE - if callback shall not be called

Note:

none

Implementation

- Check that IOC callback interrupt flag is set.
  - If flag is set, clear it
  - If flag is not set, return OSFALSE
- From OsIOCCallbackCall get the communication id for which callback shall be called
- In case of multi-core configuration:
  - If callback was invoked from the same core and if cross-core callbacks are configured - unlock semaphore OSHWSEM_RC
  - If callback was invoked from another core - set OSRC_FINISH to release another core from waiting.
- Check that the state of the receiver application is APPLICATION_ACCESSIBLE. If not, return OSFALSE.
- Copy receiver application id and pointer to callback function from corresponding IOC communication control block to isrPtr
- Return OSTRUE

Used data

- OsIocCallbackCall

- OsRC

OSIocWriteGroup

{DD_354}

Std_ReturnType OS_OSIocWriteGroup (OSWORD *commId*, OSBYTEPTR *dataRef*)

This function performs data transmission for "LastIsBest" (data semantic) communication type.
Used for unidirectional 1:1 and N:1 communication between OS-Applications located on the same core.
Data is passed by reference..

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* Data reference to be filled with the sent data

Returns:

IOC_E_OK	no errors
IOC_E_NOK	call while interrupts are disabled by OS services
IOC_E_NOK	service call from wrong context
IOC_E_NOK	access to the object denied for this application
IOC_E_NOK	an invalid address is given as a parameter to a service

Note:

none

OSIocWriteGroupAcross

{DD_355}

Std_ReturnType OS_OSIocWriteGroupAcross (OSWORD *commId*, OSBYTEPTR *dataRef*)

This function performs data transmission for "LastIsBest" (data semantic) communication type.
Used for unidirectional 1:1 and N:1 communication between OS-Applications located on the different cores.
Data is passed by reference..

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* Data reference to be filled with the sent data

Returns:

IOC_E_OK	no errors
IOC_E_NOK	call while interrupts are disabled by OS services
IOC_E_NOK	service call from wrong context
IOC_E_NOK	access to the object denied for this application
IOC_E_NOK	an invalid address is given as a parameter to a service

Note:

none

OSIocReadGroup

{DD_356}

Std_ReturnType OS_OSIocReadGroup (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data reception for "LastIsBest" (data semantic) communication type.

Used for unidirectional communication between OS-Applications located on the same core.

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* Data reference to be filled with the sent data

Returns:

IOC_E_OK	no errors
IOC_E_NOK	call while interrupts are disabled by OS services
IOC_E_NOK	service call from wrong context
IOC_E_NOK	access to the object denied for this application
IOC_E_NOK	an invalid address is given as a parameter to a service

Note:

none

OSIocReadGroupAcross

{DD_357}

Std_ReturnType OS_OSIocReadGroupAcross (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data reception for "LastIsBest" (data semantic) communication type.
Used for unidirectional communication between OS-Applications located on the different core.

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* Data reference to be filled with the sent data

Returns:

IOC_E_OK	no errors
IOC_E_NOK	call while interrupts are disabled by OS services
IOC_E_NOK	service call from wrong context
IOC_E_NOK	access to the object denied for this application
IOC_E_NOK	an invalid address is given as a parameter to a service

Note:

none

OSIocSendGroup

{DD_358}

Std_ReturnType OS_OSIocSendGroup (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data transmission for "Queued" (event semantic) communication type.
Used for unidirectional 1:1 and N:1 communication between OS-Applications located on the same core.
Data is passed by reference..

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* Data reference to be filled with the sent data

Returns:

IOC_E_OK	no errors
IOC_E_LIMIT	IOC internal communication queue is full
IOC_E_NOK	call while interrupts are disabled by OS services
IOC_E_NOK	service call from wrong context
IOC_E_NOK	access to the object denied for this application

Note:

none

OSIocSendGroupAcross

{DD_359}

Std_ReturnType OS_OSIocSendGroupAcross (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data transmission for "Queued" (event semantic) communication type.
Used for unidirectional 1:1 and N:1 communication between OS-Applications located on the different core.
Data is passed by reference.

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* Data reference to be filled with the sent data

Returns:

IOC_E_OK	no errors
IOC_E_LIMIT	IOC internal communication queue is full
IOC_E_NOK	call while interrupts are disabled by OS services
IOC_E_NOK	service call from wrong context
IOC_E_NOK	access to the object denied for this application

Note:

none

OSIocReceiveGroup

{DD_360}

Std_ReturnType OS_OSIocReceiveGroup (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data reception for "Queued" (event semantic) communication type.
Used for unidirectional communication between OS-Applications located on the same core.

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* Data reference to be filled with the sent data

Returns:

IOC_E_OK	no errors
IOC_E_NO_DATA	no data is available for reception
IOC_E_LOST_DATA	internal queue oveflow happened during last OSIocSend and data passed with it was lost. There is no error in data returned by OSIocReceive however
IOC_E_NOK	call while interrupts are disabled by OS services
IOC_E_NOK	service call from wrong context
IOC_E_NOK	access to the object denied for this application
IOC_E_NOK	an invalid address is given as a parameter to a service

Note:

none

OSIocReceiveGroupAcross

{DD_361}

Std_ReturnType OS_OSIocReceiveGroupAcross (OSWORD *commId*, OSBYTEPTR *dataRef*)

Performs data reception for "Queued" (event semantic) communication type.
Used for unidirectional communication between OS-Applications located on the different core.

Parameters:

[in] *commId* Numeric IOC communication identifier
[in] *dataRef* Data reference to be filled with the sent data

Returns:

IOC_E_OK no errors
IOC_E_NO_DATA no data is available for reception
IOC_E_LOST_DATA internal queue oveflow happened during last OSIocSend
and data passed with it was lost. There is no error in data returned by OSIocReceive however
IOC_E_NOK call while interrupts are disabled by OS services
IOC_E_NOK service call from wrong context
IOC_E_NOK access to the object denied for this application
IOC_E_NOK an invalid address is given as a parameter to a service

Note:

none

Data structures

tagIOCBUFFERCOMMCB Struct Reference

Control block for IOC buffer (last-is-best) communication

Data fields

- OSBYTEPTR dataBuf

  Pointer to the data buffer itself

- OSBYTE dataSize

  Size of a single data item in this communication

- OSCALLBACK callBack

  Pointer to the user callback function

- OSDWORD sndAppMask

  Bitmask of OS-Applications that have a right to call Write services for this communication

- OSDWORD rcvAppMask

  Bitmask of OS-Applications that have a right to call Read services for this communication

- OSAPPLICATIONTYPE rcvAppId

  Receiver application id
- OSWORD rcvCoreId

  Core ID of the receiver application
- OSIOCACT action

  Action configured for this communication

tagIOCQUEUECOMMCB Struct Reference

Control block for IOC queued communication

Data fields

- OSBYTEPTR dataQueue

  Pointer to the data queue itself
- OSBYTE dataSize

  Size of a single data item (queue element) in this communication
- OSDWORD queueSize

  Size of the queue - number of items it can hold
- OSCALLBACK callBack

  Pointer to the user callback function
- OSDWORD sndAppMask

  Bitmask of OS-Applications that have a right to call Send services for this communication
- OSDWORD rcvAppMask

  Bitmask of OS-Applications that have a right to call Receive services for this communication
- OSAPPLICATIONTYPE rcvAppId

  Receiver application id
- OSWORD rcvCoreId

  Core ID of the receiver application
- OSIOCACT action

  Action configured for this communication

- OSDWORD nItems

  Current number of data items in the queue

- OSDWORD head

  Number of first element in the queue (where Send operation will write new data)

- OSDWORD tail

  Number of last element in the queue (from where Receive operation will read data)

- OSBYTE overflow

  Overflow indication flag. It is set if Send operation wasn't able to put new data in the queue because queue was already full.

TagOSIOCACT Struct Reference

Structure of the IOC action. Action could be either 'activate task' or 'set event'

Data fields

- OSWORD taskIndex

  Task to start or set event for. If it is OSINVALID_TASK, then no action is configured.

- EventMaskType event

  Event to set. If it's empty alarm activates task.

TagOSIOCCallbackCB Struct Reference

Structure that is used for IOC callbacks invocation

Data fields

- OSWORD senderCoreId

  Core that requested to call callback

- OSWORD commId

  Communication id for which callback shall be called

Resource Management

{DD_127}

The Resource management is used to coordinate concurrent accesses of several tasks to shared resources, e.g. management entities (scheduler), program sequences (critical sections), memory or hardware areas. In general, the Resource management is provided in all Conformance Classes, but it is fully supported only beginning from the BCC2 Conformance Class. In the BCC1 Conformance Class only the scheduler is to be treated as a specific OS Resource which can be used by tasks, the Freescale AUTOSAR OS will extend BCC1 functionality to full support of Resources.

Resource management ensures that:

- two tasks or ISRs cannot "own" the same Resource at the same time,
- priority inversion cannot arise while Resources are used,
- deadlocks do not occur by use of these Resources,
- access to Resources never results in a waiting state for the task.

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

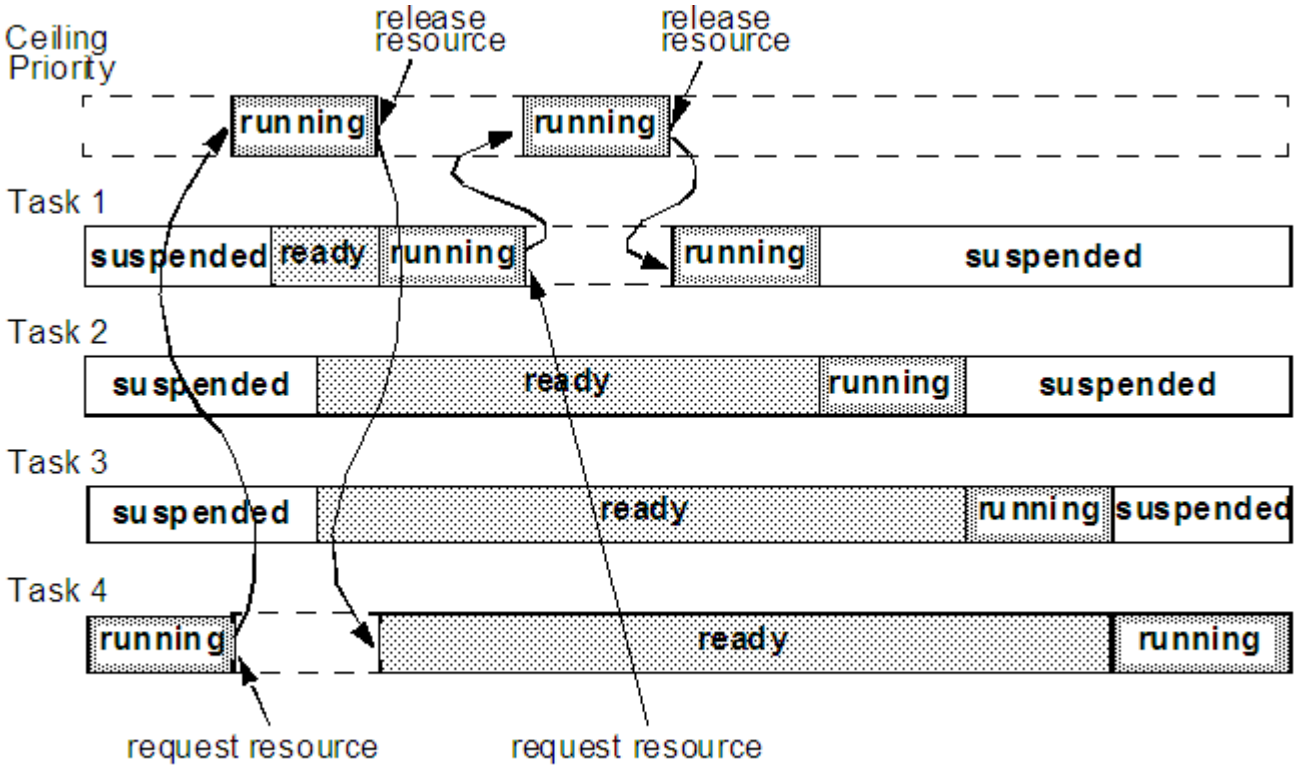
- full- or mixed-preemptive scheduling;
- non-preemptive scheduling, if the user intends to have the application code executed under other scheduling policies too;
- resource sharing between tasks and ISRs or between several ISRs.

Resources cannot be occupied by more than one task or ISR at a time. The Resource that is now occupied by a task / ISR must be released before other tasks / ISRs can get it. The AUTOSAR operating system ensures that tasks are only transferred from the ready state into the running state, if all Resources which might be occupied by that task during its execution have been released. Consequently, no situation occurs in which a task tries to access an occupied Resource. The special mechanism is used by the AUTOSAR Operating System to provide such behavior. The number of Resources is defined during compile time, and corresponding data type (typedef) is created.

OSEK Priority Ceiling Protocol

{DD_127}

The variation of Priority Ceiling Protocol called "OSEK Priority Ceiling Protocol" is implemented in the AUTOSAR Operating System as a Resource management discipline. When a task occupies a Resource the system temporary changes its priority. It is



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

Figure 0.3 OSEK Priority Ceiling Protocol

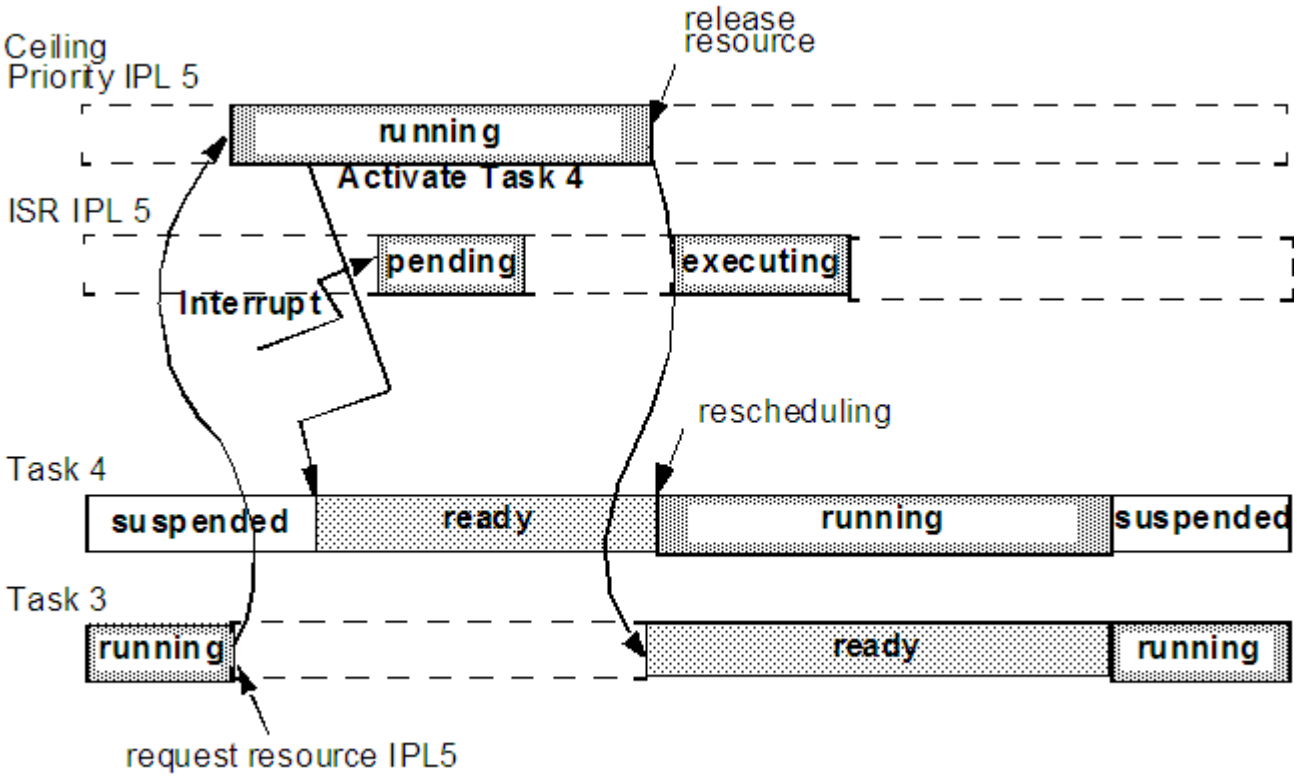
In the figure above Task 1 has the highest priority, Task 4 has the lowest Priority. The Resource has the priority greater than or equal to the Task 1 priority. When Task 4 occupies the Resource it gets the priority not less then Task 1, therefore it cannot be preempted by ready Task 1 until it release the Resource. Just after the Resource is released, Task 4 is returned to its low priority and becomes ready, and Task 1 becomes the running task. When Task 1, in its turn, occupies the Resource, its priority is also changed to the Ceiling Priority.

Extension of Resource Management for ISRs

{DD_128}

The Extension Resource management is used to coordinate concurrent accesses of several tasks and ISRs to shared resources, e.g. program sequences (critical sections), memory or hardware areas. In general, the Extension of Resource management is fully applicable only when hardware supports several interrupt processing levels (like CPU32 or MPC5xx). In some cases the hardware fully manages the hardware priority levels (CPU32), while on certain hardware it requires software algorithms (MPC5xx). In two-level CPUs (just enabled or disabled interrupt recognition) this extension also may be implemented with obvious restriction that any resource which is accessed by ISR always has hardware priority equivalent "interrupt recognition disabled". In any applicable case, processor has a mean to disable recognition of interrupts up to and including certain interrupt priority level (IPL). Hardware interrupt priority levels are statically assigned to all ISRs. All tasks have lowest possible IPL when started. During internal processing OS may disable interrupts either up to and including the highest level, or up to and including the middle level. In latest case all ISR category 2 must have their IPLs below IPL used by OS, while ISRs category 1 may have theirs IPLs above this level. When resource is shared between ISRs and task or between ISRs, it has an effective ceiling priority higher or equal the IPL of the highest ISR which accesses this resource (the calculation is done at system generation phase). When task or ISR gets resource, OS automatically sets the recognition level equal to the resource IPL,

thus effectively preventing occurring any interrupts having priority level below or equal this IPL. When this resource is released, OS



restores the recognition level, and performs rescheduling, if resource allocation was done from task.

Figure 0.4 OSEK Priority Ceiling Protocol for an Extension of Resource Management

In the figure above Task 4 has the highest priority, Task 3 has the lower Priority. The Resource has the hardware priority IPL 5 (equal to ISR IPL 5 priority). When Task 3 occupies the Resource it sets the effective IPL equal to 5, therefore all ISRs at IPL 5 and below are not recognized until task releases the Resource. During Resource occupation Task 3 activates higher priority Task 4, but no rescheduling occurs because Task 4 has lowest IPL. Just after the Resource is released, tasks are rescheduled, and Task 4 becomes running, but it is interrupted by recognized ISR IPL 5, which immediately starts execution. Task 3 is returned to its low priority and becomes ready. When ISR IPL 5 ends, Task 4 occupies CPU.

Groups of tasks

{DD_129}

The operating system allows tasks to combine aspects of preemptive and non preemptive scheduling by defining groups of tasks. For tasks which have the same or lower priority as the highest priority within a group, the tasks within the group behave like non preemptable tasks: rescheduling will only take place at the points of rescheduling. For tasks with a higher priority than the highest priority within the group, tasks within the group behave like preemptable tasks. Non preemptable tasks may be considered as the tasks with a special internal resource of highest task priority assigned.

Internal Resources

{DD_130}

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

generation. If an internal resource is assigned to a task, the internal resource is managed as follows:

- the resource is automatically taken when the task enters the running state, except when it has already taken the resource. As a result, the priority of the task is automatically changed to the ceiling priority of the internal resource.
- at the points of rescheduling the resource is automatically released (internal resources are not released when a task is preempted). The implementation may optimize, e.g. only free / take the resource within the system service Schedule if there is a need for rescheduling.

Non preemptable tasks may be seen as a special group with an internal resource of the same priority as *RES_SCHEDULER* assigned. Internal resources can be used in all cases when it is necessary to avoid unwanted rescheduling within a group of tasks. More than one group (more than one internal resource) can be defined in a system. The general restriction on some system calls that they must not be called with resources occupied does not apply to internal resources, as internal resources are handled within those calls. However, all standard resources must be released before the internal resource can be released. The tasks which have the same internal resource assigned cover a certain range of priorities. It is possible to have tasks which do not use this internal resource in the same priority range. The application has to decide if this makes sense.

Data Types

{DD_131}

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

- **-»**ResourceType The abstract data type for resource identification;

The following data types are used by the OS internally:

- -»TagOSRESCFG - resource configuration table structure
- -»TagOSRESCB - resource control block

The OS uses following global objects to handle resources:

- `const ->OSRESCFG ->OsResCfg[->OSNRESS + ->OSNRESS]` - the resource configuration table. It's generated by the SysGen.
- `->OSRESCB ->OsResources[->OSNALLRES]` - the resource control blocks table.

Macros **OSNRESS** and **OSNISRRESS** are used by the OS to access to the resource configuration table.

The macro **OSNRESS** is defined as the number of RESOURCEs with task priority, STANDARD or LINKED plus resscheduler in the OS configuration.

The macro **OSNISRRESS** is defined as the number of RESOURCEs which are used by ISRs.

The macro **OSNALLRES** is defined as (**OSNRESS + OSNISRRESS)**

Constants

{DD_192}

RES_SCHEDULER - constant of data type ResourceType corresponded to Scheduler resource

Resource Management functions

DeclareResource

{DD_293}

void DeclareTask(TaskID)

A dummy declaration, intended for compatibility with other OSEK versions

OSInitResource

{DD_132}

void OSInitResource ()

Returns:

none

Note:

none

Implementation

- Fill the resource control blocks table -»OsResources using the resource configuration table -»OsResCfg generated by the SysGen:

Used data

-»OsResources

-»OsResCfg

OS_GetResource

{DD_133}

-»StatusType OS_GetResource (-»ResourceType *resId*)

Occupies resource.

Parameters:

[in] *resId* - a reference to the resource

Returns:

Standard:
E_OK no error.
E_OS_CORE resource is bound to another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the resource identifier is invalid.
E_OS_ACCESS attempt to get resource which is already occupied by any task or ISR, or the assigned in OIL priority of the calling task or interrupt routine is higher than the calculated ceiling priority; .
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

This service changes current priority of the calling task or ISR according to ceiling priority protocol for resource management. GetResource serves to enter critical section in the code and blocks

execution of any task or ISR which can get the resource <ResID>. A critical section must always be left using -»ReleaseResource within the same task or ISR.

Particularities:

This function is fully supported in all Conformance Classes. It is Freescale OS extension of the AUTOSAR OS because OSEK/VDX specifies full support only beginning from BCC2. Nested resource occupation is only allowed if the inner critical sections are completely executed within the surrounding critical section. Nested occupation of one and the same resource is forbidden. The service call is allowed on task level and ISR level, but not in hook routines. This service is not implemented if no standard resources are defined in the configuration file. Regarding Extended Tasks, please note that -»WaitEvent within a critical section is prohibited.

Implementation

- Check the context of the service call
- Check if the resource has the invalid identifier (returns *E_OS_ID*)
- Check if the resource belong to another core (returns E_OS_CORE)
- Check if attempt to access occupied resource (returns *E_OS_ACCESS*)
- Get the reference to resource control block from -»OsResources by identifier <resId>
- If ISRs are configured

  - Check call level: if there are resources used by ISR in the OS configuration
    - If ISR resources exist and resource priority is lower than assigned ISR priority, returns *E_OS_ACCESS*
    - Else if resource is not for ISR, returns *E_OS_ACCESS*
- If it is the task resource with lower priority, returns *E_OS_ACCESS*
-
- Enter OS critical section via *OSDIS()*
- If access rights of current OS-Application to given resource are insufficient or state of OS-Application to which belong given

task is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code

- Call -»OSTPStartResLockTime to start timing protection for resource's lock time
- If there are resources used by ISR in the OS configuration

  - If ISR resources exist

    – block scheduler by setting -»OsIsrLevel to ISR level for task, which got resource with ISR priority

    – Fill resource control block table

    – Save resource in interrupts state
- If it is task level resource

  Fill resource control block table

  Set ’resources’ field of -»OsRunning to filled resource control block
- Leave the OS critical section via the macro *OSRI()*
- If there are resources used by ISR in the OS configuration and it is ISR level resource, increment -»OsISRResourceCounter
- Return E_OK

Used data

-»OsIsrLevel

-»OsRunning

-»OsISRResourceCounter

OSReleaseISRResources

{DD_134}

void OSReleaseISRResources (-»OSRESCB ** *resources*)

Releases resources of the ISR without any checking.

Parameters:

[in] *resId* - a reference to the resource of the ISR

Returns:

none

Note:

Intended for "kill" functions

Releases resources of the ISR without any checking

Particularities: The service is defined only if resources used by ISR are configured in the OS.

Implementation

- While there are ISR resource in the resource list

  Clear 'isUsed' flag in resource control block -»TagOSRESCB (flag that the resource is used)

  Decrement -»OsISRResourceCounter

  If -»OsISRResourceCounter is equal 0

  – Unblock scheduler by clearing -»OsIsrLevel (ISR level for task, which got resource with ISR priority)

Used data

-»OsISRResourceCounter

OS_ReleaseResource

{DD_294}

-»StatusType OS_ReleaseResource (-»ResourceType *resId*)

releases resource, rescheduling may occur

Parameters:

[in] *resId* - a reference to the resource

Returns:

Standard:
E_OK no error.
E_OS_CORE resource is bound to another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID the resource identifier is invalid.
E_OS_NOFUNC attempt to release a resource which is not occupied by any task or ISR, or another resource has to be released before.
E_OS_ACCESS attempt to release a resource which has a lower ceiling priority than the assigned in OIL priority of the calling task or interrupt routine. This error code is returned only if E_OS_NOFUNC was not returned.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

This call serves to leave the critical sections in the code that are assigned to the resources referenced by <ResID>. A -»ReleaseResource call is a counterpart of a -»GetResource service call. This service returns task or ISR priority to the level saved by corresponded GetResource service.

Particularities:

This function is fully supported in all Conformance Classes. It is Freescale OS extension of the AUTOSAR OS because OSEK/VDX specifies full support only beginning from BCC2. Nested resource occupation is allowed only if the inner critical sections are completely executed within the surrounding critical section. Nested occupation of one and the same resource is forbidden. The service call is allowed on task level and ISR level, but not in hook routines. This service is not implemented if no standard resources are defined in the configuration file.

Implementation

- Get the reference to ISR or Task resource field
- Check the context of the service call
- Check if the resource has the invalid identifier (returns *E_OS_ID*)
- Check if the resource belong to another core (returns E_OS_CORE)
- Check if attempt to access occupied resource (returns *E_OS_ACCESS*)
- Return *E_OS_NOFUNC* if it is the ISR resource and another resource has to be released before
- If ISRs are configured in the OS and if there are resources used by ISR in the OS configuration and resource priority is lower than assigned ISR priority, returns *E_OS_ACCESS*
- Return *E_OS_NOFUNC* if it is the Task resource and another resource has to be released before
- If ISRs are configured in the OS and it is not ISR call level and resource is not for ISR, returns *E_OS_ACCESS*
- Return *E_OS_ACCESS* if it is the Task resource and it is resource with lower priority
- Enter OS critical section via *OSDIS()*

- If access rights of current OS-Application to given resource are insufficient or state of OS-Application to which belong given task is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Call -»OSTPResetResLockTime to reset resource locking time in the task or ISR
- If there are resources used by ISR in the OS configuration and it is the ISR level resource

  Clear 'isUsed' flag in resource control block -»TagOSRESCB (flag that the resource is used)

  Decrement -»OsISRResourceCounter

  Set the current resource pointer to the next resource from the field 'nextRes' -»TagOSRESCB

  Save resource in interrupts state

  If -»OsISRResourceCounter is equal 0 (it is the first ISR resource)

  – Unblock scheduler by clearing -»OsIsrLevel (ISR level for task, which got resource with ISR priority)
- Else (it is the task level resource)

  Clear 'isUsed' flag in resource control block -»TagOSRESCB (flag that the resource is used)

  Set 'resources' field of -»OsRunning the next resource from the field 'nextRes' -»TagOSRESCB
- If it is not ISR level call *OSDISPATCH* macro (-»OSTaskForceDispatch function)
- Leave the OS critical section via the macro *OSRI()*
- Return E_OK

Used data

-»OsIsrLevel

-»OsRunning

-»OsISRResourceCounter

Data Structures

TagOSRESCFG Struct Reference

{DD_135}

```
#include <Os_resource_config.h>
```

Data Fields

- ->OSDWORD appMask

  Application identification mask value.
- ->OSPRIOTYPE prio

  Resource priority for task resources
- ->ApplicationType appId

  Application identification value

Scheduler

The algorithm deciding which task has to be started and triggering all necessary OSEK Operating System internal activities is called scheduler. It performs all actions to switch the CPU from one instruction thread to another. It is either switching from task to task or from ISR back to a task. The task execution sequence is controlled on the base of task priorities and the scheduling policy used. The scheduler is activated whenever a task switch is possible according to the scheduling policy. The principle of multitasking allows the operating system to execute various tasks concurrently. The sequence of their execution depends on the scheduling policy, therefore it has to be clearly defined. Scheduler also provides the endless idle loop if there is no task ready to run. It may occur, when all tasks are in the suspended or waiting state until the awakening signal from an Interrupt Service Routine occurs. In this case there is no currently running task in the system, and the scheduler occupies the processor performing an endless loop until the ISR awakes a task to be executed. The scheduler can be treated as a specific resource that can be occupied by any task.

Scheduling Policy

{DD_136}

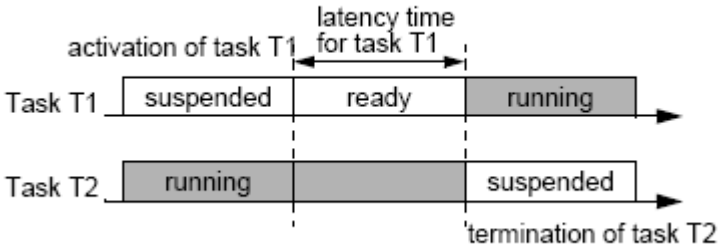
The scheduling policy being used determines whether execution of a task may be interrupted by other tasks or not. In this context, a distinction is made between full-, non- and mixed-preemptive scheduling policies. The scheduling policy affects the system performance and memory resources. In the OSEK Operating System, all listed scheduling policies are supported. Each task in an application may be preemptable or not. It is defined via the appropriate task property (preemptable / non-preemptable). Note that the interruptability of the system depends neither on the Conformance Class, nor on the scheduling policy. The desired scheduling policy is defined by the user via the tasks configuration option *SCHEDULE*. The valid values are *NON* and *FULL*. If all tasks use *NON* scheduling, scheduler works as nonpreemptive. If all tasks use *FULL* scheduling, scheduler works as full-preemptive. If some tasks use NON and other tasks use *FULL* scheduling, scheduler works as mixed-preemptive.

Non-preemptive Scheduling

{DD_137}

The scheduling policy is considered as non-preemptive, if a task switch is only performed via one of a selection of explicitly defined system services (explicit point of rescheduling). Non-preemptive scheduling imposes particular constraints on the possible timing requirements of tasks. Specifically, the lower priority non-preemptable section of a running task delays the start of a task with higher priority, up to the next point of rescheduling. The time diagram of the task execution sequence for this policy looks like the following:

Figure 0.5 Non-preemptive Scheduling



Task T2 has lower priority than task T1. Therefore, it delays task T1 up to the point of rescheduling (in this case termination of task T2). Only four following points of rescheduling exist in the OSEK OS for non-preemptive scheduling:

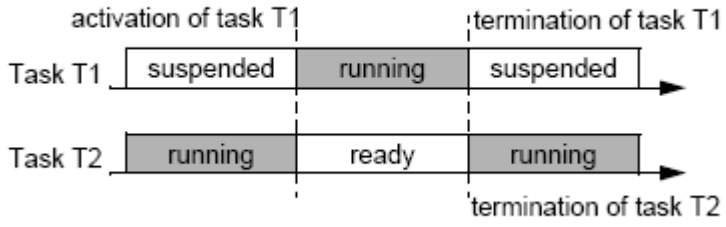
- Successful termination of a task (via the -»TerminateTask system service);
- Successful termination of a task with explicit activation of a successor task (via the -»ChainTask system service);
- Explicit call of the scheduler (via the -»Schedule system service);
- Explicit wait call, if a transition into the waiting state takes place (via the -»WaitEvent system service, Extended Tasks only). In the non-preemptive system, all tasks are non-preemptable and the task switching will take place exactly in the listed cases.

Full-preemptive Scheduling

{DD_138}

Full-preemptive scheduling means that a task which is presently running may be rescheduled at any instruction by the occurrence of trigger conditions preset by the operating system. Full-preemptive scheduling will put the running task into the ready state as soon as a higher-priority task has got ready. The task context is saved so that the preempted task can be continued at the location where it was interrupted. With full-preemptive scheduling, the latency time is independent of the run time of lower priority tasks. Certain restrictions are related to the enhanced complexity of features necessary for synchronization between tasks. As each task can theoretically be rescheduled at any location, access to data that are used jointly with other tasks must be synchronized. In a full-preemptive system all tasks are preemptable.

Figure 0.6 Full-preemptive Scheduling



Mixed-preemptive Scheduling

{DD_139}

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

preemptable task makes sense in a fullpreemptive operating system in the following cases:

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

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

Scheduler functions

OSDISPATCH

void OSDISPATCH (void)

Task dispatcher.

Returns:

none

Note:

Interrupts are disabled before call

Implementation

- If scheduler policy is mixed-preemptive or full-preemptive
  - If the function is not called from interrupt level and the Id of the current runnung task is not equal the Id of ready task then call -»OSTaskForceDispatch()

Used data

-»OsRunning

-»OsIsrLevel

OSProtectedCallTask

{DD_140}

void OSProtectedCallTask (void)

Task caller with memory protection switch.

Returns:

none

Note:

return only in case of User bug

Implementation

- Set current application to the running task application
- Call PreTaskHook
- IF application is non-Trusted
  - Disable interrupts (EI=0)
  - Enable OS interrupts (IPL = 0)
  - Open by MPU data access for this application
  - Switch CPU to User mode and enable interrupts
  - Call Task

  / / return here if no TerminateTask called in the Task
  - Call OSExceptionDispatch via OS call mechanism
- ELSE / / for trusted Task
  - Enable OS interrupts (IPL = 0)
  - Call Task

OSTaskForceDispatch

{DD_141}

void OSTaskForceDispatch (void)

Task dispatcher for BCC1 w / o FastTerminate.

Task dispatcher for protected environment (SC2..4).

Task dispatcher for ECC1, SC1 if defined **OSECC** .

Task dispatcher for BCC1 with FastTerminate.

Returns:

none

Note:

Called with OS interrupts disabled

There are several implementations for different SC.

Implementation

- Save "old" OsRunning value
- Check stack and reset TP for preempted task
- Call PostTaskHook
- Calculate new OsRunning.
- Save old Task context via OSSetJmp
- IF OSSetJmp returns non-zero value

  (it is return via OSLongJmp)
  - Enable all interrupts (EI=1)
  - Set running priority for this Task
  - Call PreTaskHook
  - return
- IF the old Task is *BASIC*
  - Save TCB address in linked list
- Call OSTaskInternalDispatch
- <unreachable point>.

OSTaskTerminateDispatch

{DD_142}

void OSTaskTerminateDispatch (void)

Reduced analog of -»OSTaskForceDispatch() (SC1, BCC) for Terminate / ChainTask services; changes Task state to SUSPENDED.

Reduced analog of -»OSTaskForceDispatch() (SC2..4) for Terminate / ChainTask services; changes Task state to SUSPENDED.

Reduced analog of -»OSTaskForceDispatch() (SC1, ECC) for Terminate / ChainTask services; changes Task state to SUSPENDED.

Returns:

never

Note:

Interrupts are disabled before call

Implementation

- Check old task stack
- Call PostTaskHook
- Reset internal priority
- Calculate new OsRunning.
- IF the new Task is not just activated
  / / it means that there were no ChainTask to itself
  - Clear OSTCBNOSUSPENDED flag for old Task
  - Set OSTCBFIRST flag for old Task
  - Remove old Task from scheduler vector
- Calculate new OsRunning.
- Call OSTaskInternalDispatch / / no return

OSDispatchOnError

{DD_143}

void OSDispatchOnError (void)

stripped Task dispatcher for error cases in SC1

Returns:

never

Note:

Called from OS*Dispatch and OSLeaveISR with new OsRunning set

Implementation

- Disable OS interrupts
- Check old task stack
- Call ErrorHook(E_OS_MISSINGEND)
- Call PostTaskHook
- Call OSKillRunningTask
- Calculate new OsRunning.
- IF the new Task is just activated
  - Clear OSTCBFIRST flag for this Task

  - Load Task SP
  - Set running priority for this Task
  - Call PreTaskHook
  - Call the Task

  / / return here if no TerminateTask called in the Task

  - GOTO beg. of this function / / this is trap for user bug

- IF the Task is Basic
  - Get and load into SPRG TCB address from linked list
- Perform OSLongJmp to resume the Task

OSTaskInternalDispatch

{DD_144}

void OSTaskInternalDispatch (void)

Main part of Task dispatcher for protected environment (SC2..4).

Returns:

never

Note:

Called from OS*Dispatch and OSLeaveISR with new OsRunning set

Implementation

- IF the new Task is just activated
  - Start Task budget
  - Clear OSTCBFIRST flag for this Task
  - Disable all interrupts (EI=0)
  - Load Task SP
  - Allow access to task stack area by MPU
  - Set running priority for this Task
  - Call OSProtectedCallTask

  / / return here if no TerminateTask called in the Task

  - Disable OS interrupts
  - Check stack and reset TP for old task
  - Call ErrorHook(E_OS_MISSINGEND)

  - Call PostTaskHook
  - Call OSKillRunningTask / /to perform appropriate cleanup
  - Calculate new OsRunning
  - GOTO beg. of this function / / this is trap for user bug

  / / resuming "new" task
- Resume Task budget
- Set this Task application as current
- IF the Task is Basic
  - Get and load into SPRG TCB address from linked list
- Disable interrupts (EI=0)
- Open by MPU data access for this application
- Allow access to task stack area by MPU
- Perform OSLongJmp to resume the Task

OS_ExceptionDispatch

{DD_145}

void OS_ExceptionDispatch (void)

Dispatch in case of Task returned w/o Terminate from OSCallTask.

Returns:

never

Note:

called from "OSProtectedCallTask," as service

Implementation

- Disable OS interrupts
- Check stack and reset TP for old task
- Call ErrorHook(E_OS_MISSINGEND)
- Call PostTaskHook
- Call OSKillRunningTask
- Calculate new OsRunning.
- Call OSTaskInternalDispatch / / no return

OS_Schedule

{DD_146}

-»StatusType OS_Schedule (void)

Yield the processor to the task with higher priority, if any.

Returns:

E_OK no error.
Extended:
E_OS_RESOURCE calling task occupies resources.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_SPINLOCK unreleased spinlock.

Note:

If a higher-priority task is in the ready state, the calling task is put into the ready state and the higher-priority task is transferred into running state. Otherwise the calling task is continued.

Particularities:

In not full preemptive applications Schedule service forces a rescheduling to other task in predefined locations. For the full preemptive task that has an internal resource assigned this service allows the rescheduling to other ready tasks in the same group (with the same internal resource) if they have a higher assigned priority. The service call is allowed on task level only.

Implementation

- IF the function is called in not allowed context
  - Call ErrorHook and return appropriate error code
- For extended status:
  - check if the current running task still occupies resources; return *E_OS_RESOURCE*
  - check if the current running task still occupies spinlocks; return *E_OS_SPINLOCK*
- Disable OS interrupts
- Reset internal Task priority
- IF this Task is not a task of highest priority and OsIsrLevel is zero
  - Call OSTaskForceDispatch
- ELSE

  - Set running priority for this Task
- Enable OS interrupts
- return E_OK

OSLOADCONTEXTADDR

{DD_147}

OSASM void OSLOADCONTEXTADDR (-»OSJMP_BUF *jmpbuf_ptr*)

Loads SPRG from Task context thus removing Task from list.

Parameters:

[in] *jmpbuf_ptr* - pointer to the task jump buffer.

Returns:

none

Note:

it is an assm inline "macro"

Implementation

- get pointer to previous Basic Task in list and save it in the SPRG

OSSAVECONTEXTADDR

{DD_148}

OSASM void OSSAVECONTEXTADDR (-»OSJMP_BUF *jmpbuf_ptr*)

Insert the Basic Task in head of Basic tasks list.

Parameters:

[in] *jmpbuf_ptr* - a pointer to the old task context buffer.

Returns:

none

Note:

it is an assm inline "macro"

implementation

- Save content of the SPRG in Task buffer pointed by *jmpbuf_ptr*
- Load into SPRG *jmpbuf_ptr* value.

Variable Documentation

- ->OSTSKCBPTR ->OsPrioLink[OSNPRIORS]

Schedule Table Management

AUTOSAR OS provides a statically defined task activation mechanism for fast tasks activations in accordance with statically defined schedule. User can define in the OIL configuration file a sequence of tasks and events to be (cyclically) activated at predefined time expyry points. It is not specified in the AUTOSAR OS allow or not to use one Counter for Alarms and Schedule Table simultaneously.

The AUTOSAR OS provides the synchronization interface for Schedule Table (SCT) similar to OSEKtime synchronization interface. Two synchronization stargegy are predefined in AUTOSAR OS: explicit and implicit. The global difference is Delay adjustment(distance between two expiry points) applied for explicit synchronization .

Data Type

{DD_149}

- -»**ScheduleTableType** - identifier of ScheduleTable.
- -»**ScheduleTableStatusType** - the status of a schedule table;
- -»**ScheduleTableStatusRefType** - reference to a variable of the data type ScheduleTableStatusType;
- -»**GlobalTimeTickType** - the global time source type.

Constants

{DD_150}

The following constants of -»ScheduleTableStatusType type are defined:

- *SCHEDULETABLE_STOPPED*
- *SCHEDULETABLE_NEXT*
- *SCHEDULETABLE_WAITING*
- *SCHEDULETABLE_RUNNING*
- *SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS*

Schedule Table functions

OSInitScheduleTables

{DD_151}

void OSInitScheduleTables (void)

Initializes Schedule Tables.

Parameters:

none

Returns:

none

Note:

Used only by OSStartUp

Implementation

- For all SCT that configured in OS:
  - Fill all appropriated fields in OSALMSCTARRAY structure.
  - Fill all appropriated fields in OsScheduleTables.
  - Fill all additional fields if Memory Protection is configured in OS.

Used data

-»OSALMSCTARRAY

-»OsScheduleTablesCfg

-»OsScheduleTables

OSKillScheduleTable

{DD_152}

OSINLINE void OSKillScheduleTable (-»OSSCTCB * *sctCB*)

Kills schedule table with given pointer.

Parameters:

[in] *sctCB* - a pointer to the schedule table

Returns:

none

Note:

none

Implementation

- Clear status to prevent start of the ’next’ SCT.
- Clear nextId field to prevent start of the ’current’ SCT if it is in ’next’ status.
- If SCT alarm is active:
  - Stop this alarm.
- Change status of the SCT to ’stopped’.

OSKillAppScheduleTables

{DD_153}

void OSKillAppScheduleTables (void)

Kills all schedule tables of given application.

Returns:

none

Note:

none

Implementation

- Check all SCT and if any belongs to current Application:
  - Stop the SCT via the internal OS service -»OSKillScheduleTable.

Used data

-»OsScheduleTables

OSStartScheduleTable

{DD_154}

OSINLINE void OSStartScheduleTable (-»OSSCTCB * *sctCB*, -»TickType *abs*)

Starts the processing schedule table.

Parameters:

[in] *sctCB* - a pointer to the schedule table
[in] *abs* The absolute tick value of the first action point

Returns:

none

Note:

Internal design supposes usage of certain 'null-point' - an additional configuration expiry point that exists when SCT has non-zero Initil Offset(the smallest expiry point offset on a SCT) AND this SCT is configured with explicit synchronization. In this case max.adjustment value(*maxRetard/Advance*) is divided into proportion parts: for the Final Delay and Initial Offset. If SCT is supposed to be stopped via service call NextScheduleTable(), divided delays take part in process of Final Delay calculation(delay from the final expiry point to the logical end of the SCT).

Implementation:

- Get pointer to the first action of expiry point with minimum offset of the SCT.
- Change SCT status to'runnung'(for v4 OSSCTCBSYNCUP is remained).
- If SCT is configured with explicit synchronization:
  - change SCT status to 'runnung and synchronous'.
- Change pointer from null-point to the first action of expiry point.
  - re-initialize initialOffset from configuration array(initialOffset can be changed in synchronization process and shall be restored when the SCT is (re)started).
- If SCT is configured with Initial Offset is not equal to zero, find absolute alarm value for initialOffset and save it in appropriated SCT field.
- Clear 'nextId' field.
- Set alarm expiration value.

Used data

-»OSALMSCTARRAY

-»OsScheduleTablesCfg

-»OsScheduleTables

OSProcessScheduleTable

{DD_155}

-»OSAlmType OSProcessScheduleTable (-»OSSCTALMCB * *almId*)

Check notification parameters of the Schedule Tables expiry points, and perform notification.

Parameters:

[in] *almId* - a pointer to the alarm

Returns:

valid new almId if next Schedule Tables started, otherwise OSALM0

Note:

none

Implementation

- Get the pointer to the SCT control block.
- Get the pointer to the first action of current expiry point.
- If it's a last expiry point of the SCT:
  - If 'next' field of the SCT is not empty, start 'next' SCT, for v4 - save deviation is remained from previous SCT and save flag of sync.direction, change status of the 'current' SCT to 'stopped' and return with pointer to new Alarm attached to started SCT.
- If it's a last expiry point of the SCT:
  - If the SCT is not periodic, change status of the SCT to 'stopped' and return with non-valid pointer.
- Perform all actions for current expiry point:
  - Set period of alarm expiration.
  - If the SCT is configured with explicit synchronization and it is the first action for the current expiry point:

    If synchronization process has already begun (OS service *OSSyncScheduleTable* was called) and deviation value is not equal to zero:

    - if deviation value is less then precision value, set status of the SCT in the 'running and synchronous' state.
    - get new delay to next expiry point depending on *maxRetard/maxAdvance* value and new deviation.

Else set status of the SCT in the 'running and synchronous' state(for all expiry points excepting null-point).

- If SCT has non-zero Initial Offset and next expiry point is the first expiry point, save "dynamic" Initial Offset (taking into account adjustment of delay via synchronization) and correct delay as Final Delay + new Initial Offset(this "dynamic" Final Delay was got in previous expiry point).
- If it is not a null-point:
- call internal OS service *OSNotifyAlarmAction* to perform appropriate action.

- Get the pointer to the next action of the expiry point.
- If the SCT has non-zero Initial Offset and SCT is periodic, save "dynamic" Final Delay (taking into account adjustment of delay via synchronization).
- If the SCT is configured w/o explicit synchronization:
  - If 'next' field of the SCT is not empty, null-point can not be skipped and delay is set as Final Delay.
- Else:
  - for v4 - save remained deviation for 'next' SCT (unconditionally).
  - If 'next' field of the SCT is empty then null-point can be skipped, repeat cycle for the first expity point.
- Return with non-valid pointer.

Used data

-»OsScheduleTables

-»OSSCTEP

-»OSSCTALMCB

OSMCStartScheduleTableAbs

{DD_323}

-»StatusType OS_StartScheduleTableAbs (-»OSSCTCB* *sctCB*, -»TickType *tickvalue*, -»OSAPPLICATIONTYPE *appId*)

Start the processing of a Schedule Table at its first expiry point after the underlaying counter reaches <tickvalue>

Parameters:

[in] *sctCB* The control block of chedule table to be started
[in] *tickvalue* The absolute tick value of the first action point
[in] *appId* - an ID of calling application

Returns:

Standard:
E_OK no error.
E_OS_STATE Schedule table is already started.

Implementation

- Check access rights of calling OS-Application to the given schedule table.
- Check that the current schedule table is not started.
- Fix the current Counter value to the variable -»*OsTimVal* via the internal OS service -»OSSetTimVal (only if the Counter is HW).
- Store parameters for case of error in called functions
- Call the internal OS service -»OSStartScheduleTable with the pointer to the SCT control block, the absolute alarm value and with sign of alarm handler existing.
- Clear parameters for case of error in called functions.
- Return E_OK

Used data

-»OsScheduleTables

OS_StartScheduleTableAbs

{DD_156}

-»StatusType OS_StartScheduleTableAbs (-»ScheduleTableType *sctId*, -»TickType *tickvalue*)

Start the processing of a Schedule Table at its first expiry point after the underlaying counter reaches <tickvalue>

Parameters:

[in] *sctId* The schedule table to be started
[in] *tickvalue* The absolute tick value of the first action point

Returns:

Standard:

E_OK no error.
E_OS_STATE Schedule table is already started.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <sctId> is not valid or implicitly synchronized.

E_OS_VALUE <tickvalue> is greater than MAXALLOWEDVALUE.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

If its input parameters are valid, the service start the processing schedule table <ScheduleTableID> at its first expiry point after underlaying counter reaches <TickValue> and return E_OK.
This service return E_OS_ID if the schedule table <ScheduleTableID> is not valid.
The service return E_OS_VALUE if the offset <TickValue> is greater than MAXALLOWEDVALUE.
The service return E_OS_STATE if schedule table <ScheduletableID> was already started or another ScheduleTable is running on this Counter.
Particularities: Allowed on task level and in ISR category 2.

Implementation

- Check the context of the service call.
- Check that given SCT id is valid.
- Get the pointer to the SCT control block.
- Get the underlaying Counter index from the SCT control block.
- Check given absolute tick value <tickvalue>.
- If given schedule table belong to the another core
  - perform remote call by calling OSRemoteCall2 macro
  - Call -»DISPATCH function
  - If status is modified via OSRemoteCall2 is not E_OK then leave OS critical section via macro *OSRI()*, return status and leave service
- else (given schedule table belong to the same core)
  - Enter OS critical section via *OSDIS()*
  - If access rights of current OS-Application to given schedule table are insufficient or state of OS-Application

to which belong given schedule table is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code

- Check that the SCT is not started.
- Fix the current Counter value to the variable -»*OsTimVal* via the internal OS service -»OSSetTimVal (only if the Counter is HW).
- Store parameters for case of error in called functions.
- Call the internal OS service -»OSStartScheduleTable with the pointer to the SCT control block, the absolute alarm value and with sign of alarm handler existing.
- Clear parameters for case of error in called functions.
- Do rescheduling if it is necessary.
- Leave the OS critical section via the macro *OSRI*.
- Return E_OK.

Used data

-»OsScheduleTables

-»OsCountersCfg

-»OsRunning

-»OsIsrLevel

OSMCStartScheduleTableRel

{DD_324}

-»StatusType OS_StartScheduleTableRel(-»OSSCTCB* *sctCB*, -»TickType *increment*, -»OSAPPLICATIONTYPE *appId*)

Start the processing of a Schedule Table at its first expiry point after offset <offset> ticks have elapsed.

Parameters:

[in] *sctCB* The control block of chedule table to be started
[in] *increment* The relative tick value between now and the first action point
[in] *appId* - an ID of calling application

Returns:

Standard:
E_OK no error.
E_OS_STATE Schedule table is already started.

Implementation

- Check access rights of calling OS-Application to the given schedule table.
- Check that the current schedule table is not started.
- Get absolute Counter value via the internal OS service -»OSAbsTickValue and fix the Counter value to the variable -»*OsTimVal* (if the Counter is HW).
- Store parameters for case of error in called functions.
- Call the internal OS service -»OSStartScheduleTable with the pointer to the SCT control block, the absolute alarm value and with sign of alarm handler existing.
- Clear parameters for case of error in called functions.
- Return E_OK

Used data

-»OsScheduleTables

OS_StartScheduleTableRel

{DD_157}

-»StatusType OS_StartScheduleTableRel (-»ScheduleTableType *sctId*, -»TickType *increment*)

Start the processing of a Schedule Table at its first expiry point after offset <offset> ticks have elapsed.

Parameters:

[in] *sctId* The schedule table to be started
[in] *increment* The relative tick value between now and the first action point

Returns:

Standard:
E_OK no error.
E_OS_STATE Schedule table is already started.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <sctId> is not valid or implicitly synchronized.

E_OS_VALUE <increment> is greater than MAXALLOWEDVALUE.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

If its input parameters are valid and the state of schedule table <ScheduleTableID> is *SCHEDULETABLE_STOPPED*, then -»StartScheduleTableRel() shall start the processing of a schedule table <ScheduleTableID>. The Initial Expiry Point shall be processed after <Offset> + Initial Offset ticks have elapsed on the underlying counter. The state of <ScheduleTableID> is set to *SCHEDULETABLE_RUNNING* before the service returns to the caller This service return E_OS_ID if the schedule table <ScheduleTableID> is not valid.
The service return *E_OS_VALUE* if the offset <Offset> is zero or greater than MAXALLOWEDVALUE.
The service return *E_OS_STATE* if schedule table <ScheduletableID> was already started.
Particularities: Allowed on task level and in ISR category 2.

Implementation

- Check the context of the service call.
- Check that given SCT id is valid.
- Get the pointer to the SCT control block.
- Get the underlaying Counter index from the SCT control block.
- Check that the SCT is not configured with implicit synchronization.
- Check given increment tick value <increment>.
- If given schedule table belong to the another core
  - perform remote call by calling OSRemoteCall2 macro
  - Call -»DISPATCH function
  - If status is modified via OSRemoteCall2 is not E_OK then leave OS critical section via macro *OSRI()*, return status and leave service
- else (given schedule table belong to the same core)
  - Enter OS critical section via *OSDIS()*
  - If access rights of current OS-Application to given schedule table are insufficient or state of OS-Application to which belong given schedule table is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Check that the SCT is already started.

- Get absolute Counter value via the internal OS service -»OSAbsTickValue and fix the Counter value to the variable -»*OsTimVal* (if the Counter is HW).
- Store parameters for case of error in called functions.
- Call the internal OS service -»OSStartScheduleTable with the pointer to the SCT control block, the absolute alarm value and with sign of alarm handler existing.
- Clear parameters for case of error in called functions.
- Do rescheduling if it is necessary.
- Leave the OS critical section via the macro *OSRI*.
- Return E_OK.

Used data

-»OsScheduleTables

-»OsCountersCfg

-»OsRunning

-»OsIsrLevel

OSMCStopScheduleTable

{DD_325}

-»StatusType OS_StartScheduleTableRel(-»OSSCTCB* *sctCB*, -»TickType *increment*, -»OSAPPLICATIONTYPE *appId*)

Stop the Schedule Table from processing any further expiry points.

Parameters:

[in] *sctCB* The control block of chedule table
[in] *appId* - an ID of calling application

Returns:

Standard:
E_OK no error.
E_OS_NOFUNC Schedule table is already started.

Implementation

- Check access rights of calling OS-Application to the given schedule table.
- Check that the current schedule table is stopped.
- Store parameters for case of error in called functions.

- Stop the SCT via the internal OS service -»OSKillScheduleTable with the pointer to the SCT control block.
- Clear parameters for case of error in called functions.
- Return E_OK

Used data

-»OsScheduleTables

OS_StopScheduleTable

{DD_158}

-»StatusType OS_StopScheduleTable (-»ScheduleTableType *sctId*)

Stop the Schedule Table from processing any further expiry points.

Parameters:

[in] *sctId* The schedule table to be stopped

Returns:

Standard:
E_OK no error.
E_OS_NOFUNC Schedule table is already started.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <sctId> is not valid or implicitly synchronized.

E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

If its input parameters are valid, the service stop the schedule table <ScheduleTableID> from processing any further expiry points and return E_OK.
This service return *E_OS_ID* if the schedule table <ScheduleTableID> is not valid.
The service return *E_OS_NOFUNC* if the schedule table with identifier <ScheduleTableID> was not started.
Particularities: Allowed on task level and in ISR category 2.

Implementation

- Check the context of the service call.
- Check that given SCT id is valid.

- Get the pointer to the SCT control block.
- If given schedule table belong to the another core
  - perform remote call by calling OSRemoteCall1 macro
  - Call -»DISPATCH function
  - If status is modified via OSRemoteCall2 is not E_OK then leave OS critical section via macro *OSRI(),* return status and leave service
- else (given schedule table belong to the same core)
  - Enter OS critical section via *OSDIS()*
  - If access rights of current OS-Application to given schedule table are insufficient or state of OS-Application to which belong given schedule table is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code.
- Check that the SCT is already stopped.
- Store parameters for case of error in called functions.
- Stop the SCT via the internal OS service -»OSKillScheduleTable with the pointer to the SCT control block.
- Clear parameters for case of error in called functions.
- Do rescheduling if it is necessary.
- Leave the OS critical section via the macro *OSRI.*
- Return E_OK.

Used data

-»OsScheduleTables

-»OsRunning

-»OsIsrLevel

OS_NextScheduleTable

{DD_159}

-»StatusType OS_NextScheduleTable (-»ScheduleTableType *sctId_current*, -»ScheduleTableType *sctId_next*)

Start the processing of Schedule Table <sctId_next> after <sctId_current> reaches its period/length.

Parameters:

[in] *sctId_current* The current schedule table
[in] *sctId_next* The next schedule table that provides its series of expiry points

Returns:

Standard:
E_OK no error.
E_OS_NOFUNC <sctId_current> was not started.
E_OS_CORE resource is bound to another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <sctId_current> or <sctId_next> is not valid or belongs to different counters.

E_OS_STATE <sctId_next> is started.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

If the input parameters are valid then -»NextScheduleTable shall start the processing of schedule table <ScheduleTableID_To> <ScheduleTableID_From>.FinalDelay ticks after the Final Expiry Point on <ScheduleTableID_From> is processed and shall return *E_OK*.
The Initial Expiry Point on <ScheduleTableID_To> shall be processed at <ScheduleTableID_From>.Final Delay + <ScheduleTable_To>.Initial Offset ticks after the Final Expiry Point on <ScheduleTableID_From> is processed.
If its input parameters are valid and the <ScheduleTableIDCurrent> is running, the service start the processing of schedule table <ScheduleTableIDNext> and stops <ScheduleTableCurrent> after <ScheduleTableCurrent> reaches its period and return E_OK.
If there were "next" ScheduleTable for the current one, it shall be moved from NEXT state, i.e. the new next table replaces the previous one. This service return E_OS_ID if the schedule table <ScheduleTableIDCurrent> or <ScheduleTableIDNext> is not valid OR (v4) if the tables are configured with different synch.strategies OR (v4) tables are explicitly synchronized but lenghts are different. The service return *E_OS_ID* if the schedule table <ScheduleTableIDNext> is driven by different counter than schedule table <ScheduleTableCurrent>. The service return *E_OS_NOFUNC* if the schedule table with identifier <ScheduleTableIDCurrent> was not started.
Particularities: Allowed on task level and in ISR category 2. If the

<ScheduleTableIDCurrent> is stopped before <ScheduleTableIDNext> is started, <ScheduleTableIDNext> is not started and its state changes to *SCHEDULETABLE_STOPPED*.

Implementation

- Check the context of the service call.
- Check that 'next' SCT id is valid.
- Check that 'current' SCT id is valid.
- Get the pointer to the 'current' SCT control block.
- Get the pointer to the 'next' SCT control block.
- Check that schedule table belongs to calling core.
- Enter OS critical section via the macro *OSDIS*.
- If access rights of current OS-Application to given schedule tables are insufficient or state of OS-Application to which belong given schedule tables is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Check that Counter index is associated with 'current' SCT the same as Counter index is associated with 'next' SCT.
- Check that the 'next' SCT is stopped.
- Check that the 'current' SCT is started.
- For v4: if the 'current' and 'next' SCT are configured with sync.startegy:
  - Leave OS critical section via macro *OSRI()* and return E_OS_ID error code if strategy are not equal.
  - If the 'current' and 'next' SCT are configured with explicit synchronization: leave OS critical section via macro *OSRI()* and return E_OS_ID error code if SCT lenghts are not equal; set syncronization flag for 'next' SCT.
- Previous 'next' SCT becomes stopped if the 'current' SCT already has 'next' SCT.
- Save pointer to the 'next' SCT control block in 'nextId' field of the 'current' SCT.
- Save pointer to the 'current' SCT control block in 'prevId' field of the 'next' SCT.
- Set status of the 'current' SCT in the 'next' state.

- If the 'current' SCT has non-zero Initial Offset,it is periodic and service call time between of last and first EP:
  - Get absolute Counter value of the service call.
  - Get remained time to the first EP of 'current' SCT.
  - If the time between the last EP and the logical end of 'current' SCT:

    - Stop alarm of the 'current' SCT and set new alarm expiration value to the end of SCT.
  - else:

    - Since service was called beyond logical end of 'current' SCT, clear OSSCTCBEND flag (this flag will be set on the next period of the SCT).
- Leave the OS critical section via the macro *OSRI*.
- Return E_OK.

Used data

-»OsScheduleTables

OS_StartScheduleTableSynchron

{DD_160}

-»StatusType OS_StartScheduleTableSynchron (-»ScheduleTableType *sctId*)

It is used for processing of Schedule Table at first expiry point after the global time is set.

Parameters:

[in] *sctId* The schedule table to be started

Returns:

Standard:
E_OK no error.
E_OS_STATE Schedule table is already started.
E_OS_CORE resource is bound to another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <sctId> is not valid.

E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

If <ScheduleTableID> is valid, -»StartScheduleTableSynchron shall set the state of <ScheduleTableID> to SCHEDULETABLE_WAITING and start the processing of schedule table <ScheduleTableID> after the synchronization count of the schedule table is set via -»SyncScheduleTable(). The Initial Expiry Point shall processed when (Duration-SyncValue)+InitialOffset ticks have elapsed on the synchronization counter where:

- Duration is <ScheduleTableID>.OsScheduleTableDuration
- SyncValue is the <Value> parameter passed to the -»SyncScheduleTable()
- InitialOffset is the shortest expiry point offset in <ScheduleTableID>.

Particularities: Allowed on task level and in ISR category 2.

Implementation

- If system hasn't explicit synchronization:
  - Return the appropriated error code.
- Check the context of the service call.
- Check that given SCT id is valid.
- Get the pointer to the SCT control block.
- Check that schedule table belongs to calling core.
- Check that the SCT is configured with explicit synchronization.
- Enter OS critical section via the macro *OSDIS*.
- If access rights of current OS-Application to given schedule table are insufficient or state of OS-Application to which belong given schedule table is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Check that the SCT is stopped.
- Set status of the 'current' SCT in the 'waiting' state.
- Leave the OS critical section via the macro *OSRI*.
- Return E_OK.

Used data

-»OsScheduleTables

OS_SyncScheduleTable

{DD_161}

-»StatusType OS_SyncScheduleTable (-»ScheduleTableType *sctId*,
-»GlobalTimeTickType *globalTime*)

It is used to synchronize the processing of the Schedule Table to global time.

Parameters:

[in] *sctId* The schedule table
[in] *globalTime* the current value of global time.

Returns:

Standard:
E_OK no error.
E_OS_CORE resource is bound to another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <sctId> is not valid.

E_OS_STATE the ScheduleTable is not started (its state is SCHEDULETABLE_STOPPED or SCHEDULETABLE_NEXT).
E_OS_VALUE <globalTime> is more or equal to ScheduleTable period.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

This service provides the OS with current global time. It is used to synchronize the processing of schedule table to global time.
Particularities: Allowed on task level and in ISR category 2. The service is applicable for ScheduleTable in waiting state (*SCHEDULETABLE_WAITING*).

Implementation

- If system hasn't explicit synchronization:
  - Return the appropriated error code.
- Check the context of the service call.
- Check that given SCT id is valid.
- Get the pointer to the SCT control block and underlaying Counter index.
- Enter OS critical section via the macro *OSDIS*.

- If access rights of current OS-Application to given schedule table are insufficient or state of OS-Application to which belong given schedule table is not APPLICATION_ACCESSIBLE, leave OS critical section via macro *OSRI()* and return E_OS_ACCESS error code
- Get absolute Counter value of the service call.
- Check that the SCT is configured with explicit synchronization.
- Check that the delivered global time is not greater then SCT duration.
- Check that the SCT is not stopped or in 'next' state.
- If the SCT is in 'waiting' state:
  - Get deviation between SCT duration and delivered global time, then get absolute Counter value for SCT start.
  - Start the SCT via the OS internal service -»OSStartScheduleTable with this Counter value.
  - The SCT gets 'running and synchronous state'.
  - Do rescheduling if it is necessary.
- If the SCT is in 'running' or 'running and synchronus' state:
  - Get time is before the next expiry point of the SCT.
  - Get a local time of the service call within the SCT.
  - If local time of the SCT equal to delivered global time:

    Clear sign of synchronization process and set status of the SCT in the 'running and synchronous' state.
  - Else get minimum deviation and set flag of increase / decrease to perform delays between SCT expiry points.
  - Set SCT status in 'running and synchronus' or 'running' state depending on less deviation then precision value or not.
- Leave the OS critical section via the macro *OSRI*.
- Return E_OK.

Used data

-»OsScheduleTables

-»OsCountersCfg

-»OsRunning

-»OsIsrLevel

OS_SetScheduleTableAsync

{DD_162}

-»StatusType OS_SetScheduleTableAsync (-»ScheduleTableType *sctId*)

Set the synchronization status of the sctId to asynchronous.

Parameters:

[in] *sctId* The schedule table

Returns:

Standard:
E_OK no error.
E_OS_CORE resource is bound to another core.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <sctId> is not valid.

E_OS_STATE the ScheduleTable is not started (its state is SCHEDULETABLE_STOPPED or SCHEDULETABLE_NEXT).
E_OS_VALUE <globalTime> is more or equal to ScheduleTable period.
E_OS_CALLEVEL call at not allowed context.
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

This service set the synchronization status of the <ScheduleTableID> to asynchronous. If this service is called for a running schedule table the OS continue process expiry points on the schedule table and stops further synchronization if it was in progress.

Particularities: Allowed on task level and in ISR category 2.

Implementation

- If system hasn't explicit synchronization:
  - Return the appropriated error code.
- Check the context of the service call.
- Check that given SCT id is valid.
- Get the pointer to the SCT control block.
- Check that the SCT not configured with explicit synchronization.
- Enter OS critical section via the macro *OSDIS*.
- If access rights of current OS-Application to given schedule table are insufficient or state of OS-Application to which belong given schedule table is not APPLICATION_ACCESSIBLE, leave

OS critical section via macro *OSRI()* and return E_OS_ACCESS error code

- Check that the SCT already in ’running’ or ’running and synchonous’ state.
- Stop further synchronization (for v4 also stop further synchronization for 'next' SCT) and set status of the SCT in the ’running’ state.
- Leave the OS critical section via the macro *OSRI*.
- Return E_OK.

Used data

-»OsScheduleTables

OS_GetScheduleTableStatus

{DD_163}

-»StatusType OS_GetScheduleTableStatus (-»ScheduleTableType *sctId*, -»ScheduleTableStatusRefType *scheduleStatus*)

Get the Schedule Table status.

Parameters:

[in] *sctId* - a schedule table identifier
[out] *scheduleStatus* - a reference to the variable of ScheduleStatusType

Returns:

Standard:
E_OK no error.
E_OS_ACCESS insufficient access rights.
Extended:
E_OS_ID <sctId> is not valid.

E_OS_CALLEVEL call at not allowed context.
E_OS_ILLEGAL_ADDRESS illegal <scheduleStatus> (only for SC3, SC4).
E_OS_DISABLEDINT call when interrupts are disabled by OS services.

Note:

If the schedule table <ScheduleTableID> is NOT started, -»GetScheduleTableStatus shall pass back *SCHEDULETABLE_STOPPED* via the reference parameter <ScheduleStatus> AND shall return *E_OK*.
If the schedule table <ScheduleTableID> was used in a -»NextScheduleTable call AND waits for the end of the current schedule table, -»GetScheduleTableStatus shall return *SCHEDULETABLE_NEXT*

via the reference parameter <ScheduleStatus> AND shall return *E_OK*. If the schedule table <ScheduleTableID> is configured with explicit synchronization AND <ScheduleTableID> was started with -»StartScheduleTableSynchron AND no synchronization count was provided to the Operating System, -»GetScheduleTableStatus shall return *SCHEDULETABLE_WAITING* via the reference parameter <ScheduleStatus> AND shall return *E_OK*. If the schedule table <ScheduleTableID> is started AND synchronous, -»GetScheduleTableStatus shall pass back *SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS* via the reference parameter <ScheduleStatus> AND shall return *E_OK*.
If the schedule table <ScheduleTableID> is started AND NOT synchronous (deviation is not within the precision interval OR the schedule table has been set asynchronous), -»GetScheduleTableStatus shall pass back *SCHEDULETABLE_RUNNING* via the reference parameter ScheduleStatus AND shall return *E_OK*. If schedule table <ScheduleTableID> is not yet started, this service passes back SCHEDULETABLE_NOT_STARTED via the reference parameter <ScheduleStatus> and return *E_OK*. If the schedule table <ScheduleTableID> is configured with hard synchronization strategy and no global time was provided to the OS, the service returns *SCHEDULETABLE_WAITING* via the reference parameter <ScheduleStatus> and returns *E_OK*.
If the schedule table <ScheduleTableID> was used in a NextScheduleTable () call and waits for the end of the current schedule table, this service *SCHEDULETABLE_NEXT* via the reference parameter <ScheduleStatus> and return *E_OK*. If schedule table <ScheduleTableID> started and synchronous, the service passes back *SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS* via the reference parameter <ScheduleStatus> and returns *E_OK*. If schedule table <ScheduleTableID> is started, but is not synchronous (deviation is not within the precision interval or the global time is not available), the service passes back *SCHEDULETABLE_RUNNUING* via the reference parameter <ScheduleStatus> and returns *E_OK*. If the identifier <ScheduleTableID> is not valid, the service returns *E_OS_ID*.
Particularities: Allowed on task level and in ISR category 2.

Implementation

- Check the context of the service call.

- Check that given SCT id is valid.
- Get the pointer to the SCT control block.
- Check access rights of current OS-Application to the SCT.
- Check access rights and the possibility to write into given pointer 'scheduleStatus'.
- Save status of the SCT by the pointer 'scheduleStatus'.
- Return E_OK.

Used data

-»OsScheduleTables

OSInitAutoScheduleTables

{DD_164}

void OSInitAutoScheduleTables (-»AppModeType *mode*)

Initialize autostarted Schedule Tables.

Parameters:

[in] *the* OS-Application mode (if defined (**OSNAPPMODES** > 1))

Returns:

none

Note:

Used only by OSStartUp

Implementation

- Get pointer to the auto-configuration control block of the SCT.
- If autostart type is 'synchron' set status of the SCT in the 'waiting' state.
  - Else get 'start' value from auto-configuration.
- If autostart type is 'absolute' fix the absolute current Counter value to the variable -»*OsTimVal* via the internal OS service -»OSSetTimVal (only if the Counter is HW).
- If autostart type is 'absolute' call the internal OS service -»OSAbsTickValue to get absolute Counter value and fix it to the variable *OsTimaVal(*only if the Counter is HW).
- Start the SCT via the OS internal service -»OSStartScheduleTable with absolute Counter value.

Used data

-»OsScheduleTables

-»OsAutoScheduleTablesCfg

Data Structures

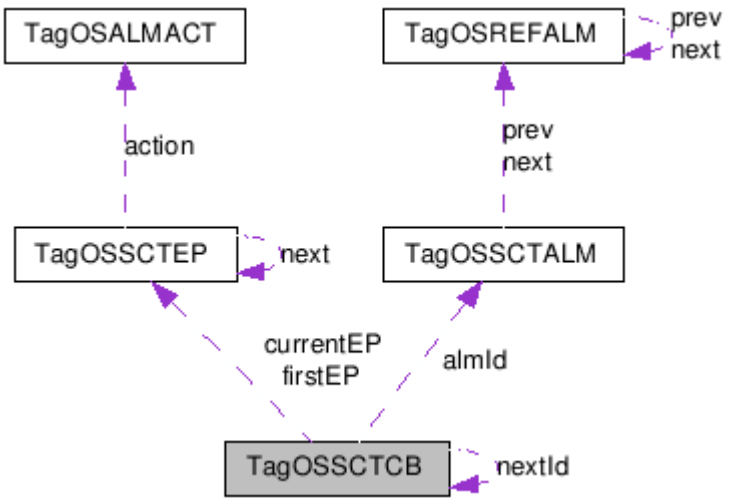
TagOSSCTCB Struct Reference

{DD_165}

```
#include <Os_schedule_table_internal_types.h>
```

Structure of the schedule table control block.

Collaboration diagram for TagOSSCTCB:



Data Fields

- const ->OSSCTEP * currentEP

  The current expiry point
- ->OSSCTCB * nextId

  The ScheduleTableId_To (used only for NextScheduleTable)
- ->OSSCTALMCB * almId

  The alarm for this SCT
- ->TickType deviation

  Deviation of SCT local time and value of the synchronization count
- ->TickType precision

Schedule table sync.PRECISION value

- ->TickType syncOffset
  The next EP offset using for sync. correction
- ->TickType length
  The length of the schedule table in ticks
- ->TickType maxSync
  Schedule table MAX_CORRECTION_SYNC values
- ->TickType maxAsync
  Schedule table MAX_CORRECTION_ASYNC values
- ->TickType initialOffset
  Initial offset
- const ->OSSCTEP * firstEP
  The pointer to first expiry point in the schedule table
- ->OSWORD ctrIndex
  Attached Counter ID
- ->OSWORD state
  The config(MS) and status(LS) of schedule table
- ->OSDWORD appMask
  Application identification mask value
- ->ApplicationType appId
  Application identification value

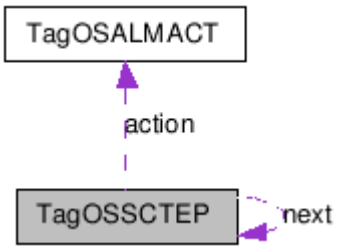
TagOSSCTEP Struct Reference

{DD_166}

```
#include <Os_schedule_table_config.h>
```

The expiry point structure.

Collaboration diagram for TagOSSCTEP:



Data Fields

- const ->OSSCTEP * next

  The next expiry point

- ->OSALMACT action

  Alarm action

- ->TickType delta

  The time space (in ticks) between the current expiry point and the next expiry point (or the end of the schedule if it's last expiry point)

- ->TickType maxRetard

  Schedule table MAXRETARD values

- ->TickType maxAdvance

  Schedule table MAXADVANCE values

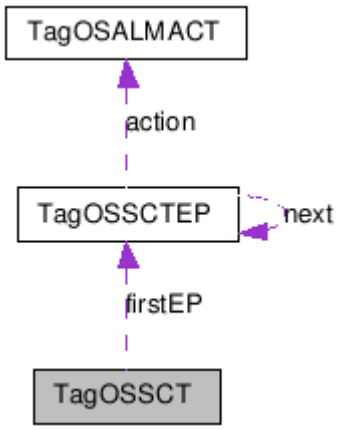
TagOSSCT Struct Reference

{DD_167}

```
#include <Os_schedule_table_config.h>
```

Structure of the schedule table cfg.

Collaboration diagram for TagOSSCT:



Data Fields

- ->OSWORD ctrIndex
  Attached Counter ID
- ->TickType length
  The length of the schedule table in ticks
- ->TickType initialOffset
  Initial offset
- const ->OSSCTEP * firstEP
  The pointer to first expiry point in the schedule table
- ->TickType precision
  Schedule table EXPLICITPRECISION value
- ->OSDWORD appMask
  Application identification mask value
- ->ApplicationType appId

Application identification value

- ->OSBYTE opts

  Config byte of the schedule table

tagSCTAUTOTYPE Struct Reference

{DD_168}

```
#include <Os_schedule_table_config.h>
```

The schedule tables which are started on startup of the OS with a given offset.

Data Fields

- ->OSWORD sctIndex
- ->TickType offset
- ->OSDWORD autostart
- ->OSBYTE type

Miscellaneous OS internal services

OSCallAppErrorHook

{DD_169}

OSINLINE void OSCallAppErrorHook (-»StatusType *error*)

calls user defined application specific hook (if any)

Parameters:

[in] *error* The status of the error

Returns:

none

Note:

none

Implementation

- Return from the function if current application is invalid.
- If application specific error hook for current application is defined:
  - If current application is nontrusted then configure MPU for application data, switch to user mode, call application specific error hook and return to supervisor mode
  - else all application specific error hook for current application

Used data

-»OsAppTable

OSDECLAREVAR

{DD_170}

```
#define OSDECLAREVAR(var_type, var) var_type var
```

OSDECLAREVAR (-»ApplicationType, OsSharedAppID_)

OS shared application Id.

Note:

Declared if defined **OSAPPLICATION**

OSDECLAREVAR (OSNEAR -»ApplicationType, OsLastApp_)

The last nontrusted application ID.

Note:

Declared if defined **OSMEMPROTECTION** and OSNNOTTRUSTEDAPPS > 1

OSDECLAREVAR (OSNEAR -»ProtectionReturnType, OsTPHookAction_)

Protection hook action desired by user when inter-arrival rate violation occurs.

Note:

Declared if defined **OSHOOKPROTECTION**

OSDECLAREVAR (OSNEAR -»OSDWORD, OsTPOverflowCnt_)

TP overflow counter.

Note:

Declared if defined **OSTPTIMSTM** and (**-»OSNTPISRARRIV** > 0 or **-»OSNTPTSKARRIV** > 0)

OSDECLAREVAR (OSNEAR -»OSTPTICKTYPE, OsTPTimVal_)

current STM count value

Note:

Declared if defined **OSTPTIMSTM**

OSDECLAREVAR (OSNEAR -»OSBYTE, OsTPForceReason_)

reason of TP forced interrupt

Note:

Declared if defined **OSTPTIMSTM**

OSDECLAREVAR (OSNEAR -»OSObjectType, OsTPResLockObject_)

TP object (TASK or ISR2) with current resource locking time.

Note:

Declared if (defined **OSNTPTSKRESLOCKTIME** > 0 or **OSNTPISRRESLOCKTIME** > 0)

OSDECLAREVAR (OSNEAR -»OSObjectType, OsTPIntLockObject_)

TP object (TASK or ISR2) with current interrupt locking time.

Note:

Declared if (defined **OSNTPTSKINTLOCKTIME** > 0 or **OSNTPISRINTLOCKTIME** > 0)

OSDECLAREVAR (OSNEAR -»OSObjectType, OsTPBudgetObject_)

TP object (TASK or ISR2) with current budget.

Note:

Declared if defined **OSTIMINGPROTECTION** and **-»**OSNTPBGTS > 0

OSDECLAREVAR (OSNEAR -»AppModeType, OsCurAppMode_)

The current Application mode.

Note:

Declared if **OSORTIDEBUGLEVEL** > 0 or **OSNAPPMODES** > 1

OSDECLAREVAR (OSNEAR -»OSTSKCBPTR, OsRunning_)

Running task (0, if no task is running).

Note:

Declared if defined **OSUSETCB**
Declared if NOT defined **OSUSETCB**

OSDECLAREVAR (OSNEAR -»OSDWORD, OsSchedulerVector2_)

Mask of activated tasks(priorities) Extended vector.

Note:

Declared if defined **OSEXTPRIORS**

OSDECLAREVAR (OSNEAR -»OSDWORD, OsSchedulerVector1_)

Mask of activated tasks(priorities).

OSDECLAREVAR (OSNEAR -»OSWORD, OsISRResourceCounter_)

ISR/Resource Counter.

Note:

Declared if defined **OSRESOURCEISR**

OSDECLAREVAR (OSNEAR -»OSDWORD, OsInitialMSR_)

Saved initial MSR.

Note:

Declared if defined **OSUSEISRLEVEL** and (**OSISRENTRYEXIT** or **OSSC1**)

OSDECLAREVAR (OSNEAR -»OSISRLEVELTYPE, OsIsrLevel_)

Number of active ISRs & context bits.

Note:

Declared if defined **OSUSEISRLEVEL**

OSDECLAREVAR (OSNEAR -»OSDWORD, OsIntCheck_)

check correct order of the nested Suspend/Resume pairs

Note:

Declared if defined **OSISRENTRYEXIT** and **OSHOOKERROR_EXT**

OSDECLAREVAR (OSNEAR -»OSINTMASKTYPE, OsOldIntMaskAll_)

SuspendAll/ResumeAllInterrupts services.

OSDECLAREVAR (OSNEAR *OSBYTE*, OsSuspendLevelAll_)

level of the nested Suspend/ResumeAll pairs

OSDECLAREVAR (OSNEAR -»OSINTMASKTYPE, OsOldInterruptMask_)

Saved interrupt mask for SuspendOS/ResumeOSInterrupts services.

OSDECLAREVAR (OSNEAR -»OSDWORD, OsOldMsr_)

Saved MSR for EnableAll/DisableAllInterrupts.

OSDECLAREVAR (OSNEAR -»OSBYTE, OsSuspendLevel_)

Level of the nested Suspend/Resume pairs.

Note:

Declared if defined **OSISRENTRYEXIT**

OSDECLAREVAR (OSNEAR volatile -»OSWORD, OSISRId_)

The id of currently running ISR.

Note:

Declared if defined **OSISRENTRYEXIT** and **OSORTIRUNNINGISRID**

OSDECLAREVAR (OSNEAR -»StatusType, OsLastError_)

The last error declaration.

Note:

Declared if defined **OSORTIDEBUGLEVEL**

OSDECLAREVAR (OSNEAR -»OSDWORD, OsObjId_)

For first parameter.

Note:

Declared if defined **OSGETSERVICEID** or **OSPARAMETERACCESS**

OSDECLAREVAR (OSNEAR -»OSServiceIdType, OsService_)

For OSErrorGetServiceId() from ErrorHook.

Note:

Declared if defined **OSGETSERVICEID** or **OSPARAMETERACCESS**

OSDECLAREVAR (OSNEAR -»OSHWTickType, OsTimVal_)

hardware Timer value

Note:

Declared if defined **OSHWCOUNTER**

OSDECLAREVAR (OSNEAR -»OSDWORD, OsCtrIncCounter_)

Counter Id.

Note:

Declared if defined **OSCOUNTER** and **OSALMINCCOUNTER**

OSDECLAREVAR (OSNEAR volatile -»OSBYTE, OsOrtiOldServiceId_)

ORTI old service Id.

Note:

Declared if defined **OSORTICURRENTSERVICEID**

OSDECLAREVAR (OSNEAR volatile -»OSBYTE, OsOrtiRunningServiceId_)

ORTI current srvice Id.

Note:

Declared if defined **OSORTICURRENTSERVICEID**

OSDECLAREVAR (OSNEAR -»ApplicationType, OsViolatorAppId_)

Application Id of runnable which has violated.

Note:

Declared if defined **OSAPPLICATION** and **OSTERMINATION** and **OSHOOKPROTECTION**

OSDECLAREVAR (OSNEAR -»ISRType, OsViolatorISRId_)

ISR2 Id of ISR2 which has violated.

Note:

Declared if defined **OSTERMINATION** and **OSHOOKPROTECTION**

OSDECLAREVAR (OSNEAR -»TaskType, OsViolatorTaskId_)

Task Id of task which has violated.

Note:

Declared if defined **OSTERMINATION** and **OSHOOKPROTECTION**

OSDECLAREVAR (OSNEAR -»OSBYTE, OsKilled_)

Flags that Task or ISR was killed or some TP context.

Note:

Declared if defined **OSTERMINATION**

OSErrorHook

{DD_171}

void OSErrorHook (-»StatusType *error*, -»OSServiceIdType *ID*, -»OSObjectType *param*)

calls user defined hook if not nested. This function is called if OS configuration parameters "OSGETSERVICEID" or "OSPARAMETERACCESS" are defined

Parameters:

[in] *error* The status of the error
[in] *ID* The service Id
[in] *param* The object

Returns:

None

Note:

2 variants - differs in number of arguments

Implementation

- Put "error" input parameter to “OsLastError” global variable. This variable contains error code of last happened error.
- Prevent nested calls of function OSErrorHook by checking context bits OSISRLEVELHOOKERROR and OSISRLEVELHOOKAPPERROR in OsIsrLevel global variable
- If counter of nested Suspend/Resume pairs is zero and current context is not context of Pre/PostTaskHook then disable OS interrupts
- else increase counter of nested Suspend/Resume pairs
- Put "ID" and "param" parameters in OsService and OsObjId global variables. The OsService variable contains ID of service function that cause error and ID of erroneous object (via the macro OSPUTPARAM).
- Save previous value of OsIsrLevel

- Set flag of error hook context
- Call user error hook function
- Restore value of OsIsrLevel variable
- Save previous value of OsIsrLevel
- Set flag of application error hook context
- Call user application error hook function
- Restore value of OsIsrLevel variable
- Clear OsService variable
- If counter of nested Suspend / Resume pairs is one and current context is not context of Pre / PostTaskHook then enable OS interrupts
- else decrease counter of nested Suspend / Resume pairs

Used data

-»OsLastError

-»OsIsrLevel

-»OsSuspendLevel

-»OsService

-»OsObjId

OSErrorHook_1

void OSErrorHook_1 (-»StatusType *error*)

calls user defined hook if not nested This function is called if OS configuration parameters "OSGETSERVICEID" or "OSPARAMETERACCESS" are NOT defined

Parameters:

[in] *error* The status of the error

Returns:

None

Note:

2 variants - differs in number of arguments

Implementation

- Put "error" input parameter to "OsLastError" global variable. This variable contains error code of last happened error.

- Prevent nested calls of function OSErrorHook by checking context bits OSISRLEVELHOOKERROR and OSISRLEVELHOOKAPPERROR in OsIsrLevel global variable
- If counter of nested Suspend / Resume pairs is zero and current context is not context of Pre / PostTaskHook then disable OS interrupts
- else increase counter of nested Suspend / Resume pairs
- Save previous value of OsIsrLevel
- Set flag of error hook context
- Call user error hook function
- Restore value of OsIsrLevel variable
- Save previous value of OsIsrLevel
- Set flag of application error hook context
- Call user application error hook function
- Restore value of OsIsrLevel variable
- If counter of nested Suspend / Resume pairs is one and current context is not context of Pre / PostTaskHook then enable OS interrupts
- else decrease counter of nested Suspend / Resume pairs

Used data

-»OsLastError

-»OsIsrLevel

-»OsSuspendLevel

OSErrorHook_noPar

void OSErrorHook_noPar (-»StatusType *error*)

calls user defined hook if not nested

Parameters:

[in] *error* The status of the error

Returns:

None

Note:

called from internal OS functions with interrupts disabled

Implementation

- Put "error" input parameter to "OsLastError" global variable. This variable contains error code of last happened error.
- Prevent nested calls of function OSErrorHook by checking context bits OSISRLEVELHOOKERROR and OSISRLEVELHOOKAPPERROR in OsIsrLevel global variable
- Save previous value of OsIsrLevel
- Set flag of error hook context
- Call user error hook function
- Restore value of OsIsrLevel variable
- Save previous value of OsIsrLevel
- Set flag of application error hook context
- Call user application error hook function
- Restore value of OsIsrLevel variable

Used data

-»OsLastError

-»OsIsrLevel

OSGetAppErrorHook

{DD_172}

OSINLINE -»OSHKERROR OSGetAppErrorHook (-»OS_AppType * *app*)

Get Application Error Hook.

Parameters:

[in] *app* - a pointer to OS-application

Returns:

Application specific error hook address

Note:

none

Implementation

- Returns application specific error hook address

Used data

-»OsAppTable

Service Protection

The main purpose of service protection is guarding against incorrect service functions parameters, using services in wrong context, attempt to manipulate with objects that belongs to another OS-Application.

The service protection functionality in SC1, SC2 classes can be confgured through the OS status parameter and in extended status contains:

- context checking of called service
- verification of disabled / suspended interrupts
- checking of input parameters

Besides described before in SC3, SC4 classes service protection has an additional functionality:

- checking read or / and write access to the memory area that addressed through reference type parameters of system service

If OS-Applications exist and extended status is defined, then executed:

- checking permissions of service caller for the object

All service protection actions are executed before entering in service critical section code.

OsIsrLevel variable is storing container for context attribute bits.There are number of context specified bits:

- `OSISRLEVELTASK` - ISR resource
- `OSISRLEVELHOOKSTARTUP` - Startup hook
- `OSISRLEVELHOOKSHUTDOWN` - Shutdown hook
- `OSISRLEVELHOOKPRETASK` - Pretask hook
- `OSISRLEVELHOOKPOSTTASK` - Posttask hook
- `OSISRLEVELHOOKERROR` - Error hook
- `OSISRLEVELHOOKPROTECTION` - Protection hook
- `OSISRLEVELCALLBACK` - alarmcalback function
- `OSISRLEVELHOOKAPPERROR` - application error hook
- `OSISRLEVELISR1` - ISR category 1

- `OSISRLEVELTASKMASK` - ISR category 2 or ISR resource

These bits are set/cleared when execution context reached/left an appropriate part of OS code.

Also OsIsrLevel variable contains bits that prevent a service execution if EnableAllInterrupts(), ResumeAllInterrupts(), ResumeOSInterrupts() were called and no corresponding DisableAllInterrupts(), SuspendAllInterrupts(), SuspendOSInterrupts() was done before. These bits are:

- OSISRLEVELDISABLEINT - indicate that DisableAllInterrupt() service was called before
- OSISRLEVELDISABLEINTALL - indicate that SuspendAllInterrupt() service was called before
- OSISRLEVELDISABLEINTAOS - indicate that SuspendOSInterrupt() service was called before

After execution an appropriate service EnableAllInterrupts(), SuspendAlInterrupts(), SuspendOSInterrupts() these bits are cleared.

Each object in SC3, SC4 classes that can be accessed via services has the "appMask" field in object configuration structure. This field specified an application that have access to the given object and can call services that operate with him. By default application to whom belongs an object has access to that object. User can grant an access to object to another application by defining parameter ACCESSING_APPLICATION in OS configuration.

Write accessibility to reference type service parameters is checked by following algorithm:

- If execution context belongs to error, protection, pre- or posttask hooks
  - if memory area that addressed by reference parameter belongs to internal or external RAM then grant write access
  - else access is denied
- else if memory area belongs to stack of current task or ISR (if service was run from ISR) then grant write access
- if current application is trusted:

  - if memory area belongs to another task / ISR stack then access is denied
  - else if memory area belongs to internal or external RAM then grant write access
- else if current application is nontrusted:
  - if memory area belongs to memory application data then grant access
  - else access is denied

Data Structures

TagOSRemoteCallCB Struct Reference

{DD_173}

```
#include <Os_multicore_internal_types.h>
```

Data Fields

- ->OSDWORD arg0
- ->OSDWORD arg1
- ->OSDWORD arg2
- ->OSDWORD serviceId
- ->StatusType retcode

Memory Protection

The memory protection subsystem exists in SC3 and SC4 classes. It protects OS-Application data and code from modification by other non-Trusted OS-Application. The memory protection operates with data, code and stack sections of the executable program. The Operation System prevents write access to OS own data sections and OS own stack. The Operation System prevents write access from non-trusted OS-Applications to: {DD_174}

- all private stacks of Task(s) / Category 2 ISR(s) except runnable own stack;
- private data sections of all other OS-Applications;
- memory mapping peripheral space except specially assigned by User address regions (via **MemData<0/1/2>**)

The Operation System permits: {DD_175}

- a Task(s) / Category 2 ISR(s) read and write access to that Task's / Category 2 ISR's own private stack;
- a non-trusted OS-Application read and write access to that OS-Application own private data sections;

The trusted OS-Applications have the same access rights as the OS with the exception that access to the stack area is limited. {DD_176}

In case of memory violation the OS calls the **ProtectionHook(E_OS_PROTECTION_MEMORY)** and performs an appropriate action defined by its return value. {DD_177}

Three special variables are available for reading inside ProtectionHook to find what has caused the violation: {DD_178}

- -»**OsViolatorAppId** - contains the Application ID that caused violation or **INVALID_OSAPPLICATION**
- -»**OsViolatorTaskId** - contains the ID of the Task that caused violation or **INVALID_TASK**
- -»**OsViolatorISRId** - contains the ID of the ISR that caused violation or **INVALID_ISR**

Freescale OS Memory Protection is based on MPU.

The OS used MPU descriptors as follows: {DD_179}

Table 0.1 and Table 0.2 are applicable for all derivatives, except Leopard in DPM mode.

Table 0.1 Usage of MPU Descriptors by OS (one MPU, SingleCore)

Number	Memory area	Region	Access rights		Note
			System mode	User mode	
0	Region1	0..addr(.osstack)	RWX	none	This descriptor covers all address space before the section .osstack
1	OS stack	addr(.osstack) .. (addr(.osstack) + sizeof(.osstack) - 1)	RW	none	This descriptor covers the section .osstack
2	Region2	(addr(.osstack) + sizeof(.osstack)) .. 0xFFFFFFFF	RWX	none	The descriptor covers all address space after the section .osstack
3	Code and Constants	addr(.begincode) .. addr(.endcode)		RX	The descriptor covers code and constants area for non-trusted applications
4	Data of non-trusted applications in MemData0 area[a]	addr(<application data start in MemData0>) .. addr(<application data end MemData0>)		RW	
5	Data of non-trusted applications in MemData1 area	addr(<application data start in MemData1>) .. addr(<application data end MemData1>)		RW	

Table 0.1 Usage of MPU Descriptors by OS (one MPU, SingleCore)

Number	Memory area	Region	Access rights		Note
			System mode	User mode	
6	Data of non-trusted applications in MemData2 area	addr(<application data start in MemData2>) .. addr(<application data end MemData2>)		RW	
7	Runable Stack	addr(<runnable stack start>) .. addr(<runnable stack end>)		RW	
8	Whole RAM	addr(<RAM start address>) .. addr(<RAM end address>)		R	The descriptor covers whole RAM

[a.] MemData0 is configured by default; MemData1,2 are used only if these areas are configured

Table 0.2 Usage of MPU Descriptors by OS (one MPU, MultiCore)

Number	Memory area	Region	Access rights				Note
			System mode		User mode		
			Core_0 (master)	Core_1 (second	Core_0 (master)	Core_1 (second	
0	Region1	0..addr(.osstack)	RWX		none		This descriptor covers all address space before the section .osstack
1	OS stack	addr(.osstack) .. (addr(.osstack) + sizeof(.osstack) - 1)	RW	-	none		This descriptor covers the section .osstack

Table 0.2 Usage of MPU Descriptors by OS (one MPU, MultiCore)

Number	Memory area	Region	Access rights				Note
			System mode		User mode		
			Core_0 (master)	Core_1 (second	Core_0 (master)	Core_1 (second	
2	OS stack	addr(.osstack2) .. (addr(.osstack2) + sizeof(.osstack2) - 1)	-	RW	none		This descriptor covers the section .osstack2 on second core
3	Region2	(addr(.osstack2) + sizeof(.osstack2)) .. 0xFFFFFFFF	RWX		none		The descriptor covers all address space after the section .osstack2 for all cores
4	Code and Constants	addr(.begincode) .. addr(.endcode).	-		RWX		The descriptor covers code and constants area for non-trusted applications on all cores
5	Appl. Data0_0	addr(<application data start in MemData0>) .. addr(<application data end MemData0>)	-		RW	-	
6	Appl. Data0_1	addr(<application data start in MemData0>) .. addr(<application data end MemData0>)	-		-	RW	
7	Appl. Data1_0[a]	addr(<application data start in MemData1>) .. addr(<application data end MemData1>)	-		RW	-	

Table 0.2 Usage of MPU Descriptors by OS (one MPU, MultiCore)

Number	Memory area	Region	Access rights				Note
			System mode		User mode		
			Core_0 (master)	Core_1 (second	Core_0 (master)	Core_1 (second	
8	Appl. Data1_1	addr(<application data start in MemData1>) .. addr(<application data end MemData1>)	-		-	RW	
9	Appl. Data2_0	addr(<application data start in MemData2>) .. addr(<application data end MemData2>)	-		RW	-	
10	Appl. Data2_1	addr(<application data start in MemData2>) .. addr(<application data end MemData2>)	-		-	RW	
11	Runable Stack_0	addr(<runnable stack start>) .. addr(<runnable stack end>)	-		RW	-	
12	Runable Stack_1	addr(<runnable stack start>) .. addr(<runnable stack end>)	-		-	RW	
13	Whole RAM	addr(<RAM start address>) .. addr(<RAM end address>)	-		R		The descriptor covers whole RAM

a. MemData1,2 are used only if these areas are configured

There are some peculiarities of MPU descriptors' assignment for Leopard in DPM mode (Table 0.3 and Table 0.4) because it has two MPUs (macro *OSDUALMPU* defined).

Memory ranges for Table 0.3 and Table 0.4 are the same as in Table 0.1 and Table 0.2 (column "Region").

Table 0.3 shows the usage of MPU descriptors when Os-Application data and stacks are located in the same phisycal RAM (RAM0 or RAM1).

Table 0.3 Usage of MPU Descriptors by OS (one physical RAM area)

Number	Memory area		Access rights			
	MPU_0	MPU_1	System mode		User mode	
			Core_0 (master)	Core_1 (second	Core_0 (master)	Core_1 (second
0	Region1	Region1	RWX		-	
1	Stack_0		RW	-	-	
2	Stack_1		-	RW	-	
3	Region2	Region2	RWX		-	
4	Code_0	Code_1	RWX		RX	
5	Runable Stack_0		RW	-	RW	-
6	Runable Stack_1		-	RW	-	RW
7	Appl. Data0_0		RWX		RW	-
8	Appl. Data1_0		RWX		RW	-
9	Appl. Data2_0		RWX		RW	-
10	Appl. Data0_1		RWX		-	RW
11	Appl. Data1_1		RWX		-	RW
12	Appl. Data2_1		RWX		-	RW

Table 0.4 shows the usage of MPU descriptors when:

- OS-Application data and stacks for Core 0 are located in RAM0;
- OS-Application data and stacks for Core 1 are located in RAM1.

Stack_0 shall be placed in RAM0, Stack_1 shall be placed in RAM1.

Table 0.4 Usage of MPU Descriptors by OS (different physical RAM areas)

Number	Memory area		Access rights			
	MPU_0	MPU_1	System mode		User mode	
			Core_0 (master)	Core_1 (second	Core_0 (master)	Core_1 (second
0	Region0	Region0	RWX		-	
1	Stack_0		RW	-	-	
2	Region1'[a]	Region1'	RWX		-	
3		Stack_1	-	RW	-	
4	Region2	Region2	RWX		-	
5	Code_0	Code_1	RWX		RX	
6	Runable Stack_0	Runable Stack_1	RW		RW	
7	Appl. Data0_0	Appl. Data0_1	RWX		RW	
8	Appl. Data1_0	Appl. Data1_1	RWX		RW	
9	Appl. Data2_0	Appl. Data2_1	RWX		RW	

a. Memory range is addr(end of .osstack) .. addr(start of .osstack2)

The table below shows the usage of MPU descriptors when OS-Application data and stacks are located either RAM0 or RAM1.

Table 0.5 Usage of MPU Descriptors by OS : single core

Number	Memory area		Access rights	
	MPU_0	MPU_1	System mode	User mode
0	Region1	Region1	RWX	-
1	Stack		RW	-
2	Region2	Region2	RWX	-

Table 0.5 Usage of MPU Descriptors by OS : single core

Number	Memory area		Access rights	
	MPU_0	MPU_1	System mode	User mode
3	Code_0	Code_1	RWX	RX
4	Runable Stack		RW	RW
5	Appl. Data0		RWX	RW
6	Appl. Data1		RWX	RW
7	Appl. Data2		RWX	RW

In tables:

- *Appl. Data0_0(1)* - Data of non-trusted applications in MemData0 area on Core_0 (master) or Core_1 (second)
- *Appl. Data1_0(1)* - Data of non-trusted applications in MemData1 area on Core_0 (master) or Core_1 (second)
- *Appl. Data2_0(1)* - Data of non-trusted applications in MemData2 area on Core_0 (master) or Core_1 (second)

Additionally, up to 6 MPU descriptors may be used for Peripheral regions protection according to OS configuration. Either MPU0 or MPU1 descriptor is used depending on Peripheral region.

Settings for MPU descriptors for OS stacks, code and constants and system regions are constant per system and are done on starting of OS.

Settings for MPU descriptors for data of non-trusted applications per application are done when OS-Application is switched.

Setting for MPU descriptor for runable stack is done when runnable is switched.

The OS controls its access to stack area via MPU descriptor for .osstack or .osstack2: the descriptor VLD bit is set to 1 when the OS needs the access to all stack space, and to 0 when the OS runs a runnable.

Note that for trusted applications the OS forbids the access to other runnable stacks.

The OS sets runable stack MPU descriptor boundaries for tasks and ISRs cat 2 using TOS and BOS fields in the runnable control block.

For basic tasks the OS fills the BOS as bottom of common stack and the TOS as stack pointer value aligned by 32 when the task is activated. The TOS and BOS values for extended task and ISRs car.2 are filled in their control blocks when the system is initialized.

The OS uses boundaries of compiler data sections for protection non-trusted application data. Proper allocation application data to data section is provided by the user. The user applies for this files Os_memmap.h and linker file generated by the system generator.

Data Type

{DD_180}

The AUTOSAR OS establishes the following data types for the OS-Applications and memory management:

- -»ApplicationType the abstract data type for OS-Application identification
- -»RestartType argument type for -»TerminateApplication() service
- -»MemoryStartAddressType data type for pointer which is able to point to any location in the MCU address space
- -»MemorySizeType data type for the size of a memory region;
- -»TrustedFunctionParameterRefType reference type for arguments of Trusted Function (it is void*).

System macros for memory access

{DD_181}

The following macros return non-zero value if the corresponded type of access is available:

- OSMEMORY_IS_READABLE(x)
- OSMEMORY_IS_WRITEABLE(x)
- OSMEMORY_IS_EXECUTABLE(x)

- OSMEMORY_IS_STACKSPACE(x) macros are applicable only to the values of type -»AccessType.

Constants

{DD_182}

- INVALID_OSAPPLICATION constant of data type
- ACCESS
- NO_ACCESS

Memory Protection functions

OSInitMemProtection

{DD_183}

void OSInitMemProtection (void)

initializes MPU descriptors and switches MPU on.

Returns:

none

Note:

none

Implementation

The function performs:

- switch off the MPU;
- initialize constant MPU descriptors (0..3 and 6 if Z0, Z1 is used);
- switch off other descriptors;
- set constant parts (rights) of dynamic MPU descriptors;
- calculate data boundaries for non-trusted application (OsMPDataRAM*). These values will be written directly to MPU descriptors (4, 7 or 6..8), when an application will be switched;
- set current OS-Application as invalid;
- set last running OS-Application as invalid;
- switch the MPU on.

Used data

-»OsLastApp

-»OsMPU_ConstRGD

-»OsMPDataRAM

-»OsMPDataRAM_1

-»OsMPDataRAM_2

OSStartMemProtection

{DD_328}

void OSStartMemProtection (void)

Start Memory Protection synchronously. It shall be called after first synchronization point

Returns:

none

Note:

none

Implementation

- Set RW rigths for osstack and osstack2;
- Switch on MPU.

OSFillRights

{DD_329}

void OSFillRights (-»OSWORD descNum, -»OSDWORD rights)

Fill rights in the given descriptor

Parameters:

[in] *descNum* The descriptor
[in] rights The rights

Returns:

none

Note:

none

Implementation

OSFillConstDesc

{DD_330}

void OSFillConstDesc (-»OSWORD descNum)

Fill constant descriptor

Parameters:

[in] *descNum* The descriptor

Returns:

none

Note:

none

Implementation

OSAppMemAccess

{DD_184}

-»AccessType OSAppMemAccess (-»ApplicationType *appId*, -»OSDWORD *start*, -»OSDWORD *end*)

Determinate access type to memory area for application.

Parameters:

[in] *appId* The OS-Application Id
[in] *start* The start address
[in] *end* The end address

Returns:

how a specific memory region can be accessed
OSMP_R - read
OSMP_W - write
OSMP_S - stack
OSMP_E - execution

Note:

none

Implementation

- If application is nontrusted:

  - If memory area between start and end addresses belongs to given application data area then return OSMP_R and OSMP_W flags
  - If memory area between start and end addresses belongs to code then return OSMP_R and OSMP_E flags
- Else if application is trusted:
  - If memory area between start and end addresses belongs to 'osstack' area then return no access
  - If memory area between start and end addresses belongs to RAM then return OSMP_S, OSMP_E, OSMP_R and OSMP_W flags
  - If memory area between start and end addresses belongs to ROM then return OSMP_E, OSMP_R and OSMP_W flags
  - If memory area between start and end addresses belongs to peripheral bridge then return OSMP_R and OSMP_W flags
- Else return no access

Used data

-»OsMPData

-»OsMPData_1

-»OsMP_Data_2

-»osbegincode

-»osendcode

-»osstack

-»OsMemRgn

OSCheckWriteAccess

{DD_185}

-»AccessType OSCheckWriteAccess (-»MemoryStartAddressType *address*, -»MemorySizeType *size*)

Checks the OS-Application write access to the data area.

Parameters:

[in] *address* The memory start address type
[in] *size* The memory region size

Returns:

OSTRUE - if write access is permitted (area is stack or application data), OSFALSE otherwise

Note:

none

Implementation

- If execution context belongs to error, protection, pre- or posttask hooks
  - if memory area that addressed by reference parameter belongs to internal or external RAM then grant write access
  - else access is denied
- else if memory area belongs to stack of current task or ISR (if service was run from ISR) then grant write access
- if current application is trusted:
  - if memory area belongs to another task / ISR stack then access is denied
  - else if memory area belongs to internal or external RAM then grant write access
- else if current application is nontrusted:
  - if memory area belongs to memory application data then grant access
  - else access is denied

Used data

-»OsIsrLevel

-»OsIsrArray

-»OsRunning

-»osstack

-»OsMPData

-»OsMPData_1

-»OsMP_Data_2

OS_CheckISRMemoryAccess

{DD_186}

-»AccessType OS_CheckISRMemoryAccess (-»ISRType *isrId*,
-»MemoryStartAddressType *address*, -»MemorySizeType *size*)

Checks whether or not the private memory section of the ISR is within the memory section.

Parameters:

[in] *isrId* The ISR Id
[in] *address* The memory start address type
[in] *size* The size of memory region

Returns:

ACCESS - access allowed.
NO_ACCESS - access from this OS-Application is not allowed or ISRID is invalid.

Note:

This service returns the access rights of the ISR on specified memory area if the ISR reference <isrId> is valid. If access rights are not valid or <ISRID> is not valid the service yields no access rights.
Particularities: Memory access checking is possible for all OS-Applications and from all OS-Applications and does not need granted rights The service call is allowed on task level, ISR cat.2 level and in ErrorHook, ProtectionHook hook routines.

Implementation

- Check service calling context
- Check that service was not called with disabled interrupts
- Check validity of isrId parameter including that given ISR is not system timer or ISR category 1
- If memory area determined by address and size parameters belongs to given ISR stack then return OSMP_S, OSMP_R, OSMP_W flags
- Else call OsAppMemAccess function

Used data

-»OsIsrLevel

-»OsIsrTable

OS_CheckTaskMemoryAccess

{DD_187}

-»AccessType OS_CheckTaskMemoryAccess (-»TaskType *taskId*,
-»MemoryStartAddressType *address*, -»MemorySizeType *size*)

checks whether or not the private memory section of the task is within the memory section

Parameters:

[in] *taskId* The Task Id
[in] *address* The memory start address type
[in] *size* The memory region size

Returns:

ACCESS - access allowed.
NO_ACCESS - access from this OS-Application is not allowed or ISRID is invalid.

Note:

This service returns the access rights of the task on specified memory area if the task reference <taskId> is valid. If access rights are not valid or <TaskID> is not valid the service yields no access rights.
Particularities: Memory access checking is possible for all OS-Applications and from all OS-Applications and does not need granted rights The service call is allowed on task level, ISR cat.2 level and in ErrorHook, ProtectionHook hook routines.

Implementation

- Check service calling context
- Check that service was not called with disabled interrupts
- Check validity of taskId parameter
- If memory area determined by address and size parameters belongs to given extended task stack then return OSMP_S, OSMP_R, OSMP_W flags
- Else call OsAppMemAccess function

Used data

-»OsIsrLevel

-»OsTaskTable

OS_CallTrustedFunction

{DD_188}

-»StatusType OS_CallTrustedFunction (-»TrustedFunctionIndexType *ix*, -»TrustedFunctionParameterRefType *ptr*)

Calls a trusted function written by user.

Parameters:

[in] *ix* The Index of the function to be called. It is generated by SysGen in the form <service_name>ID.
[in] *ptr* pointer to the parameters for the function, it is specified by the function to be called. If no parameters are provided, a NULL pointer has to be passed.
[out] *depends* on Function Index ix

Returns:

Standard:
E_OK no error
Extended:
E_OS_SERVICEID trusted function is not configured
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_CALLEVEL call at not allowed context.
E_OS_ACCESS function belong to another core.

Note:

If <FunctionIndex> is a defined function index, this service switches the processor into privileged mode and calls the function <FunctionIndex> out of list of implementation specific trusted function and returns E_OK after completion. The called trusted function must conform to the following C prototype:

```
void TRUSTED_<name of trusted service>
(TrustedFunctionIndexType,
TrustedFunctionParameterRefType);
```

(The argument is the same as the argument of CallTrustedFunction). When function <FunctionIndex> is called, it gets the same permissions (access rights) as the associated trusted OS-application. If the function <FunctionIndex> is undefined, the service return E_OS_SERVICE_ID.
Particularities: If the trusted function is called from a Category 2 ISR context it shall continue to run on the same interrupt priority and be allowed to call all system services defined for Category 2 ISR<FunctuionIndex> is generated by SG as <function name>="">ID. If the trusted function is called from a task context it shall continue to run on the same priority and be allowed to call all system services defined for tasks. Normally, a user will not directly call this service, but it will be part of some standard interface, e.g. a standard I/O interface. Applicable in SC3, SC4.

Implementation

The function performs following:

- check that context and input parameters are proper;
- save current application Id to temporarily variable;
- enter OS critical section;
- if it isn't nested call of the service for calling application then disable tasks rescheduling and disable ISRs of calling application
- increment counter of nested calls of the service for calling application
- leave OS critical section;
- call the user trusted function;
- enter OS critical section;
- decrement counter of nested calls of the service of calling application
- if counter of nested calls of the service of calling application is zero then enable tasks rescheduling and enable ISRs of calling application
- if timing protection violation has occured during call of the service then call timing protection handler
- Call -»OSDISPATCH() function
- leave OS critical section;
- return E_OK.

OS_CheckObjectAccess

{DD_189}

-»ObjectAccessType OS_CheckObjectAccess (-»ApplicationType *appId*, -»ObjectTypeType *objectType*, -»OSObjectType *objectId*)

Returns the ability of given application to access given object.

Parameters:

[in] *appId* The OS-Application Id
[in] *objectType* The type of the object
[in] *objectId* The Id of the object to be checked

Returns:

ACCESS - access allowed.
NO_ACCESS - access from this OS-Application is not allowed.

Note:

This service returns ACCESS if the OS-Application <appId> has access to queried object. The service returns NO_ACCESS if the OS-Application <appId> has no access to queried object. The service always return ACCESS if the object to be examined is the RES_SCHEDULER.
/b Particularities: If the input parameters are invalid the service returns NO_ACCESS The service call is allowed on task level, ISR cat.2 level and in ErrorHook, ProtectionHook hook routines.

Implementation

- Check service calling context
- Check that service was not called with disabled interrupts
- Check validity of appId parameter
- Check validity of objectId parameter
- If object type is task then take appropriate appMask field from OsTaskTable structure
- If object type is ISR then take appropriate appMask field from OsIsrTable structure
- If object type is alarm then take appropriate appMask field from OSALMARRAY structure
- If object type is resource then take appropriate appMask field from OsResources structure
- If object type is counter then take appropriate appMask field from OsCounters structure
- If object type is schedule table then take appropriate appMask field from OsScheduleTables structure
- If object type is message then take appropriate appMask field from OsMessage structure if it’s receive message or from OsSndMessages if it’s send massage
- Else appMask is zero
- On the basis of obtained appMask value and appId parameter return access status

Used data

-»OsIsrLevel

-»OsTaskTable

-»OsIsrTable

-»OSALMARRAY

-»OsResources

-»OsCounters

-»OsScheduleTables

OS_TerminateApplication

{DD_190}

-»StatusType OS_TerminateApplication (-»ApplicationType *appId*, -»RestartType *RestartOption*)

Kills all tasks and ISRs which belong to current application.

Parameters:

[in] *appId* The OS-Application Id
[in] *RestartOption* If RestartOption equals RESTART, the configured RESTARTTASK of the terminated OS-Application is activated.

Returns:

Standard:
E_OK no error.
E_OS_VALUE if Restart option is not of RestartType.
E_OS_CORE inaccessable another core.
E_OS_ACCESS insufficient access rights.
E_OS_VALUE if Restart option is not of RestartType.
E_OS_ID invalid application ID.
E_OS_STATE invalid application state.
Extended:
E_OS_CALLLEVEL if called from wrong context.

E_OS_DISABLEDINT call when interrupts are disabled by OS services.

E_OS_VALUE if Restart option is not of RestartType.

Note:

This service terminate the calling OS-Application, it does not returns if there were no errors.

Particularities: The service call is allowed on task level, ISR cat.2 level and in Application-specific ErrorHook. This service terminates the calling OS-Application if called from allowed context (i.e. shall kill all runnables,

disable ISRs and free all other OS resources associated with the application). If service is called from wrong context then shall return E_OS_CALLEVEL. If <RestartOption> equals RESTART service shall activate the configured RESTARTASK of terminated application if it is configured.

Implementation

The function performs:

- check that calling context and input parameter are proper;
- enter OS critical section and disable all interrupts (via bit EE in MSR);
- call OSTerminateApplication;
- if runnable was a task and the task was killed, enable interrupts (via bit EE in MSR) and call task internal dispatcher. The OS will leave critical section after dispatching.
- else (running ISR cat.2 has killed): decrease -»OsIsrLevel, long jump to interrupt dispatcher.

Used data

-»OsKilled

-»OsRunning

-»OsIsrLevel

-»OsISRBufs

OSTerminateApplication

{DD_191}

-»OSWORD OSTerminateApplication (-»RestartType *RestartOption*)

Kills all tasks and ISRs which belong to current application.

Parameters:

[in] *RestartOption* If RestartOption equals RESTART, the configured RESTARTTASK of the terminated OS-Application is activated.

Returns:

OSTRUE on success,
OSFALSE - otherwise

Note:

none

Implementation

The function may be called from the service TerminateApplication or from the places where application is killed when it is violator. The function performs:

- return OSFALSE if given application is invalid. It may be when the violation occurred in the user ErrorHook;
- restore the field appId in the control block of current runnable, because it might be changed after call of trusted function;
- kill alarms, schedule tables and tasks belonged to given application;
- if it is necessary to restart given application, activate restart task;
- kill ISRs belonged to given application;
- return OSTRUE.

Used data:

-»OsRunning

-»OsAppTable

-»OsTaskTable

-»OsTaskCfgTable

-»OsIsrArray

-»OsIsrTable

-»OsIsrCfg

OS_CheckObjectOwnership

{DD_192}

-»ApplicationType OS_CheckObjectOwnership (-»ObjectTypeType *objectType*, -»OSObjectType *objectId*)

Returns the identifier of the OS-Application to which the object belongs.

Parameters:

[in] *objectType* - type of the next parameter
[in] *objectId* - a refernce to the object

Returns:

ID of the application that owns the object
INVALID_OSAPPLICATION if no application owns the object or if the object is RES_SCHEDULER

Note:

This service returns identifier of the OS-Application to which the object belongs. The service returns *INVALID_OSAPPLICATION* if the object does not exist. The service always returns *INVALID_OSAPPLICATION* if the object to be examined is the *RES_SCHEDULER*.

Particularities: The service call is allowed on task level, ISR cat.2 level and in ErrorHook, ProtectionHook hook routines.

Implementation

- Check service calling context
- Check that service was not called with disabled interrupts
- Check validity of objectId parameter
- If object type is task then return appropriate appId field of OsTaskTable structure if object index is valid and INVALID_OSPPLICATION otherwise
- If object type is ISR then return appropriate appId field of OsIsrTable structure if object index is valid and INVALID_OSPPLICATION otherwise
- If object type is alarm then return appropriate appId field of OSALMARRAY structure if object index is valid and INVALID_OSPPLICATION otherwise
- If object type is counter then return appropriate appId field of OsCounters structure if object index is valid and INVALID_OSPPLICATION otherwise
- If object type is schedule table then return appropriate appId field of OsScheduleTables structure if object index is valid and INVALID_OSPPLICATION otherwise
- If object type is message then return appropriate appId field of OsMessages structure for receive messages (OsSndMessages for send messages) if object index is valid and INVALID_OSPPLICATION otherwise
- If object type is resource then return appropriate appId field of OsResources structure if object index is valid and INVALID_OSPPLICATION otherwise
- Else return INVALID_OSPPLICATION

Used data

-»OsIsrLevel

-»OsTaskTable

-»OsIsrTable

-»OSALMARRAY

-»OsResources

-»OsCounters

-»OsScheduleTables

OS_GetApplicationID

{DD_193}

-»ApplicationType OS_GetApplicationID (void)

shows which applications task,ISR or hook is running

Returns:

none
the current OS-Application Id.
INVALID_OSAPPLICATION if no OS-Application is running

Note:

This service returns the application identifier to which the executing TASK/ISR/hook belongs.
Particularities: The service is allowed in any context, except ISR1.

Implementation

The function checks that calling context is proper and return current application Id. Else return INVALID_OSAPPLICATION.

Used data:

OsCurApp (only for the OSes for Z0, Z1).

OSExceptionError

{DD_194}

void OSExceptionError (-»OSDWORD *srr0*, -»OSDWORD *srr1*)

Called when Instruction Error occurs.

Parameters:

[in] *srr0* The memory address to check if exception occurs in OS code itself
[in] *srr1* The problem state to check if is supervisor mode

Returns:

never

Note:

none

Implementation

The function is called when one of noncritical, critical, machine check exceptions occurs.

The function performs following:

- save to temporarily variable and clean MPU error bits;
- save MCSR register to temporarily variable and clean machine check exception;
- If there were MPU error bits, set error code E_OS_PROTECTION_MEMORY;
- else if MCSR was > 0: if it held bus error bits, set error code as E_OS_PROTECTION_MEMORY, else set error code as E_OS_PROTECTION_EXCEPTION;
- else: if ESR(ST) > 0 set error code as E_OS_PROTECTION_MEMORY, else set error code as E_OS_PROTECTION_EXCEPTION.
- if Protection hook is configured: if exception occurs in OS code itself or if exception occurs in error hooks or if exception occurs in ISR of category 1, shutdown OS. Else call OSProtectionHandler with error code;
- else shutdown OS.

Used data:

-»OsIsrLevel

OSTLBException

{DD_195}

void OSTLBException (-»OSDWORD *srr0*, -»OSDWORD *srr1*)

TLB exception (data or code).

Parameters:

[in] *srr0* The memory address to check if exception occurs in OS code itself
[in] *srr1* The problem state to check if is supervisor mode

Returns:

never

Note:

none

Implementation

The function performs following:

- call shutdown OS if Protection Hook is not configured;
- if Protection Hook is configured: if exception occurs in OS code itself or if exception occurs in error hooks or if exception occurs in ISR of category 1, shutdown OS. Else call OSProtectionHandler with error code E_OS_PROTECTION_MEMORY.

OSProtectionHandler

{DD_196}

void OSProtectionHandler (-»StatusType *ecode)*

Called from Memory and CPU exception handlers.

Parameters:

[in] *ecode* The code of error

Returns:

never

Note:

none

Implementation

The function performs:

- recognize the violator and fill the variables: OsViolatorTaskId, OsViolatorISRId, OsViolatorAppId.
- call the user Protection Hook and save an action chosen by the user;
- perform the action;
- call OSDI(), note that interrupts are disabled already;
- if the runnable is a task and the task is killed: enable interrupts(), get a task and call task internal dispatcher.
- else (runnable is ISR cat.2): decrease -»OsIsrLevel and long jump to ISR dispatcher.

OS_AllowAccess

{DD_326}

-»StatusType OS_AllowAccess (-»void)

Sets the own state of an OS-Application of caller from APPLICATION_RESTARTING to APPLICATION_ACCESSIBLE.

Parameters:

none

Returns:

Standard:
E_OK no error
Extended:
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_CALLEVEL call at not allowed context.
E_OS_STATE state of application is not APPLICATION_RESTARTING

Note:

Particularities: The service call is allowed on task and ISR cat.2 level.

Implementation

- Check service calling context
- Check that service was not called with disabled interrupts
- Enter OS critical section via *OSDIS*()
- If application is not in APPLICATION_RESTARTING state
  - Leave the OS critical section via the macro *OSRI()*
  - Return error code E_OS_STATE
- Set the application state to APPLICATION_ACCESSIBLE
- In extended status set accessible bits for all objects that belongs to application
- Leave the OS critical section via the macro *OSRI()*
- Return E_OK

Used data

-»OsIsrLevel

-»OsTaskTable

-»OsIsrTable

-»OSALMARRAY

-»OsResources

-»OsCounters

-»OsScheduleTables

OS_GetApplicationState

{DD_327}

-»StatusType OS_GetApplicationState (-»ApplicationType *appId*, -»ApplicationStateRefType *stateRef*)

Sets the own state of an OS-Application of caller from APPLICATION_RESTARTING to APPLICATION_ACCESSIBLE.

Parameters:

none

Returns:

Standard:
E_OK no error
E_OS_ID invalid applicaton ID
Extended:
E_OS_DISABLEDINT call when interrupts are disabled by OS services.
E_OS_CALLEVEL call at not allowed context.

Note:

Particularities: The service is allowed in any context, except ISR1.

Implementation

- Check service calling context
- Check that service was not called with disabled interrupts
- Check application ID
- Check the possibility to write into given pointer 'stateRef'
- Save state of given application to the 'stateRef' pointer
- Return E_OK

Used data

-»OsIsrLevel

-»OsAppTable

Linker defined symbols

{DD_197}

char OS_BEGIN_DSADDR (osstack)

Linker defined symbols for ProtectionHandler: beginning of OS stack section.

char OS_BEGIN_DSADDR (osbegincode)

Beginning of common code section.

char OS_BEGIN_DSADDR (osbssshared)

OSMPC5600: beginning of common data section.

char OS_BEGIN_DSADDR (ostext)

Linker defined symbols for ProtectionHandler: beginning of OS code section.

char OS_END_DSADDR (osstack)

Linker defined symbols for ProtectionHandler: end of OS stack section.

char OS_END_DSADDR (osendcode)

End of common code section.

char OS_END_DSADDR (osbssshared)

OSMPC5600: end of common data section.

char OS_END_DSADDR (ostext)

Linker defined symbols for ProtectionHandler: end of OS code section.

Variable Documentation

- -»OSSCTCB -»OsScheduleTables[OSNSCTS]
- -»OSDWORD -»OsExceptionStack [OSEXCEPTIONSTACKSIZE]
- const -»OSMEMRGN -»OsMemRgn[]
- const -»OsMPU_RGD -»OsMPU_ConstRGD[]

  Constant MPU region descriptors.
- const -»OSMP_DSADDR -»OsMPData[OSNNOTTRUSTEDAPPS]

  Array of application data section adresses (in ROM)

- const ->OSMP_DSADDR ->OsMPData_1[OSNNOTTRUSTEDAPPS]

  Array of application data section adresses (in ROM)

- const ->OSMP_DSADDR ->OsMPData_2[OSNNOTTRUSTEDAPPS]

  Array of application data section adresses (in ROM)

- const ->OSBYTE ->OsTrustedFunctionsAppId[OSNTRUSTEDFUNCTIONS]
- const ->OS_TRUSTED_FUNCTION ->OsTrustedFunctionsTable[]

Data Structures

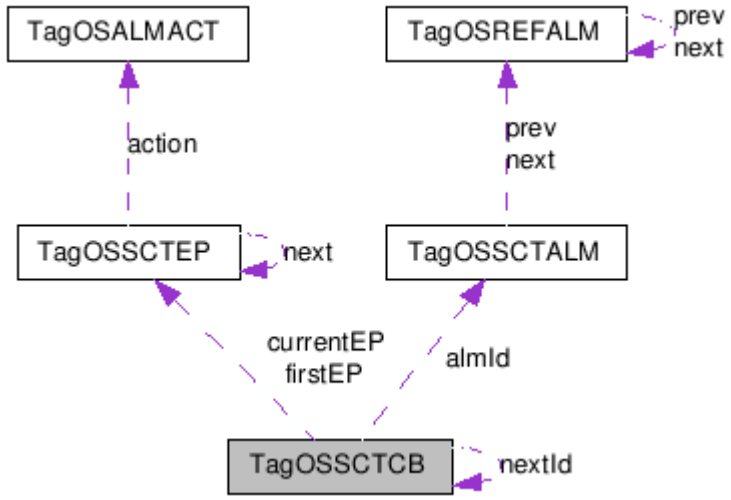
TagOSSCTCB Struct Reference

{DD_198}

```
#include <Os_schedule_table_internal_types.h>
```

Structure of the schedule table control block.

Collaboration diagram for TagOSSCTCB:



Data Fields

- const ->OSSCTEP * currentEP

  The current expiry point
- ->OSSCTCB * nextId

  The ScheduleTableId_To (used only for NextScheduleTable)
- ->OSSCTALMCB * almId

  The alarm for this SCT
- ->TickType deviation

  Deviation of ST local time and value of the synchronization count
- ->TickType precision

Schedule table sync.PRECISION value

- ->TickType syncOffset
  The next EP offset using for sync. correction
- ->TickType length
  The length of the schedule table in ticks
- ->TickType maxSync
  Schedule table MAX_CORRECTION_SYNC values
- ->TickType maxAsync
  Schedule table MAX_CORRECTION_ASYNC values
- ->TickType initialOffset
  Initial offset
- const ->OSSCTEP * firstEP
  The pointer to first expiry point in the schedule table
- ->OSWORD ctrIndex
  Attached Counter ID
- ->OSWORD state
  The config(MS) and status(LS) of schedule table
- ->OSDWORD appMask
  Application identification mask value
- ->ApplicationType appId
  Application identification value

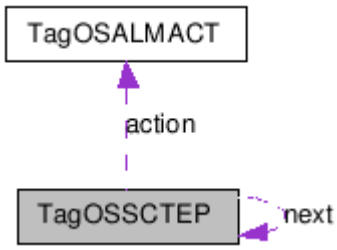
TagOSSCTEP Struct Reference

{DD_199}

```
#include <Os_schedule_table_config.h>
```

The expiry point structure.

Collaboration diagram for TagOSSCTEP:



Data Fields

- const -»OSSCTEP * next

  The next expiry point
- -»OSALMACT action

  Alarm action
- -»TickType delta

  The time space (in ticks) between the current expiry point and the next expiry point (or the end of the schedule if it's last expiry point)
- -»TickType maxRetard

  Schedule table MAXRETARD values
- -»TickType maxAdvance

  Schedule table MAXADVANCE values

OSMEMRGN Struct Reference

{DD_200}

Memory region rights structure.

Data Fields

- -»OSDWORD start
- -»OSDWORD end
- -»AccessType rights

OsMPU_RGD Struct Reference

{DD_201}

MPU region descriptor structure.

Data Fields

- ->OSDWORD word0
- ->OSDWORD word1
- ->OSDWORD word2
- ->OSDWORD word3
- ->OSBYTE id

OSMP_DSADDR Struct Reference

{DD_202}

```
#include <Os_memory_config.h>
```

Data Fields

- ->OSDWORD startaddr
- ->OSDWORD endaddr
- ->OSDWORD valid

Timing Protection

For safe and accurate timing protection it is necessary to enforce limits on the factors that determine whether or not Tasks / ISRs meet their respective deadlines. These limits are: {DD_204}

- the task execution budget and Category 2 ISR execution time;
- the execution time of each Task / Category 2 ISR while holding shared resources / disabling interrupts;
- the inter-arrival rate of each Task / Category 2 ISR.

Timing protection is applied to Tasks and Category 2 ISRs, but not to Category 1 ISRs. The Timing Protection is available in SC2 and SC4 classes. {DD_205}

Timing protection uses STM timer hardware. STM timer has four independent compare channels with common counter and prescaler. The OS uses the channels as follows {DD_206}

- channel 0 is used to count 32 bit high-order part of 64bit time. The 64bit time is used when inter-arrival rate is configured.
- channel 1 is used to handle interrupt locking time ('INTLOCK' channel)
- channel 2 is used to handle task execution budget / ISR2 execution time ('BUDGET' channel)
- channel 3 is used to handle resource locking time ('RESLOCK' channel).

There are separate interrupt handlers for each STM channel if each channel has separate INTC input. For some derivatives where several channels share one INTC input, a common interrupt handler is used. {DD_207}

Additional HW (INTC software interrupt) is used to force special OS-internal timing protection interrupt. {DD_208}

All timing protection interrupts have interrupt level higher than any interrupt level of ISR category 2 and system timers. {DD_209}

All timing protection interrupts have the same interrupt level (the OS interrupt level). Thus these interrupts cannot preempt the OS and each other. {DD_210}

Timing protection is implemented mainly via C-macro and inlined functions declared in timing protection header file. {DD_211}

If some timing protection macro or function is not necessary for some OS configuration it is declared as void macro. {DD_212}

All timing protection interrupt handlers run on own stack (only in SC3..4). {DD_213}

Note that timing protection doesn't protect interrupt locking done via the OS services {Disable/Enable}{All/OS}Interrupts because they control bit EE in CPU MSR register thereby disabling all interrupts including timing protection interrupts. {DD_214}

When timing protection violation happens, the OS calls provided by the user ProtectionHook which returned necessary action (if the hook is configured) or ShutdownOS. {DD_215}

All internal timing protection OS services check at the beginning of themselves that timing protection is configured for current runnable otherwise the timing protection is not applied. {DD_216}

It is assumed by design that any timing protection interrupt (except interrupt on channel 0) may happen only if there is real timing protection violation. {DD_217}

The OS performs all necessary reactions on timing protection violations only in timing protection interrupts (to simplify and make more robust timing protection design). If it is necessary to perform timing protection action inside the OS, the OS invokes timing protection forced interrupt (via INTC software interrupt). {DD_218}

Timing protection is used following configuration macros generated by system generator: {DD_219}

- **OSNTPBGTS** - number of tasks and ISRs with execution budget
- **OSNTPTSKBGTS** - number of tasks with execution budget
- **OSNTPISRBGTS** - number of ISR2s with execution budget
- **OSNTPTSKARRIV** - number of tasks with ISR arrival rate
- **OSNTPISRARRIV** - number of ISR2s with ISR arrival rate

- **OSNTPTSKINTLOCKTIME** - number of tasks with OS interrupt locking Time
- **OSNTPISRINTLOCKTIME** - number of ISR2s with OS interrupt locking Time
- **OSNTPTSKRESLOCKTIME** - number of tasks with at least one resource locking Time
- **OSNTPISRRESLOCKTIME** - number of ISR2s with at least one resource locking Time
- **OSNTPTSKRESLOCKS** - number of all resource locking time configurations for tasks
- **OSNTPISRRESLOCKS** - number of all resource locking time configurations for ISRs

In each point of time the system may have: {DD_220}

- Only one runnable with active execution budget or time. The variable -»OsTPBudgetObject holds the runnable Id or OSTPOBJECT0 if there is not any runnable with active budget;
- Only one runnable with active interrupt locking time. The variable -»OsTPIntLockObject holds the runnable Id or OSTPOBJECT0 if there is not any runnable with active interrupt locking time;
- Only one runnable with active resource locking time. The variable -»OsTPResLockObject holds the runnable Id or OSTPOBJECT0 if there is not any runnable with active resource locking time.

The variables {DD_221}

-»*OsTPBudgetObject*,

-»*OsTPIntLockObject*,

-»*OsTPResLockObject*

hold violating runnable Id when corresponding timing protection interrupt happens.

System ProtectionTimer

{DD_055}

For the Scalability Classes with timing protection (SC2 and SC4) the OS uses additional timer. The Protection Timer shall be Hardware based; the User has to configure used HW and timer parameters.

Timing Protection functions

OSInitializeTP

{DD_222}

void OSInitializeTP (-»AppModeType *mode*)

Initialize TP HW and start timer.

Parameters:

[in] *mode* if OSNAPPMODES > 1

Returns:

none

Note:

none

Implementation

The function is called from StartOS and performs following:

- Initialize of active timing protection runnable Ids -»OsTPBudgetObject, -»OsTPIntLockObject and -»OsTPResLockObject with the value OSTPOBJECT0;
- Initialize STM timer;
- Initialize the variable OsTPForceReason by zero. This variable is used when the OS calls timing protection forced interrupt;
- Initialize the variable OsTPOverflowCnt by zero. It is high part of 64bit time;
- Get current 64bit time;
- Fill timing protection fields for all task control blocks:
  - tpExecBudget - configured execution budget;
  - tpRemained - remained execution budget (= tpExecBudget);
  - tpTimeFrame - configured task inter-arrival rate time frame;

  - tpIntLockTime - configured task interrupt locking time;
  - tpResources - the list of allocated resources with timing protection;
  - tpNumberResLock - number of given task resource locking time configurations;
  - tpResLock - poitner to array with given task resource locking time configurations;
  - tpLast - task activate last time. It is initialized by current 64bit time If the task is autostarted, else it is reset by difference = current 64bit time - tpTimeFrame).
- Initialize timing protection fields for null task by zeroes.
- Fill timing protection fields for all ISR control blocks:
  - tpExecBudget - configured execution time;
  - tpTimeFrame - configured ISR time frame;
  - tpLast - time when ISR was started last. It is reset by difference = current 64bit time - tpTimeFrame;
  - tpNumberResLock - number of given ISR resource locking time configurations;
  - tpResLock - poitner to array with given ISR resource locking time configurations.

Used data

-»OsTPBudgetObject

-»OsTPIntLockObject

-»OsTPResLockObject

-»OsTPForceReason

-»OsTPOverflowCnt

-»OsTPTskResLockTime

-»OsTaskTable

-»OsTaskCfgTable

-»OsTPISRResLockTime

-»OsIsrCfg

-»OsIsrTable

OSTPProtectionHook

{DD_223}

-»ProtectionReturnType OSTPProtectionHook (-»OSObjectType *object*, -»StatusType *err*)

determines violator object and calls the user protection hook

Parameters:

[in] *object* The object where the error occurred
[in] *err* The code of the error occurred

Returns:

the action desired by user

Note:

none

Implementation

The function fills violator variables -»OsViolatorAppId, -»OsViolatorTaskId, -»OsViolatorISRId if runnable is known (object != OSTPOBJECT0), else fills these variables with INVALID_* constants (this situation is possible if the function was called from an alarm action, in that case runnable is unknown). Then provided by the user Protection Hook is called.

Used data

-»OsVoilatorAppId

-»OsvoilatorTaskId

-»OsVoilatorISRId

-»OsIsrTable

-»OsTaskTable

OSTPKillObjects

{DD_224}

void OSTPKillObjects (-»ProtectionReturnType *action*, -»StatusType *err*)

terminates some runnables under the action desired by the user

Parameters:

[in] *action* The action:

PRO_TERMINATETASKISR the OS shall terminate running Task or ISR and perform an appropriate cleanup.
PRO_TERMINATEAPPL the OS shall kill active OSApplication.
PRO_TERMINATEAPPL_RESTART the OS shall kill active OS-Application and then restart RESTARTTASK of this OS-Application.
PRO_SHUTDOWN the OS shall perform shutdown.
PRO_IGNORE the OS shall do nothing, allowed only in case of E_OS_PROTECTION_ARRIVAL

Returns:

none

Note:

It shall be used only in TP interrupts

Implementation

The function performs the action returned by the user ProtectionHook. The action may be following:

- PRO_TERMINATETASKISR. The function -»OSKillISR() is called if the violator is ISR2 (but if error == E_OS_PROTECTION_ISR_ARRIVAL the function -»OSKillISR() is not called because the user part of ISR has not been called actually), else the function -»OSKillTask() is called. Note when the violator is ISR, it is necessary to restore field appId in the ISR control block by configured value (application Id might be changed if a trusted function was called from this ISR).
- PRO_TERMINATEAPPL, PRO_TERMINATEAPPL_RESTART. The current application is saved. The current application is set as -»OsVoilatorAppId and current application is killed. If it was killed successfully the current application is restored by saved value, else the OS is shut downed.
- PRO_IGNORE. The function cannot be called with action PRO_IGNORE and error E_OS_PROTECTION_ARRIVAL (according to design of -»OSTPStartTaskFrame() and -»OSTPISRArrivalRate()), so we need to call -»OSShutdownOS (see OS475).
- PRO_SHUTDOWN, default. If error has internal OS error code E_OS_PROTECTION_ISR_ARRIVAL it is changed as Autosar error code E_OS_PROTECTION_ARRIVAL. Then -»OSShutdownOS is called.

Used data

-»OsIsrTable

-»OsTaskTable

-»OsVoilatorAppId

OSTPHandler

{DD_225}

OSINLINE void OSTPHandler (-»OSObjectType *object*, -»StatusType *err*)

Handler of timing protection violation.

Parameters:

[in] *object* The object Id
[in] *err* The error code

Returns:

none

Note:

It shall be used only in TP interrupts

Implementation

The function is called directly from timing protection ISR handlers when a timing protection violation occurs (except Task or ISR arrival rate violation). It calls -»OSTPProtectionHook to get the user action, then it calls -»OSTPKillObjects to kill violator runnable(s) according to provided by the user action.

Used data

none

OSISRTPViolation

{DD_226}

void OSISRTPViolation (void)

OSMPC5674F interrupt handler for 'budget',' interrupt locking time', 'resource locking time' STM channels are shared to common node.

Returns:

none

Note:

The ISR handler is used only on derivatives where three STM channels (1,2 and 3) have common INTC input.

Implementation

It is common timing protection ISR handler for execution budget / time, interrupt locking time and resource locking time violations. It performs following if the user protection hook is configured:

- If it is the interrupt on 'BUDGET' STM channel, enable interrupts, call -»OSTPHandler for object stored in -»OsTPBudgetObject, clear the HW interrupt flag in STM HW;
- Else if it is interrupt on 'INTLOCK' STM channel, enable interrupts, clear the HW interrupt flag in STM HW, call -»OSTPHandler for object stored in -»OsTPIntLockObject;
- Else if it is interrupt on 'RESLOCK' STM channel, enable interrupts, clear the HW interrupt flag in STM HW, call -»OSTPHandler for object stored in -»OsTPResLockObject;
- Else return from ISR because there are not STM channel interrupt requests (it is workaround for known INTC defect).
  - Disable interrupts
  - Call OSLeaveISR to perform necessary rescheduling or back to Interrupt Dispatcher.

    If the user Protection Hook is not configured, call -»OSShutdownOS.

Used data

-»OsTPBudgetObject
-»OsTPIntLockObject
-»OsTPResLockObject

OSISRTPTimerBudget

{DD_227}

void OSISRTPTimerBudget (void)

Interrupt handler for 'budget' STM channel.

ISR for STM timer, channel for budget.

Returns:

none

Note:

none

Implementation

If the user Protection Hook is configured:

- Check interrupt flag request on ’BUDGET’ channel and return if it is not set - it is a workaround for known INTC defect;
- Enable interrupts;
- Call -»OSTPHandler for object stored in -»OsTPBudgetObject;
- Clear the HW interrupt request flag on ’BUDGET’ channel
- Disable interrupts;
- Call -»OSTPLeaveISR to perform necessary task rescheduling or back to Interrupt Dispatcher.

  If the user Protection Hook is not configured, clean HW interrupt flag request and call -»OSShutdownOS.

Used data

-»OsTPBudgetObject

OSISRTPTimerIntLock

{DD_228}

void OSISRTPTimerIntLock (void)

Interrupt handler for 'interrupt locking time' STM channel.

ISR for STM timer, channel for interrupt locking time.

Returns:

none

Note:

none

Implementation

If the user Protection Hook is configured:

- Enable interrupts;
- Clear the HW interrupt request flag on ’INTLOCK’ channel;
- Call -»OSTPHandler for object stored in -»OsTPIntLockObject;
- Disable interrupts;
- Call -»OSTPLeaveISR to perform necessary task rescheduling or back to Interrupt Dispatcher.

If the user Protection Hook is not configured, clean HW interrupt request flag and call -»OSShutdownOS.

Used data

-»OsTPIntLockObject

OSISRTPTimerResLock

{DD_229}

void OSISRTPTimerResLock (void)

Interrupt handler for 'resource locking time' STM channel.

Returns:

none

Note:

none

Implementation

If the user Protection Hook is configured:

- Enable interrupts;
- Clear the HW interrupt request flag on ’RESLOCK’ channel;
- Call -»OSTPHandler for object stored in -»OsTPResLockObject;
- Disable interrupts;
- Call -»OSTPLeaveISR to perform necessary task rescheduling or back to Interrupt Dispatcher.

  If the user Protection Hook is not configured, clean HW interrupt request flag and call -»OSShutdownOS

Used data

-»OsTPResLockObject

OSISRTPForced

{DD_230}

void OSISRTPForced (void)

Force TP handler.

Forced ISR for alarm queue (software interrupt).

Returns:

none

Note:

none

Implementation

The forced timing protection interrupt performs deferred timing protection actions. When the interrupt is invoked, a reason bit is set in the variable -»OsTPForceReason. The interrupt handler enables interrupts, cleans HW interrupt request flag, checks bits in the variable -»OsTPForceReason and performs appropriate timing protection actions:

- If the reason is exhausted execution budget or time, it cleans the reason bit and calls -»OSTPHandler for an object stored in the variable -»OsTPBudgetObject.
- If the reason is exhausted interrupt locking time, it cleans the reason bit and calls -»OSTPHandler for an object stored in the variable -»OsTPIntLockObject.
- If the reason is exhausted resource locking time, it cleans the reason bit and calls -»OSTPHandler for an object stored in the variable -»OsTPResLockObject.

Used data

-»OsTPForceReason
-»OsTPBudgetObject
-»OsTPIntLockObject
-»OsTPResLockObject

OSISRTPTimerOVF

{DD_231}

void OSISRTPTimerOVF (void)

Interrupt handler for 'overflow' STM channel.

ISR for STM timer, channel for overflow counter.

Returns:

none

Note:

none

Implementation

It increments the variable -»OsTPOverflowCnt (high part of 64bit time) and cleans HW interrupt request flag. Note: a STM bug (FSL _9000_500) workaround is implemented here.

Used data

-»OsTPOverflowCnt

OSTPStopBudget

{DD_232}

void OSTPStopBudget (void)

Stop timing protection (the budget and/or the resource locking time) when a task or ISR is preempted by ISR2 or system timer.

Returns:

none

Note:

none

Implementation

The function performs following:

- Fix current HW timer counter to the variable -»OsTPTimVal.
- If there is a task with active executing budget, fill tpRemained field in task control block with remained time on budget and reset 'BUDGET' channel;
- Else if there is a ISR cat.2 with active executing time, fill tpRemained field in ISR control block with remained time on executing time and reset 'BUDGET' channel;
- If there is a task with active resource locking time, fill the field tpRemained in head resource control block in the list of allocated resources with TP (the field tpResources in the task control block) and reset 'RESLOCK' channel;
- Else if there is an ISR cat.2 with active resource locking time, fill tpRemained field in head resource control block in the list of allocated resources with TP (the field tpResources in the ISR control block) and reset 'RESLOCK' channel.

Used data

-»OsTPBudgetObject

-»OsTPResLockObject
-»OsRunning
-»OsIsrTable
-»OsTPTimVal

OSTPStartTaskFrameInAlm

{DD_233}

-»OSBYTE OSTPStartTaskFrameInAlm (-»OSWORD *taskId*)

start inter-arrival control when a task is activated or released

Parameters:

[in] *taskId* The Task Id

Returns:

OSFALSE or OSTRUE (if defined **OSHOOKPROTECTION**)
none (if NOT defined **OSHOOKPROTECTION**)

Note:

none

Implementation

The function is called when an alarm expires and activates or releases a task. Also the function is called for 'restart' task when an application is terminated by user with restart.

The function performs following:

- Return OSTRUE if inter-arrival rate is not configured for the task (the configured task time frame is zero).
- Get current 64bit time;
- If elapsed time from last task activation / release is less than configured task time frame:
- Get and save user protection hook action via OSTPProtectionHook if user protection hook is configured;
- Call -»OSShutdownOS if the action returned by user protection hook is not PRO_IGNORE or
  - if user protection hook is not configured.

    return OSFALSE
  - Else save current 64bit time to task control block field tpLast and return OSTRUE.

Used data

-»OSTaskTable
-»OsTPTimVal
-»OsTPHookAction

OSTPStartTaskFrame

{DD_300}

-»OSBYTE OSTPStartTaskFrame (-»OSWORD *taskId*)

Start control inter-arrival a task is activated or released.

Parameters:

[in] *taskId* The Task Id

Returns:

OSTRUE
OSFALSE if it's impossible to activate/release the task

Note:

It is implemented as a macro in API header if OSHOOKPROTECTION is not defined

Implementation

The function is called when a task is activated or released. The function performs following:

- Return if inter-arrival rate is not configured for the task (the configured task time frame is zero).
- Get current 64bit time;
- If elapsed time from last task activation / release is less then configured time frame (it is violation):
  - Get violator Id and its type (task or ISR cat.2). Note that violator is runnable who tried to activate / release task;
  - Get and save user protection hook action via call OSTPProtectionHook;
- If user action is not PRO_IGNORE, force timing protection interrupt to kill current runnable and maybe more;
  - Return OSFALSE;
- Save current 64bit time to task control block field tpLast and return OSTRUE;

Used data

-»OsTaskTable
-»OsTPTimVal
-»OsIsrArray
-»OsTPHookAction

OSTPISRArrivalRate

{DD_301}

#if defined(**OSHOOKPROTECTION**)
-»OSBYTE OSTPISRArrivalRate (-»OS_ISRTYPE * *isr*)
#else
-»void OSTPISRArrivalRate (-»OS_ISRTYPE * *isr*)
#endif

Start inter-arrival protection for ISR:

Parameters:

[in] *isr* The ISR Id

Returns:

If defined **OSHOOKPROTECTION**:
OSFALSE - there is violation
OSTRUE - no violation
If NOT defined **OSHOOKPROTECTION**:
call ShutdownOS if violation
return if no violation

Note:

OSShutdownOS will be called if violation occurs AND Protection Hook is not configured.

Implementation

The function is called from ISR dispatcher when user ISR cat.2 occurs. The function performs following:

- Check that inter-arrival rate protection is configured for given ISR (isr->tpTimeFrame > 0), and return if not.
- Get 64bit time
- If elapsed time from last given ISR occurrence is less then configured time frame:
  - Call -»OSShutdownOS if protection hook is not configured;

  - If protection hook is configured, get user protection hook action via OSTPProtectionHook with temporarily increase of -»OsISrLevel (it is necessary for GetISRID if it will be called from the user ProtectionHook), force timing protection interrupt OSTPFORCEISRARRIVAL to handle the action if the action is not PRO_IGNORE, return OSFALSE.
- Else save current 64bit time to field 'tpLast' of given ISR control block to remember last given ISR occurrence time.
- Return OSTRUE if protection hook is configured else return void.

Used data

-»OsTPTimVal
-»OsIsrLevel
-»OsTPHookAction

OSTPStartTaskResLockTime

{DD_234}

void OSTPStartTaskResLockTime (-»OSWORD *resInd*)

Start resource locking time in a task.

Parameters:

[in] *resInd* The Resource Id

Returns:

none

Note:

none

Implementation

The function is called from OSTPStartResLockTime.

The function performs following:

- Find the timing protection configuration for given Resource index in the array of resource timing protection configurations belonged to running task.
- For found configuration:
  - Save configured locking time value from the configuration to temporarily variable;

- If there is already locked resource with timing protection:
  - If remained time for the locked resource is "<=" then the saved locking time, return;
  - Else save remained time of already locked Resource as remained time minus saved locking time value;
  - Re-arm 'RESLOCK' channel with saved locking time value;
  - Add given resource to running task list of locked resources with timing protection.
  - Return.
- Return if the configuration for given resource is not found (it is possible when the timing protection for given resource is not configured for current task).

Used data

-»OsTPTimVal
-»OsRunning
-»OsResources

OSTPResetTaskResLockTime

{DD_235}

void OSTPResetTaskResLockTime (-»OSResType *resRef*)

Reset task resource locking time for given resource

Parameters:

[in] *resInd* The Resource Id

Returns:

none

Note:

none

Implementation

The function is called from OSTPResetResLockTime. The function performs following:

- Return if the head of running task's list of locked resources is not given resource;
- Save remained time for given resource;

- Remove resource from the list;
- Reset 'RESLOCK' channel;
- If the list is not empty, arm 'RESLOCK' timer channel for preempted locked resource with (remained time of the list head resource + saved remained time).

Used data

-»OsTPTimVal

-»OsRunning

OSTPStartISRResLockTime

{DD_236}

void OSTPStartISRResLockTime (-»OSWORD *resInd*)

Start resource locking time in a ISR.

Parameters:

[in] *resInd* The Resource Id

Returns:

none

Note:

none

Implementation

The function is called from OSTPStartResLockTime.

The function performs following:

- Find the timing protection configuration for given Resource index in the array of resource timing protection configurations belonged to current ISR.
- For found configuration:

  - Save configured resource locking time value from the configuration to temporarily variable;

  - If there is already locked resource with timing protection:

    – If remained time for the locked resource is "<=" then the saved locking time, return;

    – Else save remained time of already locked Resource as remained time minus saved locking time value;

- Re-arm 'RESLOCK' channel with saved locking time value;

- Add given resource to the list of locked resources with timing protection;

- Return;

- Return if the configuration for given resource is not found (it is possible when the timing protection for given resource is not configured for current ISR).

Used data

-»OsTPTimVal
-»OsIsrArray
-»OsResources

OSTPResetISRResLockTime

{DD_237}

void OSTPResetISRResLockTime (-»OSResType *resRef*)

Reset ISR resource locking time.

Parameters:

[in] *resInd* The Resource Id

Returns:

none

Note:

none

Implementation

The function is called from OSTPResetResLockTime.

The function performs following:

- Return if the head of current ISR's list of locked resources with timing protection is not given resource;
- Save remained time for given resource;
- Remove resource from the list;
- Reset 'RESLOCK' channel;
- If the list is not empty, arm 'RESLOCK' timer channel for preempted locked resource with (remained time of the list head resource + saved remained time).

Used data

-»OsTPTimVal

-»OsISRArray

OSTPTimerRemained

{DD_238}

OSINLINE -»OSTPTICKTYPE OSTPTimerRemained (int *ch*)

Calculates time remained before the compare event.

Parameters:

[in] *ch* The channel

Returns:

time remained before the compare event on a channel of timer

Note:

-»OsTPTimVal shall be updated before

Implementation

The function performs following:

- Calculate difference between current timer compare register value on a channel and current timer counter value;
- If there is channel interrupt request already, return 0, else return calculated difference.

Used data

none

OSTPResumeTaskBudget

{DD_239}

OSINLINE void OSTPResumeTaskBudget (void)

Resume timing protection execution budget and resource locking time when a task is started after preempting.

Returns:

none

Note:

none

Implementation

The function performs following:

- Fix current counter;
- If execution budget is configured for running task, start budget (call OSTPStartBudget) with remained time on budget;
- If there is locked resources with timing protection (running task's list of locked resources with timing protection tpResource is not zero), start resource locking time (call OSTPStartResLock) with remained time of locked resource.

Used data

-»OsRunning

OSTPStopTaskBudget

{DD_240}

OSINLINE void OSTPStopTaskBudget (void)

Stop task timing protection (the budget and/or the resource locking time) when a task is preempted.

Returns:

none

Note:

none

Implementation

The function performs following:

- If running task goes in wait state, reset task budget, reset timer 'BUDGET' channel and return. Note that here the running task' list tpResources shall be empty because all resources shall be released before -»WaitEvent();
- Else:
  - Fix current timer counter value;
  - If execution budget is configured for running task, save remained time on the task budget and reset timer 'BUDGET' channel

  - If there is locked resource with timing protection, save remained time before finish of configured resource locking time and reset 'RESLOCK' channel;
- Return.

Used data

-»OsRunning

-»OsTPTimVal

OSTPRestartTaskBudget

{DD_241}

OSINLINE void OSTPRestartTaskBudget (void)

Restart task budget for extended tasks.

Returns:

none

Note:

none

Implementation

The function performs following:

- Return if execution budget is not configured for running task;
- Else fix current counter value, reset task budget and start it.

Used data

-»OsRunning

-»OsTPTimVal

OSTPResetTaskBudget

{DD_242}

OSINLINE void OSTPResetTaskBudget (void)

Reset task timing protection (the budget and/or the resource locking time).

Returns:

none

Note:

none

Implementation

The function performs following:

- Reset task budget and reset 'BUDGET' timer channel;
- Clean the list of locked resources with timing protection and reset 'RESLOCK' timer channel.

Used data

-»OsRunning

OSTPResetReadyTask

{DD_243}

OSINLINE void OSTPResetReadyTask (-»OSTSKCBPTR *task*)

Reset task timing protection (the budget and/or the resource locking time) for task in ready state (not running).

Parameters:

[in] *task* - a pointer to the task

Returns:

none

Note:

It used to reset TP fields for the task when the task is not running and the task is killed

Implementation

The function performs following:

- Reset task budget;
- Clean the list of locked resources with timing protection.

Used data

none

OSTPStartISRBudget

{DD_244}

OSINLINE void OSTPStartISRBudget (-»OS_ISRTYPE * *isr*)

Start timing protection for ISR:

the execution budget
resource locking time.

Parameters:

[in] *isr* - a pointer to ISR

Returns:

none

Note:

none

Implementation

The function performs following

- If execution time is configured for given ISR cat.2, start execution time (via OSTPStartBudget);
- Clean given ISR's list of locked resources with timing protection.

Used data

-»OsTPTimVal

OSTPResetISRBudget

{DD_245}

OSINLINE void OSTPResetISRBudget (void)

Reset the ISR2 budget.

Returns:

none

Note:

none

Implementation

The function performs following:

- Reset 'BUDGET' timer channel;
- Reset 'RESLOCK' timer channel.

Used data

none

OSTPResumeISRBudget

{DD_246}

OSINLINE void OSTPResumeISRBudget (->OS_ISRTYPE * *isr*)

Resume the ISR2 budget and/or the resource locking time.

Parameters:

[in] *isr* - a pointer to ISR

Returns:

none

Note:

none

Implementation

The function performs following:

- If execution time is configured for given ISR cat.2, start it (via OSTPStartBudget);
- If given ISR's list of locked resources is not empty, start remained resource locking time.

Used data

none

OSTPStartIntLockTime

{DD_247}

OSINLINE void OSTPStartIntLockTime (void)

Start timing protection for OS interrupt locking time.

Returns:

none

Note:

none

Implementation

If current runnable is a task:

- If interrupt locking time is configured for running task, start interrupt locking time via OSTPStartIntLock with configured interrupt locking time value.
- Else (current runnable is ISR cat.2):
  - Get pointer to current ISR control block;
  - If interrupt locking time is configured for current ISR, start interrupt locking time via OSTPStartIntLock with configured interrupt locking time value.

Used data

-»OsRunning

-»OsIsrLevel

-»OsIsrArray

OSTPStartResLockTime

{DD_248}

OSINLINE void OSTPStartResLockTime (-»OSWORD *resInd*)

Start resource locking time in a task or ISR2.

Parameters:

[in] *resInd* The resource

Returns:

none

Note:

none

Implementation

If current runnable is a task, call OSTPStartResLockTime, else call OSTPStartISRResLockTime.

Used data

-»OsIsrLevel

OSTPResetResLockTime

{DD_249}

OSINLINE void OSTPResetResLockTime (-»OSResType *resRef*)

Reset resource locking time in a task or ISR2.

Parameters:

[in] *resRef* - a reference to the resource

Returns:

none

Note:

none

Implementation

If current runnable is a task, call OSTPResetResLockTime, else call OSTPResetISRResLockTime.

Used data

-»OsIsrLevel

Variable Documentation

- const -»OSTPISRCFG -»OsTPISRCfg[-»OSNTPISRBGTS]
- const -»OSTPLOCK -»OsTPISRIntLockTimeCfg[-»OSNTPISRINTLOCKTIME]
- const -»OSTPRESLOCKCFG -»OsTPISRResLockTimeCfg[-»OSNTPISRRESLOCKTIME]
- const -»OSTPTSKCFG -»OsTPTskCfg[-»OSNTPTSKBGTS]
- const -»OSTPLOCK -»OsTPTskIntLockTimeCfg[-»OSNTPTSKINTLOCKTIME]
- const -»OSTPRESLOCKCFG -»OsTPTskResLockTimeCfg[-»OSNTPTSKRESLOCKTIME]

Data Structures

TagOSTPCB Struct Reference

{DD_250}

```
#include <Os_tp_internal_types.h>
```

Common TP control block. It's used only as a reference typeduring the processing TP events.

Data Fields

- ->OSObjectType object

  Object type&id: TASK / ISR2 / Resource OBJECT_OS_INTERRUPT_LOCK

TagOSTPISRCfg Struct Reference

{DD_251}

```
#include <Os_tp_internal_types.h>
```

TP configuration of ISR2 execution time.

Data Fields

- ->OSObjectType object

  Object type&id: ISR2

- ->OSTPTICKTYPE execTime

  Execution time for ISR2

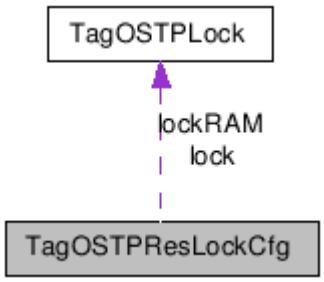
TagOSTPResLockCfg Struct Reference

{DD_252}

```
#include <Os_tp_internal_types.h>
```

TP configuration of resource locking time.

Collaboration diagram for TagOSTPResLockCfg:



Data Fields

- ->OSDWORD num

  Number of resource lock configurations

- const ->OSTPLOCK * lock

  Pointer to array of resource lock configurations

- ->OSTPLOCK * lockRAM

  Pointer to array of resource lock copies of configurations in RAM

TagOSTPTskCfg Struct Reference

{DD_253}

```
#include <Os_tp_internal_types.h>
```

TP configuration of Task execution budget

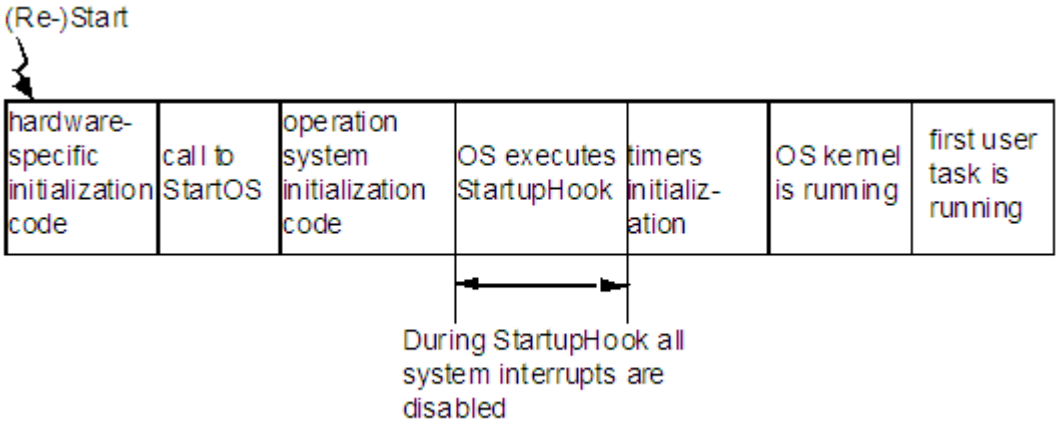
Data Fields

- -»OSObjectType object
  Object type&id: TASK
- -»OSTPTICKTYPE execBudget
  Execution budget for TASK
- -»OSTPTICKTYPE timeFrame
  Time frame for task

Set-up Routines

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

Figure 0.7 Setup in the AUTOSAR OS System Start



StartOS

{DD_254}

void StartOS (-»AppModeType *mode*)

OS data initialization from configuration tables.

Parameters:

[in] *app* - an operating mode

Returns:

only if the argument is invalid

Note:

none

Implementation

- Disable all interrupts
- IF *mode* is invalid - return
- Initialize Memory Protection
- Initialize Applications
- Initialize OS internal variables
- Initialize ISRs
- Initialize Tasks
- Fill stacks
- Initialize Resources
- Initialize Counters
- Initialize Alarms
- Initialize Schedule Tables
- Initialize Messages
- Call Startup Hooks (OS and Aplication)
- Initialize System Timers
- Initialize Timing Protection
- Start autostarted Alarms
- Start Autostarted schedule Tables
- Save MSR in OsInitialMSR
- Open by MPU stack for idle Task
- IF there is a ready Task
  - Call OSTaskForceDispatch
- go into idle loop

OS_ShutdownOS

{DD_255}

void OS_ShutdownOS (-»StatusType *error*)

Shutdown OS.

Parameters:

[in] *error* - an error code

Returns:

only if the call is not allowed

Note:

Disables all interrupts and goes into infinite loop

The special ShutdownOS service exists in AUTOSAR OS to shutdown the operating system. This service could be requested by an application or requested by the operating system due to a fatal error. When ShutdownOS service is called with a defined error code, the operating system will shut down and call the hook routine ShutdownHook. The user is free to define any system behavior in ShutdownHook e.g. not to return from the routine. If ShutdownHook returns, the operating system enters endless loop with all interrupts disabled. It is possible to restart OS by calling setjmp() before calling StartOS() and then calling longjmp() in ShutdownHook.

Implementation

- IF the function is called in not allowed context
  - Call ErrorHook and return appropriate error code
- Call OSShutdownOS

OSFillStacks

{DD_256}

OSINLINE void OSFillStacks (void)

Fill stacks with User defined pattern.

Returns:

none

Note:

none

Implementation

- Mark end of common stack
- Fill Tasks stack area
- Fill common ISR stack

OSShutdownOS

{DD_257}

void OSShutdownOS (-»StatusType *error*)

performs low-level OS shutdown.

Parameters:

[in] *error* - an error code

Returns:

never

Note:

none

Implementation

- Disable all interrupts, include critical
- Shutdown OS timers
- Call application-specific ShutdownHooks
- Call OS ShutdownHook
- Go into endless loop

GetActiveApplicationMode

{DD_299}

-»AppModeType GetActiveApplicationMode (*void*)

This service returns the current application mode.

Parameters:

none

Returns:

Current application mode

Note:

none

Implementation

- If service is called when interrupts are disabled then shall return E_OS_DISABLEDINT
- If service is called from wrong context then shall return E_OS_CALLEVEL

- Return Current application mode

OSAppStartupHooks

{DD_302}

OSINLINE void OSAppStartupHooks (*void*)

This service calls application specific startup hooks.

Parameters:

none

Returns:

none

Note:

none

Implementation

- Get the startup hook of the current application using -»OsAppTable.startup (OS data including OS application structure is not avaliable in user mode therefore startup() address is calculated before jump to user mode)
- Calculate the current stack pointer; set the same right as OS-Application has
- The application-specific startup hook with the access rights of the associated OS-Application
- Restore OS rights for osstack area
- Set OSINVALID_OSAPPLICATION

Multi-core functions

OS_ShutdownAllCores

{DD_349}

void OS_ShutdownAllCores (-»StatusType *error*)

After this service the OS on all AUTOSAR cores is shut down.

Parameters:

[in] *error* error type

Returns:

none

Note:

none

Implementation

- Return if service was called from nontrusted application
- Check context of service call
- Perform remote call for shutdown OS on another core
- Shutdown OS on current core

Used data

-»OsIsrLevel

OsRC

OS_StartCore

{DD_350}

void OS_StartCore (-»CoreIDType *coreId*, -»StatusType * *status*)

It starts the core specified by the parameter CoreID. The *status parameter allows the caller to check whether the operation was successful or not.

Parameters:

[in] *coreId* Core Id
[out] *success*

Returns:

none

Note:

If a core is started by means of this function StartOS shall be called on the core.

Implementation

- If given core ID is OS_CORE_ID_MASTER in extended status return E_OS_STATE, in standrard status return E_OK
- If given core ID is OS_CORE_ID_0 and OS is configred in multi-core configuration, in extended status return E_OS_ACCESS if service was called on second core and E_OS_STATE if second core is already started, otherwise and in standard status start second core and return E_OK

- If given core ID is OS_CORE_ID_0 and OS isn't configred in multi-core configuration, in extended status return E_OS_ACCESS, in standard status return E_OK
- If given core ID is invalid return in extended status E_OS_ID, in standard status return E_OK

Used data

None.

OS_StartNonAUTOSARCore

{DD_351}

void OS_StartNonAUTOSARCore (-»CoreIDType *coreId*, -»StatusType * *status*)

It starts the core specified by the parameter CoreID. The *status parameter allows the caller to check whether the operation was successful or not.

Parameters:

[in] *coreId* Core Id
[out] *success*

Returns:

none

Note:

it is allowed to call this function after -»StartOS()
it is not allowed to call StartOS on cores activated by this service

Implementation

- If given core ID is OS_CORE_ID_MASTER in extended status return E_OS_STATE, in standrard status return E_OK
- If given core ID is OS_CORE_ID_0 and OS is configred in multi-core configuration, start second core and return E_OK
- If given core ID is OS_CORE_ID_0 and second core is already started, in extended status return E_OS_STATE, in standard status return E_OK
- If given core ID is invalid return in extended status E_OS_ID, in standard status return E_OK

Used data

None.

OSStartedOnCore

{DD_351}

OSINLINE OSBYTE OSStartedOnCore (CoreIDType *id*)

Start core.

Parameters:

[in] id Core Id

Returns:

OSTRUE if StarttOS() have called on Z0 otherwise - OSFALSE

Note:

If a core is started by means of this function StartOS shall be called on the core.

Implementation

- If priority of software setable interrupt number 4 on main core is initialized, return OSTRUE, and return OSFALSE if otherwise
- If priority of software setable interrupt number 1 on second core is initialized, return OSTRUE, and return OSFALSE if otherwise

Variables Documentation

- const -»OS_AppType -»OsAppCfgTable[OSNAPPS]

Data Structures

OSAPP Struct Reference

`#include <Os_application_config.h>`

Data Fields

- ->OSHKSTARTUP startup
- ->OSHKSHUTDOWN shutdown
- ->OSHKERROR error
- ->OSDWORD * hookStack
- ->OSDWORD * hookStackBos
- ->OSDWORD tasks

  All tasks of the application, priority-wise
- ->OSDWORD tasks2

  Tasks of the application, priority-wise, if the tasks are more than 32
- ->OSWORD restartTask

  Index in OsTaskTable
- ->OSWORD coreId

Target-specific Functions

OS_OS2SysMode

{DD_258}

OSASM void OS_OS2SysMode (void)

Returns from ServiceDispatcher always in supervisor mode, call is allowed only from OS code.

Returns:

never

Note:

It is reduced second part of ServiceDispacher

Implementation

- If call address belongs to ostext memory area:
  - disable all interrupts
  - restore SRR0
  - restore LR
  - set user bit in SRR1
  - return from interrupt
- Else return from interrupt

OSCriticalException

{DD_259}

OSASM void OSCriticalException (void)

Handles critical exception.

Returns:

none

Note:

The processor uses CSRR0/CSRR1

Implementation

- Switch to exception stack
- Copy SRR0 to R3
- Copy SRR1 to R4

- Call OSExceprionError

OSDebugException

{DD_260}

OSASM void OSDebugException (void)

Handles debug exception.

Returns:

never

Note:

none

Implementation

- Switch to exception stack
- Load into R3 register E_OS_SYS_FATAL error code
- Call OSShutdownOS

OSDECisr

{DD_261}

OSASM void OSDECisr (void)

redirects interrupt request to INTC

Returns:

none

Note:

none

Implementation

- Save R1, R0, R3 registers on stack
- Set DIS bit in TSR register
- Invoke IRQ7 software interrupt allocated to DEC ISR
- Restore R3, R0, R1 registers
- Return form interrupt

OSExceptionError

{DD_262}

void OSExceptionError (-»OSDWORD *srr0*, -»OSDWORD *srr1*)

Called when Instruction Error occurs.

Parameters:

[in] *srr0* The memory address to check if exception occurs in OS code itself
[in] *srr1* The problem state to check if is supervisor mode

Returns:

never

Note:

none

Implementation

OSFITisr

{DD_263}

OSASM void OSFITisr (void)

Redirects interrupt request to INTC.

Returns:

none

Note:

none

Implementation

- Save R1, R0, R3 registers on stack
- Set TIS bit in TSR register
- Invoke IRQ5 software interrupt allocated for FIT ISR
- Restore R3, R0, R1 registers
- Return form interrupt

OSInterruptDispatcher

{DD_264}

OSASM void OSInterruptDispatcher (void)

Wrapper for OSInterruptDispatcher1.

Returns:

none

Note:

CPU context saving and loading

Implementation

- Save R0, R3-R12, CR, LR, CTR, XER, SRR0, SRR1 registers on stack
- Call OSInterruptDispatcher1
- Load SRR1, SRR0, XER, XTR, LR, CR, R12-R3, R0 registers
- Return form interrupt

OSLongJmp

{DD_265}

OSASM -»OSSIGNEDDWORD OSLongJmp (-»OSJMP_BUF)

Restore execution environment from buffer.

Parameters:

[in] *r3* - a pointer to environment buffer

Returns:

always 1

Note:

The following registers are restored: LR (return address), r1, CTR, r14 - r31

Implementation

- Load R1, LR, CTR, R14-R31 from buffer
- Set in R3 ’1’ return value
- Return from function

OSMachineCheckException

{DD_266}

OSASM void OSMachineCheckException (void)

Handles machine check exception.

Returns:

none

Note:

The processor uses MCSRR0/MCSRR1

Implementation

- Switch to exception stack
- Copy SRR0 to R3
- Copy SRR1 to R4
- Call OSExceprionError

OSNonCriticalException

{DD_267}

OSASM void OSNonCriticalException (void)

Handles non-critical exception.

Returns:

none

Note:

The processor uses SRR0/SRR1

Implementation

- Switch to exception stack
- Copy SRR0 to R3
- Copy SRR1 to R4
- Call OSExceprionError

OSServiceDispatcher

{DD_268}

OSASM void OSServiceDispatcher (void)

Handles system calls (sc).

Returns:

R3 - return value

Note:

R3, R4, R5 - arguments of service function R6, - number of service function

Implementation

OSServiceDispatcher

{DD_269}

OSASM void OSServiceDispatcher (void)

Handles system calls (sc).

Returns:

R3 - return value

Note:

R3, R4, R5 - arguments of service function R6, - number of service function

Implementation

- Check validity of number of service function. This value must be less or equal to the maximum allowed service number kept in OsLastServiceId constant.
- Save on stack SRR0, LR, SRR1 registers
- Enable all interrupts if they were already enabled before system call interrupt
- Calculate given service function address and call this service
- Copy EE bit value from current MSR to the value of SRR1 saved on stack
- Disable all interrupts
- Restore values of LR, SRR0, SRR1 registers from stack
- Return from interrupt

Used data

OsLastServiceId

OSSetJmp

{DD_270}

OSASM -»OSSIGNEDDWORD OSSetJmp (-»OSJMP_BUF)

Save current execution environment into buffer.

Parameters:

[in] *r3* - pointer to environment buffer

Returns:

always 0

Note:

The following registers are saved: LR (return address), r1, CTR, r14 - r31

Implementation

- Save R1, LR, CTR, R14-R31 registers into buffer
- Set in R3 zero return value
- Return from function

OSSystemCall0

{DD_271}

OSASM -»OSDWORD OSSystemCall0 (-»OSDWORD *ServiceId*)

Implementation

- Copy number of called service from R3 to R6 register
- Invoke system call interrupt
- Return from function

OSSystemCall1

{DD_272}

OSASM -»OSDWORD OSSystemCall1 (-»OSDWORD *a*, -»OSDWORD *ServiceId*)

Implementation

- Copy number of called service from R4 to R6 register
- Invoke system call interrupt
- Return from function

OSSystemCall2

{DD_273}

OSASM -»OSDWORD OSSystemCall2 (-»OSDWORD *a*, -»OSDWORD *b*, -»OSDWORD *ServiceId*)

Implementation

- Copy number of called service from R5 to R6 register
- Invoke system call interrupt
- Return from function

OSSystemCall3

{DD_274}

OSASM -»OSDWORD OSSystemCall3 (-»OSDWORD *a*, -»OSDWORD *b*, -»OSDWORD *c*, -»OSDWORD *ServiceId*)

Implementation

- Invoke system call interrupt
- Return from function

OSTLBErrorException

{DD_275}

OSASM void OSTLBErrorException (void)

Handles both code and data TLB exceptions.

Returns:

none

Note:

The processor uses SRR0/SRR1

Implementation

- Switch to exception stack
- Copy SRR0 to R3
- Copy SRR1 to R4
- Call OSExceprionError

OSTLBException

{DD_276}

void OSTLBException (-»OSDWORD *srr0*, -»OSDWORD *srr1*)

TLB exception (data or code).

Parameters:

[in] *srr0* The memory address to check if exception occurs in OS code itself
[in] *srr1* The problem state to check if it is supervisor mode

Returns:

never

Note:

none

Implementation

- If exception memory address belong to OS code or error hooks or ISR category 1 then call OSShutdownOS function with E_OS_PROTECTION_MEMORY argument
- Else call OSProtectionHandler function with E_OS_PROTECTION_MEMORY argument

File Documentation

Os_orti.c File Reference

Variables

- -»OSDWORD -»_OsOrtiStackStart = (-»OSDWORD)OS_MAIN_TOS
- -»OSDWORD -»_OsOrtiStart = (-»OSDWORD)OS_MAIN_BOS
- -»OSBYTE -»OsOrtiRunning

Os_alarm.c File Reference

- -»OSALMCB -»OsAlarms[OSNUSERALMS]
- -»OSALLALARMS -»OsAllAlarms

Os_application.c File Reference

- -»OS_AppType -»OsAppTable[OSNAPPS]

Os_counter.c File Reference

- -»OSWORD -»OsCtrIncValue[OSNCTRS-OSNHWCTRS]

  The counter increment value inspired by corresponding alarm action (only for SW counters).

  Declared if defined **OSCOUNTER** and **OSALMINCCOUNTER**

Os_isr.c File Reference

- -»OS_ISRTYPE* -»OsIsrArray[OSNIPLSP]
- -»OSJMP_BUF -»OsISRbufs[OSNIPLSP]
- -»OS_ISRTYPE -»OsIsrTable[OSNISR+1]

Os_stack.c File Reference

- -»OSDWORD -»OsISRStack[OSISRSTACKSIZE]

- ->OSDWORD ->OsStacks[(OSSTKSIZE+OSSTACKPROTECTIONLAYER)/4]
- ->OSDWORD ->OsISRStack[OSISRSTACKSIZE]

Os_mem.c File Reference

- ->OSMP_DSADDR ->OsMPDataRAM[OSNNOTTRUSTEDAPPS]
- ->OSMP_DSADDR ->OsMPDataRAM_1[OSNNOTTRUSTEDAPPS]
- ->OSMP_DSADDR ->OsMPDataRAM_2[OSNNOTTRUSTEDAPPS]

Os_task.c File Reference

- ->TaskType ->OsPrio2Task[OSNTSKS]

  References from priority to task.

  Declared if NOT defined **OSUSETCB** and defined **OSINRES**
- ->OSBYTE ->OsTaskStatus[OSNTSKS]

  Declared if **OSUSETCB** not defined
- ->OSTSKCB ->OsTaskTable

  Task control blocks table.

  Declared if defined **OSUSETCB**

Os_resource.c File Reference

- ->OSRESCB ->OsResources[OSNALLRES]

  The Resources table.

  Declared if defined **OSRESOURCE**

Os_schedule_table.c File Reference

- ->OSSCTCB ->OsScheduleTables[OSNSCTS]
- ->OSSCTALMCB ->OsSCTAlarms[OSNSCTALMS]

- -»OSBYTE -»OsTaskStatus[OSNTSKS]

  Declared if **OSUSETCB** not defined

Os_tp_v3.c File Reference

- -»OSTPRESLOCK -»OsTPISRResLockTime[-»OSNTPISRRESLOCKS]
- -»OSTPRESLOCK -»OsTPTskResLockTime[-»OSNTPTSKRESLOCKS]

{DD_284}

Os_alarm_types.h File Reference

- typedef -»AlarmBaseType* -»AlarmBaseRefType
- typedef -»OSCTR -»AlarmBaseType
- typedef -»OSObjectType -»AlarmType

Os_alarm_config.h File Reference

{DD_277}

Variables

- const -»OSALM OsAlarmsCfg[-»OSNUSERALMS]

  Alarms table
- const -»OSALMAUTOTYPE OsAutoAlarms[-»OSNAUTOALMS]

  Autostart Alarm table
- const -»OSSCT -»OsScheduleTablesCfg[OSNSCTS]
- const -»OSSCTAUTOTYPE -»OsAutoScheduleTablesCfg[OSNAUTOSCTS]
- typedef struct -»TagOSALM -»OSALM
- typedef struct -»tagALMAUTOTYPE -»OSALMAUTOTYPE
- typedef -»OSCTR* -»OSCTRPTR

Os_alarm_internal_api.h File Reference

{DD_278}

Variables

- -»OSALLALARMS -»OsAllAlarms
- -»OSSCTALMCB -»OsSCTAlarms [-»OSNSCTALMS]
- -»OSALMCB -»OsAlarms [-»OSNUSERALMS]

Os_counter_config.h File Reference

- -»OSCTRCB -»OsCounters[OSNCTRS]
  The Counters table.
- const -»OSCTR -»OsCountersCfg [OSNCTRS]
- typedef struct -»TagOSCTR -»OSCTR
- typedef -»OSCTR -»CtrInfoType

Os_counter_types.h File Reference

- typedef -»OSObjectType CounterType
  Counter ticks.

Os_counter_internal_types.h File Reference

- typedef struct -»TagOSCTRCB -»OSCTRCB
- typedef -»OSCTRCB* -»OSCTRCBPTR

Os_task_config.h File Reference

- const -»OSBYTE -»OsTaskAutostart[OSNTSKS]
- const -»OSTSK -»OsTaskCfgTable[OSNTSKS]
  Task Configuration table
- const -»OSTASKENTRY -»OsTaskEntry[OSNTSKS]
  Tasks' entries
- const -»OSBYTE -»OsTaskProperty[OSNTSKS]

Tasks' properties

- const -»OSBYTE -»OsTaskRunPrio[OSNTSKS]

  Tasks' runprio

Os_task_types.h File Reference

- typedef -»TaskType* TaskRefType
- typedef -»TaskStateType* -»TaskStateRefType
- typedef unsigned char -»TaskStateType
- typedef -»OSObjectType -»TaskType

Os_resource_config.h File Reference

{DD_281}

Variables

- const -»OSRESCFG -»OsResCfg[OSNRESS+OSNISRRESS]
- const -»OSPRIOTYPE -»OsResPrioTbl[OSNRESS+OSNISRRESS]

Os_resource_config.h File Reference

- typedef struct -»TagOSRESCFG -»OSRESCFG

Os_resource_internal_types.h File Reference

- typedef struct -»TagOSRESCB -»OSRESCB
- typedef -»OSRESCB* -»OSResType

Os_resource_types.h File Reference

- typedef -»OSObjectType -»ResourceType

Os_scheduler_internal_api.h File Reference

{DD_282}

Function Documentation

- OSINLINE void OSLOADBASICSP (void)
- OSINLINE void OSLOADCONTEXTADDR (register -»OSJMP_BUF *jmpbuf_ptr*)
- OSINLINE -»OSTASKTYPE OSMask2Task (register -»OSDWORD *scheduler_vector*, register -»OSDWORD *scheduler_vector2*)
- OSINLINE void OSSAVECONTEXTADDR (register -»OSJMP_BUF *jmpbuf_ptr*)

Os_schedule_table_types.h File Reference

- typedef -»OSObjectType -»ScheduleTableType
- typedef -»TickType -»GlobalTimeTickType
- typedef -»ScheduleTableStatusType* -»ScheduleTableStatusRefType
- typedef unsigned char -»ScheduleTableStatusType

Os_schedule_table_internal_types.h File Reference

- typedef struct -»TagOSREFALM -»OSSCTALMCB
- typedef struct -»TagOSSCTCB -»OSSCTCB

Os_schedule_table_config.h File Reference

- typedef struct -»tagSCTAUTOTYPE -»OSSCTAUTOTYPE

{DD_283}

Os_stack_internal_api.h File Reference

{DD_279}

Variables

- -»OSDWORD -»OSSTACKTOP[]
  Top of common stack
- -»OSDWORD -»OSSTACKBOTTOM[]
  Bottom of common stack

Os_setup_internal_types.h File Reference

Typedef Documentation

- typedef unsigned char AppModeType

Os_error_types.h File Reference

- typedef unsigned char OSBYTE
- typedef -»OSBYTE -»OSServiceIdType

Os_error_internal_types.h File Reference

- typedef void(* -»OSHKSTARTUP)(void)
- typedef void(* -»OSHKSHUTDOWN)(-»StatusType)
- typedef void(* -»OSHKERROR)(-»StatusType)

Os_types_basic.h File Reference

- typedef unsigned char* OSBYTEPTR
- typedef unsigned int OSDWORD
- typedef unsigned long long -»OSQWORD
- typedef signed long long -»OSSIGNEDQWORD
- typedef signed short -»OSSHORT
- typedef signed int -»OSSIGNEDDWORD

Os_types_common_public.h File Reference

- typedef unsigned char StatusType
- typedef -»OSWORD -»OSObjectType
- typedef unsigned short OSWORD
- typedef -»OSBYTE ApplicationType
- typedef -»OSDWORD EventMaskType
- typedef -»OSDWORD TickType

  Type for timers ticks
- typedef -»TickType* -»TickRefType

  OSEK: Reference to counter value
- typedef -»CtrInfoType* -»CtrInfoRefType
- typedef -»OSObjectType -»CounterType

  Counter ticks
- typedef -»OSObjectType -»ISRType
- typedef -»EventMaskType* -»EventMaskRefType
- typedef -»OSBYTE ApplicationType

Os_types_common_internal.h File Reference

- typedef struct -»TagOSALMACT -»OSALMACT
- typedef struct -»TagOSALMCB -»OSALMCB
- typedef struct -»TagOSREFALM* -»OSAlmType
- typedef -»OSDWORD -»OSISRLEVELTYPE
- typedef -»OSDWORD OSJMP_BUF[OSJBLEN]
- typedef signed char -»OSPRIOTYPE
- typedef -»OSTASKTYPE(* OSTASKENTRY)(void)
- typedef -»OSDWORD -»OSTASKTYPE
- typedef -»OSDWORD -»OSTPTICKTYPE
- typedef -»OSDWORD -»OSINTMASKTYPE
- typedef void(* -»OSVOIDFUNCVOID)(void)
- typedef void(* -»OSCALLBACK)(void)
- typedef struct -»TagOSTSKCB -»OSTSKCB

- typedef struct -»TagOSTSK -»OSTSK

Enumeration Type Documentation

enum -»OSISRTYPE

Enumerator:

- *OSNONTRUSTEDISR2*
- *OSTRUSTEDISR2*
- *OSISR1*
- *OSSYSINTERRUPT*
- *OSTPISR1*

Os_multicore_types.h File Reference

- typedef -»OSObjectType -»SpinlockIDType
- typedef -»OSDWORD -»CoreIDType
- typedef -»OSWORD -»TryToGetSpinlockType

Os_multicore_internal_types.h File Reference

- typedef struct -»OSTag_RemoteCallCB -»OSREMOTECALLCB

Os_tp_internal_types.h File Reference

- typedef struct -»TagOSTPAlm -»OSTPALMCB
- typedef struct -»TagOSTPCB -»OSTPCB
- typedef struct -»TagOSTPISRCfg -»OSTPISRCFG
- typedef struct -»TagOSTPResLockCfg -»OSTPRESLOCKCFG
- typedef struct -»TagOSTPTskCfg -»OSTPTSKCFG

Os_tp_internal_types_v3.h File Reference

- typedef struct -»TagOSTPResLock -»OSTPRESLOCK

Os_isr_config.h File Reference

Variables

- const -»OSSHORT -»OsIsr[OSNINTC]
- const -»OSISRCFGTYPE -»OsIsrCfg[OSNISR+1]
- typedef struct -»tagISRCFGTYPE -»OSISRCFGTYPE

Os_isr_internal_api.h File Reference

{DD_280}

Hook definitions

- void PreIsrHook(void);
- void PostIsrHook(void);

Os_memory_types.h File Reference

- typedef -»OSBYTE -»AccessType
- typedef void* -»MemoryStartAddressType
- typedef -»OSDWORD -»MemorySizeType
- typedef void* -»TrustedFunctionParameterRefType
- typedef -»OSWORD -»TrustedFunctionIndexType
- typedef -»OSBYTE -»ObjectAccessType
- typedef -»OSBYTE -»ObjectTypeType

enum -»RestartType

Enumerator:

- *NO_RESTART*
- *RESTART*

Os_memory_config.h File Reference

- typedef void(* -»OS_TRUSTED_FUNCTION)(-»TrustedFunctionIndexType, -»TrustedFunctionParameterRefType)

Os_memory_internal_types.h File Reference

- typedef void(* -»OSSRV)(void)

Os_hook_api.h File Reference

- typedef unsigned char -»ProtectionReturnType

Os_msg_api.h File Reference

- typedef void* -»ApplicationDataRef
- typedef unsigned char -»FlagType

Os_platform_timers.h File Reference

{DD_285}

- typedef -»OSDWORD -»OSHWTickType

Os_multicore_internal_api.h File Reference

Enumeration Type Documentation

Enumerator:

- ***OsRCIdTerminateApplication***
- ***OsRCIdActivateTask***
- ***OsRCIdNotifyAlarmAction***
- ***OsRCIdChainTask***
- ***OsRCIdGetTaskState***
- ***OsRCIdSetEvent***
- ***OsRCIdGetAlarm***
- ***OsRCIdSetRelAlarm***
- ***OsRCIdSetAbsAlarm***
- ***OsRCIdCancelAlarm***
- ***OsRCIdStartScheduleTableRel***
- ***OsRCIdStartScheduleTableAbs***
- ***OsRCIdStopScheduleTable***
- ***OsRCIdGetCounterValue***
- ***OsRCIdGetElapsedCounterValue***
- ***OsRCIdShutdownAllCores***
- ***OsRCIdIocAction***