

embOS

Real-Time Operating System User Guide & Reference Manual for embOS-Base and embOS-MPU

Document: UM01001
Software Version: 5.20.0
Revision: 0
Date: November 14, 2024



A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 1995-2024 SEGGER Microcontroller GmbH, Monheim am Rhein / Germany

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel. +49 2173-99312-0
Fax. +49 2173-99312-28
E-mail: support@segger.com*
Internet: www.segger.com

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at <https://www.segger.com/legal/privacy-policy/>.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: November 14, 2024

Software	Revision	Date	By	Description
5.20.0	0	241114	TS	Update to latest software version.
5.18.3	0	230928	MC	Update to latest software version.
5.18.2	0	230731	MC	Update to latest software version.
5.18.1	0	230403	TS	Update to latest software version.
5.18.0	1	221014	TS	Chapter "Introduction and Basic Concepts" and "Shipment" updated. Minor spelling & wording corrections.
5.18.0	0	220915	TS	Update to latest software version. Chapter "Introduction and Basic Concepts" and others improved. New chapter "Kernel" added. New API functions added: <ul style="list-style-type: none"> • OS_TASK_GetStatus() • OS_DEBUG_GetError() Minor spelling & wording corrections.
5.16.2	0	220330	TS	Update to latest software version. Minor spelling & wording corrections.
5.16.1	0	220112	TS	Update to latest software version. Minor spelling & wording corrections.
5.16.0	0	211201	TS	Update to latest software version. New API functions OS_TIME_ConvertCycles2ms(), OS_TIME_Convertms2Cycles() and OS_MAILBOX_IsInUse() added. Chapter "Performance and Resource Usage" updated. Minor spelling & wording corrections.
5.14.0	0	210504	TS	Update to latest software version. New API function OS_INT_PreserveAndDisable() in chapter "Interrupts" added. Chapter "Readers-Writer Lock" added. Minor spelling & wording corrections.
5.12.0	0	200929	TS	Update to latest software version. New function OS_EVENT_ResetMask() in chapter "Event Objects" added. Minor spelling & wording corrections.
5.10.2	0	200706	TS	Update to latest software version.
5.10.1	0	200617	MM	Update to latest software version. Tickless support chapter updated.
5.10.0	0	200519	TS	Update to latest software version. Additional software examples added. Minor spelling & wording corrections.
5.8.2	1	200203	TS	Minor spelling & wording corrections.
5.8.2	0	191217	MC	Update to latest software version.
5.8.1	0	191111	TS	Event object description improved.
5.8.0	0	191028	TS	Description of new API functions OS_TIME_Convertms2Ticks() and OS_TIME_ConvertTicks2ms() added.
5.06	2	190619	TS	Minor spelling & wording corrections.
5.06	1	190401	TS	Minor spelling & wording corrections.
5.06	0	190219	TS	New API functions added. Minor spelling & wording corrections.
5.04	0	180913	TS	New API function OS_MUTEX_IsMutex(). Minor spelling & wording corrections.
5.02	0	180626	TS	New API function OS_STAT_AddLoadMeasurementEx(). Minor spelling & wording corrections.
5.00	1	180524	TS	OS_TASK_Delay() parameter description corrected. Timeout parameter description added where necessary.

Software	Revision	Date	By	Description
5.00	0	180508	TS	New API names. Chapter "Debugging" updated. Minor spelling & wording corrections.
4.40	0	171220	MC	Introductory description in chapter "Software timers" expanded. Description of limitations in chapter "Mailboxes" corrected. Description of limitations in chapter "Queues" added. Description of embOS trial edition in chapter "Shipment" updated. Description of OS_WD_Config() updated for change in parameters. List of error codes in chapter "Debugging" updated. Minor spelling & wording corrections.
4.38	1	170928	MC	Minor spelling & wording corrections.
4.38	0	170919	TS	First version generated with emDoc. New function in chapter "Tasks" added: <ul style="list-style-type: none"> OS_SetDefaultTaskStartHook() New functions in chapter "Debugging" added: <ul style="list-style-type: none"> OS_SetObjName() OS_GetObjName() Minor corrections/updates.
4.36	0	170711	TS	New library mode OS_LIBMODE_SAFE added in chapter "Basic Concepts". New functions in chapter "Stacks" added: <ul style="list-style-type: none"> OS_GetStackCheckLimit() OS_SetStackCheckLimit() New functions in chapter "MPU" added: <ul style="list-style-type: none"> OS_MPU_AddSanityCheckBuffer() OS_MPU_SanityCheck() Chapter "Source Code" updated. New functions in chapter "Task Routines" added: <ul style="list-style-type: none"> OS_Config_Stop() OS_Stop() Minor corrections/updates
4.34	0	170308	TS	New functions in chapter "Event Objects" added: <ul style="list-style-type: none"> OS_EVENT_GetMaskMode() OS_EVENT_SetMaskMode()
4.32	0	170105	RH/TS	Chapter "Watchdog" added. New functions in chapter "Event Objects" added: <ul style="list-style-type: none"> OS_EVENT_GetMask() OS_EVENT_SetMask() OS_EVENT_WaitMask() OS_EVENT_WaitMaskTimed() New functions in chapter "Mailboxes" added: <ul style="list-style-type: none"> OS_PutMailTimed() OS_PutMailTimed1()
4.30	0	161130	MC/TS	Chapter "Basic Concepts", "Time Measurement", "MPU", "Profiling" and "Updates" updated. Chapters, "System Tick", "Low Power Support", "Configuration (BSP)" updated and re-structured. Chapter "Resource Semaphores" updated.
4.26	0	160907	RH	Chapter "embOSView", "Interrupts" and "MPU" updated. Minor corrections/updates.
4.24	0	160628	MC	Chapter "Multi-core Support" added. Chapter "Debugging" updated.
4.22	0	160525	MC	New functions in chapter "Queues" added: <ul style="list-style-type: none"> OS_Q_PutEx() OS_Q_PutBlockedEx() OS_Q_PutTimedEx()
4.20	0	160421	TS	Chapter "MPU - Memory Protection" added. OS_AddExtendTaskContext() added.
4.16	0	160210	TS	Minor corrections/updates
4.14a	0	160115	TS	Minor corrections/updates
4.14	0	151029	TS	Chapter "Interrupts" updated. Description of new API function OS_SetDefaultTaskContextExtension() added. Chapter "System Variables": embOS info routines added. Chapter "Shipment" updated. Chapter "Low Power Support" updated. Chapter "Interrupts": Description of

Software	Revision	Date	By	Description
				<ul style="list-style-type: none"> • OS_INT_PRIO_PRESERVE() and • OS_INT_PRIO_RESTORE() added. Chapter "Software Timerss": Description of <ul style="list-style-type: none"> • OS_TriggerTimer() and • OS_TriggerTimerEx() added.
4.12b	0	150922	TS	Update to latest software version.
4.12a	0	150916	TS	Description of API function OS_InInterrupt() added.
4.12	0	150715	TS	New funtions in chapter "Mailboxes" added: <ul style="list-style-type: none"> • OS_Mail_GetPtr() • OS_Mail_getPtrCond() • OS_Mail_Purge() Chapter "Debugging" with new error codes updated.
4.10b	1	150703	MC	Minor spelling and wording corrections.
4.10b	0	150527	TS	Minor spelling and wording corrections. Chapter "Source Code of Kernel and Library" updated. New chapter "embOS Shipment". New chapter "Update". New chapter "Low Power Support".
4.10a	0	150519	MC	Minow spelling and wording corrections. Chapter "embOSView": added JTAG Chain configuration.
4.10	0	150430	TS	Chapter "embOSView" updated.
4.06b	0	150324	MC	Minow spelling and wording corrections.
4.06a	0	150318	MC	Minow spelling and wording corrections.
4.06	0	150312	TS	Updated to latest software version.
4.04a	0	141201	TS	Updated to latest software version.
4.04	0	141112	TS	Chapter "Tasks" <ul style="list-style-type: none"> • Task priority description updated. Chapter "Debugging" <ul style="list-style-type: none"> • New error number
4.02a	0	140918	TS	Update to latest software version. Minor corrections.
4.02	0	140818	TS	New functions in chapter "Time Measurement" added: <ul style="list-style-type: none"> • OS_Config_SystemTimer() • OS_GetTime_us() • OS_GetTime_us64()
4.00a	0	140723	TS	New functions added in chapter "System Tick": <ul style="list-style-type: none"> • OS_StopTicklesMode() New functions added in chapter "Profiling": <ul style="list-style-type: none"> • OS_STAT_Start() • OS_STAT_Stop() • OS_STAT_GetTaskExecTime()
4.00	0	140606	TS	Tickless support added.
3.90a	0	140410	AW	Software-Update, OS_TerminateTask() modified / corrected.
3.90	1	140312	SC	Added cross-references to the API-lists.
3.90	0	140303	AW	New functions to globally enable / disable Interrupts: <ul style="list-style-type: none"> • OS_INTERRUPT_MaskGlobal() • OS_INTERRUPT_UnmaskGlobal() • OS_INTERRUPT_PreserveGlobal() • OS_INTERRUPT_RestoreGlobal() • OS_INTERRUPT_PreserveAndMaskGlobal()
3.88h	0	131220	AW	New functions added, chapter "System Tick": <ul style="list-style-type: none"> • OS_GetNumIdleTicks() • OS_AdjustTime() Chapter "System Variables": Description of internal variable OS_Global.TimeDex corrected.
3.88g	1	131104	TS	Corrections.
3.88g	0	131030	TS	Update to latest software version.
3.88f	0	130922	TS	Update to latest software version.

Software	Revision	Date	By	Description
3.88e	0	130906	TS	Update to latest software version.
3.88d	0	130904	AW	Update to latest software version.
3.88c	0	130808	TS	Update to latest software version.
3.88b	0	130528	TS	Update to latest software version.
3.88a	0	130503	AW	Software update. Event handling modified, the reset behaviour of events can be controlled. New functions added, chapter "Events": <ul style="list-style-type: none"> • OS_EVENT_CreateEx() • OS_EVENT_SetResetMode() • OS_EVENT_GetResetMode() Mailboxes message size limits enlarged.
3.88	0	130219	TS	Minor corrections.
3.86n	0	121210	AW/TS	Update to latest software version.
3.86l	0	121122	AW	Software update. OS_AddTickHook() function corrected. Several functions modified to allow most of MISRA rule checks.
3.86k	0	121004	TS	Chapter "Queue": <ul style="list-style-type: none"> • OS_Q_GetMessageSize() and • OS_Q_PeekPtr() added.
3.86i	0	120926	TS	Update to latest software version.
3.86h	0	120906	AW	Software update, OS_EVENT handling with timeout corrected.
3.86g	0	120806	AW	Software update, OS_RetriggerTimer() corrected. Task events explained more in detail. Additional software examples in the manual.
3.86f	0	120723	AW	Task event modified, default set to 32 bit on 32-bit CPUs. Chapter 4: <ul style="list-style-type: none"> • New API function OS_AddOnTerminateHook() • OS_ERR_TIMESLICE removed. A time slice value of zero is legal when creating tasks.
3.86e	0	120529	AW	Update to latest software version with corrected functions: <ul style="list-style-type: none"> • OS_GetSysStackBase() • OS_GetSysStackSize() • OS_GetSysStackSpace() • OS_GetSysStackUsed() • OS_GetIntStackBase() • OS_GetIntStackSize() • OS_GetIntStackSpace() • OS_GetIntStackUsed() could not be used in release builds of embOS. Manual corrections: <ul style="list-style-type: none"> • Several index entries corrected. • OS_EnterRegion() described more in detail.
3.86d	0	120510	TS	Update to latest software version.
3.86c	0	120508	TS	Update to latest software version.
3.86b	0	120502	TS	Chapter "Mailbox" <ul style="list-style-type: none"> • OS_PeekMail() added. Chapter "Support" added. Chapter "Debugging": <ul style="list-style-type: none"> • Application defined error codes added.
3.86	0	120323	AW	Timeout handling for waitable objects modified. A timeout will be returned from the waiting function, when the object was not available during the timeout time. Previous implementation of timeout functions might have returned a signaled state when the object was signaled after the timeout when the calling task was blocked for a longer period by higher prioritized tasks Modified functions: <ul style="list-style-type: none"> • OS_UseTimed() • OS_WaitCSemaTimed() • OS_GetMailTimed() • OS_WaitMailTimed() • OS_Q_GetPtrTimed()

Software	Revision	Date	By	Description
				<ul style="list-style-type: none"> • OS_EVENT_WaitTimed() • OS_MEMF_AllocTimed() <p>New chapter "Extending the Task Context" added. New functions added and described in the manual:</p> <ul style="list-style-type: none"> • OS_GetTaskName() • OS_GetTimeSliceRem() <p>Handling of queues described more in detail:</p> <ul style="list-style-type: none"> • OS_Q_GetPtr() • OS_Q_GetPtrCond() • OS_Q_GetPtrTimed() • OS_Q_Purge() <p>Chapter "Priority Inversion / Inheritance" updated. Function names OS_Timing_Start() and OS_Timing_End() corrected in the API table.</p>
3.84c	1	120130	AW/TS	<p>Since version 3.822 of embOS, all pointer parameter pointing to objects which were not modified by the function were declared as const, but the manual was not updated accordingly. The prototype descriptions of the following API functions are corrected now:</p> <ul style="list-style-type: none"> • OS_GetTimerValue() • OS_GetTimerStatus() • OS_GetTimerPeriod() • OS_GetSemaValue() • OS_GetResourceOwner() • OS_Q_IsInUse() • OS_Q_GetMessageCnt() • OS_IsTask() • OS_GetEventsOccured() • OS_GetCSemaValue() • OS_TICK_RemoveHook() • OS_MEMF_IsInPool() • OS_MEMF_GetMaxUsed() • OS_MEMF_GetNumBlocks() • OS_MEMF_GetBlockSize() • OS_GetSuspendCnt() • OS_GetPriority() • OS_EVENT_Get() • OS_Timing_Getus() <p>Chapter "Preface":</p> <ul style="list-style-type: none"> • Segger Logo replaced <p>Chapter "Mailbox":</p> <ul style="list-style-type: none"> • OS_CREATEMB() changed to OS_CreateMB() <p>Chapter "Queues":</p> <ul style="list-style-type: none"> • Typos corrected
3.84c	0	120104	TS	<p>Chapter "Events":</p> <ul style="list-style-type: none"> • Return value of OS_EVENT_WaitTimed() explained in more detail
3.84b	0	111221	TS	<p>Chapter "Queues":</p> <ul style="list-style-type: none"> • OS_Q_PutBlocked() added
3.84a	0	111207	TS	General updates and corrections.
3.84	0	110927	TS	<p>Chapter "Stacks":</p> <ul style="list-style-type: none"> • OS_GetSysStackBase() added • OS_GetSysStackSize() added • OS_GetSysStackUsed() added • OS_GetSysStackSpace() added • OS_GetIntStackBase() added • OS_GetIntStackSize() added • OS_GetIntStackUsed() added • OS_GetIntStackSpace() added
3.82x	0	110829	TS	<p>Chapter "Debugging":</p> <ul style="list-style-type: none"> • New error code "OS_ERR_REGIONCNT" added
3.82w	0	110812	TS	New embOS generic sources. Chapter "Debugging" updated.
3.82v	0	110715	AW	OS_Terminate() renamed to OS_TerminateTask().
3.82u	0	110630	TS	New embOS generic sources. Chapter 13: Fixed size memory pools modified.
3.82t	0	110503	TS	New embOS generic sources. Trial time limitation increased.

Software	Revision	Date	By	Description
3.82s	0	110318	AW	Chapter "Timer" API functions table corrected. All functions can be called from main(), task, ISR or Timer. Chapter 6: OS_UseTimed() added. Chapter 9: OS_Q_IsInUse() added.
3.82p	0	110112	AW	Chapter "Mailboxes": <ul style="list-style-type: none"> OS_PutMail() OS_PutMailCond() OS_PutMailFront() OS_PutMailFrontCond() parameter declaration changed. Chapter 4.3 API functions table corrected. OS_Suspend() cannot be called from ISR or Timer.
3.82o	0	110104	AW	Chapter "Mailboxes": <ul style="list-style-type: none"> OS_WaitMailTimed() added
3.82n	0	101206	AW	Chapter "Taskroutines": <ul style="list-style-type: none"> OS_ResumeAllSuspendedTasks() added OS_SetInitialSuspendCnt() added OS_SuspendAllTasks() added Chapter "Time Measurement": <ul style="list-style-type: none"> Description of OS_GetTime32() corrected Chapter "List of Error Codes": <ul style="list-style-type: none"> New error codes added
3.82k	0	100927	TS	Chapter "Taskroutines": <ul style="list-style-type: none"> OS_Delayus() added OS_Q_Delete() added
3.82i	0	100917	TS	General updates and corrections
3.82h	0	100621	AW	Chapter "Event Objects": <ul style="list-style-type: none"> Samples added Chapter "Configuration of Target System": <ul style="list-style-type: none"> Detailed description of OS_Idle() added
3.82f	1	100505	TS	Chapter "Profiling" added Chapter "System Tick": <ul style="list-style-type: none"> OS_TickHandleNoHook() added
3.82f	0	100419	AW	Chapter "Tasks": <ul style="list-style-type: none"> OS_IsRunning() added Description of OS_Start() added
3.82e	0	100309	TS	Chapter "Working with embOS - Recommendations" added. Chapter "Basics": <ul style="list-style-type: none"> Priority inversion image added Chapter "Interrupt": <ul style="list-style-type: none"> subchapter "Using OS functions from high priority interrupts" added Added text at chapter 22 "Performance and resource usage"
3.82	0	090922	TS	API function overview now contains information about allowed context of cution usage (main, task, ISR or timer) TOC format corrected
3.80	0	090612	AW	Scheduler optimized for higher task switching speed.
3.62c	0	080903	SK	Chapter structure updated. Chapter "Interrupts": <ul style="list-style-type: none"> OS_LeaveNestableInterruptNoSwitch() removed OS_LeaveInterruptNoSwitch() removed Chapter "System Tick": <ul style="list-style-type: none"> OS_TICK_Config() added
3.60	2	080722	SK	Contact address updated.
3.60	1	080617	SK	General updates. Chapter "Mailboxes": <ul style="list-style-type: none"> OS_GetMailCond() / OS_GetMailCond1() corrected
3.60	0	080117	OO	General updates. Chapter "System Tick" added.
3.52	1	071026	AW	Chapter "Task Routines": <ul style="list-style-type: none"> OS_SetTaskName() added
3.52	0	070824	OO	Chapter "Task Routines": <ul style="list-style-type: none"> OS_ExtendTaskContext() added Chapter "Interrupts": <ul style="list-style-type: none"> Updated

Software	Revision	Date	By	Description
				<ul style="list-style-type: none"> • OS_CallISR() added • OS_CallNestableISR() added
3.50c	0	070814	AW	Chapter "List of Libraries" updated, XR library type added.
3.40c	3	070716	OO	Chapter "Performance and Resource Usage" updated.
3.40c	2	070625	SK	Chapter "Debugging", error codes updated: <ul style="list-style-type: none"> • OS_ERR_ISR_INDEX added • OS_ERR_ISR_VECTOR added • OS_ERR_RESOURCE_OWNER added • OS_ERR_CSEMA_OVERFLOW added Chapter "Task Routines": <ul style="list-style-type: none"> • OS_Yield() added Chapter "Counting Semaphores" updated <ul style="list-style-type: none"> • OS_SignalCSema(), additional information adjusted Chapter "Performance and Resource Usage" updated: <ul style="list-style-type: none"> • Minor changes in wording.
3.40a	1	070608	SK	Chapter "Counting Semaphores" updated: <ul style="list-style-type: none"> • OS_SetCSemaValue() added • OS_CreateCSema(): Data type of parameter InitValue changed from unsigned char to unsigned int • OS_SignalCSemaMax(): Data type of parameter MaxValue changed from unsigned char to unsigned int • OS_SignalCSema(): Additional information updated
3.40	0	070516	SK	Chapter "Performance and Resource Usage" added. Chapter "Configuration of your Target System (RTOSInit.c)" renamed to "Configuration of your Target System". Chapter "STOP/WAIT/IDLE modes" moved into chapter "Configuration of your Target System". Chapter "Time-related Routines" renames to "Time Measurement".
3.32o	9	070422	SK	Chapter 4: OS_CREATETIMER_EX(), additional information corrected.
3.32m	8	070402	AW	Chapter 4: Extended timer added. Chapter 8: API overview corrected, OS_Q_GetMessageCount()
3.32j	7	070216	AW	Chapter 6: OS_CSemaRequest() function added.
3.32e	6	061220	SK	About: Company description added. Some minor formatting changes.
3.32e	5	061107	AW	Chapter 7: OS_GetMessageCnt() return value corrected to unsigned int.
3.32d	4	061106	AW	Chapter 8: OS_Q_GetPtrTimed() function added.
3.32a	3	061012	AW	Chapter 3: OS_CreateTaskEx() function, description of parameter pContext corrected. Chapter 3: OS_CreateTaskEx() function, type of parameter TimeSlice corrected. Chapter 3: OS_CreateTask() function, type of parameter TimeSlice corrected. Chapter 9: OS_GetEventOccured() renamed to OS_GetEventsOccured(). Chapter 10: OS_EVENT_WaitTimed() added.
3.32a	2	060804	AW	Chapter 3: OS_CREATETASK_EX() function added. Chapter 3: OS_CreateTaskEx() function added.
3.32	1	060717	OO	Event objects introduced. Chapter 10 inserted which describes event objects. Previous chapter "Events" renamed to "Task Events".
3.30	1	060519	OO	New software version.
3.28	5	060223	OO	All chapters: Added API tables. Some minor changes.
3.28	4	051109	AW	Chapter 7: OS_SignalCSemaMax() function added. Chapter 14: Explanation of interrupt latencies and high / low priorities added.
3.28	3	050926	AW	Chapter 6: OS_DeleteRSema() function added.
3.28	2	050707	AW	Chapter 4: OS_GetSuspendCnt() function added.
3.28	1	050425	AW	Version number changed to 3.28 to fit to current embOS version. Chapter 18.1.2: Type return value of OS_GetTime32() corrected.

Software	Revision	Date	By	Description
3.26	0	050209	AW	Chapter 4: OS_Terminate() modified due to new features of version 2.26. Chapter 24: Source code version: additional compile time switches and build process of libraries explained more in detail.
3.24	0	011115	AW	Chapter 6: Some prototype declarations showed in OS_SEMA instead of OS_RSEMA. Corrected.
3.22	1	040816	AW	Chapter 8: New Mailbox functions added <ul style="list-style-type: none"> • OS_PutMailFront() • OS_PutMailFront1() • OS_PutMailFrontCond() • OS_PutMailFrontCond1()
3.20	5	040621	RS/AW	Software timers: Maximum timeout values and OS_TIMER_MAX_TIME described. Chapter 14: Description of rules for interrupt handlers revised. OS_LeaveNestableInterruptNoSwitch() added which was not described before.
3.20	4	040329	AW	OS_CreateCSema() prototype declaration corrected. Return type is void. OS_Q_GetMessageCnt() prototype declaration corrected. OS_Q_Clear() function description added. OS_MEMF_FreeBlock() prototype declaration corrected.
3.20	2	031128	AW	OS_CREATEEMB() Range for parameter MaxnofMsg corrected. Upper limit is 65535, but was declared 65536 in previous manuals.
3.20	1	040831	AW	Code samples modified: Task stacks defined as array of int, because most CPUs require alignment of stakc on integer aligned addresses.
3.20	1	031016	AW	Chapter 4: Type of task priority parameter corrected to unsigned char. Chapter 4: OS_DelayUntil(): Sample program modified. Chapter 4: OS_Suspend() added. Chapter 4: OS_Resume() added. Chapter 5: OS_GetTimerValue(): Range of return value corrected. Chapter 6: Sample program for usage of resource semaphores modified. Chapter 6: OS_GetResourceOwner(): Type of return value corrected. Chapter 8: OS_CREATEEMB(): Types and valid range of parameter corrected. Chapter 8: OS_WaitMail() added Chapter 10: OS_WaitEventTimed(): Range of timeout value specified.
3.12	1	021015	AW	Chapter 8: OS_GetMailTimed() added Chapter 11 (Heap type memory management) inserted Chapter 12 (Fixed block size memory pools) inserted
3.10	3	020926 020924 020910	KG	Index and glossary revised. Section 16.3 (Example) added to Chapter 16 (Time-related Routines). Revised for language/grammar. Version control table added. Screenshots added: superloop, cooperative/preemptive multitasking, nested interrupts, low-res nad hi-res measurement. Section 1.3 (Typographic conventions) changed to table. Section 3.2 added (Single-task system). Section 3.8 merged with section 3.9 (How the OS gains control). Chapter 4 (Configuration for your target system) moved to after Chapter 15 (System variables) Chapter 16 (Time-related routines) added.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
Keyword	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
Parameter	Parameters in API functions.
Sample	Sample code in program examples.
Sample comment	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

1	Introduction and Basic Concepts	17
1.1	What is embOS?	18
1.2	embOS editions	19
1.3	embOS ports	20
1.4	Singletasking systems (superloop)	22
1.5	Multitasking systems	24
1.6	Threads vs. Processes	26
1.7	Scheduling	27
1.8	Polling vs. Event based programming	32
1.9	Synchronization and communication primitives	33
1.10	How the OS gains control	34
1.11	Valid context for embOS API	35
1.12	Blocking and Non blocking embOS API	36
1.13	embOS API with timeout	37
1.14	RTOS objects	41
1.15	embOS types	43
1.16	Callback / Hook routines	44
1.17	embOS library modes	45
2	Kernel	48
2.1	Introduction	49
2.2	API functions	50
3	Task	60
3.1	Introduction	61
3.2	Cooperative vs. preemptive task switches	62
3.3	Extending the task context	63
3.4	API functions	65
4	Software Timer	110
4.1	Introduction	111
4.2	API functions	114
5	Task Event	139
5.1	Introduction	140
5.2	API functions	141

6	Event Object	155
6.1	Introduction	156
6.2	API functions	159
7	Mutex	181
7.1	Introduction	182
7.2	API functions	185
8	Semaphore	199
8.1	Introduction	200
8.2	API functions	202
9	Readers-Writer Lock	213
9.1	Introduction	214
9.2	API functions	215
10	Mailbox	227
10.1	Introduction	228
10.2	API functions	231
11	Queue	265
11.1	Introduction	266
11.2	API functions	268
12	Multi Object Wait	294
12.1	Introduction	295
12.2	Condition routines	297
12.3	API functions	305
13	Watchdog	311
13.1	Introduction	312
13.2	API functions	313
14	Multi-core Support	320
14.1	Introduction	321
14.2	API functions	323
15	Interrupts	331
15.1	What are interrupts?	332
15.2	Interrupt latency	333
15.3	Rules for interrupt handlers	338
15.4	Interrupt control	350
16	Critical Region	367
16.1	Introduction	368
16.2	API functions	369
16.3	Disabling context transitions	372
17	Time Measurement	374
17.1	Introduction	375
17.2	Low-resolution measurement	377
17.3	High-resolution measurement	383

18	Low Power Support	401
18.1	Introduction	402
18.2	Starting power save modes in OS_Idle()	402
18.3	Tickless support	403
18.4	Peripheral power control	415
19	Heap Type Memory Management	421
19.1	Introduction	422
19.2	API functions	423
20	Fixed Block Size Memory Pool	427
20.1	Introduction	428
20.2	API functions	430
21	System Tick	444
21.1	Introduction	445
21.1.1	Hooking into the system tick	445
21.2	API functions	446
22	Debugging	454
22.1	Runtime application errors	455
22.2	Human readable object identifiers	464
22.3	embOS API trace	469
23	Profiling	473
23.1	Task execution	474
23.1.1	API functions	474
23.2	Task specific CPU load measurement	477
23.2.1	API functions	479
23.3	CPU load measurement	485
23.3.1	API functions	486
24	embOSView	491
24.1	Introduction	492
24.2	Setup embOSView for communication	494
24.3	Setup target for communication	498
24.4	Sharing the SIO for terminal I/O	505
24.5	embOSView API trace	508
25	MPU - Memory Protection	528
25.1	Introduction	529
25.2	Memory Access permissions	532
25.3	ROM placement of embOS	533
25.4	Allowed embOS API in unprivileged tasks	534
25.5	Device driver	535
25.6	API functions	537
26	Stacks	556
26.1	Introduction	557
26.2	API functions	559
27	Board Support Packages	574
27.1	Introduction	575
27.2	How to create a new board support package	575
27.3	Example	576

27.4	Mandatory routines	577
27.5	Optional routines	581
27.6	Settings	584
27.7	UART routines for embOSView	585
28	System Variables	586
28.1	Introduction	587
28.2	OS_Global	588
28.3	OS information routines	589
28.3.1	API functions	589
29	Source Code	596
29.1	Introduction	597
29.2	Building embOS libraries	598
29.3	Compile time switches	599
29.4	Source code project	601
29.4.1	Compiler options	601
30	Shipment	602
30.1	Introduction	603
30.2	Object code package	604
30.3	Source code package	605
31	Update	606
31.1	Introduction	607
31.2	How to update an existing project	608
31.3	embOS migration guide	609
32	Support	616
32.1	Contacting support	617
32.1.1	Where can I find the license number?	617
33	Performance and Resource Usage	618
33.1	Introduction	619
33.2	Resource Usage	619
33.3	Performance	620
34	Supported Development Tools	627
34.1	Reentrance	628
34.2	Compiler version	629
34.3	Compiler options	629
34.4	C/C++ standard	630
35	Glossary	631

Chapter 1

Introduction and Basic Concepts

1.1 What is embOS?

embOS is a priority-controlled multitasking system, designed to be used as an embedded operating system for the development of real-time applications for a variety of microcontrollers.

embOS is a high-performance tool that has been optimized for minimal memory consumption in both RAM and ROM, as well as high speed and versatility. Throughout the development process of embOS, the limited resources of microcontrollers have always been kept in mind. The internal structure of the real-time operating system (RTOS) has been optimized in a variety of applications with different customers, to fit the needs of industry. Fully source-compatible implementations of embOS are available for a variety of microcontrollers, making it well worth the time and effort to learn how to structure real-time programs with an RTOS.

embOS is highly modular. This means that only those functions that are required are linked into an application, keeping the ROM size very small. A couple of files are supplied in source code to make sure that you do not lose any flexibility by using embOS libraries and that you can customize the system to fully fit your needs.

The tasks you create can easily and safely communicate with each other using a number of communication mechanisms such as semaphores, mailboxes, and events.

Some features of embOS include:

- Time resolution can be freely selected (default is 1 millisecond).
- Easily accessible time variable.
- Preemptive scheduling:
Guarantees that of all tasks in `READY` state the one with the highest priority executes, except for situations in which priority inheritance applies.
- Round-robin scheduling for tasks with identical priorities.
- Preemptions can be disabled for entire tasks or for sections of a program.
- Up to 4,294,967,296 priorities. Every task can have an individual priority, which means that the response of tasks can be precisely defined according to the requirements of the application.
- Unlimited number of tasks, software timers and all other synchronization and communication primitives like event objects, semaphores, mutexes, mailboxes and queues. (limited only by the amount of available memory).
- Size and number of messages can be freely defined when initializing mailboxes.
- Up to 32-bit events for every task.
- Power management.
- Calculation time in which embOS is idle can automatically be spent in power save mode. Power-consumption is minimized.
- Full interrupt support:
Interrupts may call any function except those that require waiting for data, as well as create, delete or change the priority of a task. Interrupts can wake up or suspend tasks and directly communicate with tasks using all available communication methods (mailboxes, semaphores, events).
- Disabling interrupts for very short periods allows minimal interrupt latency.
- Nested interrupts are permitted.
- embOS has its own, optional interrupt stack.
- Application samples for an easy start.
- Debug build performs runtime checks that catch common programming errors early on.
- Profiling and stack-check may be implemented by choosing specified libraries.
- Monitoring during runtime is available using embOSView via UART, Debug Communications Channel (DCC) and memory read/write, or else via Ethernet.
- Very fast and efficient, yet small code.
- Minimal RAM usage.
- API can be called from assembly, C or C++ code.
- Board support packages (BSP) as source code available.

1.2 embOS editions

embOS is the general term for four different embOS editions:

- embOS-Base
- embOS-Ultra
- embOS-MPU
- embOS-Safe

1.2.1 embOS-Base

embOS-Base is a preemptive RTOS designed to be the foundation for developing embedded applications. Now in its 4th decade of continuous use and enhancement, its reliability and performance underpin the firmware in every J-Link and J-Trace.

1.2.2 embOS-Ultra

embOS-Ultra is a revolutionary RTOS using cycle-resolution timing to improve timing and performance and to reduce energy consumption. Using SEGGER's innovative Cycle-based Scheduling, embOS-Ultra is the first choice for applications requiring ultra low power or extremely high precision. Its time resolution is based on the CPU cycle and provides the highest precision possible. Removing iterative scheduler calls from the kernel reduces the energy consumption significantly.

1.2.3 embOS-MPU

embOS-MPU adds comprehensive memory protection to embOS-Base which tightens the safety of embedded devices. All unprivileged tasks are 100% sandboxed, making devices suitable for any safety-critical application. embOS-MPU uses the hardware's memory protection unit and additional implemented software mechanisms to prevent one task from affecting the entire system. This guarantees that even if a bug occurs in one task, all other tasks and the operating system itself continue their execution, enhancing both the stability and safety of embedded applications.

1.2.4 embOS-Safe

embOS-Safe is the pre-certified version of embOS-Base, embOS-Ultra and embOS-MPU. embOS-Safe has been certified in accordance with TÜV SÜD Germany. Certification is compliant with IEC 61508 SIL 3, IEC 62304 Class C, and ISO 26262 ASIL D.

1.3 embOS ports

embOS is available for many core and compiler combinations. The embOS sources are written in C but a small part is written in assembler and therefore core and compiler specific. Hence, an embOS port is always technically limited to one core or core family and one compiler. An embOS port includes several board support packages for different devices and evaluation boards. Each board support package includes a project for a specific IDE. In most embOS ports the same IDE is used for all board support packages.

1.3.1 Additional documentation

Some embOS aspects are core and compiler specific and explained in a separate embOS manual which is shipped in the according embOS port shipment. For example an embOS port could provide additional core specific API functions which are described in the core and compiler specific manual.

Example Cover of embOS-Base Cortex-M ES Manual



1.3.2 Naming convention

All embOS ports use the same naming convention: `embOS_<edition>_<core>_<compiler>`. For example: `embOS_Base_CortexM_ES`, embOS-Base for Cortex-M and Embedded Studio

1.3.3 Version number convention

SEGGER releases new embOS versions with new features and bug fixes. As soon as a new embOS version is released embOS ports are updated to this version.

Generic embOS

Each release of the generic embOS sources has a unique version number:

```
V<Major>.<Minor>.<Patch>
```

For example:

```
v5.20.0
```

Major: 5
Minor: 20
Patch: 0

Major and minor values are used for new features. The patch value is used for bug fixes only.

embOS Ports

An updated embOS port has the same version number as the used generic embOS sources, plus an additional revision for the port. This is because an embOS port may be updated for changes in the CPU/compiler specific part, while still using the same generic embOS sources. The complete version number for a specific embOS port is defined as:

```
V<Major>.<Minor>.<Patch>.<Revision>
```

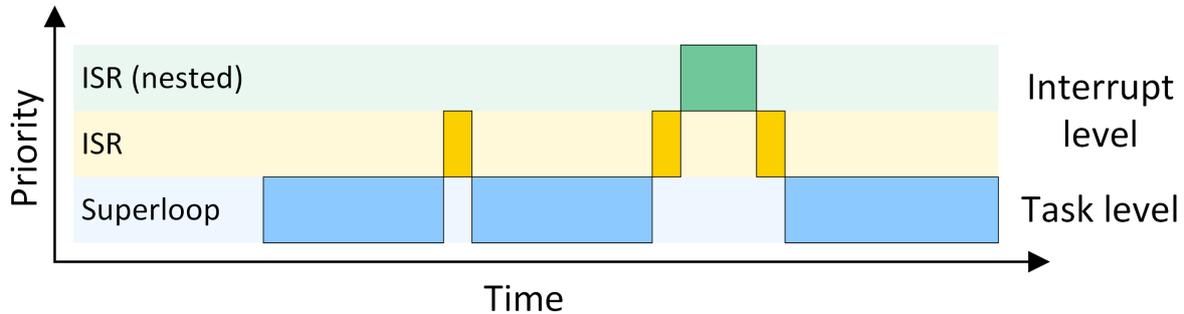
For example:

```
v5.20.0.0
```

Major: 5
Minor: 20
Patch: 0
Revision: 0

1.4 Singletasking systems (superloop)

The classic way of designing embedded systems does not use the services of an RTOS, which is also called "superloop design". Typically, no real time kernel is used, so interrupt service routines (ISRs) are used for the real-time parts of the application and for critical operations (at interrupt level). This type of system is typically used in small, simple systems or if real-time behavior is not critical.



Typically, since no real-time kernel and only one stack is used, both program (ROM) size and RAM size are smaller for simple applications when compared to using an RTOS. Obviously, there are no inter-task synchronization problems with a superloop application. However, superloops can become difficult to maintain if the program becomes too large or uses complex interactions. As sequential processes cannot interrupt themselves, reaction times depend on the execution time of the entire sequence, resulting in a poor real-time behavior.

1.4.1 Advantages & disadvantages

Advantages

- Simple structure (for small applications)
- Low stack usage (only one stack required)

Disadvantages

- No "delay" capability
- Higher power consumption due to the lack of a power save mode in most architectures
- Difficult to maintain as program grows
- Timing of all software components depends on all other software components: Small change in one place can have major side effects in other places
- Defeats modular programming
- Real time behavior only with interrupts

1.4.2 Using embOS in superloop applications

In a true superloop application, no tasks are used, hence the biggest advantage of using an RTOS cannot be utilized unless the application is re-written for multitasking. However, even with just one single task, using embOS offers the following advantages:

- Software timers are available
- Power saving: Idle mode can be used
- Future extensions can be put in a separate task

1.4.3 Migrating from superloop to multi-tasking

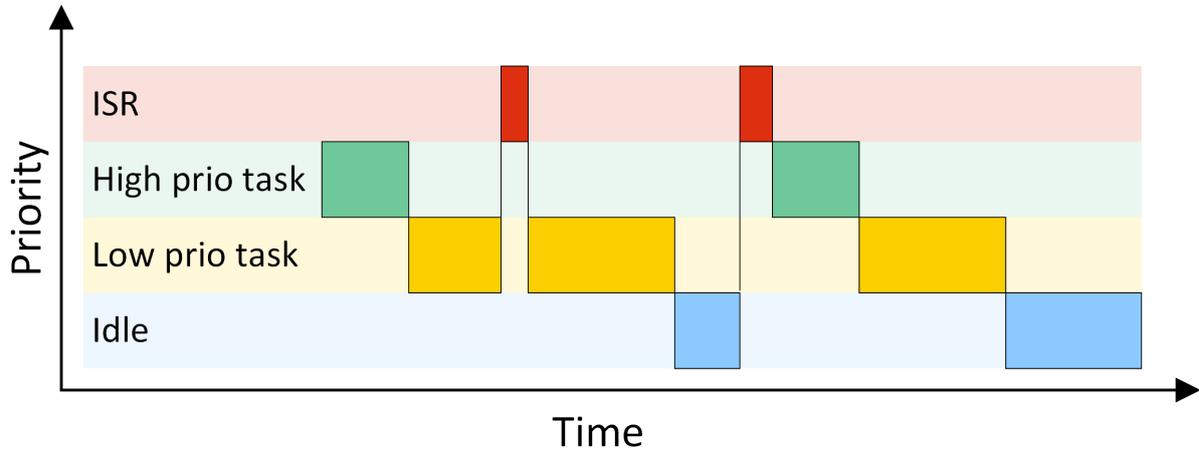
A common situation is that an application exists for some time and has been designed as a single-task super-loop-application. At some point, the disadvantages of this approach result in a decision to use an RTOS. The typical question now usually is: How do I do this?

The easiest way is to start with one of the sample applications that come with embOS and to add the existing "super-loop code" into one task. At this point, you should also ensure that the stack size of this task is sufficient. Later, additional functionality is added to the

software and can be put in one or more additional tasks; the functionality of the super-loop can also be distributed over multiple tasks.

1.5 Multitasking systems

In a multitasking system, there are different ways to distribute CPU time among different tasks. This process is called scheduling.



1.5.1 Task switches

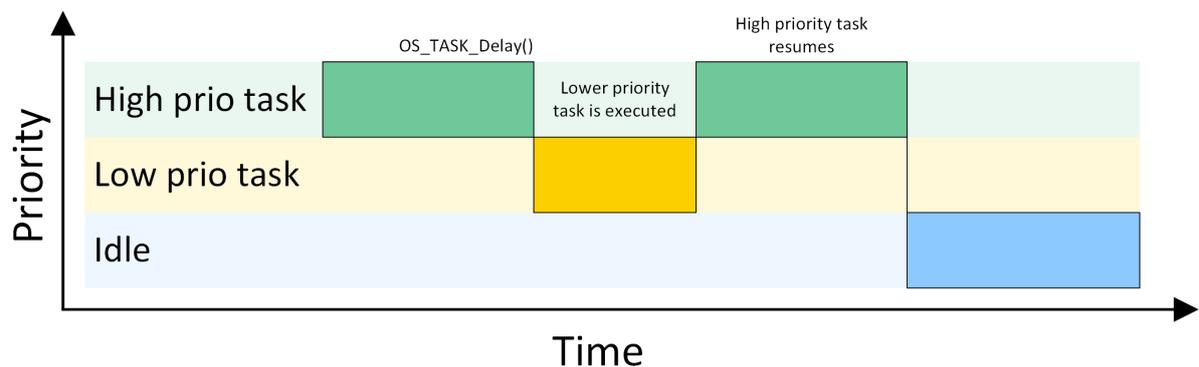
There are two types of task switches, also called context switches: Cooperative and preemptive task switches.

A cooperative task switch is performed by the task itself. As its name indicates, it requires the cooperation of the task: it suspends itself by calling a blocking RTOS function, e.g. `OS_TASK_Delay()` or `OS_TASKEVENT_GetBlocked()`.

A preemptive task switch, on the other hand, is a task switch that is caused externally. For example, a task of higher priority becomes ready for execution and, as a result, the scheduler suspends the current task in favor of that task.

1.5.2 Cooperative multitasking

Cooperative multitasking requires all tasks to cooperate by using blocking functions. A task switch can only take place if the running task blocks itself by calling a blocking function such as `OS_TASK_Delay()` or `OS_MAILBOX_GetBlocked()`. This is illustrated in the diagram below.

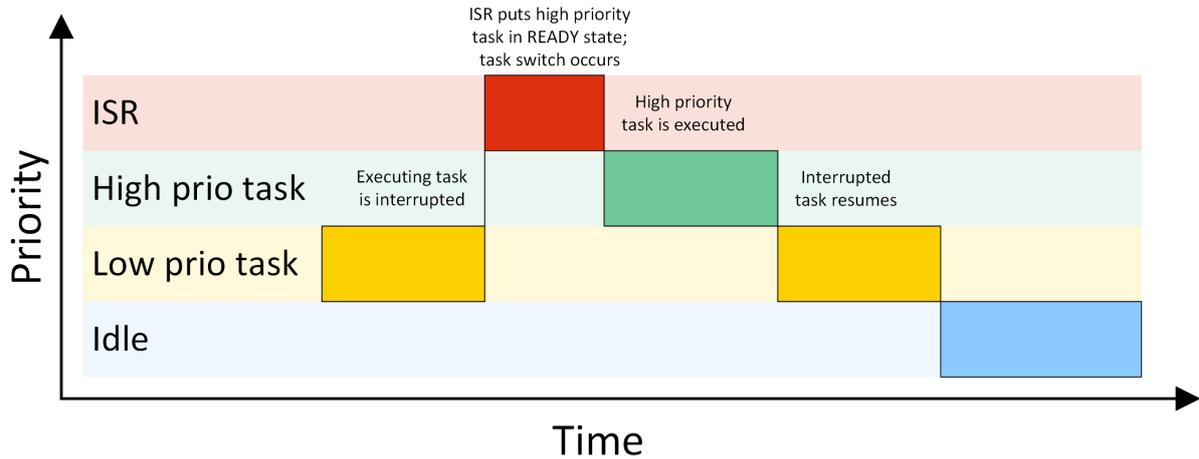


If tasks in a pure cooperative multi-tasking system do not cooperate, the system “hangs”. This means that other tasks have no chance of being executed by the CPU while the first task is being carried out. Even if an ISR makes a higher-priority task ready to run, the interrupted task will be resumed and completes before the task switch is made.

A pure cooperative multi-tasking system has the disadvantage of longer reaction times when high priority tasks become ready for execution. This makes their usage in embedded real-time systems uncommon.

1.5.3 Preemptive multitasking

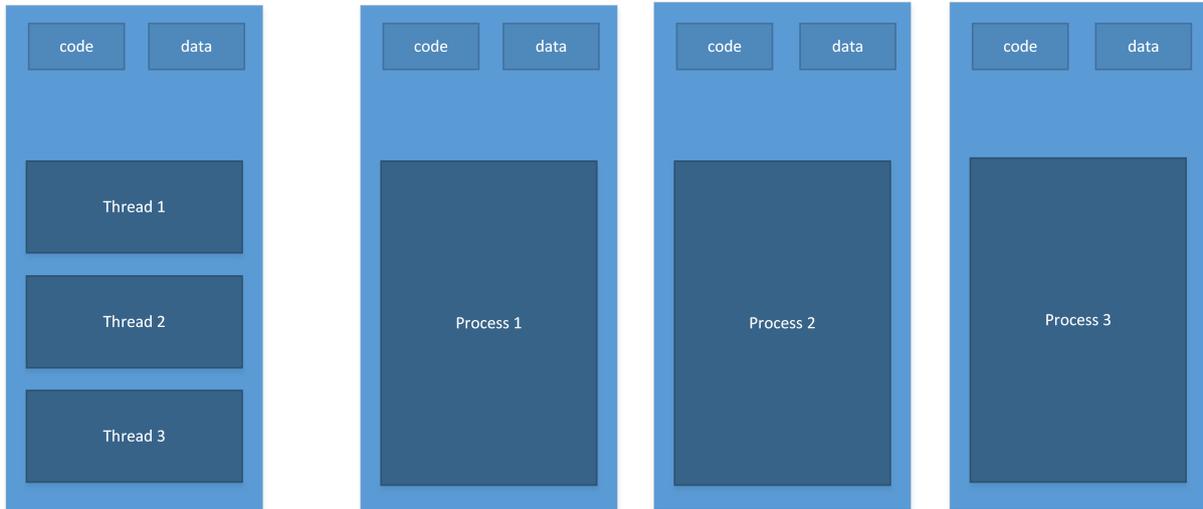
An RTOS like embOS operates with preemptive multitasking. The highest-priority task in the `READY` state always executes as long as the task is not suspended by a call of any blocking operating system function. A high-priority task waiting for an event is signaled `READY` as soon as the event occurs. The event can be set by an interrupt handler, which then activates the task immediately. Other tasks with lower priority are suspended (preempted) for as long as the high-priority task is executing. Usually, an RTOS utilizes a timer interrupt that interrupts tasks and thereby allows to perform task switches whenever timed task switches are necessary.



Preemptive multitasking may be switched off in sections of a program where task switches are prohibited, known as critical regions. embOS itself will also temporarily disable preemptive task switches during critical operations, which might be performed during the execution of some embOS API functions.

1.6 Threads vs. Processes

In this context, a task is a program running on the CPU core of a microcontroller. Without a multitasking kernel (an RTOS), only one task can be executed by the CPU. This is called a single-task system. The RTOS, on the other hand, allows the execution of multiple tasks on a single CPU. All tasks execute as if they completely “owned” the entire CPU. The tasks are scheduled for execution, meaning that the RTOS can activate and deactivate each task according to its priority, with the highest priority task being executed in general.



Threads are tasks that share the same memory layout, hence any two threads can access the same memory locations. If virtual memory is used, the same virtual to physical translation and access rights are used.

With embOS, all tasks are threads: they all have the same memory access rights and translation (in systems with virtual memory).

Processes are tasks with their own memory layout. Two processes cannot normally access the same memory locations. Different processes typically have different access rights and (in case of MMUs) different translation tables. Processes are not supported with the current version of embOS.

1.7 Scheduling

There are different algorithms used by schedulers to determine which task to execute. But all schedulers have one thing in common: they distinguish between tasks that are ready to be executed (in the `READY` state) and other tasks that are suspended for some reason (delay, waiting for mailbox, waiting for semaphore, waiting for event, etc). The scheduler selects one of the tasks in the `READY` state and activates it (executes the body of this task). The task which is currently executing is referred to as the running task. The main difference between schedulers is the way they distribute computation time between tasks in the `READY` state.

1.7.1 Priority-controlled scheduling algorithm

In real-world applications, different tasks require different response times. For example, in an application that controls a motor, a keyboard, and a display, the motor usually requires faster reaction time than the keyboard and the display. E.g., even while the display is being updated, the motor needs to be controlled. This renders preemptive multitasking essential. Round-robin might work, but as it cannot guarantee any specific reaction time, a more suitable algorithm should be used.

In priority-controlled scheduling, every task is assigned a priority. Depending on these priorities, a task is chosen for execution according to one simple rule:

Note

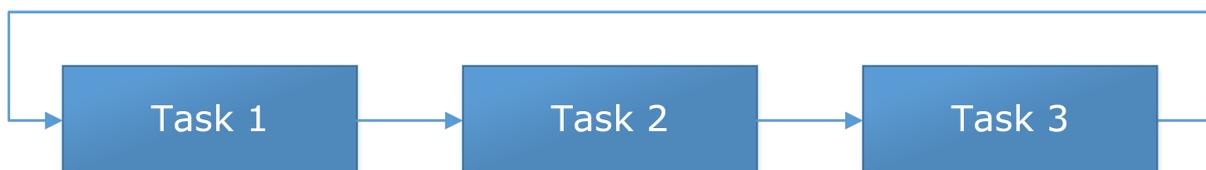
The scheduler activates the task that has the highest priority of all tasks and is ready for execution.

This means that every time a task with a priority higher than the running task becomes ready, it becomes the running task, and the previous task gets preempted. However, the scheduler can be switched off in sections of a program where task switches are prohibited, known as critical regions.

embOS uses a priority-controlled scheduling algorithm with round-robin between tasks of identical priority. One hint at this point: round-robin scheduling is a nice feature because you do not need to decide whether one task is more important than another. Tasks with identical priority cannot block each other for longer periods than their time slices. But round-robin scheduling also costs time if two or more tasks of identical priority are ready and no task of higher priority is, because execution constantly switches between the identical-priority tasks. It usually is more efficient to assign distinct priority to each task, thereby avoiding unnecessary task switches.

1.7.2 Round-robin scheduling algorithm

With round-robin scheduling, the scheduler has a list of tasks and, when deactivating the running task, it activates the next task that is in the `READY` state. Round-robin can be used with either preemptive or cooperative multitasking. It works well if you do not need to guarantee response time. Round-robin scheduling can be illustrated as follows:



The possession of the CPU changes periodically after a predefined execution time among all tasks with the same priority. This time is specified in `time slices` and may be defined individually for each task.

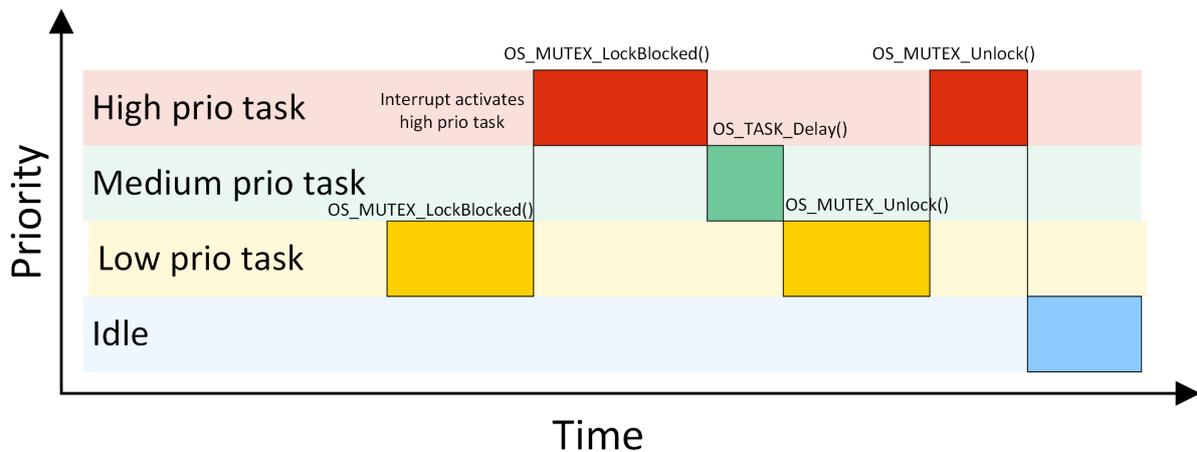
1.7.3 Priority inversion / priority inheritance

The rule the scheduler obeys is:

Activate the task that has the highest priority of all tasks in the `READY` state.

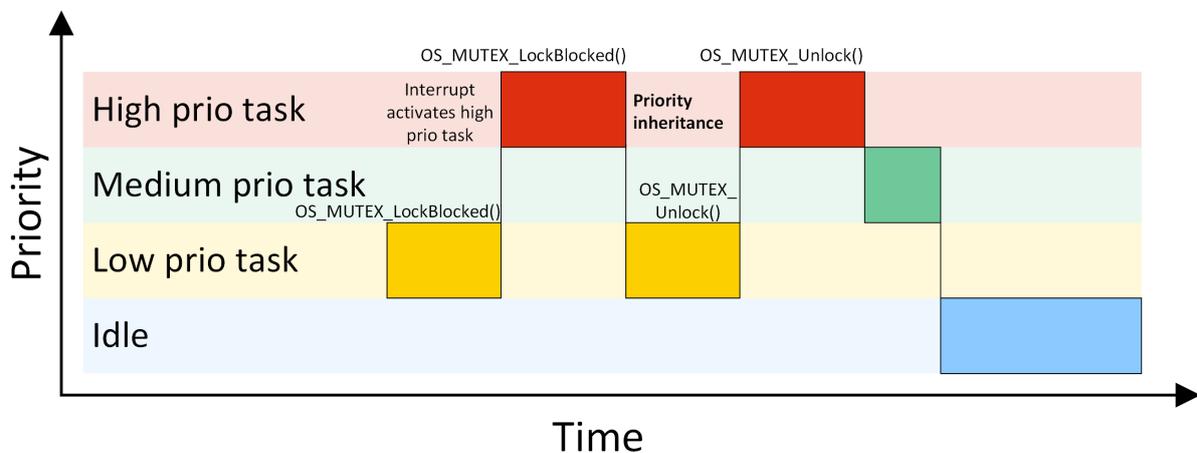
But what happens if the highest-priority task is blocked because it is waiting for a resource owned by a lower-priority task? According to the above rule, it would wait until the low-priority task is resumed and releases the resource. Up to this point, everything works as expected. Problems arise when a task with medium priority becomes ready during the execution of the higher prioritized task.

When the higher priority task is suspended waiting for the resource, the task with the medium priority will run until it finishes its work, because it has a higher priority than the low-priority task. In this scenario, a task with medium priority runs in place of the task with high priority. This is known as **priority inversion**.



The low priority task claims the mutex with `OS_MUTEX_LockBlocked()`. An interrupt activates the high priority task, which also calls `OS_MUTEX_LockBlocked()`. Meanwhile a task with medium priority becomes ready and runs when the high priority task is suspended. The task with medium priority eventually calls `OS_TASK_Delay()` and is therefore suspended. The task with lower priority now continues and calls `OS_MUTEX_Unlock()` to release the mutex. After the low priority task releases the mutex, the high priority task is activated and claims the mutex.

To avoid this situation, embOS temporarily raises the low-priority task to high priority until it releases the resource. This unblocks the task that originally had the highest priority and can now be resumed. This is known as **priority inheritance**.



With priority inheritance, the low priority task inherits the priority of the waiting high priority task as long as it holds the mutex. The lower priority task is activated instead of the medium priority task when the high priority task tries to claim the mutex.

1.7.4 Change of task status

A task may be in one of several states at any given time. When a task is created, it is placed into the `READY` state.

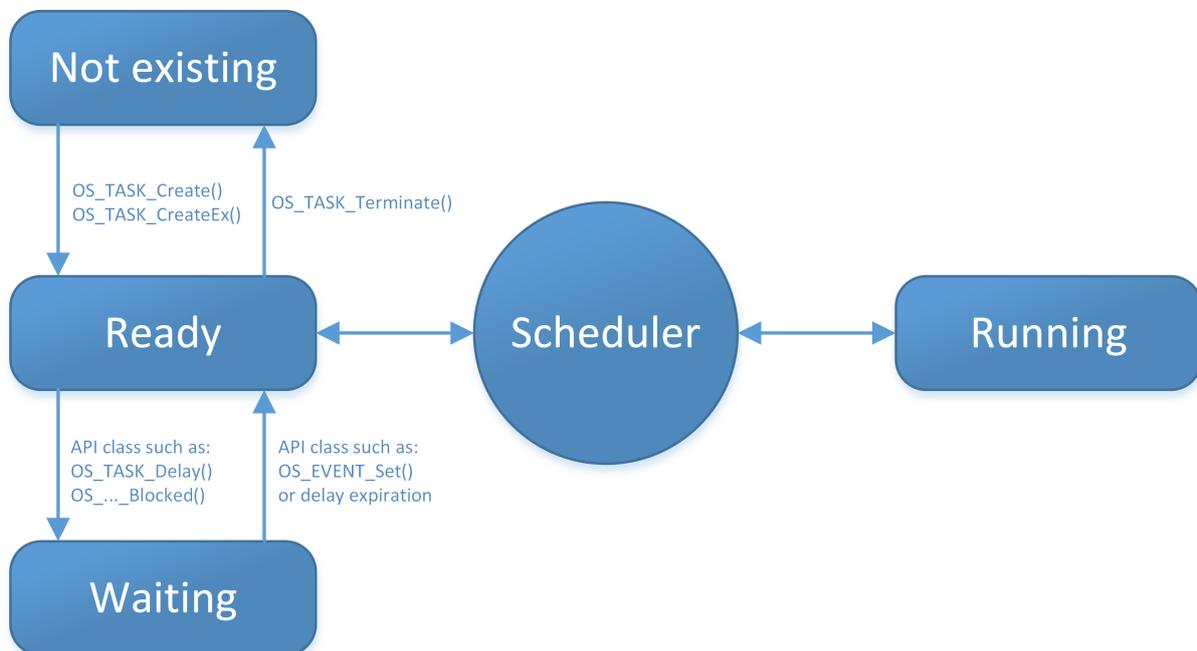
A task in the `READY` state is activated as soon as there is no other task in the `READY` state with higher priority. Only one task may be running at a time. If a task with higher priority becomes `READY`, this higher priority task is activated and the preempted task remains in the `READY` state.

The running task may be delayed for or until a specified time; in this case it is placed into the `WAITING` state and the next-highest-priority task in the `READY` state is activated.

The running task might need to wait for an event (or semaphore, mailbox or queue). If the event has not yet occurred, the task is placed into the waiting state and the next-highest-priority task in the `READY` state is activated.

A non-existent task is one that is not yet available to embOS; it either has been terminated or was not created at all.

The following illustration shows all possible task states and transitions between them.



1.7.5 How task switching works

A real-time multitasking system lets multiple tasks run like multiple single-task programs, quasi-simultaneously, on a single CPU. A task consists of three parts in the multitasking world:

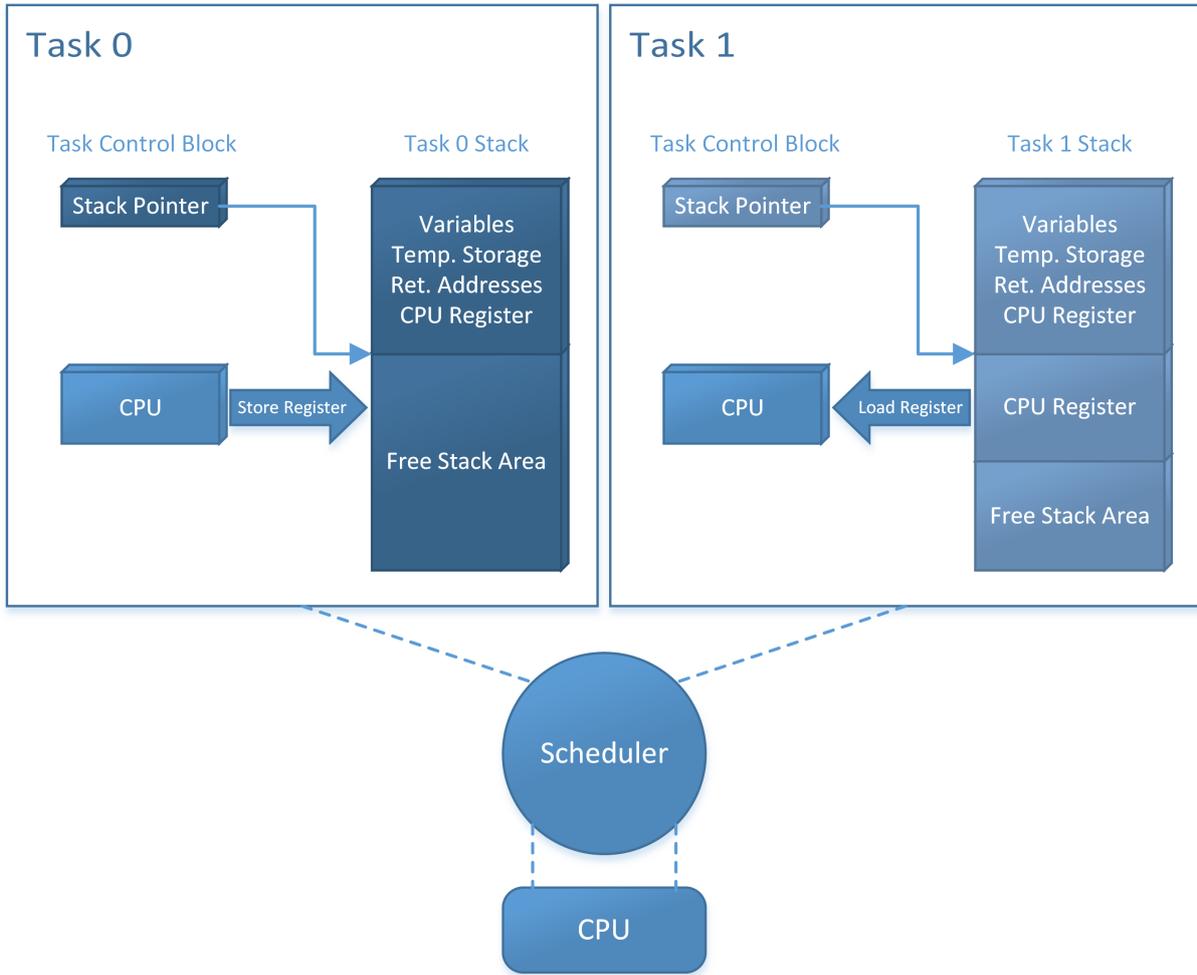
- The program code, which typically resides in ROM
- A stack, residing in a RAM area that can be accessed by the stack pointer
- A task control block, residing in RAM.

The task's stack has the same function as in a single-task system: storage of return addresses of function calls, parameters and local variables, and temporary storage of intermediate results and register values. Each task can have a different stack size. More information can be found in chapter *Stacks* on page 556.

The task control block (TCB) is a data structure assigned to a task when it is created. The TCB contains status information for the task, including the stack pointer, task priority, current task status (ready, waiting, reason for suspension) and other management data. Knowledge of the stack pointer allows access to the other registers, which are typically stored (pushed onto) the stack when the task is created and each time it is suspended. This information allows an interrupted task to continue execution exactly where it left off. TCBs are only accessed by the RTOS.

1.7.6 Switching stacks

The following diagram demonstrates the process of switching from one stack to another.



The scheduler deactivates the task to be suspended (Task 0) by saving the processor registers on its stack. It then activates the higher-priority task (Task 1) by loading the stack pointer (SP) and the processor registers from the values stored on Task 1’s stack.

Deactivating a task

The scheduler deactivates the task to be suspended (Task 0) as follows:

1. Save (push) the processor registers on the task’s stack.
2. Save the stack pointer in the Task Control Block.

Activating a task

The scheduler activates the higher-priority task (Task 1) by performing the sequence in reverse order:

1. Load (pop) the stack pointer (SP) from the Task Control Block.
2. Load the processor registers from the values stored on Task 1’s stack.

1.8 Polling vs. Event based programming

The easiest way to communicate between different pieces of code is by using global variables. In an application without RTOS you could set a flag in an UART interrupt routine and poll in main() for the flag until it is set.

```
static int      UartRxFlag;
static unsigned char Data;

void UartRxISR(void) {
    UartRxFlag = 1;
    Data = UART_RX_REGISTER;
}

int main(void) {
    while (1) {
        if (UartRxFlag != 0) {
            printf("Uart: %u", Data);
            UartRxFlag = 0;
        }
    }
    return 0;
}
```

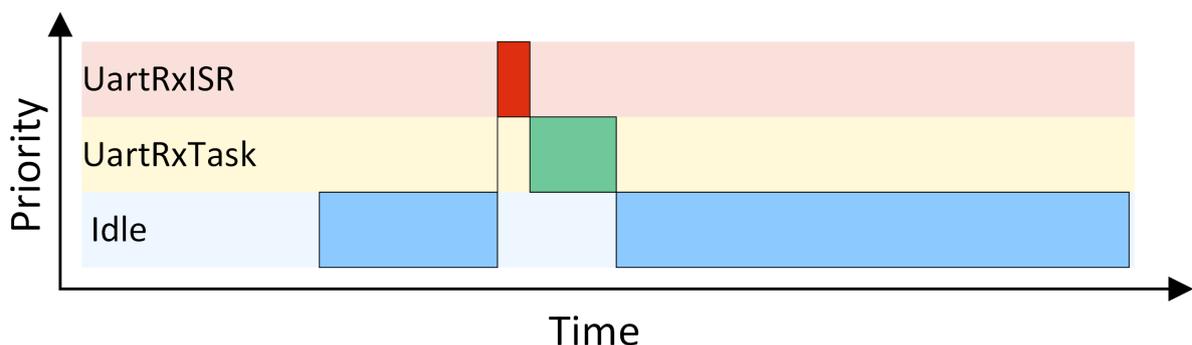
This has the disadvantage that the CPU cannot execute any other part of the application while it waits for new UART characters.

An RTOS offers the opportunity to implement an event based application. Such an event can be an interrupt. UartRxTask() calls OS_MAILBOX_GetBlocked() and is suspended until a new message is stored in the mailbox. UartRxISR() stores a new message (the received character) in the mailbox with OS_MAILBOX_Put(). Therefore UartRxTask() is executed only when a new UART character is received and does not waste any precious computation time and energy. Additionally the CPU can execute other parts of the application in the meantime.

```
void UartRxISR(void) {
    unsigned char Data;

    OS_INT_Enter();
    Data = UART_RX_REGISTER;
    OS_MAILBOX_Put(&Mailbox, &Data);
    OS_INT_Leave();
}

void UartRxTask(void) {
    unsigned char c;
    while (1) {
        OS_MAILBOX_GetBlocked(&Mailbox, &c);
        printf("Uart: %u", c);
    }
}
```



1.9 Synchronization and communication primitives

1.9.1 Synchronization primitives

In a multitasking (multithreaded) program, multiple tasks work completely separately. Because they all work in the same application, it will be necessary for them to synchronize with each other. Semaphores, mutexes and readers-write locks are used for task synchronization and to manage resources of any kind.

For details and samples, refer to the chapters *Mutexes* on page [100](#), *Semaphores* on page [101](#) and *Readers-Writer Locks* on page [102](#).

1.9.2 Event driven primitives

A task can wait for a particular event without consuming any CPU time. The idea is as simple as it is convincing, there is no sense in polling if we can simply activate a task once the event it is waiting for occurs. This saves processor cycles and energy and ensures that the task can respond to the event without delay. Typical applications for events are those where a task waits for some data, a pressed key, a received command or character, or the pulse of an external real-time clock.

For further details, refer to the chapters *Task Events* on page [103](#) and *Event Objects* on page [104](#).

1.9.3 Communication primitives

A mailbox is a data buffer managed by the RTOS. It is used for sending a message from a task or an ISR to a task. It works without conflicts even if multiple tasks and interrupts try to access the same mailbox simultaneously. embOS activates any task that is waiting for a message in a mailbox the moment it receives new data and, if necessary, switches to this task.

A queue works in a similar manner, but handles larger messages than mailboxes, and each message may have an individual size.

For more information, refer to the chapters *Mailboxes* on page [105](#) and *Queues* on page [106](#).

1.10 How the OS gains control

Upon CPU reset, the special-function registers are set to their default values. After reset, program execution begins: The PC register is set to the start address defined by the start vector or start address (depending on the CPU). This start address is usually in a startup module shipped with the C compiler, and is sometimes part of the standard library.

The startup code performs the following:

- Loads the stack pointer(s) with the default values, which is for most CPUs the end of the defined stack segment(s)
- Initializes all data segments to their respective values
- Calls the `main()` function.

The `main()` function is the part of your program which takes control immediately after the C startup. Normally, embOS works with the standard C startup module without any modification. If there are any changes required, they are documented in the CPU & Compiler Specifics manual of the embOS documentation.

With embOS, the `main()` function is still part of your application program. Essentially, `main()` creates one or more tasks and then starts multitasking by calling `OS_Start()`. From this point, the scheduler controls which task is executed.

```
Startup_code()
main()
  OS_Init();
  OS_Inithw();
  OS_TASK_CREATE();
  OS_Start();
```

The `main()` function will not be interrupted by any of the created tasks because those tasks execute only following the call to `OS_Start()`. It is therefore usually recommended to create all or most of your tasks here, as well as your control structures such as mailboxes and semaphores. Good practice is to write software in the form of modules which are (up to a point) reusable. These modules usually have an initialization routine, which creates any required task(s) and control structures. A typical `main()` function looks similar to the following example:

Example

```
int main(void) {
  OS_Init();           // Initialize embOS (must be first)
  OS_Inithw();        // Initialize hardware for embOS (in RTOSInit.c)
  // Call Init routines of all program modules which in turn will create
  // the tasks they need ... (Order of creation may be important)
  MODULE1_Init();
  MODULE2_Init();
  MODULE3_Init();
  MODULE4_Init();
  MODULE5_Init();
  OS_Start();         // Start multitasking
  return 0;
}
```

With the call to `OS_Start()`, the scheduler starts the highest-priority task created in `main()`. Note that `OS_Start()` is called only once during the startup process and does not return.

1.11 Valid context for embOS API

Some embOS functions may only be called from specific locations inside your application. We distinguish between `main()` (before the call of `OS_Start()`), privileged and unprivileged task, interrupt routine and embOS software timer.

With embOS-Base there are privileged tasks only. With embOS-MPU a task is a unprivileged task after the call to `OS_MPU_SwitchToUnprivState()`.

Note

Please consult the embOS API tables to determine whether an embOS function is allowed from within a specific execution context. Please find the API tables at beginning of each chapter.

Example

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TASK_Delay()</code>	Suspends the calling task for a specified amount of system ticks, or waits actively when called from <code>main()</code> .	•	•	•		

This table entry says it is allowed to call `OS_TASK_Delay()` from `main()` and a privileged/unprivileged task but not from an embOS software timer or an interrupt handler. Please note the differentiation between privileged and unprivileged tasks is relevant only for embOS-MPU. With embOS all tasks are privileged.

Debug check

An embOS debug build will check for violations of these rules and call `OS_Error()` with an according error code:

Error code	Description
<code>OS_ERR_ILLEGAL_IN_MAIN</code>	Not a legal API call from <code>main()</code> .
<code>OS_ERR_ILLEGAL_IN_TASK</code>	Not a legal API call after <code>OS_Start()</code> .
<code>OS_ERR_ILLEGAL_AFTER_OSSTART</code>	<code>OS_Start()</code> called twice.
<code>OS_ERR_OS_INT_ENTER_CALLED</code>	<code>OS_INT_Enter()</code> has been called, but CPU is not in ISR state.
<code>OS_ERR_ILLEGAL_IN_TIMER</code>	Not a legal API call from an embOS software timer.
<code>OS_ERR_OS_INT_ENTER_NOT_CALLED</code>	<code>OS_INT_Enter()</code> has not been called, but CPU is in ISR state.
<code>OS_ERR_ILLEGAL_OUT_ISR</code>	Not a legal API call outside an interrupt.

1.12 Blocking and Non blocking embOS API

Most embOS API comes in three different version: Non blocking, blocking and blocking with a timeout. The embOS API uses a specific naming convention for those API functions. API functions which do not block a task have no suffix. API functions which could block a task have the suffix "Blocked". API functions which could block a task but have a timeout have the suffix "Timed".

Blocking API functions (with or without a timeout) must not be called from any context other than a task context.

Non blocking API

Non blocking API functions always return at once, irrespective of the state of the OS object. The return value can be checked in order to find out if e.g. new data is available in a mailbox.

```
static OS_MAILBOX MyMailbox;
static char Buffer[10];

void Task(void) {
    char r;
    while (1) {
        r = OS_MAILBOX_Get(MyMailbox, Buffer);
        if (r == 0u) {
            // Process message
        }
    }
}
```

Blocking API

Blocking API functions suspend the task until it is activated again by another embOS API function. The task does not cause any CPU load while it is waiting for the next activation.

```
static OS_MAILBOX MyMailbox;
static char Buffer[10];

void Task(void) {
    while (1) {
        // Suspend task until a new message is available
        OS_MAILBOX_GetBlocked(MyMailbox, Buffer);
        // Process message
    }
}
```

Blocking API with timeout

These API functions have an additional timeout. They are blocking until the timeout occurs.

```
static OS_MAILBOX MyMailbox;
static char Buffer[10];

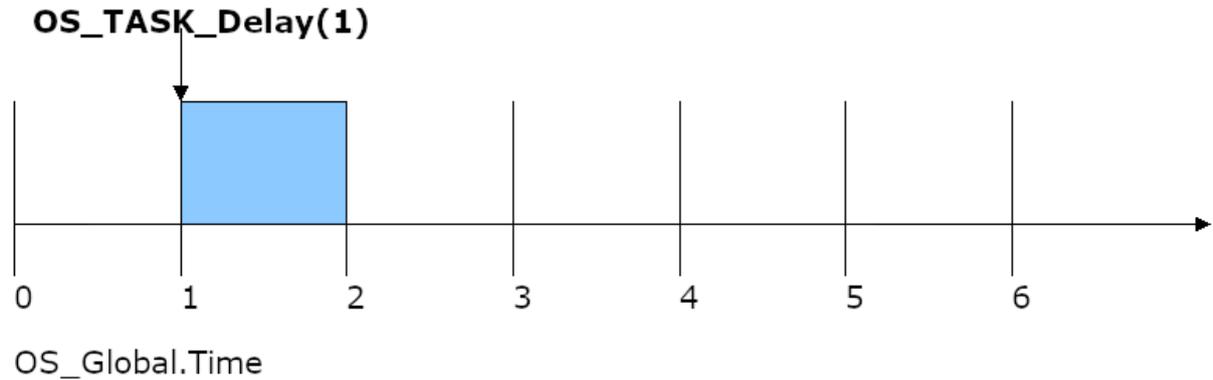
void Task(void) {
    char r;
    while (1) {
        // Suspend task until a new message is available or the timeout occurs
        r = OS_MAILBOX_GetTimed(MyMailbox, Buffer, 10);
        if (r == 0u) {
            // Process message
        }
    }
}
```

1.13 embOS API with timeout

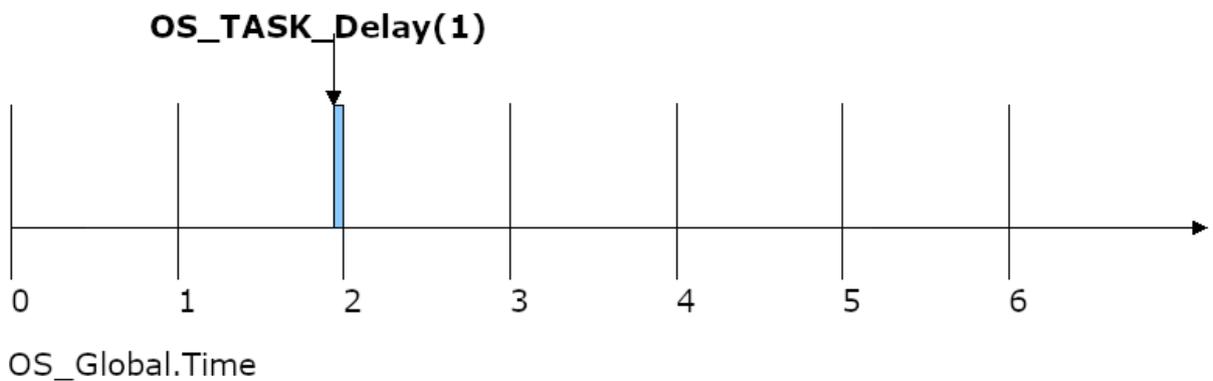
1.13.1 Usage

The embOS system tick in `OS_Global.Time` is based on a hardware timer. The hardware timer periodically generates an interrupt which increments `OS_Global.Time`. embOS API functions like `OS_TASK_Delay()` and `OS_TASKEVENT_GetTimed()` expect a timeout value as a parameter. The timeout unit is system ticks.

If for example `OS_TASK_Delay(1)` was called shortly after the timer interrupt the actual timeout is nearly one system tick.



But if `OS_TASK_Delay(1)` is called shortly before the next timer interrupt the actual timeout will be less than a full system tick or even almost zero.



The actual timeout depends on when the API function is called in relation to the next timer interrupt. A timeout of 1 could cause the API function to almost return immediately.

Note

`OS_TASK_Delay(1)` does not guarantee a timeout of a full system tick. If you need a timeout of at least one full system tick please use `OS_TASK_Delay(2)` instead.

1.13.2 Implementation details

`OS_Global.Time` holds the system time (in system ticks) since reset and always is a signed 32-bit variable. `OS_Global.Time` starts at `0x00000000` and is incremented with every system tick in the interrupt handler (assuming `OS_TICK_Config()` is not used). After `OS_Global.Time` reaches `0xFFFFFFFF`, it starts at `0x00000000` again. With a typical system tick period of 1 millisecond, this happens after ~49 days.

Note

You must not rely on `OS_Global.Time` as a timestamp since reset because it overflows after `0xFFFFFFFF` system ticks. Please use `OS_TIME_Get_us64()` instead.

When calling `OS_TASK_Delay()`, or an API function with a timeout (e.g. `OS_EVENT_GetTimed()`), embOS calculates the end time and stores it in `OS_Global.TimeDex`. The end time is the current time plus the desired timeout. `OS_Global.TimeDex` is also a signed integer value.

```
OS_Global.TimeDex = OS_Global.Time + Timeout
```

Example:

```
OS_Global.Time = 5
```

```
OS_TASK_Delay(10)
```

```
OS_Global.TimeDex = 15
```

With each system tick, embOS checks whether the current system time is equal or greater than `OS_Global.TimeDex`. This is implemented as a subtraction of signed values. This calculation guarantees that overflows are handled correctly as long as the timeout value limitation (explained below) is respected.

```
if ((OS_Global.Time - OS_Global.TimeDex) >= 0) {
    // Timeout has expired
} else {
    // Timeout has not yet expired
}
```

Note

You must not choose a timeout value which violates the following limitation:
The maximum timeout is half of the available range minus one.

$1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs.

$1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Example

With a 32-bit CPU and a one millisecond system tick the maximum timeout is ~24 days.

Description

The actual width of embOS timing variables is core specific, but for the following examples we assume 8-bit variables for easier understanding. The range is 0x00 to 0xFF, where 0x00 to 0x7F represent positive values and 0x80 to 0xFF negative values.

```
0x00  0
0x01  1
...
0x7F 127
0x80 -128
...
0xFF -1
```

Four cases exist: Both OS_Global.Time and OS_Global.TimeDex are positive values, both are negative values, and one positive and one negative value (and vice versa).

OS_Global.Time and OS_Global.TimeDex are positive

```
OS_Global.Time      = 100  (0x64)
Timeout             = 20   (0x14)
OS_Global.TimeDex   = 120  (0x78)
OS_Global.Time - OS_Global.TimeDex = -20 < 0 => Timeout has not yet expired

OS_Global.Time      = 121  (0x79)
OS_Global.TimeDex   = 120  (0x78)
OS_Global.Time - OS_Global.TimeDex = 1  >= 0 => Timeout has expired
```

OS_Global.Time and OS_Global.TimeDex are negative

```
OS_Global.Time      = -128 (0x80)
Timeout             = 8    (0x08)
OS_Global.TimeDex   = -120 (0x88)
OS_Global.Time - OS_Global.TimeDex = -8 < 0 => Timeout has not yet expired

OS_Global.Time      = -119 (0x89)
OS_Global.TimeDex   = -120 (0x78)
OS_Global.Time - OS_Global.TimeDex = 1  >= 0 => Timeout has expired
```

OS_Global.Time is positive and OS_Global.TimeDex is negative

```
OS_Global.Time      = 120  (0x88)
Timeout             = 16   (0x10)
OS_Global.TimeDex   = -120 (0x88)
OS_Global.Time - OS_Global.TimeDex = -16 < 0 => Timeout has not yet expired

OS_Global.Time      = -119 (0x89)
OS_Global.TimeDex   = -120 (0x88)
OS_Global.Time - OS_Global.TimeDex = 1  >= 0 => Timeout has expired
```

OS_Global.Time is negative and OS_Global.TimeDex is positive

```
OS_Global.Time      = -1
Timeout             = 16
OS_Global.TimeDex   = 15
OS_Global.Time - OS_Global.TimeDex = -16 < 0 => Timeout has not yet expired

OS_Global.Time      = 16
OS_Global.TimeDex   = 15
OS_Global.Time - OS_Global.TimeDex = 1  >= 0 => Timeout has expired
```

Limitation

This check may only be performed if the difference between `OS_Global.Time` and `OS_Global.TimeDex` is less than half of the available range minus one. Otherwise, it is undecidable whether `OS_Global.Time` has lapped `OS_Global.TimeDex`. The following example shows how the calculation fails if the timeout limit is violated.

```
8-bit range => Maximum timeout value = 128 - 1 = 127

OS_Global.Time      = 0
Invalid Timeout     = 130
OS_Global.TimeDex   = 130
OS_Global.Time - OS_Global.TimeDex = 126 > 0
=> Wrong result, Timeout has not yet expired
```

Conclusion

As long as the timeout limitation is not violated, an overflow of `OS_Global.Time` is no problem. As shown in the above examples all calculations are performed correctly. Therefore you will find the timeout limitation in the timeout parameter description of all according API functions.

1.14 RTOS objects

Most embOS API functions require an according RTOS object. The RTOS object is based on a C structure and stores application specific information. For example, if you create a new task with `OS_TASK_Create()`, you will need to specify a task control block. `OS_TASK_Create()` expects a pointer to an RTOS object of the type `OS_TASK` to store information like the task priority.

Examples for RTOS objects:

- `OS_TASK`
- `OS_TIMER`
- `OS_EVENT`
- `OS_MUTEX`
- `OS_SEMAPHORE`
- `OS_RWLOCK`
- `OS_MAILBOX`
- `OS_QUEUE`
- `OS_WD`

It is the developer's responsibility to allocate RAM for the RTOS object. The memory can be allocated statically or dynamically. Whether it is preferable to use static or dynamic memory allocation depends on the application. Both methods can be used with embOS and also within the same embOS application.

The RTOS object must be located in RAM; it is the developer's responsibility to allocate sufficient memory for it. Furthermore, the RTOS object must not be located at address `0x00`. If the target has RAM at address `0x00`, the linker file should define the RAM start at e.g. address `0x04`. Consequently, `NULL` must never be passed to an embOS API function that expects an RTOS object as parameter (unless explicitly stated otherwise in the respective API function description).

Static allocation

```
static OS_MUTEX _Mutex;

int main(void) {
    ...
    OS_MUTEX_Create(&_Mutex);
    ...
    return 0;
}
```

Dynamic allocation

```
static OS_MUTEX* _pMutex;

int main(void) {
    ...
    _pMutex = (OS_MUTEX*)malloc(sizeof(OS_MUTEX));
    if (_pMutex != NULL) {
        OS_MUTEX_Create(_pMutex);
    }
    ...
    return 0;
}
```

Note

An RTOS object may be modified by an embOS API function only. You must not modify an RTOS object directly. For example, you must not free memory containing an RTOS object which is still in use.

Bad examples

Write to a member of an RTOS object:

```
static OS_MUTEX _Mutex;

int main(void) {
    ...
    OS_MUTEX_Create(&_Mutex);
    _Mutex.UseCnt = 42;
    ...
    return 0;
}
```

Memory freed while the RTOS object is still in use:

```
static OS_MUTEX* _pMutex;

void Task(void) {
    while (1) {
        OS_MUTEX_LockBlocked(_pMutex);
    }
}

int main(void) {
    ...
    _pMutex = (OS_MUTEX*)malloc(sizeof(OS_MUTEX));
    OS_MUTEX_Create(_pMutex);
    free(_pMutex);
    ...
    OS_Start();
    return 0;
}
```

1.15 embOS types

In addition to embOS object types, embOS uses further data types for API routine arguments and return values.

Per default they are defined as:

Type	C type
OS_I8	signed char
OS_U8	unsigned char
OS_I16	signed short
OS_U16	unsigned short
OS_I32	signed long
OS_U32	unsigned long
OS_I64	signed long long
OS_U64	unsigned long long
OS_UINT	unsigned int
OS_BOOL	unsigned char (0 = false / 1 = true)
OS_TIME	int
OS_TASKEVENT	unsigned char (8/16-bit CPU) / unsigned long (32-bit CPU)
OS_TASK_PRIO	unsigned char (8/16-bit CPU) / unsigned long (32-bit CPU)

1.16 Callback / Hook routines

Both terms are used with embOS and mean the same. Some embOS API functions use a function pointer parameter for a callback routine. The callback routine must be implemented by the application and is defined as determined by the function pointer type. The following function pointer types are used:

Type	Definition
OS_ROUTINE_VOID	void Routine(void)
OS_ROUTINE_VOID_PTR	void Routine(void* p)
OS_ROUTINE_BOOL_VOID_PTR	OS_BOOL Routine(void* p, void* pParam)
OS_ROUTINE_CHAR	void Routine(OS_U8 Data)
OS_ROUTINE_WD_PTR	void Routine(OS_CONST_PTR OS_WD* pWD)
OS_ROUTINE_TASK_PTR	void Routine(OS_CONST_PTR OS_TASK* pTask)
OS_ROUTINE_TASK_PTR_ERRORCODE	void Routine(OS_CONST_PTR OS_TASK* pTask, OS_MPU_ERRORCODE ErrorCode)

Example

```
void OS_WD_Config(OS_ROUTINE_VOID* pfTrigger, OS_ROUTINE_WD_PTR* pfReset);
```

```
static void _TriggerRoutine(void) {
    ...
}

static void _ResetRoutine(OS_CONST_PTR OS_WD* pWD) {
    ...
}

int main(void) {
    ...
    OS_WD_Config(&_TriggerRoutine, &_ResetRoutine);
    ...
    return 0;
}
```

1.17 embOS library modes

embOS comes in different builds or versions of the libraries. The reason for different builds is that requirements vary during development. While developing software, the performance (and resource usage) is not as important as in the final version which usually goes as release build into the product. But during development, even small programming errors should be caught by use of assertions. These assertions are compiled into the debug build of the embOS libraries and make the code a little bigger (about 50%) and also slightly slower than the release or stack-check build used for the final product.

This concept gives you the best of both worlds: a compact and very efficient build for your final product (release or stack-check build of the libraries), and a safer (though bigger and slower) build for development which will catch most common application programming errors. Of course, you may also use the release build of embOS during development, but it will not catch these errors.

The features are enabled and disabled with compile-time switches in the C source code. For example the macro `OS_DEBUG` controls whether the debug code is included in the build. Please have a look in the chapter *Compile time switches* on page 599 for more details.

The following features are included in the different embOS builds:

Debug code

The embOS debug code detects application programming errors like calling an API function from an invalid context. An application using an embOS debug library has to include `OS_Error.c`. `OS_Error.c` contains the `OS_Error()` function which will be called if a debug assertion fails. It is advisable to always use embOS debug code during development.

Stack Check

The embOS stack check detects overflows of task stacks, system stack and interrupt stack. Furthermore, it enables additional information in embOSView and IDE RTOS plug-ins, and provides additional embOS API regarding stack information. An application using an embOS stack check library has to include `OS_Error.c`. `OS_Error.c` contains the `OS_Error()` function which will be called if a stack overflow occurs.

Profiling

The embOS profiling code makes precise information available about the execution time of individual tasks. You may always use the profiling libraries, but they induce larger task control blocks as well as additional ROM and runtime overhead. This overhead is usually acceptable, but for best performance you may want to use non-profiling builds of embOS if you do not use this feature.

Libraries including support for profiling do also include the support for SystemView.

embOS API Trace

embOS API trace saves information about called API in a trace buffer. The trace data can be visualized in e.g. SystemView.

embOSView API Trace

embOSView API trace saves information about called API in a trace buffer. The trace data can be visualized in embOSView.

Round-Robin

Round-Robin lets all tasks at the same priority execute periodically for a pre-defined period of time.

Object Names

Tasks and OS object names can be used to easily identify a task or e.g. a mailbox in tools like embOSView, SystemView or IDE RTOS plug-ins.

Task Context Extension

For some applications it might be useful or required to have individual data in tasks that are unique to the task or to execute specific actions at context switch. With the task context extension support each task control block includes function pointer to a save and a restore routine which are executed during the context switch from and to the task.

1.17.1 Available library modes

In your application program, you need to let the compiler know which build of embOS you are using. This is done by adding the corresponding define to your preprocessor settings and linking the appropriate library file. If the preprocessor setting does not match the library, a linker error will occur. Using the preprocessor define, `RTOS.h` will set embOS structures to the same configuration that was used during the creation of the library, thus ensuring identical structure definitions in both the application and the library. If no preprocessor setting is given, `OS_Config.h` will be included and will set a library mode automatically (see `OS_Config.h`).

Name / Define	Debug Code	Stack Check	Profiling	embOS API Trace	embOS/View API Trace	Round-Robin	Object Names	Task Context Extension
<code>OS_LIBMODE_XR</code>								
<code>OS_LIBMODE_R</code>						•	•	•
<code>OS_LIBMODE_S</code>		•				•	•	•
<code>OS_LIBMODE_SP</code>		•	•	•		•	•	•
<code>OS_LIBMODE_D</code>	•	•				•	•	•
<code>OS_LIBMODE_DP</code>	•	•	•	•		•	•	•
<code>OS_LIBMODE_DT</code>	•	•	•	•	•	•	•	•
<code>OS_LIBMODE_SAFE</code>	•	•	•	•		•	•	•

1.17.2 OS_Config.h

`OS_Config.h` is part of every embOS port and located in the `Start\Inc` folder. Use of `OS_Config.h` is optional but makes it easier to define the embOS library mode: Instead of defining `OS_LIBMODE_*` in your preprocessor settings, you may define `DEBUG=1` in your preprocessor settings in debug compile configurations and define nothing in the preprocessor settings in release compile configurations. Subsequently, `OS_Config.h` will automatically define `OS_LIBMODE_DP` for debug compile configurations and `OS_LIBMODE_R` for release compile configurations. `OS_Config.h` will be included only when no `OS_LIBMODE_*` is defined.

Compile Configuration	Preprocessor Define	Define Set by <code>OS_Config.h</code>
Debug	<code>DEBUG=1</code>	<code>OS_LIBMODE_DP</code>
Release		<code>OS_LIBMODE_R</code>

Note

The macro `DEBUG` may not be appropriate for a specific project. For example the identifier may already be used in other 3rd party software or it is preferred to use another macro like `NDEBUG`. You can customize `OS_Config.h` to fully fit your needs and keep it with your next embOS update.

1.17.3 OS_LIBMODE_SAFE

OS_LIBMODE_SAFE is usually used with embOS-Safe only for the safety certified embOS library. OS_LIBMODE_SAFE excludes/adds some embOS features which are not mentioned above:

embOS API not supported with OS_LIBMODE_SAFE

- Heap Type Memory Management
- CPU Load Measurement
- Spinlock API
- embOSView / embOSView trace
- Thread local storage

Additional embOS API supported with OS_LIBMODE_SAFE

- Configurable stack check limit
- MPU sanity check buffer

Chapter 2

Kernel

2.1 Introduction

The embOS kernel is started with `OS_Start()` in `main()` after the kernel was initialized with `OS_Init()`. Typically, applications will also initialize the required hardware, and create at least one task before calling `OS_Start()`. `OS_Start()` usually never returns but runs the embOS scheduler which decides which task to run next. It is possible to stop and de-initialize the kernel with `OS_Stop()` and `OS_DeInit()`.

Example

```
int main(void) {
    OS_Init();    // Initialize embOS
    OS_Inithw(); // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_Start();  // Start embOS

    return 0;
}
```

Interrupts in main()

`OS_Start()` enables interrupts, but interrupts may also be used in `main()`. It is not necessary to disable interrupts in `main()`. When using embOS interrupts in `main()`, please ensure they are enabled after `OS_Init()` only. It is good practice to call `OS_Init()` as first instruction in `main()`.

```
void UART_ISR(void) {
    // Handle UART interrupt
}

int main(void) {
    OS_Init();    // Initialize embOS
    UART_Init(); // Initialize UART and UART interrupts
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_Start();  // Start embOS

    return 0;
}
```

2.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_ConfigStop()</code>	Configures the <code>OS_Stop()</code> function.	•				
<code>OS_DeInit()</code>	De-initializes the embOS kernel.	•				
<code>OS_Init()</code>	Initializes the embOS kernel.	•				
<code>OS_IsRunning()</code>	Determines whether the embOS kernel was started by a call to <code>OS_Start()</code> .	•	•	•	•	•
<code>OS_Start()</code>	Starts the embOS kernel.	•				
<code>OS_Stop()</code>	Stops the embOS kernel and returns from <code>OS_Start()</code> .		•			

2.2.1 OS_ConfigStop()

Description

Configures the `OS_Stop()` function.

Prototype

```
void OS_ConfigStop(OS_MAIN_CONTEXT* pContext,
                  void* Addr,
                  OS_U32 Size);
```

Parameters

Parameter	Description
<code>pContext</code>	Pointer to an object of type <code>OS_MAIN_CONTEXT</code> .
<code>Addr</code>	Address of the buffer which is used to save the <code>main()</code> stack.
<code>Size</code>	<code>Size</code> of the buffer.

Additional information

This function configures the `OS_Stop()` function. When configured, `OS_Start()` saves the context and stack from within `main()`, which subsequently are restored by `OS_Stop()`. The `main()` context and stack are saved to the resources configured by `OS_ConfigStop()`. Only the stack that was actually used during `main()` is saved. Therefore, the size of the buffer depends on the used stack. The structure `OS_MAIN_CONTEXT` is core and compiler specific; it is specifically defined with each embOS port.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
227: OS_ERR_ILLEGAL_IN_TASK
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#include "RTOS.h"
#include "stdio.h"

#define BUFFER_SIZE (32u)
static OS_U8 Buffer[BUFFER_SIZE]; // Buffer for main stack copy
static OS_MAIN_CONTEXT MainContext; // Main context control structure
static OS_STACKPTR int StackHP[128]; // Task stack
static OS_TASK TCBHP; // Task control block

static void HPTask(void) {
    OS_TASK_Delay(50);
    OS_INT_Disable();
    OS_Stop();
}

int main(void) {
    int TheAnswerToEverything = 42;
    OS_Init(); // Initialize embOS
    OS_InitHW(); // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_ConfigStop(&MainContext, Buffer, BUFFER_SIZE);
    OS_Start(); // Start embOS
```

```
//  
// We arrive here because OS_Stop() was called.  
// The local stack variable still has its value.  
//  
printf("%d", TheAnswerToEverything);  
while (TheAnswerToEverything == 42) {  
}  
return 0;  
}
```

2.2.2 OS_DeInit()

Description

De-initializes the embOS kernel.

Prototype

```
void OS_DeInit(void);
```

Additional information

OS_DeInit() can be used to de-initializes the embOS kernel and the hardware which was initialized in OS_Init(). OS_DeInit() is usually used after returning from OS_Start(). It does not de-initialize the hardware which was configured in e.g. OS_InitHW() but it resets all embOS variables to their default values.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
227: OS_ERR_ILLEGAL_IN_TASK
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define BUFFER_SIZE      (32u)

static OS_STACKPTR int StackHP[128] // Task stacks
static OS_TASK      TCBHP;         // Task control blocks
static OS_U8        Buffer[BUFFER_SIZE];
static OS_MAIN_CONTEXT MainContext;

static void HPTask(void) {
    while (1) {
        OS_TASK_Delay(50);
        OS_Stop();
    }
}

int main(void) {
    OS_Init();           // Initialize embOS
    OS_InitHW();        // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_ConfigStop(&MainContext, Buffer, BUFFER_SIZE);
    OS_Start();         // Start embOS
    OS_DeInit();
    OS_DeInitHW();
    DoSomethingElse();
    //
    // Start embOS for the 2nd time
    //
    OS_Init();
    OS_InitHW();
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_ConfigStop(&MainContext, Buffer, BUFFER_SIZE);
    OS_Start();
    return 0;
}
```

2.2.3 OS_Init()

Description

Initializes the embOS kernel.

Prototype

```
void OS_Init(void);
```

Additional information

In library mode `OS_LIBMODE_SAFE` all RTOS variables are explicitly initialized. All other library modes presume that, according to the C standard, all initialized variables have their initial value and all non initialized variables are set to zero.

Note

`OS_Init()` must be called prior to any other embOS API. When using embOS API in C++ constructors, please be aware C++ constructors might be executed before `main()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
227: OS_ERR_ILLEGAL_IN_TASK
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
    while (1) {
        OS_TASK_Delay(50);
    }
}

static void LPTask(void) {
    while (1) {
        OS_TASK_Delay(200);
    }
}

/*****
 *
 *      main()
 */
int main(void) {
    OS_Init();           // Initialize embOS
    OS_InitHW();        // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_Start();         // Start embOS
    return 0;
}
```

```
}
```

2.2.4 OS_IsRunning()

Description

Determines whether the embOS kernel was started by a call to `OS_Start()`.

Prototype

```
OS_BOOL OS_IsRunning(void);
```

Return value

= 0 Kernel is not started.
≠ 0 Kernel is running, `OS_Start()` has been called.

Additional information

This function may be helpful for some functions which might be called from `main()` or from running tasks. As long as the kernel is not started and a function is called from `main()`, blocking task switches are not allowed. A function which may be called from a task or `main()` may use `OS_IsRunning()` to determine whether a subsequent call to a blocking API function is allowed.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintStatus() {  
    OS_BOOL b;  
  
    b = OS_IsRunning();  
    if (b == 0) {  
        printf("embOS scheduler not started, yet.\n");  
    } else {  
        printf("embOS scheduler is running.\n");  
    }  
}
```

2.2.5 OS_Start()

Description

Starts the embOS scheduler.

Prototype

```
void OS_Start(void);
```

Additional information

This function starts the embOS scheduler, which will activate and start the task with the highest priority.

OS_Start() marks embOS as running; this may be examined by a call of the function OS_IsRunning(). OS_Start() automatically enables interrupts. It must be called from main() context only.

embOS will reuse the main stack after OS_Start() was called. Therefore, local data located on the main stack may not be used after calling OS_Start(). If OS_Stop() is used, OS_ConfigStop() will save the main stack and restore it upon stopping embOS.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

165: OS_ERR_INIT_NOT_CALLED

228: OS_ERR_ILLEGAL_AFTER_OSSTART

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
    while (1) {
        OS_TASK_Delay(50);
    }
}

static void LPTask(void) {
    while (1) {
        OS_TASK_Delay(200);
    }
}

/*****
 *
 *      main()
 */
int main(void) {
    OS_Init();           // Initialize embOS
    OS_InitHW();        // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_Start();         // Start embOS
    return 0;
}
```

2.2.6 OS_Stop()

Description

Stops the embOS kernel and returns from `OS_Start()`.

Prototype

```
void OS_Stop(void);
```

Additional information

This function stops the embOS kernel and the application returns from `OS_Start()`. `OS_ConfigStop()` must be called prior to `OS_Stop()`. `OS_Stop()` restores context and stack to their state prior to calling `OS_Start()`. `OS_Stop()` does not deinitialize any hardware. It's the application's responsibility to de-initialize all hardware that was initialized, for example, during `OS_InitHW()`.

It is possible to restart embOS after `OS_Stop()`. To do so, `OS_Init()` must be called and any task must be recreated. It also is the application's responsibility to initialize all embOS variables to their default values. With the embOS source code, this can easily be achieved using the compile time switch `OS_INIT_EXPLICITLY`.

With some cores it is not possible to save and restore the `main()` stack. This is e.g. true for 8051. Hence, in that case no functionality should be implemented that relies on the stack to be preserved. But `OS_Stop()` can be used anyway.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
250: OS_ERR_CONFIG_OSSTOP
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#include "RTOS.h"
#include "stdio.h"

#define BUFFER_SIZE      (32u)
static OS_U8             Buffer[BUFFER_SIZE];
static OS_MAIN_CONTEXT  MainContext;

static OS_STACKPTR int  StackHP[128];
static OS_TASK          TCBHP;

static void HPTask(void) {
    OS_TASK_Delay(50);
    OS_Stop();
}

int main(void) {
    int TheAnswerToEverything = 42;
    OS_Init();
    OS_InitHW();
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_ConfigStop(&MainContext, Buffer, BUFFER_SIZE);
    OS_Start();
    //
    // We arrive here because OS_Stop() was called.
```

```
// The local stack variable still has its value.  
//  
printf("%d", TheAnswerToEverything);  
while (1) {  
}  
return 0;  
}
```

Chapter 3

Task

3.1 Introduction

A task that should run under embOS needs a task control block (TCB), a task stack, and a task body written in C. The following rules apply to task routines:

- The task routine can either not take parameters (void parameter list), in which case `OS_TASK_Create()` is used to create it, or take a single void pointer as parameter, in which case `OS_TASK_CreateEx()` is used to create it.
- The task routine must not return.
- The task routine must be implemented as an endless loop or it must terminate itself (see examples below).

3.1.1 Example of a task routine as an endless loop

```
void Task1(void) {
    while(1) {
        DoSomething();           // Do something
        OS_TASK_Delay(10);      // Give other tasks a chance to run
    }
}
```

3.1.2 Example of a task routine that terminates itself

```
void Task2(void) {
    char DoSomeMore;
    do {
        DoSomeMore = DoSomethingElse(); // Do something
        OS_TASK_Delay(10);              // Give other tasks a chance to run
    } while (DoSomeMore);
    OS_TASK_Terminate(NULL);           // Terminate this task
}
```

There are different ways to create a task: On the one hand, embOS offers a simple macro to facilitate task creation, which is sufficient in most cases. However, if you are dynamically creating and deleting tasks, a function is available allowing “fine-tuning” of all parameters. For most applications, at least initially, we recommend using the macro.

3.2 Cooperative vs. preemptive task switches

In general, preemptive task switches are an important feature of an RTOS. Preemptive task switches are required to guarantee responsiveness of high-priority, time critical tasks. However, it may be desirable to disable preemptive task switches for certain tasks in some circumstances. The default behavior of embOS is to allow preemptive task switches in all circumstances.

3.2.1 Disabling preemptive task switches for tasks of equal priority

In some situations, preemptive task switches between tasks running at identical priorities are not desirable. To inhibit time slicing of equal-priority tasks, the time slice of the tasks running at identical priorities must be set to zero as in the example below:

```
#include "RTOS.h"

#define PRIO_COOP      10
#define TIME_SLICE_NULL 0

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task control blocks

static void TaskEx(void* pData) {
    while (1) {
        OS_TASK_Delay((OS_TIME)pData);
    }
}

/*****
 *
 *     main()
 */
int main(void) {
    OS_Init();           // Initialize embOS
    OS_Inithw();        // Initialize required hardware
    BSP_Init();         // Initialize LED ports
    OS_TASK_CreateEx(&TCBHP, "HP Task", PRIO_COOP, TaskEx, StackHP,
                    sizeof(StackHP), TIME_SLICE_NULL, (void *) 50);
    OS_TASK_CreateEx(&TCBLP, "LP Task", PRIO_COOP, TaskEx, StackLP,
                    sizeof(StackLP), TIME_SLICE_NULL, (void *) 200);
    OS_Start();        // Start embOS
    return 0;
}
```

3.2.2 Completely disabling preemptions for a task

This is simple: The first line of code should be `OS_TASK_EnterRegion()` as shown in the following sample:

```
void MyTask(void* pContext) {
    OS_TASK_EnterRegion(); // Disable preemptive context switches
    while (1) {
        // Do something. In the code, make sure that you call a blocking
        // function periodically to give other tasks a chance to run.
    }
}
```

This will entirely disable preemptive context switches from that particular task and will therefore affect the timing of higher-priority tasks. Do not use this carelessly.

3.3 Extending the task context

For some applications it might be useful or required to have individual data in tasks that are unique to the task. Local variables, declared in the task, are unique to the task and remain valid, even when the task is suspended and resumed again. When the same task function is used for multiple tasks, local variables in the task may be used, but cannot be initialized individually for every task. embOS offers different options to extend the task context.

3.3.1 Passing one parameter to a task during task creation

Very often it is sufficient to have just one individual parameter passed to a task. Using the `OS_TASK_CREATEEX()` or `OS_TASK_CreateEx()` function to create a task allows passing a void-pointer to the task. The pointer may point to individual data, or may represent any data type that can be held within a pointer.

3.3.2 Extending the task context individually at runtime

Sometimes it may be required to have an extended task context for individual tasks to store global data or special CPU registers such as floating-point registers in the task context. The standard libraries for file I/O, locale support and others may require task-local storage for specific data like `errno` and other variables. embOS enables extension of the task context for individual tasks during runtime by a call of `OS_TASK_SetContextExtension()`. The sample application file `OS_ExtendTaskContext.c` delivered in the application samples folder of embOS demonstrates how the individual task context extension can be used.

3.3.3 Extending the task context by using own task structures

When complex data is needed for an individual task context, the `OS_TASK_CREATEEX()` or `OS_TASK_CreateEx()` functions may be used, passing a pointer to individual data structures to the task. Alternatively you may define your own task structure which can be used. Note, that the first item in the task structure must be an embOS task control structure `OS_TASK`. This can be followed by any amount and type of additional data of different types.

The following code shows the example application `OS_ExtendedTask.c` which is delivered in the sample application folder of embOS.

```
#include "RTOS.h"

/*****
 *
 *      Types, local
 *
 *****/

//
// Custom task structure with extended task context.
//
typedef struct {
    OS_TASK Task;      // OS_TASK has to be the first element
    OS_TIME Timeout;  // Any other data type may be used to extend the context
    char*   pString;  // Any number of elements may be used to extend the context
} MY_APP_TASK;

/*****
 *
 *      Static data
 *
 *****/

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
```

```

static MY_APP_TASK      TCBHP, TCBLP;           // Task-control-blocks

/*****
 *
 *      Local functions
 *
 *****/

/*****
 *
 *      MyTask()
 */
static void MyTask(void) {
    MY_APP_TASK* pThis;
    OS_TIME      Timeout;
    char*        pString;

    pThis = (MY_APP_TASK*)OS_TASK_GetID();
    while (1) {
        Timeout = pThis->Timeout;
        pString = pThis->pString;
        OS_COM_SendString(pString);
        OS_TASK_Delay(Timeout);
    }
}

/*****
 *
 *      Global functions
 *
 *****/

/*****
 *
 *      main()
 */
int main(void) {
    OS_Init();    // Initialize embOS
    OS_Inithw(); // Initialize required hardware
    //
    // Create the extended tasks just as normal tasks.
    // Note that the first parameter has to be of type OS_TASK
    //
    OS_TASK_CREATE(&TCBHP.Task, "HP Task", 100, MyTask, StackHP);
    OS_TASK_CREATE(&TCBLP.Task, "LP Task", 50, MyTask, StackLP);
    //
    // Give task contexts individual data
    //
    TCBHP.Timeout = 200;
    TCBHP.pString = "HP task running\n";
    TCBLP.Timeout = 500;
    TCBLP.pString = "LP task running\n";
    OS_Start();   // Start embOS
    return 0;
}

```

3.4 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TASK_AddContextExtension()</code>	Adds the specified task context extension.		•			
<code>OS_TASK_AddTerminateHook()</code>	Adds the specified hook (call-back) routine to the list of routines which are called when a task is terminated.	•	•			
<code>OS_TASK_Create()</code>	Creates a new task.	•	•			
<code>OS_TASK_CreateEx()</code>	Creates a new task and passes a parameter to the task.	•	•			
<code>OS_TASK_Delay()</code>	Suspends the calling task for a specified amount of system ticks, or waits actively when called from main().	•	•	•		
<code>OS_TASK_Delay_us()</code>	Actively waits for the given time (in microseconds) to expire.	•	•	•		
<code>OS_TASK_DelayUntil()</code>	Suspends the calling task until a specified time, or waits actively when called from main().	•	•	•		
<code>OS_TASK_GetID()</code>	Returns a pointer to the task control block structure of the currently scheduled task.	•	•	•	•	•
<code>OS_TASK_GetName()</code>	Returns a pointer to the name of the specified task.	•	•	•	•	•
<code>OS_TASK_GetNumTasks()</code>	Returns the number of tasks.	•	•	•	•	•
<code>OS_TASK_GetPriority()</code>	Returns the task priority of the specified task.	•	•	•	•	•
<code>OS_TASK_GetStatus()</code>	Returns the current task status of the the specified task.	•	•	•	•	•
<code>OS_TASK_GetSuspendCnt()</code>	Returns the suspension count and thus suspension state of the specified task.	•	•	•	•	•
<code>OS_TASK_GetTimeSliceRem()</code>	Returns the remaining time slice value of the specified task in system ticks.	•	•	•	•	•
<code>OS_TASK_IsTask()</code>	Determines whether the specified task control block belongs to a valid task.	•	•	•	•	•
<code>OS_TASK_Index2Ptr()</code>	Returns the task control block of the task with the specified Index.	•	•	•	•	•
<code>OS_TASK_RemoveAllTerminateHooks()</code>	Removes all hook functions from the <code>OS_ON_TERMINATE_HOOK</code> list which contains the list of functions that are called when a task is terminated.	•	•			

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TASK_RemoveTerminateHook()</code>	<code>OS_TASK_RemoveTerminateHook()</code> removes the specified hook function from the <code>OS_ON_TERMINATE_HOOK</code> list which contains the list of functions that are called when a task is terminated.	•	•			
<code>OS_TASK_Resume()</code>	Resumes the specified suspended task.	•	•	•	•	
<code>OS_TASK_ResumeAll()</code>	Decrements the suspend count of all tasks that have a nonzero suspend count and resumes these tasks when their respective suspend count reaches zero.	•	•		•	
<code>OS_TASK_SetContextExtension()</code>	Makes global variables or processor registers task-specific.		•			
<code>OS_TASK_SetDefaultContextExtension()</code>	Sets the specified context extension as the default task context extension.	•	•			
<code>OS_TASK_SetDefaultStartHook()</code>	Sets a default hook routine which is executed before a task starts.	•	•			
<code>OS_TASK_SetInitialSuspendCnt()</code>	Sets the initial suspend count for newly created tasks to 1 or 0.	•	•		•	•
<code>OS_TASK_SetName()</code>	Set the specified task name for the specified task.	•	•		•	•
<code>OS_TASK_SetPriority()</code>	Assigns the specified task priority to the specified task.	•	•			
<code>OS_TASK_SetTimeSlice()</code>	Assigns the specified time-slice period to a specified task.	•	•		•	•
<code>OS_TASK_Suspend()</code>	Suspends the specified task and increments the task's suspend count.	•	•	•		
<code>OS_TASK_SuspendAll()</code>	Suspends all tasks except the running task.	•	•		•	•
<code>OS_TASK_Terminate()</code>	Ends (terminates) the specified task.	•	•			
<code>OS_TASK_Wake()</code>	Ends delay of the specified task immediately.	•	•	•	•	
<code>OS_TASK_Yield()</code>	Calls the scheduler to force a task switch.		•	•		

3.4.1 OS_TASK_AddContextExtension()

Description

Adds the specified task context extension. The task context can be extended with `OS_TASK_SetContextExtension()` only once. Additional task context extensions can be added with `OS_TASK_AddContextExtension()`. `OS_TASK_AddContextExtension()` can also be called for the first task context extension.

The function `OS_TASK_AddContextExtension()` requires an additional parameter of type `OS_EXTEND_TASK_CONTEXT_LINK` which is used to create a task specific linked list of task context extensions.

Prototype

```
void OS_TASK_AddContextExtension
(OS_EXTEND_TASK_CONTEXT_LINK* pExtendContextLink,
 OS_CONST_PTR OS_EXTEND_TASK_CONTEXT *pExtendContext);
```

Parameters

Parameter	Description
<code>pExtendContextLink</code>	Pointer to the <code>OS_EXTEND_TASK_CONTEXT_LINK</code> structure.
<code>pExtendContext</code>	Pointer to the <code>OS_EXTEND_TASK_CONTEXT</code> structure which contains the addresses of the specific save and restore functions that save and restore the extended task context during task switches.

Additional information

The object of type `OS_EXTEND_TASK_CONTEXT_LINK` is task specific and must only be used for one task. It can be located e.g. on the task stack. `pExtendContext`, `pExtendContext->pfSave` and `pExtendContext->pfRestore` must not be NULL.

Note

embOS interrupts must not be enabled in the save and restore functions.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
116: OS_ERR_EXTEND_CONTEXT
128: OS_ERR_INV_TASK
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void HPTask(void) {
    OS_EXTEND_TASK_CONTEXT_LINK p;
    //
    // Extend task context by VFP registers
    //
    OS_TASK_SetContextExtension(&_SaveRestoreVFP);
    //
    // Extend task context by global variable
    //
```

```
OS_TASK_AddContextExtension(&p, &_SaveRestoreGlobalVar);  
a = 1.2;  
while (1) {  
    b = 3 * a;  
    GlobalVar = 1;  
    OS_TASK_Delay(10);  
}
```

3.4.2 OS_TASK_AddTerminateHook()

Description

Adds the specified hook (callback) routine to the list of routines which are called when a task is terminated.

Prototype

```
void OS_TASK_AddTerminateHook(OS_ON_TERMINATE_HOOK* pHook,
                             OS_ROUTINE_TASK_PTR* pfRoutine);
```

Parameters

Parameter	Description
<code>pHook</code>	Pointer to a variable of type <code>OS_ON_TERMINATE_HOOK</code> which will be inserted into the linked list of routines to be called during <code>OS_TASK_Terminate()</code> .
<code>pfRoutine</code>	Pointer to the routine of type <code>OS_ROUTINE_TASK_PTR</code> which shall be called when a task is terminated.

Additional information

For some applications, it may be useful to allocate memory or objects specific to tasks. For other applications, it may be useful to have task-specific information on the stack. When a task is terminated, the task-specific objects may become invalid. A callback routine may be hooked into `OS_TASK_Terminate()` by calling `OS_TASK_AddTerminateHook()` to allow the application to invalidate all task-specific objects before the task is terminated.

The callback routine `pfRoutine` of type `OS_ROUTINE_TASK_PTR` receives the address of the terminated task as its parameter (see description in chapter *Callback / Hook routines* on page 44).

Note

The variable of type `OS_ON_TERMINATE_HOOK` must reside in memory as a global or static variable. It may be located on a task stack, as local variable, but it must not be located on any stack of any task that might be terminated.

If a task terminates itself, its task control block and task stack are still used until the scheduler switches to another task or `OS_Idle()`. You must not use the task control block or task stack for anything else before the scheduler was executed. For example you must not free the task control block or task stack in the hook routine when using heap memory for the task control block or task stack.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
OS_ON_TERMINATE_HOOK _TerminateHook;

void TerminateHookFunc(OS_CONST_PTR OS_TASK* pTask) {
    // This function is executed upon calling OS_TASK_Terminate().
    if (pTask == &MyTask) {
```

```
    free(MytaskBuffer);
  }
}
...
int main(void) {
    OS_TASK_AddTerminateHook(&_amp;_TerminateHook, TerminateHookFunc);
    ...
}
```

3.4.3 OS_TASK_Create()

Description

Creates a new task.

Prototype

```
void OS_TASK_Create(      OS_TASK*      pTask,
                        const char*     sName,
                        OS_TASK_PRIO    Priority,
                        OS_ROUTINE_VOID* pfRoutine,
                        void             OS_STACKPTR *pStack,
                        OS_UINT          StackSize,
                        OS_UINT          TimeSlice);
```

Parameters

Parameter	Description
pTask	Pointer to a task control block of type <code>OS_TASK</code> .
sName	Pointer to the name of the task. Can be <code>NULL</code> if not used. embOS does not copy the task name, but uses the pointer exclusively. When using an embOS build without task name support, this parameter is ignored.
Priority	Priority of the task. Must be within the following range: $1 \leq \text{Priority} \leq 2^8 - 1 = 0xFF$ for 8/16-bit CPUs $1 \leq \text{Priority} \leq 2^{32} - 1 = 0xFFFFFFFF$ for 32-bit CPUs Higher values indicate higher priorities. The type <code>OS_TASK_PRIO</code> is defined as a 32-bit value for 32-bit CPUs and as an 8-bit value for 8 or 16-bit CPUs by default.
pfRoutine	Pointer to the routine of type <code>OS_ROUTINE_VOID</code> that shall run as the task body.
pStack	Pointer to an area of memory in RAM that will serve as stack area for the task. The size of this block of memory determines the size of the stack area.
StackSize	Size of stack in bytes.
TimeSlice	Time slice value for round-robin scheduling. Has an effect only if other tasks are running at the same priority. It denotes the time (in embOS system ticks) that the task will run before it suspends, and must be in the following range: $0 \leq \text{TimeSlice} \leq 255$.

Additional information

`OS_TASK_Create()` creates a task and makes it ready for execution. The newly created task will be activated by the scheduler as soon as there is no other task with higher priority ready for execution.

`OS_TASK_Create()` can be called either from `main()` during initialization or from any other task. The recommended strategy is to create all tasks during initialization in `main()` to keep the structure of your application easy to maintain.

The absolute value of [Priority](#) is of no importance, only the value in comparison to the priorities of other tasks matters. If there is another task with the same priority, the new task will be scheduled prior to the existing one.

In embOS builds that do not support round-robin, unique priorities must be assigned to each individual task.

The stack indicated by [pStack](#) must reside in an area that the CPU can address as stack. Most CPUs cannot use the entire memory area as stack and require the stack to be aligned to a multiple of the processor word size.

A [TimeSlice](#) value of zero is allowed and disables round-robin task switches (see sample in chapter *Disabling preemptive task switches for tasks of equal priority* on page 62).

Note

With embOS-MPU `OS_MPU_ConfigMem()` must be called before creating any task.

Note

Up until embOS V5.8.2, `OS_TASK_Create()` expected the task name and time-slice parameters to be omitted in `OS_LIBMODE_XR`. From embOS V5.10.0 onward, `OS_TASK_Create()` expects all parameters to be present independent of the library mode. This means existing applications which call `OS_TASK_Create()` in `OS_LIBMODE_XR` need to be updated accordingly.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
120: OS_ERR_TASK_STACK
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
165: OS_ERR_INIT_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
170: OS_ERR_2USE_TASK
202: OS_ERR_TASK_PRIORITY
```

For details, refer to the chapter *Runtime application errors* on page 455.

Note

embOS offers a macro that calls `OS_TASK_Create()` with two pre-defined parameters, `OS_TASK_CREATE()`, allowing to more easily create tasks. `OS_TASK_CREATE()` determines the value of [StackSize](#) automatically using `sizeof()`. This is possible only if the memory area has been defined at compile time. Furthermore, `OS_TASK_CREATE()` uses a default [TimeSlice](#) of 2. If the macro shall be used, its definition is as follows:

```
#define OS_TASK_CREATE(pTask, pName, Priority, pRoutine, pStack) \
    OS_TASK_Create((pTask), \
                   (pName), \
                   (OS_PRIO)(Priority), \
                   (pRoutine), \
                   (void OS_STACKPTR*)(pStack), \
                   sizeof(pStack), \
                   2u \
                   )
```

Example

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
    while (1) {
        OS_TASK_Delay(50);
    }
}
```

```
static void LPTask(void) {
    while (1) {
        OS_TASK_Delay(200);
    }
}

int main(void) {
    OS_Init();           // Initialize embOS
    OS_Inithw();        // Initialize required hardware
    OS_TASK_Create(&TCBHP, "HP Task", 100, HPTask, StackHP, sizeof(StackHP), 2);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_Start();         // Start embOS
    return 0;
}
```

3.4.4 OS_TASK_CreateEx()

Description

Creates a new task and passes a parameter to the task.

Prototype

```
void OS_TASK_CreateEx(    OS_TASK*          pTask,
                        const char*       sName,
                        OS_TASK_PRIO     Priority,
                        OS_ROUTINE_VOID_PTR* pfRoutine,
                        void              OS_STACKPTR *pStack,
                        OS_UINT           StackSize,
                        OS_UINT           TimeSlice,
                        void*             pContext);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> .
<code>sName</code>	Pointer to the name of the task. Can be <code>NULL</code> if not used. embOS does not copy the task name, but uses the pointer exclusively. When using an embOS build without task name support, this parameter is ignored.
<code>Priority</code>	<code>Priority</code> of the task. Must be within the following range: $1 \leq \text{Priority} \leq 2^8 - 1 = 0xFF$ for 8/16-bit CPUs $1 \leq \text{Priority} \leq 2^{32} - 1 = 0xFFFFFFFF$ for 32-bit CPUs Higher values indicate higher priorities. The type <code>OS_TASK_PRIO</code> is defined as a 32-bit value for 32-bit CPUs and as an 8-bit value for 8 or 16-bit CPUs by default.
<code>pfRoutine</code>	Pointer to the routine of type <code>OS_ROUTINE_VOID_PTR</code> that shall run as the task body.
<code>pStack</code>	Pointer to an area of memory in RAM that will serve as stack area for the task. The size of this block of memory determines the size of the stack area.
<code>StackSize</code>	Size of stack in bytes.
<code>TimeSlice</code>	Time slice value for round-robin scheduling. Has an effect only if other tasks are running at the same priority. It denotes the time (in embOS system ticks) that the task will run before it suspends, and must be in the following range: $0 \leq \text{TimeSlice} \leq 255$.
<code>pContext</code>	Parameter passed to the created task.

Additional information

This function works the same way as `OS_TASK_Create()`, but allows passing a parameter, `pContext`, to the task. Using a `void` pointer as additional parameter gives the flexibility to pass any kind of data to the task function.

Note

With embOS-MPU `OS_MPU_ConfigMem()` must be called before creating any task.

Note

Up until embOS V5.8.2, `OS_TASK_CreateEx()` expected the task name and time-slice parameters to be omitted in `OS_LIBMODE_XR`. From embOS V5.10.0 onward,

`OS_TASK_CreateEx()` expects all parameters to be present independent of the library mode. This means existing applications which call `OS_TASK_CreateEx()` in `OS_LIB-MODE_XR` need to be updated accordingly.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
120: OS_ERR_TASK_STACK
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
165: OS_ERR_INIT_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
170: OS_ERR_2USE_TASK
202: OS_ERR_TASK_PRIORITY
```

For details, refer to the chapter *Runtime application errors* on page 455.

Note

embOS offers a macro that calls `OS_TASK_CreateEx()` with two pre-defined parameters, `OS_TASK_CREATEEX()`, allowing to more easily create tasks. `OS_TASK_CREATEEX()` determines the value of `StackSize` automatically using `sizeof()`. This is possible only if the memory area has been defined at compile time. Furthermore, `OS_TASK_CREATEEX()` uses a default `TimeSlice` of 2. If the macro shall be used, its definition is as follows:

```
#define OS_TASK_CREATEEX(pTask, pName, Priority, pRoutine, pStack, pContext)
    OS_TASK_CreateEx( (pTask),
                      (pName),
                      (OS_PRIO)(Priority),
                      (pRoutine),
                      (void OS_STACKPTR*)(pStack),
                      sizeof(pStack),
                      2u,
                      (pContext)
                    )
```

Example

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task control blocks

static void Task(void* pContext) {
    while (1) {
        OS_TASK_Delay((int)pContext);
    }
}

int main(void) {
    OS_Init();           // Initialize embOS
    OS_InitHW();        // Initialize required hardware
    OS_TASK_CreateEx(&TCBHP, "HP Task", 100, Task,
                    StackHP, sizeof(StackHP), 2, (void*) 50);
    OS_TASK_CREATEEX(&TCBLP, "LP Task", 50, Task,
                    StackLP, (void*)200);
    OS_Start();         // Start embOS
    return 0;
}
```

3.4.5 OS_TASK_Delay()

Description

Suspends the calling task for a specified amount of system ticks, or waits actively when called from main().

Prototype

```
void OS_TASK_Delay(OS_TIME t);
```

Parameters

Parameter	Description
<code>t</code>	Number of system ticks to delay. Must be within the following range: $0 \leq t \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs $0 \leq t \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs Please note that these are signed values.

Additional information

The parameter `t` specifies the time interval in system ticks during which the task is suspended. The actual delay will be in the following range: $t - 1 \leq \text{delay} \leq t$, depending on when the interrupt for the scheduler occurs. After the expiration of the delay, the task is made ready and activated according to the rules of the scheduler. A delay can be ended prematurely by another task or by an interrupt handler calling `OS_TASK_Wake()`.

If `OS_TASK_Delay()` is called from `main()`, it will actively wait for the timeout to expire. Therefore, interrupts must be enabled.

Note

Up until embOS V4.40 and when called within a critical region, `OS_TASK_Delay()` actively waits for the timeout to expire.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

154: `OS_ERR_INTERRUPT_DISABLED`

160: `OS_ERR_ILLEGAL_IN_ISR`

161: `OS_ERR_ILLEGAL_IN_TIMER`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Hello(void) {
    printf("Hello");
    printf("The next output will occur in 5000 system ticks.\n");
    OS_TASK_Delay(5000);
    printf("Delay is over.\n");
}
```

3.4.6 OS_TASK_DelayUntil()

Description

Suspends the calling task until a specified time, or waits actively when called from main().

Prototype

```
void OS_TASK_DelayUntil(OS_TIME t);
```

Parameters

Parameter	Description
<code>t</code>	Specified time. Must be within the following range: $1 \leq (t - OS_Global.Time) \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs $1 \leq (t - OS_Global.Time) \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs Please note that these are signed values.

Additional information

`OS_TASK_DelayUntil()` suspends the calling task until the global time-variable `OS_Global.Time` (see *OS_Global.Time* on page 588) reaches the specified value. The main advantage of this function is that it avoids potentially accumulating delays. The additional condition towards parameter `t` ensures proper behavior even when an overflow of the embOS system tick timer occurs.

If `OS_TASK_DelayUntil()` is called from `main()`, it will actively wait for the timeout to expire. Therefore, interrupts must be enabled.

Note

Up until embOS V4.40 and when called within a critical region, `OS_TASK_DelayUntil()` actively waits for the timeout to expire.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

154: `OS_ERR_INTERRUPT_DISABLED`

160: `OS_ERR_ILLEGAL_IN_ISR`

161: `OS_ERR_ILLEGAL_IN_TIMER`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
int sec, min;

void TaskShowTime(void) {
    OS_TIME t0;
    t0 = OS_TIME_GetTicks();
    while (1) {
        ShowTime(); // Routine to display time
        t0 += 1000;
        OS_TASK_DelayUntil(t0);
        if (sec < 59) {
            sec++;
        } else {
            sec = 0;
        }
    }
}
```

```
        min++;  
    }  
}
```

If the example above used `OS_TASK_Delay()` instead of `OS_TASK_DelayUntil()`, this could lead to accumulating overhead between delays if `OS_TASK_Delay()` is not called exactly each second (which may e.g. happen if interrupts or higher priority tasks are executed instead). This would cause the simple “clock” to be slow. Using `OS_TASK_DelayUntil()` avoids this accumulating overhead.

3.4.7 OS_TASK_Delay_us()

Description

Actively waits for the given time (in microseconds) to expire.

Prototype

```
void OS_TASK_Delay_us(OS_U16 us);
```

Parameters

Parameter	Description
us	Minimum number of microseconds to wait. Must be within the following range: $1 \leq us \leq 2^{15} - 1 = 0x7FFF$. Please note that these are signed values.

Additional information

This function can be used for short delays. `OS_TASK_Delay_us()` must only be called with interrupts enabled and after `OS_Init()` and `OS_TIME_ConfigSysTimer()` have been called. Furthermore, the embOS system tick timer must be running.

`OS_TASK_Delay_us()` does not suspend the calling task, thus all tasks with lower priority cannot be scheduled while the calling tasks executes `OS_TASK_Delay_us()`.

`OS_TASK_Delay_us()` does not block preemptive task switches and does not disable interrupts, which may extend the configured delay period for potentially unlimited periods of time. Furthermore, returning from `OS_TASK_Delay_us()` adds some overhead to the configured delay period. For these reasons, the actual delay period may not necessarily be exact and the function guarantees a minimum delay only.

Note

For embOS V5.06 and later `OS_TIME_ConfigSysTimer()` must be called before calling `OS_TASK_Delay_us()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
158: OS_ERR_DELAYUS_INTERRUPT_DISABLED
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Hello(void) {
    printf("Hello");
    printf("The next output will occur in 500 microseconds.\n");
    OS_TASK_Delay_us(500);
    printf("Delay is over.\n");
}
```

3.4.8 OS_TASK_GetID()

Description

Returns a pointer to the task control block structure of the currently scheduled task. This pointer is unique for the task and is used as a task Id.

Prototype

```
OS_TASK *OS_TASK_GetID(void);
```

Return value

= NULL No task is executing.
≠ NULL Pointer to the task control block of the currently running task.

Additional information

When called from a task, this function may be used for determining which task is currently executing. This can be helpful if the action(s) of a function depend(s) on which task is executing it.

If called from an interrupt service routine, this function may be used to determine the interrupted task (if any).

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintCurrentTaskID(void) {  
    OS_TASK* pTask;  
    pTask = OS_TASK_GetID();  
    printf("Task ID 0x%x\n", pTask);  
}
```

3.4.9 OS_TASK_GetName()

Description

Returns a pointer to the name of the specified task.

Prototype

```
char *OS_TASK_GetName(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

A pointer to the name of the task. `NULL` indicates that the task has no name. If `NULL` is passed for `pTask`, the function returns the name of the running task. If there is no currently running task, the return value is `"OS_Idle()"`. If `pTask` is not `NULL` it must specify a valid task.

When using an embOS build without task name support, `OS_TASK_GetName()` returns `"n/a"` in any case. The embOS `OS_LIBMODE_XR` library mode does not support task names.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintTaskName(void) {
    char* s;
    s = OS_TASK_GetName(NULL);
    printf("Task name: %s\n", s);
}
```

3.4.10 OS_TASK_GetNumTasks()

Description

Returns the number of tasks.

Prototype

```
int OS_TASK_GetNumTasks(void);
```

Return value

Number of tasks.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintNumberOfTasks(void) {  
    int NumTasks;  
    NumTasks = OS_TASK_GetNumTasks();  
    printf("Number of tasks %d\n", NumTasks);  
}
```

3.4.11 OS_TASK_GetPriority()

Description

Returns the task priority of the specified task.

Prototype

```
OS_TASK_PRIO OS_TASK_GetPriority(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

Priority of the specified task (range 1 to 255 for 8/16-bit CPUs and up to 4294967295 for 32-bit CPUs).

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintPriority(const OS_TASK* pTask) {
    OS_PRIO Prio;
    Prio = OS_TASK_GetPriority(pTask);
    printf("Priority of task 0x%x = %u\n", pTask, Prio);
}
```

3.4.12 OS_TASK_GetStatus()

Description

Returns the current task status of the the specified task.

Prototype

```
OS_TASK_STATUS OS_TASK_GetStatus(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

Task status.

Possible return values are:

```
READY_FOR_EXECUTION
DELAYED
WAITS_FOR_TASKEVENT
WAITS_FOR_TASKEVENT_WITH_TIMEOUT
WAITS_FOR_MUTEX
WAITS_FOR_MUTEX_WITH_TIMEOUT
WAITS_FOR_COMMUNICATION
WAITS_FOR_SEMAPHORE
WAITS_FOR_SEMAPHORE_WITH_TIMEOUT
WAITS_FOR_MEMPOOL
WAITS_FOR_MEMPOOL_WITH_TIMEOUT
WAITS_FOR_MESSAGE_IN_QUEUE
WAITS_FOR_MESSAGE_IN_QUEUE_WITH_TIMEOUT
WAITS_FOR_SPACE_IN_MAILBOX
WAITS_FOR_SPACE_IN_MAILBOX_WITH_TIMEOUT
WAITS_FOR_MESSAGE_IN_MAILBOX
WAITS_FOR_MESSAGE_IN_MAILBOX_WITH_TIMEOUT
WAITS_FOR_EVENTOBJECT
WAITS_FOR_EVENTOBJECT_WITH_TIMEOUT
WAITS_FOR_SPACE_IN_QUEUE
WAITS_FOR_SPACE_IN_QUEUE_WITH_TIMEOUT
WAITS_FOR_MULTIPLE_OBJECTS
WAITS_FOR_MULTIPLE_OBJECTS_WITH_TIMEOUT
RUNNING
SUSPENDED
```

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintTaskStatus(void) {
    OS_TASK_STATUS status;

    status = OS_TASK_GetStatus(&TCB);
    printf("Task status: %u\n", status);
}
```

3.4.13 OS_TASK_GetSuspendCnt()

Description

Returns the suspension count and thus suspension state of the specified task. This function may be used to examine whether a task is suspended by previous calls of `OS_TASK_Suspend()`.

Prototype

```
OS_U8 OS_TASK_GetSuspendCnt(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

Suspension count of the specified task.

- = 0 Task is not suspended.
- > 0 Task is suspended by at least one call of `OS_TASK_Suspend()`.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void ResumeTask(OS_TASK* pTask) {
    OS_U8 SuspendCnt;
    SuspendCnt = OS_TASK_GetSuspendCnt(pTask);
    while (SuspendCnt > 0u) {
        OS_TASK_Resume(pTask); // May cause a task switch
        SuspendCnt--;
    }
}
```

3.4.14 OS_TASK_GetTimeSliceRem()

Description

Returns the remaining time slice value of the specified task in system ticks.

Prototype

```
OS_U8 OS_TASK_GetTimeSliceRem(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

Remaining time slice value of the task in system ticks.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

The return value is valid only when using an embOS build with round-robin support. In all other builds it will be 0. The embOS `OS_LIBMODE_XR` library mode does not support round-robin.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`128: OS_ERR_INV_TASK`

`164: OS_ERR_OS_INT_ENTER_NOT_CALLED`

`167: OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintRemainingTimeSlices(void) {
    OS_U8 slices;

    slices = OS_TASK_GetTimeSliceRem(NULL);
    printf("Remaining Time Slices: %d\n", slices);
}
```

3.4.15 OS_TASK_IsTask()

Description

Determines whether the specified task control block belongs to a valid task.

Prototype

```
OS_BOOL OS_TASK_IsTask(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> .

Return value

= 0 TCB is not used by any task.
 ≠ 0 TCB is used by a task.

Additional information

This function checks if the specified task is present in the internal task list. When a task is terminated it is removed from the internal task list. In applications that create and terminate tasks dynamically, this function may be useful to determine whether the task control block and stack for one task may be reused for another task.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintTCBStatus(OS_TASK* pTask) {
    OS_BOOL b;

    b = OS_TASK_IsTask(pTask);
    if (b == 0u) {
        printf("TCB can be reused for another task.\n");
    } else {
        printf("TCB refers to a valid task.\n");
    }
}
```

3.4.16 OS_TASK_Index2Ptr()

Description

Returns the task control block of the task with the specified Index.

Prototype

```
OS_TASK *OS_TASK_Index2Ptr(int TaskIndex);
```

Parameters

Parameter	Description
<code>TaskIndex</code>	Index of a task control block in the task list. This is a zero based index. <code>TaskIndex</code> 0 identifies the first task control block.

Return value

= NULL No task control block with this index found.
 ≠ NULL Pointer to the task control block with the index `TaskIndex`.

Additional information

The order of the tasks in the task list is not defined by the order of task creation and changes at runtime. Tasks are ordered in the task list by their task priority. Therefore `OS_TASK_Index2Ptr()` is usually used only to iterate through the complete task list (e.g. to print all task names).

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintAllTaskNames(void) {
    OS_TASK* pTask;
    int      TaskIndex;

    TaskIndex = 0;
    OS_TASK_EnterRegion();
    do {
        pTask = OS_TASK_Index2Ptr(TaskIndex);
        if (pTask != NULL) {
            printf("%s\n", pTask->sName);
        }
        TaskIndex++;
    } while (pTask != NULL);
    OS_TASK_LeaveRegion();
}
```

3.4.17 OS_TASK_RemoveAllTerminateHooks()

Description

Removes all hook functions from the `OS_ON_TERMINATE_HOOK` list which contains the list of functions that are called when a task is terminated.

Prototype

```
void OS_TASK_RemoveAllTerminateHooks(void);
```

Additional information

`OS_TASK_RemoveAllTerminateHooks()` removes all hook functions which were previously added by `OS_TASK_AddTerminateHook()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
OS_ON_TERMINATE_HOOK _TerminateHook;

void TerminateHookFunc(OS_CONST_PTR OS_TASK* pTask) {
    // This function is called when OS_TASK_Terminate() is called.
    if (pTask == &MyTask) {
        free(MytaskBuffer);
    }
}
...
int main(void) {
    OS_TASK_AddTerminateHook(&_TerminateHook, TerminateHookFunc);
    OS_TASK_RemoveAllTerminateHooks();
    ...
}
```

3.4.18 OS_TASK_RemoveTerminateHook()

Description

OS_TASK_RemoveTerminateHook() removes the specified hook function from the OS_ON_TERMINATE_HOOK list which contains the list of functions that are called when a task is terminated.

Prototype

```
void OS_TASK_RemoveTerminateHook(OS_CONST_PTR OS_ON_TERMINATE_HOOK *pHook);
```

Parameters

Parameter	Description
pHook	Pointer to a variable of type OS_ON_TERMINATE_HOOK.

Additional information

OS_TASK_RemoveTerminateHook() removes the specified hook routine which was previously added by OS_TASK_AddTerminateHook().

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
OS_ON_TERMINATE_HOOK _TerminateHook;

void TerminateHookFunc(OS_CONST_PTR OS_TASK* pTask) {
    // This function is called when OS_TASK_Terminate() is called.
    if (pTask == &MyTask) {
        free(MytaskBuffer);
    }
}
...
int main(void) {
    OS_TASK_AddTerminateHook(&_TerminateHook, TerminateHookFunc);
    OS_TASK_RemoveTerminateHook(&_TerminateHook);
    ...
}
```

3.4.19 OS_TASK_Resume()

Description

Resumes the specified suspended task.

Prototype

```
void OS_TASK_Resume(OS_TASK* pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> .

Additional information

`OS_TASK_Resume()` decrements the specified task's suspend count. When the resulting value is zero, the execution of the specified task is resumed.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
161: OS_ERR_ILLEGAL_IN_TIMER
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
201: OS_ERR_RESUME_BEFORE_SUSPEND
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example of `OS_TASK_Suspend()`.

3.4.20 OS_TASK_ResumeAll()

Description

Decrements the suspend count of all tasks that have a nonzero suspend count and resumes these tasks when their respective suspend count reaches zero.

Prototype

```
void OS_TASK_ResumeAll(void);
```

Additional information

This function may be helpful to synchronize or start multiple tasks at the same time. The function resumes all tasks, no specific task must be addressed. The function may be used together with the functions `OS_TASK_SuspendAll()` and `OS_TASK_SetInitialSuspendCnt()`.

The function may cause a task switch when a task with higher priority than the calling task is resumed. The task switch will be executed after all suspended tasks are resumed.

The function may be called even when no task is suspended.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

161: OS_ERR_ILLEGAL_IN_TIMER

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example of `OS_TASK_SetInitialSuspendCnt()`.

3.4.21 OS_TASK_SetContextExtension()

Description

Makes global variables or processor registers task-specific. The function may be used for a variety of purposes. Typical applications are:

- Global variables such as "errno" in the C library, making the C-lib functions thread-safe.
- Additional, optional CPU / registers such as MAC / EMAC registers (multiply and accumulate unit) if they are not saved in the task context per default.
- Coprocessor registers such as registers of a VFP (floating-point co-processor).
- Data registers of an additional hardware unit such as a CRC calculation unit.

This allows the user to extend the task context as required. A major advantage is that the task extension is task-specific. This means that the additional information (such as floating-point registers) needs to be saved only by tasks that actually use these registers. The advantage is that the task switching time of other tasks is not affected. The same is true for the required stack space: Additional stack space is required only for the tasks which actually save the additional registers.

Prototype

```
void OS_TASK_SetContextExtension
(OS_CONST_PTR OS_EXTEND_TASK_CONTEXT *pExtendContext);
```

Parameters

Parameter	Description
<code>pExtendContext</code>	Pointer to the <code>OS_EXTEND_TASK_CONTEXT</code> structure which contains the addresses of the specific save and restore functions that save and restore the extended task context during task switches.

Additional information

The `OS_EXTEND_TASK_CONTEXT` structure is defined as follows:

```
typedef struct OS_EXTEND_TASK_CONTEXT {
void* (*pfSave) ( void* pStack);
void* (*pfRestore)(const void* pStack);
} OS_EXTEND_TASK_CONTEXT;
```

Note

In embOS V4.16 and earlier the `OS_EXTEND_TASK_CONTEXT` structure was defined as follows:

```
typedef struct OS_EXTEND_TASK_CONTEXT_STRUCT {
void (*pfSave) ( void OS_STACKPTR * pStack);
void (*pfRestore)(const void OS_STACKPTR * pStack);
} OS_EXTEND_TASK_CONTEXT;
```

The Save/Restore functions did not return the stack pointer. When updating from embOS V4.16 and earlier to embOS V4.20 and later please update your Save/Restore functions accordingly.

`pExtendContext`, `pExtendContext->pfSave` and `pExtendContext->pfRestore` must not be NULL. The save and restore functions must be declared according the function type used in the structure. The sample below shows how the task stack must be addressed to save and restore the extended task context.

The embOS kernel pushes during the context switch the task context on the task stack. The stack pointer value after the push operation is stored in the task control block and passed to the Save routine. The Save routine pushes the extended task context data onto the task stack. To do so it must reserve the necessary bytes on the task stack and return the adjusted stack pointer. The adjusted stack pointer is used for the next task context extension Save routine (if any).

The Restore routines are executed in the same order. For example task context extensions A, B and C are used. When the kernel saves the task context the Save routines of task context extensions A, B and C are called. When the kernel restores the task context the Restore routines of the task context extensions A, B and C are called in the same order.

The embOS `OS_LIBMODE_XR` library mode does not support task context extension.

Note

The task context can be extended only once per task with `OS_TASK_SetContextExtension()`. The function must not be called multiple times for one task. Additional task context extensions can be set with `OS_TASK_AddContextExtension()`.

Note

embOS interrupts must not be enabled in the save and restore functions.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
116: OS_ERR_EXTEND_CONTEXT
128: OS_ERR_INV_TASK
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```

#include "RTOS.h"

//
// Custom structure with task context extension.
// In this case, the extended task context consists of just
// a single member, which is a global variable.
//
typedef struct {
    int GlobalVar;
} CONTEXT_EXTENSION;

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task control blocks
static int          GlobalVar;

static void OS_STACKPTR* _Save(void OS_STACKPTR* pStack) {
    CONTEXT_EXTENSION* p;

    p = (CONTEXT_EXTENSION*)pStack;
    #if (OS_STACK_GROWS_TOWARD_HIGHER_ADDR == 1)
        p++;
    #else
        p--;
    #endif
    p->GlobalVar = GlobalVar;
    return (void OS_STACKPTR*)p;
}

static void OS_STACKPTR* _Restore(const void OS_STACKPTR* pStack) {
    const CONTEXT_EXTENSION* p;

    p = (CONTEXT_EXTENSION*)pStack;
    #if (OS_STACK_GROWS_TOWARD_HIGHER_ADDR == 1)
        p++;
    #else
        p--;
    #endif
    GlobalVar = p->GlobalVar;
    return (void OS_STACKPTR*)p;
}

const OS_EXTEND_TASK_CONTEXT _SaveRestore = {
    _Save, // Function pointer to save the task context
    _Restore // Function pointer to restore the task context
};

static void HPTask(void) {
    OS_TASK_SetContextExtension(&_SaveRestore);
    GlobalVar = 1;
    while (1) {
        OS_TASK_Delay(10);
    }
}

static void LPTask(void) {
    OS_TASK_SetContextExtension(&_SaveRestore);
    GlobalVar = 2;
    while (1) {
        OS_TASK_Delay(50);
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_InitHW(); // Initialize required hardware
}

```

```
OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);  
OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);  
OS_Start();      // Start embOS  
return 0;  
}
```

3.4.22 OS_TASK_SetDefaultContextExtension()

Description

Sets the specified context extension as the default task context extension.

Prototype

```
void OS_TASK_SetDefaultContextExtension
(OS_CONST_PTR OS_EXTEND_TASK_CONTEXT *pExtendContext);
```

Parameters

Parameter	Description
<code>pExtendContext</code>	Pointer to the <code>OS_EXTEND_TASK_CONTEXT</code> structure which contains the addresses of the specific save and restore functions that save and restore the extended task context during task switches.

Additional information

After calling this function all newly started tasks will automatically use this context extension. The same task context extension is used for all tasks.

`pExtendContext`, `pExtendContext->pfSave` and `pExtendContext->pfRestore` must not be NULL. An embOS debug build calls `OS_Error(OS_ERR_EXTEND_CONTEXT)` when one of the function pointers is NULL).

Note

embOS interrupts must not be enabled in the save and restore functions.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
116: OS_ERR_EXTEND_CONTEXT
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
extern const OS_EXTEND_TASK_CONTEXT _SaveRestore;

int main(void) {
    OS_Init();           // Initialize embOS
    OS_Inithw();        // Initialize required hardware
    OS_TASK_SetDefaultContextExtension(&_SaveRestore);
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_Start();         // Start embOS
    return 0;
}
```

3.4.23 OS_TASK_SetDefaultStartHook()

Description

Sets a default hook routine which is executed before a task starts. May be used to perform additional initialization for newly created tasks.

Prototype

```
void OS_TASK_SetDefaultStartHook(OS_ROUTINE_VOID* pfRoutine);
```

Parameters

Parameter	Description
<code>pfRoutine</code>	Pointer to the routine of type <code>OS_ROUTINE_VOID</code> which shall be called when a task is started.

Additional information

After calling `OS_TASK_SetDefaultStartHook()` all newly created tasks will automatically call this hook routine when the tasks are started for the first time. The same hook routine is used for all tasks. If `NULL` is passed no hook routine gets executed.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void _HookRoutine(void) { // This routine is automatically executed before
    DoSomething();        // HPTask() gets executed
}

void HPTask(void) {
    while (1) {
        OS_TASK_Delay(10);
    }
}

int main(void) {
    OS_Init();           // Initialize embOS
    OS_InitHW();        // Initialize required hardware
    OS_TASK_SetDefaultStartHook(_HookRoutine); // Set task start hook routine
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_Start();         // Start embOS
    return 0;
}
```

3.4.24 OS_TASK_SetInitialSuspendCnt()

Description

Sets the initial suspend count for newly created tasks to 1 or 0. May be used to create tasks which are initially suspended.

Prototype

```
void OS_TASK_SetInitialSuspendCnt(OS_U8 SuspendCnt);
```

Parameters

Parameter	Description
<code>SuspendCnt</code>	1: Tasks will be created in suspended state. 0: Tasks will be created normally, unsuspended.

Additional information

Can be called at any time from `main()`, any task, ISR or software timer. After calling this function with nonzero `SuspendCnt`, all newly created tasks will be automatically suspended with a suspend count of one. This function may be used to inhibit further task switches, which may be useful during system initialization.

Note

When this function is called from `main()` to initialize all tasks in suspended state, at least one task must be resumed before the system is started by a call of `OS_Start()`. The initial suspend count should be reset to allow normal creation of tasks before the system is started.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
```

```
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
//
// High priority task started first after OS_Start().
//
void InitTask(void) {
    OS_TASK_SuspendAll();
    // Prevent execution of all other existing tasks.
    OS_TASK_SetInitialSuspendCnt(1);
    // Prevent execution of subsequently created tasks.
    ... // New tasks may be created, but will not execute.
    ... // Even when InitTask() blocks itself, no other task may execute.
    OS_TASK_SetInitialSuspendCnt(0); // Reset initial suspend count for new tasks.
    OS_TASK_ResumeAll();
    // Resume all tasks that were blocked before or
    // were created in suspended state. May cause a
    // task switch.
    while (1) {
        ... // Do the normal work.
    }
}
```

3.4.25 OS_TASK_SetName()

Description

Set the specified task name for the specified task. Allows modification of a task name at runtime.

Prototype

```
void OS_TASK_SetName( OS_TASK* pTask,
                     const char* sName );
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.
<code>sName</code>	Pointer to a null-terminated string which is used as task name. embOS does not copy the task name, but uses the pointer exclusively.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

When using an embOS build without task name support, `OS_TASK_SetName()` performs no modifications at all. The embOS `OS_LIBMODE_XR` library mode does not support task names.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_TASK_SetName(NULL, "Initializer Task");
    while (1) {
        OS_TASK_Delay(100);
    }
}
```

3.4.26 OS_TASK_SetPriority()

Description

Assigns the specified task priority to the specified task.

Prototype

```
void OS_TASK_SetPriority(OS_TASK*    pTask,
                       OS_TASK_PRIO Priority);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.
<code>Priority</code>	<code>Priority</code> of the task. Must be within the following range: $1 \leq \text{Priority} \leq 2^8 - 1 = 0xFF$ for 8/16-bit CPUs $1 \leq \text{Priority} \leq 2^{32} - 1 = 0xFFFFFFFF$ for 32-bit CPUs Higher values indicate higher priorities. The type <code>OS_TASK_PRIO</code> is defined as 32-bit value for 32-bit CPUs and 8-bit value for 8 or 16-bit CPUs per default.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Calling this function might lead to an immediate task switch.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_TASK_SetPriority(NULL, 20);    // Change priority of this task to 20.
    while (1) {
        OS_TASK_Delay(100);
    }
}
```

3.4.27 OS_TASK_SetTimeSlice()

Description

Assigns the specified time-slice period to a specified task.

Prototype

```
OS_U8 OS_TASK_SetTimeSlice(OS_TASK* pTask,
                           OS_U8   TimeSlice);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.
<code>TimeSlice</code>	New time slice period for the task in system ticks. Must be within the following range: $0 \leq \text{TimeSlice} \leq 255$.

Return value

Previous time slice period of the task in system ticks.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Setting the time slice period only affects tasks running in round-robin mode. The new time slice period is interpreted as a reload value: It is used with the next activation of the task, but does not affect the remaining time slice of a running task.

A time slice value of zero is allowed, but disables round-robin task switches (see *Disabling preemptive task switches for tasks of equal priority* on page 62).

`OS_TASK_SetTimeSlice()` assigns a time-slice only when using an embOS build with round-robin support. The return value is valid only when using an embOS build without round-robin support. The embOS `OS_LIBMODE_XR` library mode does not support round-robin. In all other builds it will be 0.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_TASK_SetTimeSlice(NULL, 4);    // Give this task a higher time slice
    while (1) {
        OS_TASK_Delay(100);
    }
}
```

3.4.28 OS_TASK_Suspend()

Description

Suspends the specified task and increments the task's suspend count.

Prototype

```
void OS_TASK_Suspend(OS_TASK* pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> .

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Execution of the specified task is suspended immediately and the task's suspend count is incremented. The task can only be restarted by a call of `OS_TASK_Resume()` or `OS_TASK_ResumeAll()`.

Every task has a suspend count with a maximum value of 3, thus you must not call `OS_TASK_Suspend()` more often than the maximum value without calling `OS_TASK_Resume()`. `OS_TASK_Suspend()` must not be called from an interrupt handler or software timer as this function may cause an immediate task switch.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
200: OS_ERR_SUSPEND_TOO_OFTEN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void HighPrioTask(void) {
    OS_TASK_Suspend(NULL);
    // Suspends itself, low priority task will be executed
}

void LowPrioTask(void) {
    OS_TASK_Resume(&HighPrioTCB); // Resumes the high priority task
}
```

3.4.29 OS_TASK_SuspendAll()

Description

Suspends all tasks except the running task.

Prototype

```
void OS_TASK_SuspendAll(void);
```

Additional information

This function may be used to inhibit task switches. It may be useful during application initialization or supervising.

The calling task will not be suspended.

After calling `OS_TASK_SuspendAll()`, the calling task may block or suspend itself. No other task will be activated unless one or more tasks are resumed again. The tasks may be resumed individually by a call of `OS_TASK_Resume()` or all at once by a call of `OS_TASK_ResumeAll()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example of `OS_TASK_SetInitialSuspendCnt()`.

3.4.30 OS_TASK_Terminate()

Description

Ends (terminates) the specified task.

Prototype

```
void OS_TASK_Terminate(OS_TASK* pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> . A value of <code>NULL</code> terminates the current task.

Additional information

The specified task will terminate immediately. The memory used for stack and task control block can be reassigned.

All resources which are held by a task are released upon its termination. Any task may be terminated regardless of its state.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_TASK_Terminate(&TCBHP); // Terminate HPTask()
    DoSomething();
    OS_TASK_Terminate(NULL); // Terminate itself
}
```

3.4.31 OS_TASK_Wake()

Description

Ends delay of the specified task immediately.

Prototype

```
void OS_TASK_Wake(OS_TASK* pTask);
```

Parameters

Parameter	Description
pTask	Pointer to a task control block of type OS_TASK.

Additional information

Places the specified task, which is already suspended for a certain amount of time by a call of OS_TASK_Delay() or OS_TASK_DelayUntil(), back into the READY state.

The specified task will be activated immediately if it has a higher priority than the task that had the highest priority before. If the specified task is not in the WAITING state (e.g. when it has already been activated, or the delay has already expired, or for some other reason), calling this function has no effect.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
161: OS_ERR_ILLEGAL_IN_TIMER
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
    while (1) {
        OS_TASK_Delay(50);
    }
}

static void LPTask(void) {
    while (1) {
        OS_TASK_Delay(10);
        OS_TASK_Wake(&TCBHP); // Wake HPTask() which is in delay state
    }
}

/*****
 *
 *     main()
 */
int main(void) {
    OS_Init();           // Initialize embOS
    OS_InitHW();        // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
}
```

```
OS_Start();    // Start embOS
return 0;
}
```

3.4.32 OS_TASK_Yield()

Description

Calls the scheduler to force a task switch.

Prototype

```
void OS_TASK_Yield(void);
```

Additional information

If the task is running on round-robin, it will be suspended if there is another task with equal priority ready for execution.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
    while (1) {
        DoSomething();
    }
}

static void LPTask(void) {
    while (1) {
        DoSomethingElse();
        //
        // This task doesn't need the complete time slice.
        // Give another task with the same priority the chance to run
        //
        OS_TASK_Yield();
    }
}

/*****
 *
 *     main()
 */
int main(void) {
    OS_Init();           // Initialize embOS
    OS_InitHW();        // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 100, LPTask, StackLP);
    OS_Start();         // Start embOS
    return 0;
}
```

Chapter 4

Software Timer

4.1 Introduction

A software timer is an object that calls a user-specified routine after a specified delay. An unlimited number of software timers can be created.

embOS software timers can be stopped, started and re-triggered much like hardware timers. When defining a timer, you specify a routine to be called after the expiration of the delay. Timer routines are similar to interrupt routines: they have a priority higher than the priority of any task. For that reason they should be kept short just like interrupt routines.

Software timers are called by embOS with interrupts enabled, so they can be interrupted by any hardware interrupt. But software timers run to completion and cannot interrupt each other or be interrupted by a preemptive task switch. Generally, software timer run in single-shot mode, which means they expire exactly once and call their callback routine exactly once. By calling `OS_TIMER_Restart()` from within the callback routine, the timer is restarted with its initial delay time and therefore functions as a periodic timer.

The state of timers can be checked by the functions `OS_TIMER_GetStatus()`, `OS_TIMER_GetRemainingPeriod()` and `OS_TIMER_GetPeriod()`.

Example

```
#include "RTOS.h"
#include "BSP.h"

static OS_TIMER Timer0, Timer1;

static void Callback0(void) {
    BSP_ToggleLED(0);
    OS_TIMER_Restart(&Timer0);
}

static void Callback1(void) {
    BSP_ToggleLED(1);
    OS_TIMER_Restart(&Timer1);
}

int main(void) {
    OS_Init();           // Initialize embOS
    OS_InitHW();        // Initialize required hardware
    BSP_Init();         // Initialize LED ports
    OS_TIMER_CREATE(&Timer0, Callback0, 50u);
    OS_TIMER_CREATE(&Timer1, Callback1, 200u);
    OS_Start();         // Start embOS
    return 0;
}
```

Extended software timers

Sometimes it may be useful to pass a parameter to the timer callback function. This allows the callback function to be shared between different software timers. Since version 3.32m of embOS, the extended timer structure and related extended timer functions were implemented to allow parameter passing to the callback function. Except for the different callback function with parameter passing, extended timers behave exactly the same as regular embOS software timers and may be used in parallel with these.

Example

```
#include "RTOS.h"
#include "BSP.h"

static OS_TIMER_EX Timer0, Timer1;

static void Callback(void* Led) {
    BSP_ToggleLED((int)Led);
}
```

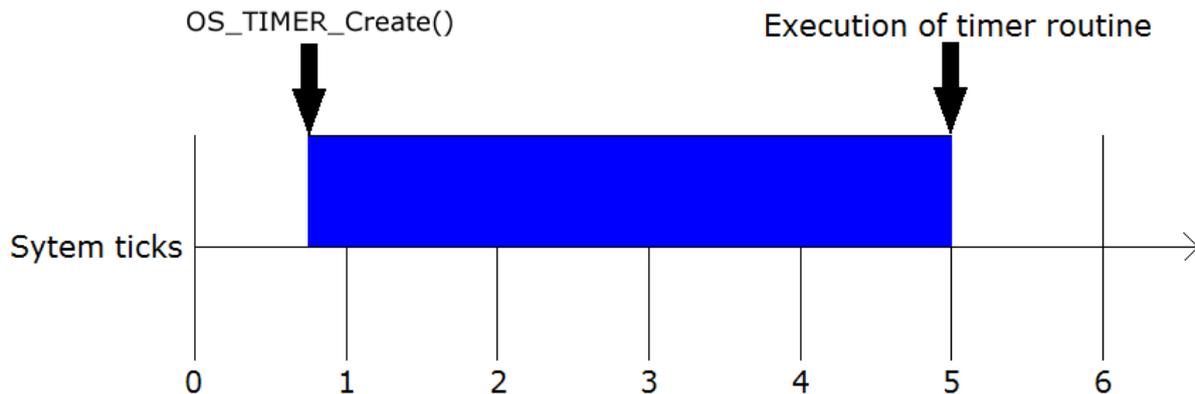
```
    OS_TIMER_RestartEx(OS_TIMER_GetCurrentEx());
}

int main(void) {
    OS_Init();           // Initialize embOS
    OS_Inithw();        // Initialize required hardware
    BSP_Init();         // Initialize LED ports
    OS_TIMER_CREATEEX(&Timer0, Callback, 50u, (void*)0);
    OS_TIMER_CREATEEX(&Timer1, Callback, 200u, (void*)1);
    OS_Start();         // Start embOS
    return 0;
}
```

Minimum timeout / period

Software timer periods elapse with the appropriate embOS system tick. This means that the actual timeout period can actually be slightly shorter than the configured timeout period. For example, if the system tick is configured to occur once every millisecond, and the timer is configured for a timeout of 1, the actual timeout duration is somewhere between 0 and 1 millisecond.

The following diagram illustrates how software timer timeouts work. We can see that the timer configuration is performed prior to the first system tick, that is: at system time 0. The timeout period is configured to 5 system ticks, therefore the callback is called upon the 5th system tick. For example, if the the system ticks occurs at 1 millisecond, 2 millisecond, (...), 5 millisecond, and the timer was started at 0.8 millisecond, the actual timer period would equal 4.2 millisecond.



Maximum timeout / period

The timeout value is stored as an integer, thus a 16-bit value on 8/16-bit CPUs, a 32-bit value on 32-bit CPUs. The comparisons are done as signed comparisons because expired time-outs are permitted. This means that only 15 bits can be used on 8/16-bit CPUs, 31 bits on 32-bit CPUs. Another factor to take into account is the maximum time spent in critical regions. Timers may expire during critical regions, but because the timer routine cannot be called from a critical region (timers are "put on hold"), the maximum time that the system continuously spends in a critical region needs to be deducted. In most systems, this is no more than a single tick. However, to be safe, we have assumed that your system spends no more than a maximum of 255 consecutive system ticks in a critical region and defined a macro for the maximum timeout value. This macro, `OS_TIMER_MAX_TIME`, defaults to `0x7F00` on 8/16-bit systems and to `0x7FFFFFF00` on 32-bit Systems as defined in `RTOS.h`. If your system spends more than 255 consecutive ticks in a critical section, effectively disabling the scheduler during this time (which is not recommended), you must ensure your application uses shorter timeouts.

4.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TIMER_Create()</code>	Creates a software timer without starting it.	•	•		•	•
<code>OS_TIMER_CreateEx()</code>	Creates an extended software timer without starting it.	•	•		•	•
<code>OS_TIMER_Delete()</code>	Stops and deletes the specified software timer.	•	•		•	•
<code>OS_TIMER_DeleteEx()</code>	Stops and deletes an extended software timer.	•	•		•	•
<code>OS_TIMER_GetCurrent()</code>	Returns a pointer to the software timer object whose callback is currently executing.	•	•	•	•	•
<code>OS_TIMER_GetCurrentEx()</code>	Returns a pointer to the data structure of the extended software timer that just expired.	•	•	•	•	•
<code>OS_TIMER_GetPeriod()</code>	Returns the reload value of the specified software timer.	•	•	•	•	•
<code>OS_TIMER_GetPeriodEx()</code>	Returns the current reload value of an extended software timer.	•	•	•	•	•
<code>OS_TIMER_GetRemainingPeriod()</code>	Returns the remaining timer value of the specified software timer.	•	•	•	•	•
<code>OS_TIMER_GetRemainingPeriodEx()</code>	Returns the remaining timer value of an extended software timer.	•	•	•	•	•
<code>OS_TIMER_GetStatus()</code>	Returns the current timer status of the specified software timer.	•	•	•	•	•
<code>OS_TIMER_GetStatusEx()</code>	Returns the current timer status of an extended software timer.	•	•	•	•	•
<code>OS_TIMER_Restart()</code>	Restarts the specified software timer with its initial time value.	•	•	•	•	•
<code>OS_TIMER_RestartEx()</code>	Restarts an extended software timer with its initial time value.	•	•	•	•	•
<code>OS_TIMER_SetPeriod()</code>	Sets a new timer reload value for the specified software timer.	•	•	•	•	•
<code>OS_TIMER_SetPeriodEx()</code>	Sets a new timer reload value for an extended software timer.	•	•	•	•	•
<code>OS_TIMER_Start()</code>	Starts the specified software timer.	•	•	•	•	•
<code>OS_TIMER_StartEx()</code>	Starts an extended software timer.	•	•	•	•	•
<code>OS_TIMER_Stop()</code>	Stops the specified software timer.	•	•	•	•	•
<code>OS_TIMER_StopEx()</code>	Stops an extended software timer.	•	•	•	•	•
<code>OS_TIMER_Trigger()</code>	Ends the specified software timer at once and calls the timer callback function.		•	•	•	
<code>OS_TIMER_TriggerEx()</code>	Ends an extended software timer at once and calls the timer callback function.		•	•	•	

4.2.1 OS_TIMER_Create()

Description

Creates a software timer without starting it.

Prototype

```
void OS_TIMER_Create(OS_TIMER*      pTimer,
                    OS_ROUTINE_VOID* pfTimerRoutine,
                    OS_TIME         Period);
```

Parameters

Parameter	Description
<code>pTimer</code>	Pointer to a software timer object of type <code>OS_TIMER</code> .
<code>pfTimerRoutine</code>	Pointer to the routine of type <code>OS_ROUTINE_VOID</code> which shall be called by the kernel after the timer period has expired.
<code>Period</code>	Initial period in embOS system ticks. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Period} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Period} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Additional information

Once the period has expired, the callback routine will be called immediately (unless the current task is in a critical region or has interrupts disabled). The timer is not automatically started. This must be done explicitly by a call of `OS_TIMER_Start()` or `OS_TIMER_Restart()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
165: OS_ERR_INIT_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
171: OS_ERR_2USE_TIMER
205: OS_ERR_TIMER_PERIOD_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TIMER Timer;

static void Callback(void) {
    BSP_ToggleLED(0);
    OS_TIMER_Restart(&Timer); // Make timer periodic
}

void InitTask(void) {
    OS_TIMER_Create(&Timer, Callback, 100u);
    OS_TIMER_Start(&Timer);
}
```

Note

embOS offers a macro that calls the functions `OS_TIMER_Create()` and `OS_TIMER_Start()` sequentially, allowing to more easily create software timers. As the macro

does “hide” the called functions, however, we typically suggest to call these functions directly. If the macro shall still be used, its definition is as follows:

```
#define OS_TIMER_CREATE(pTimer, cb, Period) \
    OS_TIMER_Create(pTimer, cb, Period); \
    OS_TIMER_Start(pTimer)
```

4.2.2 OS_TIMER_CreateEx()

Description

Creates an extended software timer without starting it.

Prototype

```
void OS_TIMER_CreateEx(OS_TIMER_EX*      pTimerEx,
                      OS_ROUTINE_VOID_PTR* pfTimerRoutine,
                      OS_TIME            Period,
                      void*              pData);
```

Parameters

Parameter	Description
<code>pTimerEx</code>	Pointer to an extended software timer object of type <code>OS_TIMER_EX</code> .
<code>pfTimerRoutine</code>	Pointer to the routine of type <code>OS_ROUTINE_VOID_PTR</code> which shall be called by the kernel after the timer period has expired.
<code>Period</code>	Initial period in embOS system ticks. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Period} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Period} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.
<code>pData</code>	A void pointer which is used as parameter for the extended timer callback function.

Additional information

Once the period is expired, the callback routine will be called immediately (unless the current task is in a critical region or has interrupts disabled). The timer is not automatically started. This must be done explicitly by a call of `OS_TIMER_StartEx()` or `OS_TIMER_RestartEx()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

165: `OS_ERR_INIT_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

171: `OS_ERR_2USE_TIMER`

205: `OS_ERR_TIMER_PERIOD_INVALID`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TIMER_EX TimerEx0, TimerEx1;

static void Callback(void* pData) {
    BSP_ToggleLED((int)pData);
    OS_TIMER_RestartEx(NULL); // Make timer periodic
}

void InitTask(void) {
    OS_TIMER_CreateEx(&TimerEx0, Callback, 50u, (void*)0);
    OS_TIMER_CreateEx(&TimerEx1, Callback, 200u, (void*)1);
    OS_TIMER_StartEx(&TimerEx0);
    OS_TIMER_StartEx(&TimerEx1);
}
```

```
}
```

Note

embOS offers a macro that calls the functions `OS_TIMER_CreateEx()` and `OS_TIMER_StartEx()` sequentially, allowing to more easily create extended software timers. As the macro does “hide” the called functions, however, we typically suggest to call these functions directly. If the macro shall still be used, its definition is as follows:

```
#define OS_TIMER_CREATEEX(pTimer, cb, Period, pData) \
    OS_TIMER_CreateEx(pTimer, cb, Period, pData); \
    OS_TIMER_StartEx(pTimer)
```

4.2.3 OS_TIMER_Delete()

Description

Stops and deletes the specified software timer.

Prototype

```
void OS_TIMER_Delete(OS_TIMER* pTimer);
```

Parameters

Parameter	Description
<code>pTimer</code>	Pointer to a software timer object of type <code>OS_TIMER</code> .

Additional information

The timer is stopped and therefore removed from the linked list of running timers.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

129: `OS_ERR_INV_TIMER`

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TIMER Timer;

void Task(void) {
    //
    // Create and implicitly start timer
    //
    OS_TIMER_CREATE(&Timer, Callback, 100u);
    ...
    //
    // Delete timer
    //
    OS_TIMER_Delete(&Timer);
}
```

4.2.4 OS_TIMER_DeleteEx()

Description

Stops and deletes an extended software timer.

Prototype

```
void OS_TIMER_DeleteEx(OS_TIMER_EX* pTimerEx);
```

Parameters

Parameter	Description
<code>pTimerEx</code>	Pointer to an extended software timer object of type <code>OS_TIMER_EX</code> .

Additional information

The extended software timer is stopped and removed from the linked list of running timers. In debug builds of embOS, the timer is also marked invalid.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

129: `OS_ERR_INV_TIMER`

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TIMER_EX TimerEx;

void Task(void) {
    //
    // Create and implicitly start timer
    //
    OS_TIMER_CREATEEX(&TimerEx, Callback, 100u, (void*)&TCB);
    ...
    //
    // Delete timer
    //
    OS_TIMER_DeleteEx(&TimerEx);
}
```

4.2.5 OS_TIMER_GetCurrent()

Description

Returns a pointer to the software timer object whose callback is currently executing.

Prototype

```
OS_TIMER *OS_TIMER_GetCurrent(void);
```

Return value

= NULL No software timer callback is currently being executed.

≠ NULL Pointer to the software timer object of type OS_TIMER.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#include "RTOS.h"

static OS_TIMER Timer0, Timer1;

static void Callback(void) {
    OS_TIMER* pTimer = OS_TIMER_GetCurrent();
    OS_TIMER_Restart(pTimer); // Make timer periodic
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_Inithw(); // Initialize required hardware
    OS_TIMER_CREATE(&Timer0, Callback, 50u);
    OS_TIMER_CREATE(&Timer1, Callback, 200u);
    OS_Start(); // Start embOS
    return 0;
}
```

4.2.6 OS_TIMER_GetCurrentEx()

Description

Returns a pointer to the extended software timer object whose callback is currently executing.

Prototype

```
OS_TIMER_EX* OS_TIMER_GetCurrentEx(void);
```

Return value

= NULL No software timer callback is currently being executed.
≠ NULL Pointer to the software timer object of type OS_TIMER_EX.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#include "RTOS.h"

static OS_TIMER_EX TimerEx0, TimerEx1;

static void Callback(void* pData) {
    OS_TIMER* pTimerEx = OS_TIMER_GetCurrentEx();
    OS_TIMER_RestartEx(pTimerEx); // Make timer periodic
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_Inithw(); // Initialize required hardware
    OS_TIMER_CREATEEX(&TimerEx0, Callback, 50u, (void*)0);
    OS_TIMER_CREATEEX(&TimerEx1, Callback, 200u, (void*)1);
    OS_Start(); // Start embOS
    return 0;
}
```

4.2.7 OS_TIMER_GetPeriod()

Description

Returns the reload value of the specified software timer.

Prototype

```
OS_TIME OS_TIMER_GetPeriod(OS_CONST_PTR OS_TIMER *pTimer);
```

Parameters

Parameter	Description
<code>pTimer</code>	Pointer to a software timer object of type <code>OS_TIMER</code> .

Return value

The returned value is the reload value of a software timer.

Additional information

The period returned is the reload value of the timer which was set as initial value when the timer was created or which was modified by a call of `OS_TIMER_SetPeriod()`. This reload value will be used as time period when the timer is retrigged by `OS_TIMER_Restart()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
129: OS_ERR_INV_TIMER  
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void PrintPeriod((OS_TIMER* pTimer) {  
    int period;  
  
    period = OS_TIMER_GetPeriod(pTimer);  
    printf("Period %u\n", period);  
}
```

4.2.8 OS_TIMER_GetPeriodEx()

Description

Returns the current reload value of an extended software timer.

Prototype

```
OS_TIME OS_TIMER_GetPeriodEx(OS_CONST_PTR OS_TIMER_EX* pTimerEx);
```

Parameters

Parameter	Description
<code>pTimerEx</code>	Pointer to an extended software timer object of type <code>OS_TIMER_EX</code> .

Return value

The returned value is the current reload value of an extended software timer.

Additional information

The period returned is the reload value of the timer which was set as initial value when the timer was created or which was modified by a call of `OS_TIMER_SetPeriodEx()`. This reload value will be used as time period when the timer is re-triggered by `OS_TIMER_RestartEx()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
129: OS_ERR_INV_TIMER
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void PrintPeriod(OS_TIMER_EX* pTimerEx) {
    int period;

    period = OS_TIMER_GetPeriodEx(pTimerEx);
    printf("Period %u\n", period);
}
```

4.2.9 OS_TIMER_GetRemainingPeriod()

Description

Returns the remaining timer value of the specified software timer.

Prototype

```
OS_TIME OS_TIMER_GetRemainingPeriod(OS_CONST_PTR OS_TIMER *pTimer);
```

Parameters

Parameter	Description
<code>pTimer</code>	Pointer to a software timer object of type <code>OS_TIMER</code> .

Return value

The returned timer value is the remaining timer time in embOS system ticks until expiration of the timer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

129: OS_ERR_INV_TIMER

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void PrintRemainingPeriod(OS_TIMER* pTimer) {
    int period;

    period = OS_TIMER_GetRemainingPeriod(pTimer);
    printf("Remaining %u\n", period);
}
```

4.2.10 OS_TIMER_GetRemainingPeriodEx()

Description

Returns the remaining timer value of an extended software timer.

Prototype

```
OS_TIME OS_TIMER_GetRemainingPeriodEx(OS_CONST_PTR OS_TIMER_EX* pTimerEx);
```

Parameters

Parameter	Description
<code>pTimerEx</code>	Pointer to an extended software timer object of type <code>OS_TIMER_EX</code> .

Return value

The returned time value is the remaining timer value in embOS system ticks until expiration of the extended software timer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

129: OS_ERR_INV_TIMER

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void PrintRemainingPeriod(OS_TIMER_EX* pTimer) {
    int period;

    period = OS_TIMER_GetRemainingPeriodEx(pTimer);
    printf("Remaining %u\n", period);
}
```

4.2.11 OS_TIMER_GetStatus()

Description

Returns the current timer status of the specified software timer.

Prototype

```
OS_BOOL OS_TIMER_GetStatus(OS_CONST_PTR OS_TIMER *pTimer);
```

Parameters

Parameter	Description
<code>pTimer</code>	Pointer to a software timer object of type <code>OS_TIMER</code> .

Return value

Denotes whether the specified timer is running or not:

- = 0 Timer has stopped.
- ≠ 0 Timer is running.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
129: OS_ERR_INV_TIMER  
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void PrintStatus(OS_TIMER* pTimer) {  
    if (OS_TIMER_GetStatus(pTimer) == 0) {  
        printf("Timer has stopped");  
    } else {  
        printf("Timer is running");  
    }  
}
```

4.2.12 OS_TIMER_GetStatusEx()

Description

Returns the current timer status of an extended software timer.

Prototype

```
OS_BOOL OS_TIMER_GetStatusEx(OS_CONST_PTR OS_TIMER_EX* pTimerEx);
```

Parameters

Parameter	Description
<code>pTimerEx</code>	Pointer to an extended software timer object of type <code>OS_TIMER_EX</code> .

Return value

Denotes whether the specified timer is running or not:

- = 0 Timer has stopped.
- ≠ 0 Timer is running.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
129: OS_ERR_INV_TIMER  
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void PrintStatus(OS_TIMER_EX* pTimerEx) {  
    if (OS_TIMER_GetStatusEx(pTimerEx) == 0) {  
        printf("Timer has stopped");  
    } else {  
        printf("Timer is running");  
    }  
}
```

4.2.13 OS_TIMER_Restart()

Description

Restarts the specified software timer with its initial time value.

Prototype

```
void OS_TIMER_Restart(OS_TIMER* pTimer);
```

Parameters

Parameter	Description
<code>pTimer</code>	Pointer to a software timer object of type <code>OS_TIMER</code> .

Additional information

`OS_TIMER_Restart()` restarts the software timer using the initial time value programmed at creation of the timer or which was set using the function `OS_TIMER_SetPeriod()`.

`OS_TIMER_Restart()` can be called regardless the state of the timer. A running timer will restart using the full initial time. A timer that was stopped before or had expired will be restarted.

If `NULL` is passed for `pTimer`, the currently running timer is restarted. This can be used from the software timer callback function only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`129: OS_ERR_INV_TIMER`

`164: OS_ERR_OS_INT_ENTER_NOT_CALLED`

`167: OS_ERR_CPU_STATE_ILLEGAL`

`171: OS_ERR_2USE_TIMER`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example for `OS_TIMER_Create()` on page 115.

4.2.14 OS_TIMER_RestartEx()

Description

Restarts an extended software timer with its initial time value.

Prototype

```
void OS_TIMER_RestartEx(OS_TIMER_EX* pTimerEx);
```

Parameters

Parameter	Description
<code>pTimerEx</code>	Pointer to an extended software timer object of type <code>OS_TIMER_EX</code> .

Additional information

`OS_TIMER_RestartEx()` restarts the extended software timer using the initial time value which was programmed at creation of the timer or which was set using the function `OS_TIMER_SetPeriodEx()`. `OS_TIMER_RestartEx()` can be called regardless the state of the timer. A running timer will continue using the full initial time. A timer that was stopped before or had expired will be restarted.

If `NULL` is passed for `pTimer`, the currently running timer is restarted. This can be used from the software timer callback function only. If no timer is currently running, `OS_Error()` is called with the error code `OS_ERR_INV_TIMER`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

129: OS_ERR_INV_TIMER

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example for `OS_TIMER_CreateEx()` on page 117.

4.2.15 OS_TIMER_SetPeriod()

Description

Sets a new timer reload value for the specified software timer.

Prototype

```
void OS_TIMER_SetPeriod(OS_TIMER* pTimer,
                       OS_TIME   Period);
```

Parameters

Parameter	Description
<code>pTimer</code>	Pointer to a software timer object of type <code>OS_TIMER</code> .
<code>Period</code>	Timer period in embOS system ticks. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Period} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Period} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Additional information

`OS_TIMER_SetPeriod()` sets the initial time value of the specified software timer. `Period` is the reload value of the timer to be used as initial value when the timer is retriggered by `OS_TIMER_Restart()`.

A call of `OS_TIMER_SetPeriod()` does not affect the remaining time period of a software timer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
129: OS_ERR_INV_TIMER
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
205: OS_ERR_TIMER_PERIOD_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TIMER Timer;

static void TimerPulse(void) {
    TogglePulseOutput(); // Toggle output
    OS_TIMER_Restart(&Timer); // Make timer periodic
}

void InitTask(void) {
    //
    // Create and implicitly start timer with first pulse in 500 system ticks
    //
    OS_TIMER_CREATE(&Timer, TimerPulse, 500u);
    //
    // Set timer period to 200 system ticks for further pulses
    //
    OS_TIMER_SetPeriod(&Timer, 200u);
}
```

4.2.16 OS_TIMER_SetPeriodEx()

Description

Sets a new timer reload value for an extended software timer.

Prototype

```
void OS_TIMER_SetPeriodEx(OS_TIMER_EX* pTimerEx,
                        OS_TIME      Period);
```

Parameters

Parameter	Description
<code>pTimerEx</code>	Pointer to an extended software timer object of type <code>OS_TIMER_EX</code> .
<code>Period</code>	Initial period in embOS system ticks. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Period} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs $1 \leq \text{Period} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs

Additional information

`OS_TIMER_SetPeriodEx()` sets the initial time value of the specified extended software timer. `Period` is the reload value of the timer to be used as initial value when the timer is re-triggered the next time by `OS_TIMER_RestartEx()`.

A call of `OS_TIMER_SetPeriodEx()` does not affect the remaining time period of an extended software timer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
129: OS_ERR_INV_TIMER
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
205: OS_ERR_TIMER_PERIOD_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TIMER_EX Timer;
static OS_TASK      TCB;

static void TimerPulse(void* pTask) {
    if (pTask != NULL) {
        OS_TASKEVENT_Set(0x01, (OS_TASK*)pTask);
    }
    OS_TIMER_RestartEx(&Timer); // Make timer periodic
}

void InitTask(void) {
    //
    // Create and implicitly start Pulse Timer with first pulse in 500 system ticks
    //
    OS_TIMER_CREATEEX(&Timer, TimerPulse, 500, (void*)&TCB);
    //
    // Set timer period to 200 system ticks for further pulses
    //
    OS_TIMER_SetPeriodEx(&Timer, 200);
}
```

4.2.17 OS_TIMER_Start()

Description

Starts the specified software timer.

Prototype

```
void OS_TIMER_Start(OS_TIMER* pTimer);
```

Parameters

Parameter	Description
<code>pTimer</code>	Pointer to a software timer object of type <code>OS_TIMER</code> .

Additional information

`OS_TIMER_Start()` is used for the following reasons:

- Start a timer which was created by `OS_TIMER_Create()`. The timer will start with its initial timer value.
- Restart a timer which was stopped by calling `OS_TIMER_Stop()`. In this case, the timer will continue with the remaining time value which was preserved upon stopping the timer.

Note

This function has no effect on running timers. It also has no effect on timers that are not running, but have expired: use `OS_TIMER_Restart()` to restart those timers.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
129: OS_ERR_INV_TIMER
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
171: OS_ERR_2USE_TIMER
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example for `OS_TIMER_Create()` on page 115.

4.2.18 OS_TIMER_StartEx()

Description

Starts an extended software timer.

Prototype

```
void OS_TIMER_StartEx(OS_TIMER_EX* pTimerEx);
```

Parameters

Parameter	Description
<code>pTimerEx</code>	Pointer to an extended software timer object of type <code>OS_TIMER_EX</code> .

Additional information

`OS_TIMER_StartEx()` is used for the following reasons:

- Start an extended software timer which was created by `OS_TIMER_CreateEx()`. The timer will start with its initial timer value.
- Restart a timer which was stopped by calling `OS_TIMER_StopEx()`. In this case, the timer will continue with the remaining time value which was preserved upon stopping the timer.

Note

This function has no effect on running timers. It also has no effect on timers that are not running, but have expired. Use `OS_TIMER_RestartEx()` to restart those timers.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

129: OS_ERR_INV_TIMER

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example for `OS_TIMER_CreateEx()` on page 117.

4.2.19 OS_TIMER_Stop()

Description

Stops the specified software timer.

Prototype

```
void OS_TIMER_Stop(OS_TIMER* pTimer);
```

Parameters

Parameter	Description
<code>pTimer</code>	Pointer to a software timer object of type <code>OS_TIMER</code> .

Additional information

The actual value of the software timer (the time until expiration) is maintained until `OS_TIMER_Start()` lets the timer continue. The function has no effect on timers that are not running, but have expired.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

129: OS_ERR_INV_TIMER

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TIMER TIMER100;

static void Task(void) {
    OS_TIMER_Restart(&TIMER100); // Start the timer
    ...
    OS_TIMER_Stop(&TIMER100);    // Stop the timer
}
```

4.2.20 OS_TIMER_StopEx()

Description

Stops an extended software timer.

Prototype

```
void OS_TIMER_StopEx(OS_TIMER_EX* pTimerEx);
```

Parameters

Parameter	Description
<code>pTimerEx</code>	Pointer to an extended software timer object of type <code>OS_TIMER_EX</code> .

Additional information

The actual value of the extended software timer (the time until expiration) is maintained until `OS_TIMER_StartEx()` lets the timer continue. The function has no effect on timers that are not running, but have expired.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

129: `OS_ERR_INV_TIMER`

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TIMER_EX TIMER100;

static void Task(void) {
    OS_TIMER_RestartEx(&TIMER100); // Start the timer
    ...
    OS_TIMER_StopEx(&TIMER100);   // Stop the timer
}
```

4.2.21 OS_TIMER_Trigger()

Description

Ends the specified software timer at once and calls the timer callback function.

Prototype

```
void OS_TIMER_Trigger(OS_TIMER* pTimer);
```

Parameters

Parameter	Description
<code>pTimer</code>	Pointer to a software timer object of type <code>OS_TIMER</code> .

Additional information

`OS_TIMER_Trigger()` can be called regardless of the state of the timer. A running timer will be stopped and the callback function is called. For a timer that was stopped before or had expired the callback function will not be executed.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
129: OS_ERR_INV_TIMER
161: OS_ERR_ILLEGAL_IN_TIMER
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TIMER TIMERUartRx;

void TimerUart(void) {
    HandleUartRx();
}

void UartRxIntHandler(void) {
    OS_TIMER_Trigger(&TIMERUartRx); // Character received, stop the software timer
}

void UartSendNextCharachter(void) {
    OS_TIMER_Start(&TIMERUartRx);
    // Send next UART character and wait for Rx character
}

int main(void) {
    OS_TIMER_Create(&TIMERUartRx, TimerUart, 20);
}
```

4.2.22 OS_TIMER_TriggerEx()

Description

Ends an extended software timer at once and calls the timer callback function.

Prototype

```
void OS_TIMER_TriggerEx (OS_TIMER_EX* pTimerEx);
```

Parameters

Parameter	Description
<code>pTimerEx</code>	Pointer to an extended software timer object of type <code>OS_TIMER_EX</code> .

Additional information

`OS_TIMER_TriggerEx()` can be called regardless of the state of the timer. A running timer will be stopped and the callback function is called. For a timer that was stopped before or had expired the callback function will not be executed.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
129: OS_ERR_INV_TIMER
161: OS_ERR_ILLEGAL_IN_TIMER
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TIMER_EX TIMERUartRx;
static OS_U32      UartNum;

void TimerUart(void* pNum) {
    HandleUartRx((OS_U32)pNum);
}

void UartRxIntHandler(void) {
    OS_TIMER_TriggerEx(&TIMERUartRx);
    // Character received, stop the software timer
}

void UartSendNextCharachter(void) {
    OS_TIMER_StartEx(&TIMERUartRx);
    // Send next UART character and wait for Rx character
}

int main(void) {
    UartNum = 0;
    OS_TIMER_CreateEx(&TIMERUartRx, TimerUart, 20, (void*)&UartNum);
}
```

Chapter 5

Task Event

5.1 Introduction

Task events are a way of communicating between a task and another task, software timer or embOS interrupt handler. In contrast to event objects, task events are messages to a single, specified task. In other words, a task event is sent to a specified task.

The purpose of a task event is to enable a task to wait for a particular event (or for one of several events) to occur. This task can be kept inactive until the event is signaled by another task, a software timer or an embOS interrupt handler. An event can be, for example, the change of an input signal, the expiration of a hardware timer, a key press or the reception of a character.

Every task has an individual event bit mask. The width of the bit mask depends on the CPU. The bit mask width with 8/16-bit CPUs is 8 bit. The bit mask width with 32-bit CPUs is 32 bit. This means that 8 or 32 different events can be signaled to and distinguished by every task.

The width of the event bit mask can be modified with the macro `OS_TASKEVENT`. Changing the definition of `OS_TASKEVENT` can only be done when using the embOS sources in a project, or when the libraries are rebuilt from sources with the modified definition. Please have a look in the chapter *Source Code* on page 596 for more details.

Example

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
    OS_TASKEVENT MyEvents;

    while (1) {
        MyEvents = OS_TASKEVENT_GetBlocked(3); // Wait for event bits 0 or 1
        if (MyEvents & 1) {
            _HandleEvent0();
        } else {
            _HandleEvent1();
        }
    }
}

static void LPTask(void) {
    while (1) {
        OS_TASK_Delay(200);
        OS_TASKEVENT_Set(&TCBHP, 1);
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_Inithw(); // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_Start(); // Start embOS
    return 0;
}
```

5.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TASKEVENT_Clear()</code>	Returns the actual state of events and then clears all events of the specified task.	•	•	•	•	•
<code>OS_TASKEVENT_ClearEx()</code>	Returns the actual state of events and then clears the specified events for the specified task.	•	•	•	•	•
<code>OS_TASKEVENT_Get()</code>	Returns a list of events that have occurred for the specified task.	•	•	•	•	•
<code>OS_TASKEVENT_GetBlocked()</code>	Waits for one of the events specified in the bit mask and clears the event memory when the function returns.		•	•		
<code>OS_TASKEVENT_GetSingleBlocked()</code>	Waits for one of the specified events and clears only those events that were specified in the event mask.		•	•		
<code>OS_TASKEVENT_GetSingleTimed()</code>	Waits for one of the specified events for a given time and clears only those events that were specified in the event mask.		•	•		
<code>OS_TASKEVENT_GetTimed()</code>	Waits for the specified events for a given time, and clears all task events when the function returns.		•	•		
<code>OS_TASKEVENT_Set()</code>	Signals event(s) to the specified task.	•	•	•	•	•

5.2.1 OS_TASKEVENT_Clear()

Description

Returns the actual state of events and then clears all events of the specified task.

Prototype

```
OS_TASKEVENT OS_TASKEVENT_Clear(OS_TASK* pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

All events that have been signaled before clearing. If `pTask` is `NULL`, the function clears all events of the currently running task.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_TASKEVENT MyEvents;

    MyEvents = OS_TASKEVENT_Clear(NULL);

    while (1) {
        //
        // Wait for event 0 or 1 to be signaled
        //
        MyEvents = OS_TASKEVENT_GetBlocked(3);
    }
}
```

5.2.2 OS_TASKEVENT_ClearEx()

Description

Returns the actual state of events and then clears the specified events for the specified task.

Prototype

```
OS_TASKEVENT OS_TASKEVENT_ClearEx(OS_TASK*    pTask,
                                   OS_TASKEVENT EventMask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.
<code>EventMask</code>	The bit mask containing the event bits which shall be cleared.

Return value

All events that have been signaled before clearing. If `pTask` is `NULL`, the function clears the events of the currently running task.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_TASKEVENT MyEvents;

    MyEvents = OS_TASKEVENT_ClearEx(NULL, 1);

    while (1) {
        //
        // Wait for event 0 or 1 to be signaled
        //
        MyEvents = OS_TASKEVENT_GetBlocked(3);
    }
}
```

5.2.3 OS_TASKEVENT_Get()

Description

Returns a list of events that have occurred for the specified task.

Prototype

```
OS_TASKEVENT OS_TASKEVENT_Get(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

All events that have been signaled.

Additional information

By calling this function, all events remain signaled: event memory is not cleared. This is one way for a task to query which events are signaled. The task is not suspended if no events are signaled.

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintEvents(void) {
    OS_TASKEVENT MyEvents;

    MyEvents = OS_TASKEVENT_Get(NULL);
    printf("Events %u\n", MyEvents);
}
```

5.2.4 OS_TASKEVENT_GetBlocked()

Description

Waits for one of the events specified in the bit mask and clears the event memory when the function returns.

Prototype

```
OS_TASKEVENT OS_TASKEVENT_GetBlocked(OS_TASKEVENT EventMask);
```

Parameters

Parameter	Description
<code>EventMask</code>	The event bit mask containing the event bits, which shall be waited for.

Return value

All events that have been signaled.

Additional information

If none of the specified events are signaled, the task is suspended. The first of the specified events will wake the task. These events are signaled by another task, a software timer or an interrupt handler. Any bit that is set in the event mask enables the corresponding event.

When a task waits on multiple events, all of the specified events shall be requested by a single call of `OS_TASKEVENT_GetBlocked()` and all events must be handled when the function returns.

Note that all events of the task are cleared when the function returns, even those events that were not set in the parameters in the `EventMask`. The calling function must handle the returned value, otherwise events may get lost. Consecutive calls of `OS_TASKEVENT_GetBlocked()` with different event masks will not work, as all events are cleared when the function returns. If this is not desired, `OS_TASKEVENT_GetSingleBlocked()` may be used instead.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_TASKEVENT MyEvents;

    while(1) {
        //
        // Wait for event 0 or 1 to be signaled
        //
        MyEvents = OS_TASKEVENT_GetBlocked(3);
        //
        // Handle all events
        //
        if (MyEvents & 1) {
            _HandleEvent0();
        }
    }
}
```

```
    }  
    if (MyEvents & 2) {  
        _HandleEvent1();  
    }  
}
```

For another example, see `OS_TASKEVENT_Set()`.

5.2.5 OS_TASKEVENT_GetSingleBlocked()

Description

Waits for one of the specified events and clears only those events that were specified in the event mask.

Prototype

```
OS_TASKEVENT OS_TASKEVENT_GetSingleBlocked(OS_TASKEVENT EventMask);
```

Parameters

Parameter	Description
<code>EventMask</code>	The event bit mask containing the event bits, which shall be waited for and reset.

Return value

All requested events that have been signaled and were specified in the `EventMask`.

Additional information

If none of the specified events are signaled, the task is suspended. The first of the requested events will wake the task. These events are signaled by another task, a software timer, or an interrupt handler. Any bit in the event mask may enable the corresponding event. When the function returns, it delivers all of the requested events. The requested events are cleared in the event state of the task. All other events remain unchanged and will not be returned.

`OS_TASKEVENT_GetSingleBlocked()` may be used in consecutive calls with individual requests. Only requested events will be handled, no other events can get lost. When the function waits on multiple events, the returned value must be evaluated because the function returns when at least one of the requested events was signaled. When the function requests a single event, the returned value does not need to be evaluated.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_TASKEVENT MyEvents;

    while(1) {
        //
        // Wait for event 0 or 1 to be signaled
        //
        MyEvents = OS_TASKEVENT_GetSingleBlocked(3);
        //
        // Handle all events
        //
        if (MyEvents & 1) {
            _HandleEvent0();
        }
        if (MyEvents & 2) {
            _HandleEvent1();
        }
    }
}
```

```
}  
}  
}
```

5.2.6 OS_TASKEVENT_GetSingleTimed()

Description

Waits for one of the specified events for a given time and clears only those events that were specified in the event mask.

Prototype

```
OS_TASKEVENT OS_TASKEVENT_GetSingleTimed(OS_TASKEVENT EventMask,
                                          OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>EventMask</code>	The event bit mask containing the event bits, which shall be waited for and reset.
<code>Timeout</code>	Maximum time in system ticks until the event must be signaled. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 No event available within the specified timeout.
 ≠ 0 All events that have been signaled.

Additional information

If none of the specified events in the event mask are available, the task is suspended for the given time. The first of the specified events will wake the task if the event is signaled by another task, a software timer or an interrupt handler within the specified `Timeout` time.

If no event is signaled within the specified timeout, the calling task gets activated and return zero.

Any bit in the event mask may enable the corresponding event. All unmasked events remain unchanged.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_TASKEVENT MyEvents;

    while(1) {
        //
        // Wait for event 0 and 1 to be signaled within 10 system ticks
        //
        MyEvents = OS_TASKEVENT_GetSingleTimed(3, 10);
        if (MyEvents == 0) {
            _HandleTimeout();
        }
    }
}
```

```
    } else {  
        if (MyEvents & 1) {  
            _HandleEvent0();  
        }  
        if (MyEvents & 2) {  
            _HandleEvent1();  
        }  
    }  
}
```

5.2.7 OS_TASKEVENT_GetTimed()

Description

Waits for the specified events for a given time, and clears all task events when the function returns.

Prototype

```
OS_TASKEVENT OS_TASKEVENT_GetTimed(OS_TASKEVENT EventMask,
                                     OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>EventMask</code>	The event bit mask containing the event bits, which shall be waited for.
<code>Timeout</code>	Maximum time in system ticks until the events must be signaled. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 No event available within the specified timeout.
 ≠ 0 All events that have been signaled.

Additional information

If none of the specified events in the event mask are available, the task is suspended for the given time. The first of the specified events will wake the task if the event is signaled by another task, a software timer or an interrupt handler within the specified `Timeout` time.

If no event is signaled within the specified timeout, the calling task gets activated and return zero.

Note that the function returns all events that were signaled until the task continues execution, even those which were not requested. The calling function must handle the returned value, otherwise events may get lost. Consecutive calls of `OS_TASKEVENT_GetTimed()` with different event masks will not work, as all events are cleared when the function returns. If this is not desired, `OS_TASKEVENT_GetSingleTimed()` may be used instead.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_TASKEVENT MyEvents;

    while(1) {
        // Wait for event 0 and 1 to be signaled within 10 system ticks
        MyEvents = OS_TASKEVENT_GetTimed(3, 10);
        if ((MyEvents & 3) == 0) {
```

```
    _HandleTimeout();  
  } else {  
    if (MyEvents & 1) {  
      _HandleEvent0();  
    }  
    if (MyEvents & 2) {  
      _HandleEvent1();  
    }  
  }  
}  
}
```

5.2.8 OS_TASKEVENT_Set()

Description

Signals event(s) to the specified task.

Prototype

```
void OS_TASKEVENT_Set(OS_TASK* pTask,
                     OS_TASKEVENT Event);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> .
<code>Event</code>	The event bit mask containing the event bits, which shall be signaled.

Additional information

If the specified task is waiting for one of these events, it will be put in the READY state and activated according to the rules of the scheduler.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

The task that handles the serial input and the keyboard waits for a character to be received either via the keyboard (`EVENT_KEYPRESSED`) or serial interface (`EVENT_SERIN`):

```
#define EVENT_KEYPRESSED (1u << 0)
#define EVENT_SERIN      (1u << 1)

static OS_STACKPTR int Stack0[96]; // Task stacks
static OS_TASK      TCB0;         // Data area for tasks (task control blocks)

void Task0(void) {
    OS_TASKEVENT MyEvent;
    while(1)
        MyEvent = OS_TASKEVENT_GetBlocked(EVENT_KEYPRESSED | EVENT_SERIN)
        if (MyEvent & EVENT_KEYPRESSED) {
            // Handle key press
        }
        if (MyEvent & EVENT_SERIN) {
            // Handle serial reception
        }
    }
}

void Key_ISR(void) { // ISR for external interrupt
    OS_TASKEVENT_Set(&TCB0, EVENT_KEYPRESSED); // Notify task that key was pressed
}

void UART_ISR(void) { // ISR for UART interrupt
    OS_TASKEVENT_Set(&TCB0, EVENT_SERIN);
    // Notify task that a character was received
}
```

```
}  
  
void InitTask(void) {  
    OS_TASK_CREATE(&TCB0, "HPTask", 100, Task0, Stack0);  
}
```

Chapter 6

Event Object

6.1 Introduction

Event objects are another type of communication and synchronization object. In contrast to task-events, event objects are standalone objects which are not owned by any task.

The purpose of an event object is to enable one or multiple tasks to wait for a particular event to occur. The tasks can be kept suspended until the event is set by another task, a software timer, or an interrupt handler. An event can be, for example, the change of an input signal, the expiration of a timer, a key press, the reception of a character, or a complete command.

Compared to a task event, the signaling function does not need to know which task is waiting for the event to occur.

Using event object API

There are two groups of event object API functions. The first group does not have "mask" as part of their name and operates on the complete event object. These functions are `OS_EVENT_Get()`, `OS_EVENT_GetBlocked()`, `OS_EVENT_GetTimed()`, `OS_EVENT_Pulse()`, and `OS_EVENT_Set()`. The second group does have "mask" as part of the API name and operates on a event object bit mask. These functions are `OS_EVENT_GetMask()`, `OS_EVENT_GetMaskBlocked()`, `OS_EVENT_GetMaskMode()`, `OS_EVENT_GetMaskTimed()`, `OS_EVENT_SetMask()`, and `OS_EVENT_SetMaskMode()`. Any event object is in non-signaled state when the event object value is zero, and in signaled state when the event object value is unequal to zero. We do not recommend to use both API groups on the same event object. For example, you must not wait for an event object with `OS_EVENT_GetBlocked()` and signal that event object with `OS_EVENT_SetMask()`, but with `OS_EVENT_Set()`.

Reset mode

Since version 3.88a of embOS, the reset behavior of the event can be controlled by different reset modes which may be passed as parameter to the new function `OS_EVENT_CreateEx()` or may be modified by a call of `OS_EVENT_SetResetMode()`.

- `OS_EVENT_RESET_MODE_SEMIAUTO`:
This reset mode is the default mode used with all previous versions of embOS. The reset behavior unfortunately is not consistent and depends on the function called to set or wait for an event. This reset mode is defined for compatibility with older embOS versions (prior version 3.88a). Calling `OS_EVENT_Create()` sets the reset mode to `OS_EVENT_RESET_MODE_SEMIAUTO` to be compatible with older embOS versions.
- `OS_EVENT_RESET_MODE_AUTO`:
This mode sets the reset behavior of an event object to automatic clear. When an event is set, all waiting tasks are resumed and the event is cleared automatically. An exception to this is when a task called `OS_EVENT_GetTimed()` and the timeout expired before the event was signaled, in which case the function returns with timeout and the event is not cleared automatically.
- `OS_EVENT_RESET_MODE_MANUAL`:
This mode sets the event to manual reset mode. When an event is set, all waiting tasks are resumed and the event object remains signaled. The event must be reset by one task which was waiting for the event.

Mask mode

Since version 4.34 of embOS, the mask bits behavior of the event object can be controlled by different mask modes which may be passed to the new function `OS_EVENT_CreateEx()` or may be modified by a call of `OS_EVENT_SetMaskMode()`.

- `OS_EVENT_MASK_MODE_OR_LOGIC`:
This mask mode is the default mode. Only one of the bits specified in the event object bit mask must be signaled.
- `OS_EVENT_MASK_MODE_AND_LOGIC`:
With this mode all specified event object mask bits must be signaled.

6.1.1 Examples

Activate a task from interrupt by an event object

The following code example shows usage of an event object which is signaled from an ISR handler to activate a task. The waiting task should reset the event after waiting for it.

```
static OS_EVENT _Event;

static void _ISRHandler(void) {
    OS_INT_Enter();
    //
    // Wake up task to do the rest of the work
    //
    OS_EVENT_Set(&_Event);
    OS_INT_Leave();
}

static void Task(void) {
    while (1) {
        OS_EVENT_GetBlocked(&_Event);
        //
        // Do the rest of the work (which has not been done in the ISR)
        //
        ...
    }
}
```

Activating multiple tasks using a single event object

The following sample program shows how to synchronize multiple tasks with one event object.

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128], StackHW[128];
static OS_TASK      TCBHP, TCBLP, TCBHW;
static OS_EVENT     HW_Event;

static void HPTask(void) {
    //
    // Wait until HW module is set up
    //
    OS_EVENT_GetBlocked(&HW_Event);
    while (1) {
        OS_TASK_Delay(50);
    }
}

static void LPTask(void) {
    //
    // Wait until HW module is set up
    //
    OS_EVENT_GetBlocked(&HW_Event);
    while (1) {
        OS_TASK_Delay(200);
    }
}

static void HWTask(void) {
    //
    // Wait until HW module is set up
    //
    OS_TASK_Delay(100);
    //
    // Init done, send broadcast to waiting tasks
}
```

```

//
OS_EVENT_Set(&HW_Event);
while (1) {
    OS_TASK_Delay(40);
}
}

int main(void) {
    OS_Init();                // Initialize embOS
    OS_Inithw();             // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_TASK_CREATE(&TCBHW, "HWTTask", 25, HWTTask, StackHW);
    OS_EVENT_Create(&HW_Event);
    OS_Start();              // Start multitasking
    return 0;
}

```

Using event object mask bits

The following sample program shows how to use event object mask bits.

```

#include "RTOS.h"

#define EVENT1_BITMASK (1u << 0)
#define EVENT2_BITMASK (1u << 1)

static OS_STACKPTR int StackTask1[128], StackTask2[128], StackLP[128];
static OS_TASK      TCBTask1, TCBTask2, TCBLP;
static OS_EVENT     _Event;

static void Task1(void) {
    OS_EVENT_GetMaskBlocked(&_Event, EVENT1_BITMASK);
    while (1) {
        OS_TASK_Delay(50);
    }
}

static void Task2(void) {
    OS_EVENT_GetMaskBlocked(&_Event, EVENT2_BITMASK);
    while (1) {
        OS_TASK_Delay(50);
    }
}

static void LPTask(void) {
    OS_EVENT_SetMask(&_Event, EVENT1_BITMASK);
    OS_EVENT_SetMask(&_Event, EVENT2_BITMASK);
    while (1) {
        OS_TASK_Delay(200);
    }
}

int main(void) {
    OS_Init();                // Initialize embOS
    OS_Inithw();             // Initialize required hardware
    OS_TASK_CREATE(&TCBTask1, "Task 1", 100, Task1, StackTask1);
    OS_TASK_CREATE(&TCBTask2, "Task 2", 100, Task2, StackTask2);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_EVENT_Create(&_Event);
    OS_Start();              // Start multitasking
    return 0;
}

```

6.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_EVENT_Create()</code>	Creates an event object.	•	•		•	•
<code>OS_EVENT_CreateEx()</code>	Creates an extended event object and sets its reset behavior as well as mask bits behavior.	•	•		•	•
<code>OS_EVENT_Delete()</code>	Deletes the specified event object.	•	•			
<code>OS_EVENT_Get()</code>	Retrieves the current state of the specified event object.	•	•	•	•	•
<code>OS_EVENT_GetBlocked()</code>	Waits for the specified event object and suspends the task until the event has been signaled.		•	•		
<code>OS_EVENT_GetMask()</code>	Returns the bits of the specified event object that match the given <code>EventMask</code> .	•	•	•		
<code>OS_EVENT_GetMaskBlocked()</code>	Waits for the specified event bits in <code>EventMask</code> , depending on the current mask mode.		•	•		
<code>OS_EVENT_GetMaskMode()</code>	Retrieves the current mask mode (mask bits behavior) of the specified event object.	•	•	•	•	•
<code>OS_EVENT_GetMaskTimed()</code>	Waits for the specified event bits <code>EventMask</code> with timeout, depending on the current mask mode.		•	•		
<code>OS_EVENT_GetResetMode()</code>	Returns the reset mode (reset behavior) of the specified event object.	•	•	•	•	•
<code>OS_EVENT_GetTimed()</code>	Waits for an event and suspends the task for a specified time or until the specified event has been signaled.		•	•		
<code>OS_EVENT_Pulse()</code>	Signals the specified event object and resumes waiting tasks, then resets the event object to non-signaled state.	•	•	•	•	•
<code>OS_EVENT_Reset()</code>	Resets the specified event object to non-signaled state.	•	•	•	•	•
<code>OS_EVENT_ResetMask()</code>	Resets the specified mask bits in the specified event object to non-signaled state.	•	•	•	•	•
<code>OS_EVENT_Set()</code>	Sets the specified event object to signaled state, or resumes tasks which are waiting at the event object.	•	•	•	•	•
<code>OS_EVENT_SetMask()</code>	Sets the event mask bits of the specified event object.	•	•	•	•	•
<code>OS_EVENT_SetMaskMode()</code>	Sets the mask mode of the specified event object to OR/AND logic.	•	•	•	•	•
<code>OS_EVENT_SetResetMode()</code>	Sets the reset behavior of the specified event object to automatic, manual or semi-auto.	•	•	•	•	•

6.2.1 OS_EVENT_Create()

Description

Creates an event object.

Prototype

```
void OS_EVENT_Create(OS_EVENT* pEvent);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .

Additional information

On creation, the event is set to non-signaled state.

The event is created with the default reset behavior which is semi-auto. Since version 3.88a of embOS, the reset behavior of the event can be modified by a call of the function `OS_EVENT_SetResetMode()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

165: `OS_ERR_INIT_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

178: `OS_ERR_2USE_EVENT`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void HPTask(void) {
    OS_EVENT_GetMaskBlocked(&_Event, 3); // Wait for bit 0 AND 1 to be set
}

void LPTask(void) {
    OS_EVENT_SetMask(&_Event, 1); // Resumes HPTask due to OR logic
}

int main(void) {
    ...
    OS_EVENT_Create(&_Event);
    ...
    return 0;
}
```

6.2.2 OS_EVENT_CreateEx()

Description

Creates an extended event object and sets its reset behavior as well as mask bits behavior.

Prototype

```
void OS_EVENT_CreateEx(OS_EVENT* pEvent,
                      unsigned int Mode);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .
<code>Mode</code>	Specifies the reset and mask bits behavior of the event object. You can use one of the predefined reset modes: <code>OS_EVENT_RESET_MODE_SEMIAUTO</code> <code>OS_EVENT_RESET_MODE_MANUAL</code> <code>OS_EVENT_RESET_MODE_AUTO</code> and one of the mask modes: <code>OS_EVENT_MASK_MODE_OR_LOGIC</code> <code>OS_EVENT_MASK_MODE_AND_LOGIC</code> which are described under additional information.

Additional information

On creation, the event is set to non-signaled state.

Since version 3.88a of embOS, the reset behavior of the event can be controlled by different reset modes which may be passed as parameter to the new function `OS_EVENT_CreateEx()` or may be modified by a call of `OS_EVENT_SetResetMode()`.

- `OS_EVENT_RESET_MODE_SEMIAUTO`:
This reset mode is the default mode used with all previous versions of embOS. The reset behavior unfortunately is not consistent and depends on the function called to set or wait for an event. This reset mode is defined for compatibility with older embOS versions (prior version 3.88a). Calling `OS_EVENT_Create()` sets the reset mode to `OS_EVENT_RESET_MODE_SEMIAUTO` to be compatible with older embOS versions.
- `OS_EVENT_RESET_MODE_AUTO`:
This mode sets the reset behavior of an event object to automatic clear. When an event is set, all waiting tasks are resumed and the event is cleared automatically. An exception to this is when a task called `OS_EVENT_GetTimed()` and the timeout expired before the event was signaled, in which case the function returns with timeout and the event is not cleared automatically.
- `OS_EVENT_RESET_MODE_MANUAL`:
This mode sets the event to manual reset mode. When an event is set, all waiting tasks are resumed and the event object remains signaled. The event must be reset by one task which was waiting for the event.

Since version 4.34 of embOS, the mask bits behavior of the event object can be controlled by different mask modes which may be passed to the new function `OS_EVENT_CreateEx()` or may be modified by a call of `OS_EVENT_SetMaskMode()`.

- `OS_EVENT_MASK_MODE_OR_LOGIC`:
This mask mode is the default mode. Only one of the bits specified in the event object bit mask must be signaled.
- `OS_EVENT_MASK_MODE_AND_LOGIC`:
With this mode all specified event object mask bits must be signaled.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED
165: OS_ERR_INIT_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
178: OS_ERR_2USE_EVENT

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void HPTask(void) {
    OS_EVENT_GetMaskBlocked(&_amp;_Event, 3); // Wait for bit 0 AND 1 to be set
}

void LPTask(void) {
    OS_EVENT_SetMask(&_amp;_Event, 1); // Does not resume HPTask
    OS_EVENT_SetMask(&_amp;_Event, 2);
    // Resume HPTask since both bits are now set
}

int main(void) {
    ...
    OS_EVENT_CreateEx(&_amp;_Event, OS_EVENT_RESET_MODE_AUTO |
                      OS_EVENT_MASK_MODE_AND_LOGIC);
    ...
    return 0;
}
```

6.2.3 OS_EVENT_Delete()

Description

Deletes the specified event object.

Prototype

```
void OS_EVENT_Delete(OS_EVENT* pEvent);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .

Additional information

Before deleting an event object, make sure that no task is waiting for the event object.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
167: OS_ERR_CPU_STATE_ILLEGAL  
212: OS_ERR_EVENT_DELETE  
220: OS_ERR_EVENT_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;  
  
void Task(void) {  
    ...  
    OS_EVENT_Delete(&_Event);  
    ...  
}
```

6.2.4 OS_EVENT_Get()

Description

Retrieves the current state of the specified event object.

Prototype

```
OS_BOOL OS_EVENT_Get(OS_CONST_PTR OS_EVENT *pEvent);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .

Return value

= 0 Event object is not set to signaled state.
≠ 0 Event object is set to signaled state.

Additional information

By calling this function, the actual state of the event object remains unchanged.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

220: `OS_ERR_EVENT_INVALID`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void Task(void) {
    OS_BOOL Status;

    Status = OS_EVENT_Get(&_Event);
    printf("Event Object Status: %d\n", Status);
}
```

6.2.5 OS_EVENT_GetBlocked()

Description

Waits for the specified event object and suspends the task until the event has been signaled.

Prototype

```
void OS_EVENT_GetBlocked(OS_EVENT* pEvent);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .

Additional information

The state of the event object after calling `OS_EVENT_GetBlocked()` depends on the reset mode of the event object which was set by creating the event object by a call of `OS_EVENT_CreateEx()` or `OS_EVENT_SetResetMode()`.

The event is consumed when `OS_EVENT_RESET_MODE_AUTO` is selected. The event is not consumed when `OS_EVENT_RESET_MODE_MANUAL` is selected. With `OS_EVENT_RESET_MODE_SEMIAUTO` the event is consumed only when it was already set before.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
220: OS_ERR_EVENT_INVALID
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void HPTask(void) {
    OS_EVENT_GetBlocked(&_Event); // Suspends the task
}

void LPTask(void) {
    OS_EVENT_Pulse(&_Event);      // Signals the HPTask
}
```

6.2.6 OS_EVENT_GetMask()

Description

Returns the bits of the specified event object that match the given [EventMask](#).

Prototype

```
OS_TASKEVENT OS_EVENT_GetMask(OS_EVENT* pEvent,
                               OS_TASKEVENT EventMask);
```

Parameters

Parameter	Description
pEvent	Pointer to an event object of type <code>OS_EVENT</code> .
EventMask	The bit mask containing the event bits which shall be retrieved.

Return value

All events that have been signaled and were specified in the [EventMask](#).

Additional information

The returned event mask bits are consumed unless `OS_EVENT_RESET_MODE_MANUAL` is selected. The state of the event object after calling `OS_EVENT_GetMask()` depends on the reset mode of the event object which was set by creating the event object by a call of `OS_EVENT_CreateEx()` or `OS_EVENT_SetResetMode()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

220: `OS_ERR_EVENT_INVALID`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void Task(void) {
    OS_TASKEVENT EventMask;

    EventMask = ~0; // Request all event bits
    EventMask = OS_EVENT_GetMask(&_Event, EventMask);
    printf("Signaled Event Bits: 0x%X\n", EventMask);
}
```

6.2.7 OS_EVENT_GetMaskBlocked()

Description

Waits for the specified event bits in `EventMask`, depending on the current mask mode. The task is suspended until the event(s) have been signaled.

Prototype

```
OS_TASKEVENT OS_EVENT_GetMaskBlocked(OS_EVENT*   pEvent,
                                     OS_TASKEVENT EventMask);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .
<code>EventMask</code>	The event bit mask containing the event bits, which shall be waited for.

Return value

All requested events that have been signaled and were specified in the `EventMask`.

Additional information

It returns the bits of the event object that match the given `EventMask`. The returned event mask bits are consumed unless `OS_EVENT_RESET_MODE_MANUAL` is selected. The state of the event object after calling `OS_EVENT_GetMaskBlocked()` depends on the reset mode of the event object which was set by creating the event object by a call of `OS_EVENT_CreateEx()` or `OS_EVENT_SetResetMode()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
220: OS_ERR_EVENT_INVALID
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void Task(void) {
    ...
    //
    //  Waits either for the first or second, or for
    //  both event bits to be signaled, depending on
    //  the specified mask mode.
    //
    OS_EVENT_GetMaskBlocked(&_Event, 0x3);
    ...
}
```

6.2.8 OS_EVENT_GetMaskMode()

Description

Retrieves the current mask mode (mask bits behavior) of the specified event object.

Prototype

```
OS_EVENT_MASK_MODE OS_EVENT_GetMaskMode(OS_CONST_PTR OS_EVENT *pEvent);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .

Return value

The mask mode which is currently set.

Modes are defined in enum `OS_EVENT_MASK_MODE`.

`OS_EVENT_MASK_MODE_OR_LOGIC (0x00u)`: Mask bits are used with OR logic (default).

`OS_EVENT_MASK_MODE_AND_LOGIC (0x04u)`: Mask bits are used with AND logic.

Additional information

Since version 4.34 of embOS, the mask mode of an event object can be controlled by the `OS_EVENT_CreateEx()` function or set after creation using the new function `OS_EVENT_SetMaskMode()`. If needed, the current setting of the mask mode can be retrieved with `OS_EVENT_GetMaskMode()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

220: `OS_ERR_EVENT_INVALID`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void Task(void) {
    OS_EVENT_MASK_MODE MaskMode;

    MaskMode = OS_EVENT_GetMaskMode(&_Event);
    if (MaskMode == OS_EVENT_MASK_MODE_OR_LOGIC) {
        printf("Logic: OR\n");
    } else {
        printf("Logic: AND\n");
    }
}
```

6.2.9 OS_EVENT_GetMaskTimed()

Description

Waits for the specified event bits `EventMask` with timeout, depending on the current mask mode.

Prototype

```
OS_TASKEVENT OS_EVENT_GetMaskTimed(OS_EVENT*   pEvent,
                                   OS_TASKEVENT EventMask,
                                   OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .
<code>EventMask</code>	The event bit mask containing the event bits, which shall be waited for.
<code>Timeout</code>	Maximum time in embOS system ticks until events must be signaled. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 No event available within the specified timeout.
 ≠ 0 All events that have been signaled and were specified in the `EventMask`.

Additional information

The task is suspended for the specified time or until the event(s) have been signaled. It returns the bits of the event object that match the given `EventMask`. The returned event mask bits are consumed unless `OS_EVENT_RESET_MODE_MANUAL` is selected. The state of the event object after calling `OS_EVENT_GetMaskTimed()` depends on the reset mode of the event object which was set by creating the event object by a call of `OS_EVENT_CreateEx()` or `OS_EVENT_SetResetMode()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
220: OS_ERR_EVENT_INVALID
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void Task(void) {
    ...
    //
    // Waits either for the first or second, or for
    // both event bits to be signaled, depending on
    // the specified mask mode. The task resumes after
    // 1000 system ticks, if the needed event bits were not
```

```
// signaled.  
//  
OS_EVENT_GetMaskTimed(&_Event, 0x3, 1000);  
...  
}
```

6.2.10 OS_EVENT_GetResetMode()

Description

Returns the reset mode (reset behavior) of the specified event object.

Prototype

```
OS_EVENT_RESET_MODE OS_EVENT_GetResetMode(OS_CONST_PTR OS_EVENT *pEvent);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .

Return value

The reset mode which is currently set.

Modes are defined in enum `OS_EVENT_RESET_MODE`.

`OS_EVENT_RESET_MODE_SEMIAUTO (0x00u)`: As previous mode (default).

`OS_EVENT_RESET_MODE_MANUAL (0x01u)`: Event remains set, has to be reset by task.

`OS_EVENT_RESET_MODE_AUTO (0x02u)`: Event is reset automatically.

Additional information

Since version 3.88a of embOS, the reset mode of an event object can be controlled by the new `OS_EVENT_CreateEx()` function or set after creation using the new function `OS_EVENT_SetResetMode()`. If needed, the current setting of the reset mode can be retrieved with `OS_EVENT_GetResetMode()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

220: `OS_ERR_EVENT_INVALID`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void Task(void) {
    OS_EVENT_RESET_MODE ResetMode;

    ResetMode = OS_EVENT_GetResetMode(&_Event);
    if (ResetMode == OS_EVENT_RESET_MODE_SEMIAUTO) {
        printf("Reset Mode: SEMIAUTO\n");
    } else if (ResetMode == OS_EVENT_RESET_MODE_MANUAL) {
        printf("Reset Mode: MANUAL\n");
    } else {
        printf("Reset Mode: AUTO\n");
    }
}
```

6.2.11 OS_EVENT_GetTimed()

Description

Waits for an event and suspends the task for a specified time or until the specified event has been signaled.

Prototype

```
char OS_EVENT_GetTimed(OS_EVENT* pEvent,
                      OS_TIME   Timeout);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .
<code>Timeout</code>	Maximum time in embOS system ticks until the event must be signaled. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Success, the event was signaled within the specified time.
 ≠ 0 If the event was not signaled within the specified time.

Additional information

The event is consumed unless `OS_EVENT_RESET_MODE_MANUAL` is selected.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
220: OS_ERR_EVENT_INVALID
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void Task(void) {
    ...
    if (OS_EVENT_GetTimed(&_Event, 1000) == 0) {
        // event was signaled within timeout time, handle event
    } else {
        // event was not signaled within timeout time, handle timeout
    }
    ...
}
```

6.2.12 OS_EVENT_Pulse()

Description

Signals the specified event object and resumes waiting tasks, then resets the event object to non-signaled state.

Prototype

```
void OS_EVENT_Pulse(OS_EVENT* pEvent);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .

Additional information

If any tasks are waiting at the event object, the tasks are resumed. The event object remains in non-signaled state, regardless the reset mode.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

220: `OS_ERR_EVENT_INVALID`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void HPTask(void) {
    OS_EVENT_GetBlocked(&_Event); // Suspends the task
}

void LPTask(void) {
    OS_EVENT_Pulse(&_Event); // Signals the HPTask
}
```

6.2.13 OS_EVENT_Reset()

Description

Resets the specified event object to non-signaled state.

Prototype

```
void OS_EVENT_Reset(OS_EVENT* pEvent);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .

Additional information

`OS_EVENT_Reset()` may also be used with event mask and resets all mask bits.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

220: OS_ERR_EVENT_INVALID

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;  
  
void Task(void) {  
    ...  
    OS_EVENT_Reset(&_Event);  
    ...  
}
```

6.2.14 OS_EVENT_ResetMask()

Description

Resets the specified mask bits in the specified event object to non-signaled state.

Prototype

```
void OS_EVENT_ResetMask(OS_EVENT* pEvent,
                        OS_TASKEVENT EventMask);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .
<code>EventMask</code>	The event bit mask containing the event bits which shall be cleared.

Additional information

`OS_EVENT_ResetMask()` resets only the event mask bits specified in `EventMask`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

220: OS_ERR_EVENT_INVALID

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void Task(void) {
    ...
    OS_EVENT_ResetMask(&_Event, 1);
    ...
}
```

6.2.15 OS_EVENT_Set()

Description

Sets the specified event object to signaled state, or resumes tasks which are waiting at the event object.

Prototype

```
void OS_EVENT_Set(OS_EVENT* pEvent);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .

Additional information

If no tasks are waiting at the event object, the event object is set to signaled state. Any task that is already waiting for the event object will be resumed. The state of the event object after calling `OS_EVENT_Set()` then depends on the reset mode of the event object.

- With reset mode `OS_EVENT_RESET_MODE_SEMIAUTO`:
This is the default mode when the event object was created with `OS_EVENT_Create()`. This was the only mode available in embOS versions prior version 3.88a. If tasks were waiting, the event is reset when the waiting tasks are resumed.
- With reset mode `OS_EVENT_RESET_MODE_AUTO`:
The event object is automatically reset when waiting tasks are resumed and continue operation.
- With reset mode `OS_EVENT_RESET_MODE_MANUAL`:
The event object remains signaled when waiting tasks are resumed and continue operation. The event object must be reset by the calling task.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

220: `OS_ERR_EVENT_INVALID`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Examples on how to use the `OS_EVENT_Set()` function are shown in *Examples* on page 157.

6.2.16 OS_EVENT_SetMask()

Description

Sets the event mask bits of the specified event object.

Prototype

```
void OS_EVENT_SetMask(OS_EVENT* pEvent,
                     OS_TASKEVENT EventMask);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .
<code>EventMask</code>	The event bit mask containing the event bits, which shall be signaled.

Additional information

Any task that is already waiting for matching event mask bits on this event object will be resumed. `OS_EVENT_SetMask()` does not clear any event mask bits.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

220: `OS_ERR_EVENT_INVALID`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void Task(void) {
    OS_TASKEVENT EventMask;

    ...
    EventMask = 1 << ((sizeof(OS_TASKEVENT) * 8) - 1); // Set MSB event bit
    OS_EVENT_SetMask(&_amp;_Event, EventMask);           // Signal MSB event bit
    ...
}
```

6.2.17 OS_EVENT_SetMaskMode()

Description

Sets the mask mode of the specified event object to OR/AND logic.

Prototype

```
void OS_EVENT_SetMaskMode(OS_EVENT*      pEvent,
                          OS_EVENT_MASK_MODE MaskMode);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .
<code>MaskMode</code>	Event Mask mode. Modes are defined in enum <code>OS_EVENT_MASK_MODE</code> . <code>OS_EVENT_MASK_MODE_OR_LOGIC (0x00u)</code> : Mask bits are used with OR logic (default). <code>OS_EVENT_MASK_MODE_AND_LOGIC (0x04u)</code> : Mask bits are used with AND logic.

Additional information

Since version 4.34 of embOS, the mask bits behavior of the event object can be controlled by different mask modes which may be passed to the new function `OS_EVENT_CreateEx()` or may be modified by a call of `OS_EVENT_SetMaskMode()`. The following mask modes are defined and can be used as parameter:

- `OS_EVENT_MASK_MODE_OR_LOGIC`:
This mask mode is the default mode. Only one of the bits specified in the event object bit mask must be signaled.
- `OS_EVENT_MASK_MODE_AND_LOGIC`:
With this mode all specified event mask bits must be signaled.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
220: OS_ERR_EVENT_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;

void Task(void) {
    ...
    // Set the mask mode for the event object to AND logic
    OS_EVENT_SetMaskMode(&_Event, OS_EVENT_MASK_MODE_AND_LOGIC);
    ...
}
```

6.2.18 OS_EVENT_SetResetMode()

Description

Sets the reset behavior of the specified event object to automatic, manual or semi-auto.

Prototype

```
void OS_EVENT_SetResetMode(OS_EVENT* pEvent,
                           OS_EVENT_RESET_MODE ResetMode);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .
<code>ResetMode</code>	Controls the reset mode of the event object. <code>OS_EVENT_RESET_MODE_SEMIAUTO (0x00u)</code> : As previous mode (default). <code>OS_EVENT_RESET_MODE_MANUAL (0x01u)</code> : Event remains set, has to be reset by task. <code>OS_EVENT_RESET_MODE_AUTO (0x02u)</code> : Event is reset automatically.

Additional information

Implementation of event objects in embOS versions before 3.88a unfortunately was not consistent with respect to the state of the event after calling `OS_EVENT_Set()` or `OS_EVENT_GetBlocked()` functions. The state of the event was different when tasks were waiting or not.

Since embOS version 3.88a, the state of the event (reset behavior) can be controlled after creation by the new function `OS_EVENT_SetResetMode()`, or during creation by the new `OS_EVENT_CreateEx()` function. The following reset modes are defined and can be used as parameter:

- `OS_EVENT_RESET_MODE_SEMIAUTO`:
This reset mode is the default mode used with all previous versions of embOS. The reset behavior unfortunately is not consistent and depends on the function called to set or wait for an event. This reset mode is defined for compatibility with older embOS versions (prior version 3.88a). Calling `OS_EVENT_Create()` sets the reset mode to `OS_EVENT_RESET_MODE_SEMIAUTO` to be compatible with older embOS versions.
- `OS_EVENT_RESET_MODE_AUTO`:
This mode sets the reset behavior of an event object to automatic clear. When an event is set, all waiting tasks are resumed and the event is cleared automatically. An exception to this is when a task called `OS_EVENT_GetTimed()` and the timeout expired before the event was signaled, in which case the function returns with timeout and the event is not cleared automatically.
- `OS_EVENT_RESET_MODE_MANUAL`:
This mode sets the event to manual reset mode. When an event is set, all waiting tasks are resumed and the event object remains signaled. The event must be reset by one task which was waiting for the event.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

220: OS_ERR_EVENT_INVALID

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_EVENT _Event;  
  
void Task(void) {  
    // Set the reset mode for the event object to manual  
    OS_EVENT_SetResetMode(&_amp;_Event, OS_EVENT_RESET_MANUAL);  
}
```

Chapter 7

Mutex

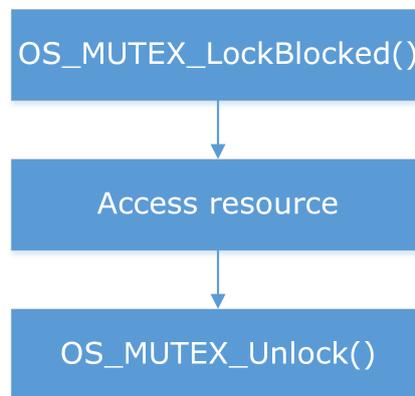
7.1 Introduction

Mutexes are used for managing resources by avoiding conflicts caused by simultaneous use of a resource. The resource managed can be of any kind: a part of the program that is not reentrant, a piece of hardware like the display, a flash memory that can only be written to by a single task at a time, a motor in a CNC control that can only be controlled by one task at a time, and a lot more.

The basic procedure is as follows:

Any task that uses a resource first claims it calling the `OS_MUTEX_LockBlocked()` or `OS_MUTEX_Lock()` routines of embOS. If the mutex is available, the program execution of the task continues, but the mutex is blocked for other tasks. If a second task now tries to acquire the same mutex while it is in use by the first task, this second task is suspended until the first task releases the mutex. However, if the first task that uses the mutex calls `OS_MUTEX_LockBlocked()` again for that mutex, it is not suspended because the mutex is blocked only for other tasks.

The following diagram illustrates the process of using a mutex:



A mutex contains a counter that keeps track of how many times the mutex has been claimed by calling `OS_MUTEX_Lock()` or `OS_MUTEX_LockBlocked()` by a particular task. It is released when that counter reaches zero, which means the `OS_MUTEX_Unlock()` routine must be called exactly the same number of times as `OS_MUTEX_LockBlocked()` or `OS_MUTEX_Lock()`. If it is not, the mutex remains blocked for other tasks.

On the other hand, a task cannot release a mutex that it does not own by calling `OS_MUTEX_Unlock()`. In debug builds of embOS, a call of `OS_MUTEX_Unlock()` for a mutex that is not owned by this task will result in a call to the error handler `OS_Error()`.

Example of using a mutex

Here, two tasks access a (debug) terminal completely independently from each other. The terminal is a resource that needs to be protected with a mutex. One task may not interrupt another task which is writing to the terminal, as otherwise the following might occur:

- Task A begins writing to the terminal
- Task B interrupts Task A and writes to the terminal
- Task A is resumed and its output is written at a wrong position

To avoid this type of situation, every time the terminal is to be accessed by a task it is first claimed by a call to `OS_MUTEX_LockBlocked()` (and is automatically waited for if the mutex is blocked). After the terminal has been written to, it is released by a call to `OS_MUTEX_Unlock()`.

The sample application file `OS_Mutexes.c` delivered in the application samples folder of embOS demonstrates how mutex can be used in the above scenario:

```
#include "RTOS.h"
#include <stdio.h>

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task-control-blocks
static OS_MUTEX     Mutex;

static void _Write(char const* s) {
    OS_MUTEX_LockBlocked(&Mutex);
    printf(s);
    OS_MUTEX_Unlock(&Mutex);
}

static void HPTask(void) {
    while (1) {
        _Write("HPTask\n");
        OS_TASK_Delay(50);
    }
}

static void LPTask(void) {
    while (1) {
        _Write("LPTask\n");
        OS_TASK_Delay(200);
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_Inithw(); // Initialize hardware for embOS
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_MUTEX_Create(&Mutex); // Creates mutex
    OS_Start(); // Start multitasking
    return 0;
}
```

Priority inversion / priority inheritance

embOS supports priority inheritance as a solution for the priority inversion problem when a mutex is used by multiple tasks. Please have a look in the chapter *Priority inversion / priority inheritance* on page 28 for more details.

Deadlock

Occasionally, you might want to access two resources at once. Perhaps you are using one of the resources, and then discover that the other resource is needed as well. A problem exists if two tasks attempt to claim both resources but lock the associated mutexes in different orders.

HPTask runs first, claims `Mutex_A` and then calls `OS_TASK_Delay()` which executes a task switch to LPTask. LPTask claims `Mutex_B` and tries to claim `Mutex_A`. Since `Mutex_A` is already acquired by HPTask it cannot be acquired by LPTask and LPTask is blocked. When the delay has expired HPTask tries to claim `Mutex_B` which is already acquired by LPTask. Both tasks are blocked now.

```
static void HPTask(void) {
    while (1) {
        OS_MUTEX_LockBlocked(&Mutex_A);
        OS_TASK_Delay(1);
        OS_MUTEX_LockBlocked(&Mutex_B);
        OS_MUTEX_Unlock(&Mutex_B);
        OS_MUTEX_Unlock(&Mutex_A);
    }
}

static void LPTask(void) {
    while (1) {
        OS_MUTEX_LockBlocked(&Mutex_B);
        OS_MUTEX_LockBlocked(&Mutex_A);
        OS_MUTEX_Unlock(&Mutex_A);
        OS_MUTEX_Unlock(&Mutex_B);
    }
}
```

The best way to avoid this problem is to make sure that when tasks lock multiple mutexes, the tasks do so in the same order. When locks are always taken in a prescribed order, deadlock should not occur.

However, this technique cannot always be used. Sometimes, you must take the mutexes in an order other than prescribed. To prevent deadlock in such a situation, use `OS_MUTEX_Lock()` instead of the blocking API. One task must release its mutexes when the task discovers that deadlock would otherwise be inevitable.

```
static void HPTask(void) {
    while (1) {
        OS_MUTEX_LockBlocked(&Mutex_A);
        OS_TASK_Delay(1);
        OS_MUTEX_LockBlocked(&Mutex_B);
        OS_MUTEX_Unlock(&Mutex_B);
        OS_MUTEX_Unlock(&Mutex_A);
    }
}

static void LPTask(void) {
    while (1) {
        OS_MUTEX_LockBlocked(&Mutex_B);
        if (OS_MUTEX_Lock(&Mutex_A) == 0) {
            OS_MUTEX_Unlock(&Mutex_B);
        } else {
            OS_MUTEX_Unlock(&Mutex_A);
            OS_MUTEX_Unlock(&Mutex_B);
        }
        OS_TASK_Delay(1);
    }
}
```

7.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_MUTEX_Create()</code>	Creates a mutex.	•	•			
<code>OS_MUTEX_Delete()</code>	Deletes a specified mutex.	•	•			
<code>OS_MUTEX_GetOwner()</code>	Returns the mutex owner if any.	•	•	•		
<code>OS_MUTEX_GetValue()</code>	Returns the value of the usage counter of a specified mutex.	•	•	•		
<code>OS_MUTEX_IsMutex()</code>	Returns whether a mutex has already been created.	•	•	•		
<code>OS_MUTEX_Lock()</code>	Requests a specified mutex and blocks it for other tasks if it is available.	•	•	•		
<code>OS_MUTEX_LockBlocked()</code>	Claims a mutex and blocks it for other tasks.	•	•	•		
<code>OS_MUTEX_LockTimed()</code>	Tries to claim a mutex and blocks it for other tasks if it is available within a specified time.	•	•	•		
<code>OS_MUTEX_Unlock()</code>	Releases a mutex currently in use by a task.	•	•	•		

7.2.1 OS_MUTEX_Create()

Description

Creates a mutex.

Prototype

```
void OS_MUTEX_Create(OS_MUTEX* pMutex);
```

Parameters

Parameter	Description
<code>pMutex</code>	Pointer to a mutex object of type <code>OS_MUTEX</code> .

Additional information

After creation, the mutex is not locked. The mutex counter value is zero.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
165: OS_ERR_INIT_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL  
165: OS_ERR_INIT_NOT_CALLED  
175: OS_ERR_2USE_MUTEX
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MUTEX _Mutex;  
  
int main(void) {  
    ...  
    OS_MUTEX_Create(&_Mutex);  
    ...  
    return 0;  
}
```

7.2.2 OS_MUTEX_Delete()

Description

Deletes a specified mutex.

Prototype

```
void OS_MUTEX_Delete(OS_MUTEX* pMutex);
```

Parameters

Parameter	Description
<code>pMutex</code>	Pointer to a mutex object of type <code>OS_MUTEX</code> .

Additional information

The memory of that mutex may be reused for other purposes or may be used for creating another mutex using the same memory. Before deleting a mutex, make sure that no task is claiming the mutex.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
133: OS_ERR_INV_MUTEX
138: OS_ERR_MUTEX_DELETE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MUTEX _Mutex;

int Task(void) {
    ...
    OS_MUTEX_Delete(&_Mutex);
    ...
    return 0;
}
```

7.2.3 OS_MUTEX_GetOwner()

Description

Returns the mutex owner if any. When a task is currently using (blocking) the mutex the task Id (address of task according task control block) is returned.

Prototype

```
OS_TASK *OS_MUTEX_GetOwner(OS_CONST_PTR OS_MUTEX *pMutex);
```

Parameters

Parameter	Description
<code>pMutex</code>	Pointer to a mutex object of type <code>OS_MUTEX</code> .

Return value

= NULL The mutex is not used by any task.
 ≠ NULL Task Id (address of the task control block).

Additional information

If a mutex was used in `main()` the return value of `OS_MUTEX_GetOwner()` is ambiguous. The return value `NULL` can mean it is currently used in `main()` or it is currently unused. Therefore, `OS_MUTEX_GetOwner()` must not be used to check if a mutex is available. Please use `OS_MUTEX_GetValue()` instead.

It is also good practice to free all used mutexes in `main()` before calling `OS_Start()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
133: OS_ERR_INV_MUTEX
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please find an example at `OS_MUTEX_GetValue()`.

7.2.4 OS_MUTEX_GetValue()

Description

Returns the value of the usage counter of a specified mutex.

Prototype

```
int OS_MUTEX_GetValue(OS_CONST_PTR OS_MUTEX *pMutex);
```

Parameters

Parameter	Description
<code>pMutex</code>	Pointer to a mutex object of type <code>OS_MUTEX</code> .

Return value

The counter value of the mutex.

A value of zero means the mutex is available.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
133: OS_ERR_INV_MUTEX
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MUTEX _Mutex;

void CheckMutex(void) {
    int Value;
    OS_TASK* Owner;

    Value = OS_MUTEX_GetValue(&_Mutex);
    if (Value == 0) {
        printf("Mutex is currently unused");
    } else {
        Owner = OS_MUTEX_GetOwner(&_Mutex);
        if (Owner == NULL) {
            printf("Mutex was used in main()");
        } else {
            printf("Mutex is currently used in task 0x%X", Owner);
        }
    }
}
```

7.2.5 OS_MUTEX_IsMutex()

Description

Returns whether a mutex has already been created.

Prototype

```
OS_BOOL OS_MUTEX_IsMutex(OS_CONST_PTR OS_MUTEX *pMutex);
```

Parameters

Parameter	Description
<code>pMutex</code>	Pointer to a mutex object of type <code>OS_MUTEX</code> .

Return value

= 0 Mutex has not been created or was deleted.
 ≠ 0 Mutex has already been created.

Additional information

`OS_MUTEX_IsMutex()` returns 1 if a mutex was created with `OS_MUTEX_Create()` and not yet deleted with `OS_MUTEX_Delete()`. `OS_MUTEX_IsMutex()` returns 0 if a mutex was not yet created with `OS_MUTEX_Create()` or it was deleted with `OS_MUTEX_Delete()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MUTEX _Mutex;

int main(void) {
    ...
    if (OS_MUTEX_IsMutex(&_Mutex) != (OS_BOOL)0) {
        printf("Mutex has already been created");
    } else {
        printf("Mutex has not yet been created");
    }
    ...
    return 0;
}
```

7.2.6 OS_MUTEX_Lock()

Description

Requests a specified mutex and blocks it for other tasks if it is available. Continues execution in any case.

Prototype

```
char OS_MUTEX_Lock(OS_MUTEX* pMutex);
```

Parameters

Parameter	Description
<code>pMutex</code>	Pointer to a mutex object of type <code>OS_MUTEX</code> .

Return value

= 0 Mutex was not available.
 ≠ 0 Mutex was available, now in use by calling task.

Error codes

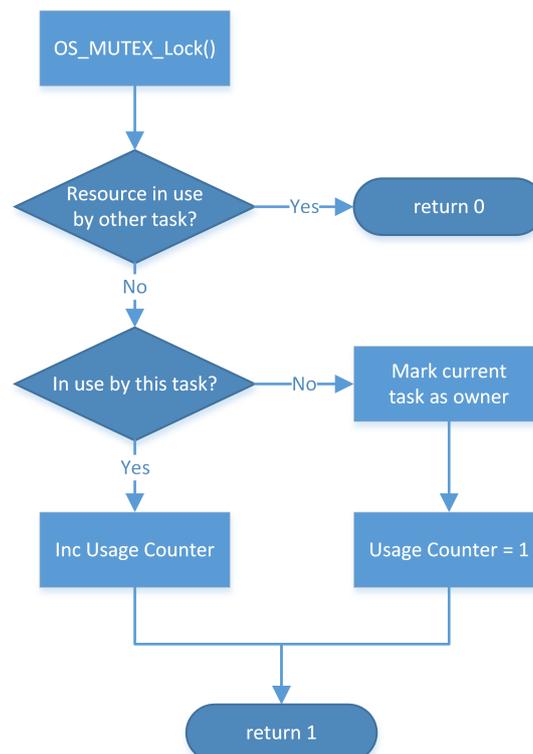
With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
133: OS_ERR_INV_MUTEX
159: OS_ERR_MUTEX_OVERFLOW
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Additional information

The following diagram illustrates how `OS_MUTEX_Lock()` works:



Example

```
if (OS_MUTEX_Lock(&Mutex_LCD)) {  
    DispTime();           // Access the resource LCD  
    OS_MUTEX_Unlock(&Mutex_LCD); // Resource LCD is no longer needed  
} else {  
    ... // Do something else  
}
```

7.2.7 OS_MUTEX_LockBlocked()

Description

Claims a mutex and blocks it for other tasks.

Prototype

```
int OS_MUTEX_LockBlocked(OS_MUTEX* pMutex);
```

Parameters

Parameter	Description
<code>pMutex</code>	Pointer to a mutex object of type <code>OS_MUTEX</code> .

Return value

The counter value of the mutex.

A value greater than one denotes the mutex was already locked by the calling task.

Additional information

The following situations are possible:

- Case A: The mutex is not in use.
If the mutex is not used by a task, which means the counter of the mutex is zero, the mutex will be blocked for other tasks by incrementing the counter and writing a unique code for the task that uses it into the mutex.
- Case B: The mutex is used by this task.
The counter of the mutex is incremented. The program continues without a break.
- Case C: The mutex is being used by another task.
The execution of this task is suspended until the mutex is released. In the meantime if the task blocked by the mutex has a higher priority than the task blocking the mutex, the blocking task is assigned the priority of the task requesting the mutex. This is called priority inheritance. Priority inheritance can only temporarily increase the priority of a task, never reduce it.

An unlimited number of tasks can wait for a mutex. According to the rules of the scheduler, of all the tasks waiting for the mutex the task with the highest priority will acquire the mutex and continue program execution.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
133: OS_ERR_INV_MUTEX
159: OS_ERR_MUTEX_OVERFLOW
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

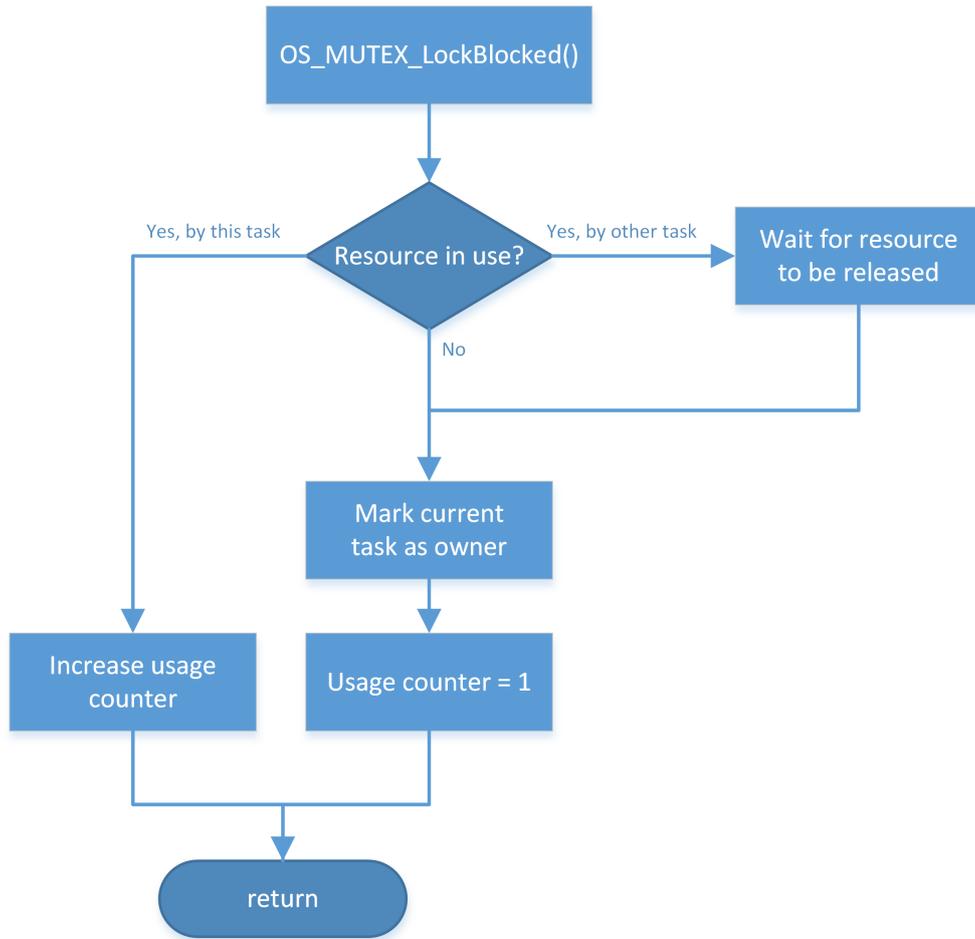
For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MUTEX _Mutex;

void Task(void) {
    ...
    OS_MUTEX_LockBlocked(&_Mutex);
    ...
    OS_MUTEX_Unlock(&_Mutex);
    ...
}
```

The following diagram illustrates how OS_MUTEX_LockBlocked() works:



7.2.8 OS_MUTEX_LockTimed()

Description

Tries to claim a mutex and blocks it for other tasks if it is available within a specified time.

Prototype

```
int OS_MUTEX_LockTimed(OS_MUTEX* pMutex,
                      OS_TIME Timeout);
```

Parameters

Parameter	Description
<code>pMutex</code>	Pointer to a mutex object of type <code>OS_MUTEX</code> .
<code>Timeout</code>	Maximum time in system ticks until the mutex must be available. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Failed, mutex not available before timeout.
 ≠ 0 Success, mutex available, current usage count of mutex.

A value greater than one denotes the mutex was already locked by the calling task.

Additional information

The following situations are possible:

- Case A: The mutex is not in use.
If the mutex is not used by a task, which means the counter of the mutex is zero, the mutex will be blocked for other tasks by incrementing the counter and writing a unique code for the task that uses it into the mutex.
- Case B: The mutex is used by this task.
The counter of the mutex is incremented. The program continues without a break.
- Case C: The mutex is being used by another task.
The execution of this task is suspended until the mutex is released or the timeout time expired. In the meantime if the task blocked by the mutex mutex has a higher priority than the task blocking the mutex, the blocking task is assigned the priority of the task requesting the mutex. This is called priority inheritance. Priority inheritance can only temporarily increase the priority of a task, never reduce it.
If the mutex becomes available during the timeout, the calling task claims the mutex and the function returns a value greater than zero, otherwise, if the mutex does not become available, the function returns zero.

When the calling task is blocked by higher priority tasks for a period longer than the timeout value, it may happen that the mutex becomes available before the calling task is resumed. Anyhow, the function will not claim the mutex because it was not available within the requested time.

An unlimited number of tasks can wait for a mutex. According to the rules of the scheduler, of all the tasks waiting for the mutex the task with the highest priority will acquire the mutex and continue program execution.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
133: OS_ERR_INV_MUTEX
159: OS_ERR_MUTEX_OVERFLOW
```

160: *OS_ERR_ILLEGAL_IN_ISR*
161: *OS_ERR_ILLEGAL_IN_TIMER*
167: *OS_ERR_CPU_STATE_ILLEGAL*

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MUTEX _Mutex;

void Task(void) {
    ...
    if (OS_MUTEX_LockTimed(&_Mutex, 100)) {
        ... // Mutex acquired
    } else {
        ... // Timeout
    }
    ...
}
```

7.2.9 OS_MUTEX_Unlock()

Description

Releases a mutex currently in use by a task.

Prototype

```
void OS_MUTEX_Unlock(OS_MUTEX* pMutex);
```

Parameters

Parameter	Description
<code>pMutex</code>	Pointer to a mutex object of type <code>OS_MUTEX</code> .

Additional information

`OS_MUTEX_Unlock()` may be used on a mutex only after that mutex has been locked by calling `OS_MUTEX_Lock()`, `OS_MUTEX_LockBlocked()`, or `OS_MUTEX_LockTimed()`. `OS_MUTEX_Unlock()` decrements the usage counter of the mutex, which must never become negative.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
133: OS_ERR_INV_MUTEX
150: OS_ERR_UNUSE_BEFORE_USE
156: OS_ERR_MUTEX_OWNER
159: OS_ERR_MUTEX_OVERFLOW
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please find an example at `OS_MUTEX_Lock()`.

Chapter 8

Semaphore

8.1 Introduction

A semaphore is a mechanism that can be used to provide synchronization of tasks. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 are called binary semaphores.

One way to use semaphores is for signaling from one task (or ISR/software timer) to another task. For example, if two tasks need to execute the same total number of times over the long run: A counting semaphore can be created with an initial count of zero (no 'tokens' in it). Every time the first task runs, it puts a token into the semaphore, thus incrementing the semaphore's count. The second task of the pair waits at the semaphore for tokens to appear, and runs once for each new token, thus consuming the token and decrementing the semaphore's count. If the first task runs with moderate bursts, the second task will eventually 'catch up' to the same total number of executions.

Binary semaphores can be used for signaling from task to task, too, in situations where signals (counts, tokens) will not accumulate or need not be counted.

Counting semaphores are also used for regulating the access of tasks to multiple equivalent serially-shareable resources. For instance, 10 tasks may wish to share 4 identical printers. In this case, a counting semaphore can be created and initialized with 4 tokens. Tasks are then programmed to take a token before printing, and return the token after printing is done.

Example of using counter semaphore for signaling

Here, an interrupt is issued every time data is received from a peripheral source. The interrupt service routine then signals the arrival of data to a worker task, which subsequently processes that data. When the worker task is blocked from execution, e.g. by a higher-priority task, the semaphore's counter effectively tracks the number of data packets to be processed by the worker task, which will be executed for that exact number of times when resumed.

The following sample application shows how semaphores can be used in the above scenario:

```
#include "RTOS.h"
#include <stdio.h>

static OS_STACKPTR int Stack[128];           // Task stack
static OS_TASK       TCB;                   // Task control block
static OS_SEMAPHORE  Sema;                 // Semaphore
static OS_TIMER      Timer;                // Timer to emulate interrupt

static void Task(void) {
    while(1) {
        OS_SEMAPHORE_TakeBlocked(&Sema);    // Wait for signaling of received data
        printf("Task is processing data\n"); // Act on received data
    }
}

static void TimerCallback(void) {
    // Software timer function to emulate an interrupt
    OS_SEMAPHORE_Give(&Sema);              // Signal data reception
    OS_TIMER_Restart(&Timer);
}

int main(void) {
    OS_Init();                             // Initialize embOS
    OS_Inithw();                            // Initialize required hardware
    OS_TIMER_Create(&Timer, TimerCallback, 10);
    OS_TIMER_Start(&Timer);
    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);
    OS_SEMAPHORE_Create(&Sema, 0);         // Creates semaphore
    OS_Start();                             // Start embOS
    return 0;
}
```

Example of using semaphore for regulating the access to shareable resources:

Ten tasks need to print messages on four available printers. The access to the printer must not be interrupted by another task. It is not essential for a task which actual printer is used and the `Printer()` function does not care about this aspect (this is a limitation of the example but not relevant). The example creates the semaphore with 4 tokens. Each token represents one printer. If a task wants to use one of the printers it takes one token and give it back after the print job is done. When no token (printer) is available the task is suspended until a token is again available.

```
#include "RTOS.h"
#include <stdio.h>

#define NUM_PRINTERS 4
#define NUM_TASKS 10

static OS_STACKPTR int Stack[NUM_TASKS][128]; // Task stack
static OS_TASK TCB[NUM_TASKS]; // Task control block
static OS_SEMAPHORE Sema; // Semaphore

static void Print(const char* s) {
    OS_SEMAPHORE_TakeBlocked(&Sema);
    // Print message on one of the available printers
    OS_SEMAPHORE_Give(&Sema);
}

static void Task(void) {
    while(1) {
        Print("Hello World");
    }
}

int main(void) {
    OS_U32 i;

    OS_Init(); // Initialize embOS
    OS_Inithw(); // Initialize required hardware
    for (i = 0u; i < NUM_TASKS; i++) {
        OS_TASK_CREATE(&TCB[i], "Task", 100, Task, Stack[i]);
    }
    OS_SEMAPHORE_Create(&Sema, NUM_PRINTERS); // Creates semaphore
    OS_Start(); // Start embOS
    return 0;
}
```

8.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_SEMAPHORE_Create()</code>	Creates a semaphore with a specified initial token count value.	•	•			
<code>OS_SEMAPHORE_Delete()</code>	Deletes the specified semaphore.	•	•			
<code>OS_SEMAPHORE_GetValue()</code>	Returns the semaphore token counter value of the specified semaphore.	•	•	•	•	•
<code>OS_SEMAPHORE_Give()</code>	Increments the semaphore token counter of the specified semaphore.	•	•	•	•	•
<code>OS_SEMAPHORE_GiveMax()</code>	Increments the semaphore token count of the specified semaphore up to the specified maximum value.	•	•	•	•	•
<code>OS_SEMAPHORE_SetValue()</code>	Sets the semaphore token counter value of the specified semaphore.	•	•	•		
<code>OS_SEMAPHORE_Take()</code>	Decrements the semaphore token counter value of the specified semaphore, if it was not zero.	•	•	•	•	•
<code>OS_SEMAPHORE_TakeBlocked()</code>	Decrements the semaphore token counter value of the specified semaphore.		•	•		
<code>OS_SEMAPHORE_TakeTimed()</code>	Decrements the semaphore token counter value of the specified semaphore if a semaphore token is available within the specified time.		•	•		

8.2.1 OS_SEMAPHORE_Create()

Description

Creates a semaphore with a specified initial token count value.

Prototype

```
void OS_SEMAPHORE_Create(OS_SEMAPHORE* pSema,
                        OS_UINT      InitValue);
```

Parameters

Parameter	Description
<code>pSema</code>	Pointer to a semaphore object of type <code>OS_SEMAPHORE</code> .
<code>InitValue</code>	Initial semaphore token count value: $0 \leq \text{InitValue} \leq 2^{16} - 1 = 0xFFFF$ for 8/16-bit CPUs. $0 \leq \text{InitValue} \leq 2^{32} - 1 = 0xFFFFFFFF$ for 32-bit CPUs.

Additional information

Note

embOS offers a macro that calls `OS_SEMAPHORE_Create()` with an initial count value of 0, allowing to more easily create semaphores. If the macro shall be used, its definition is as follows:

```
#define OS_SEMAPHORE_CREATE(ps) OS_SEMAPHORE_Create((ps), 0)
```

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
165: OS_ERR_INIT_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
174: OS_ERR_2USE_SEMAPHORE
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_SEMA _Sema;

int main(void) {
    ...
    OS_SEMAPHORE_Create(&_amp;Sema, 8);
    ...
    return 0;
}
```

8.2.2 OS_SEMAPHORE_Delete()

Description

Deletes the specified semaphore.

Prototype

```
void OS_SEMAPHORE_Delete(OS_SEMAPHORE* pSema);
```

Parameters

Parameter	Description
<code>pSema</code>	Pointer to a semaphore object of type <code>OS_SEMAPHORE</code> .

Additional information

Before deleting a semaphore, make sure that no task is waiting for it.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
132: OS_ERR_INV_SEMAPHORE
137: OS_ERR_SEMAPHORE_DELETE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_SEMA _Sema;

void Task(void) {
    ...
    OS_SEMAPHORE_Delete(&_amp;Sema);
    ...
}
```

8.2.3 OS_SEMAPHORE_GetValue()

Description

Returns the semaphore token counter value of the specified semaphore.

Prototype

```
int OS_SEMAPHORE_GetValue(OS_CONST_PTR OS_SEMAPHORE *pSema);
```

Parameters

Parameter	Description
<code>pSema</code>	Pointer to a semaphore object of type <code>OS_SEMAPHORE</code> .

Return value

The current semaphore token counter value.

Additional information

`OS_SEMAPHORE_GetValue()` can be used to get the current semaphore token value before calling e.g. `OS_SEMAPHORE_TakeBlocked()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
132: OS_ERR_INV_SEMAPHORE
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_SEMA _Sema;

void PrintSemaValue(void) {
    int Value;

    Value = OS_SEMAPHORE_GetValue(&_Sema);
    printf("Sema Value: %d\n", Value)
}
```

8.2.4 OS_SEMAPHORE_SetValue()

Description

Sets the semaphore token counter value of the specified semaphore.

Prototype

```
OS_U8 OS_SEMAPHORE_SetValue(OS_SEMAPHORE* pSema,
                             OS_UINT      Value);
```

Parameters

Parameter	Description
<code>pSema</code>	Pointer to a semaphore object of type <code>OS_SEMAPHORE</code> .
<code>Value</code>	Count value of the semaphore: $0 \leq \text{Value} \leq 2^{16} - 1 = 0xFFFF$ for 8/16-bit CPUs. $0 \leq \text{Value} \leq 2^{32} - 1 = 0xFFFFFFFF$ for 32-bit CPUs.

Return value

= 0: In any case. The return value can safely be ignored.

Additional information

If one or more tasks are waiting for this semaphore, they will be put in the READY state and activated according to the rules of the scheduler.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
132: OS_ERR_INV_SEMAPHORE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_SEMA _Sema;

void Task(void) {
    ...
    OS_SEMAPHORE_SetValue(&_Sema, 0);
    ...
}
```

8.2.5 OS_SEMAPHORE_Give()

Description

Increments the semaphore token counter of the specified semaphore.

Prototype

```
void OS_SEMAPHORE_Give(OS_SEMAPHORE* pSema);
```

Parameters

Parameter	Description
<code>pSema</code>	Pointer to a semaphore object of type <code>OS_SEMAPHORE</code> .

Additional information

`OS_SEMAPHORE_Give()` increments the semaphore token count. If one or more tasks are waiting for this semaphore, they will be put in the READY state and activated according to the rules of the scheduler.

The semaphore token counter can have a maximum value of `0xFFFF` for 8/16-bit CPUs or `0xFFFFFFFF` for 32-bit CPUs. It is the responsibility of the application to make sure that this limit is not exceeded.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
121: OS_ERR_SEMAPHORE_OVERFLOW
132: OS_ERR_INV_SEMAPHORE
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Semaphores* on page .

8.2.6 OS_SEMAPHORE_GiveMax()

Description

Increments the semaphore token count of the specified semaphore up to the specified maximum value.

Prototype

```
void OS_SEMAPHORE_GiveMax(OS_SEMAPHORE* pSema,
                          OS_UINT      MaxValue);
```

Parameters

Parameter	Description
<code>pSema</code>	Pointer to a semaphore object of type <code>OS_SEMAPHORE</code> .
<code>MaxValue</code>	Count value of the semaphore: $1 \leq \text{MaxValue} \leq 2^{16} - 1 = 0xFFFF$ for 8/16-bit CPUs. $1 \leq \text{MaxValue} \leq 2^{32} - 1 = 0xFFFFFFFF$ for 32-bit CPUs.

Additional information

As long as current value of the semaphore token counter is below the specified maximum value, `OS_SEMAPHORE_GiveMax()` increments its token counter. If one or more tasks are waiting for this semaphore, they will be put in the READY state and activated according to the rules of the scheduler.

Calling `OS_SEMAPHORE_GiveMax()` with a `MaxValue` of 1 makes a counting semaphore behave like a binary semaphore.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
132: OS_ERR_INV_SEMAPHORE
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_SEMA _Sema;

void Task(void) {
    ...
    OS_SEMAPHORE_GiveMax(&_Sema, 8);
    ...
}
```

8.2.7 OS_SEMAPHORE_Take()

Description

Decrements the semaphore token counter value of the specified semaphore, if it was not zero.

Prototype

```
OS_BOOL OS_SEMAPHORE_Take(OS_SEMAPHORE* pSema);
```

Parameters

Parameter	Description
<code>pSema</code>	Pointer to a semaphore object of type <code>OS_SEMAPHORE</code> .

Return value

= 0 Failed, no semaphore token available (semaphore token counter is zero).
 ≠ 0 Success, semaphore token was available and semaphore token counter was decremented once.

Additional information

This function never suspends the calling task. It may therefore also be called from an embOS interrupt routine. If the semaphore token counter is not zero, the counter is decremented. If the semaphore token counter is zero, `OS_SEMAPHORE_Take()` does not modify the semaphore token counter.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
132: OS_ERR_INV_SEMAPHORE
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_SEMA _Sema;

void Task(void) {
    ...
    if (OS_SEMAPHORE_Take(&_Sema) != 0) {
        printf("Semaphore decremented successfully.\n");
    } else {
        printf("Semaphore not signaled.\n");
    }
    ...
}
```

8.2.8 OS_SEMAPHORE_TakeBlocked()

Description

Decrements the semaphore token counter value of the specified semaphore.

Prototype

```
void OS_SEMAPHORE_TakeBlocked(OS_SEMAPHORE* pSema);
```

Parameters

Parameter	Description
<code>pSema</code>	Pointer to a semaphore object of type <code>OS_SEMAPHORE</code> .

Additional information

If the counter of the semaphore is not zero, the counter is decremented and program execution continues.

If the counter is zero, `OS_SEMAPHORE_TakeBlocked()` waits until the counter is incremented by another task, a timer or an interrupt handler by a call to `OS_SEMAPHORE_Give()`. The counter is then decremented and program execution continues. An unlimited number of tasks can wait for a semaphore. According to the rules of the scheduler, of all the tasks waiting for the semaphore, the task with the highest priority will continue program execution.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
132: OS_ERR_INV_SEMAPHORE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Semaphores* on page .

8.2.9 OS_SEMAPHORE_TakeTimed()

Description

Decrements the semaphore token counter value of the specified semaphore if a semaphore token is available within the specified time.

Prototype

```
OS_BOOL OS_SEMAPHORE_TakeTimed(OS_SEMAPHORE* pSema,
                               OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>pSema</code>	Pointer to a semaphore object of type <code>OS_SEMAPHORE</code> .
<code>Timeout</code>	Maximum time in system ticks until the semaphore token must be available. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Failed, semaphore not available before timeout.
 ≠ 0 Success, semaphore was available and counter decremented.

Additional information

If the semaphore token counter is not zero, the counter is decremented and program execution continues.

If the counter is zero, `OS_SEMAPHORE_TakeTimed()` waits until the semaphore token counter is incremented by another task, a timer, or an interrupt handler by a call to `OS_SEMAPHORE_Give()`. The counter is then decremented and program execution continues. If the semaphore token counter was incremented within the specified time the program execution continues, but returns a value of zero.

When the calling task is blocked by higher priority tasks for a period longer than the timeout value, it may happen that the semaphore becomes available after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the semaphore was not available within the requested time. In this case, the state of the semaphore is not modified by `OS_SEMAPHORE_TakeTimed()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
132: OS_ERR_INV_SEMAPHORE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_SEMA _Sema;

void Task(void) {
    ...
    if (OS_SEMAPHORE_TakeTimed(&_Sema, 100)) {
```

```
    ... // Semaphore acquired
  } else {
    ... // Timeout
  }
  ...
}
```

Chapter 9

Readers-Writer Lock

9.1 Introduction

A readers-writer lock is a synchronization primitive that solves the readers-writer problem. A readers-writer lock allows concurrent access for read-only operations, while write operations require exclusive access. This means that multiple tasks can read the data in parallel but an exclusive lock is needed for writing or modifying data. When a writer is writing the data, all other writers or readers will be blocked until the writer has finished writing. A common use might be to control access to a data structure in memory that cannot be updated atomically and is invalid (and should not be read by another task) until the update is complete. An embOS readers-writer lock is implemented using semaphores and mutexes.

```
#include "RTOS.h"
#include "stdio.h"

#define NUM_READERS 2

static OS_STACKPTR int StackRd1[128], StackRd2[128], StackWr[128];
static OS_TASK      TCBRd1, TCBRd2, TCBWr;
static OS_RWLOCK    Lock;
static OS_U32       GlobalVar;

static void RdTask(void) {
    while (1) {
        OS_RWLOCK_RdLockBlocked(&Lock);
        printf("%u\n", GlobalVar);
        OS_RWLOCK_RdUnlock(&Lock);
    }
}

static void WrTask(void) {
    while (1) {
        OS_RWLOCK_WrLockBlocked(&Lock);
        GlobalVar++;
        OS_RWLOCK_WrUnlock(&Lock);
        OS_TASK_Delay(10);
    }
}

int main(void) {
    OS_Init();    // Initialize embOS
    OS_Inithw(); // Initialize required hardware
    OS_TASK_CREATE(&TCBRd1, "Reader Task 1", 100, RdTask, StackRd1);
    OS_TASK_CREATE(&TCBRd2, "Reader Task 2", 100, RdTask, StackRd2);
    OS_TASK_CREATE(&TCBWr, "Writer Task", 101, WrTask, StackWr);
    OS_RWLOCK_Create(&Lock, NUM_READERS);
    OS_Start();   // Start embOS
    return 0;
}
```

9.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_RWLOCK_Create()</code>	Creates a readers-writer lock.	•	•			
<code>OS_RWLOCK_Delete()</code>	Deletes the specified readers-writer lock.	•	•			
<code>OS_RWLOCK_RdLock()</code>	Claims the specified readers-writer lock and blocks it for writer tasks.	•	•	•	•	•
<code>OS_RWLOCK_RdLockBlocked()</code>	Claims the specified readers-writer lock and blocks it for writer tasks.		•	•		
<code>OS_RWLOCK_RdLockTimed()</code>	Claims the specified readers-writer lock if the lock is available within the specified timeout and blocks it for writer tasks.		•	•		
<code>OS_RWLOCK_RdUnlock()</code>	Releases the specified readers-writer lock currently used by the reader task.	•	•	•	•	•
<code>OS_RWLOCK_WrLock()</code>	Claims the specified readers-writer lock and blocks it for writer and reader tasks.	•	•	•		
<code>OS_RWLOCK_WrLockBlocked()</code>	Claims the specified readers-writer lock and blocks it for writer and reader tasks.		•	•		
<code>OS_RWLOCK_WrLockTimed()</code>	Claims the specified readers-writer lock if the lock is available within the specified timeout and blocks it for writer and reader tasks.		•	•		
<code>OS_RWLOCK_WrUnlock()</code>	Releases the specified readers-writer lock currently used by the writer task.	•	•	•		

9.2.1 OS_RWLOCK_Create()

Description

Creates a readers-writer lock.

Prototype

```
void OS_RWLOCK_Create(OS_RWLOCK* pLock,
                     OS_UINT   NumReaders);
```

Parameters

Parameter	Description
<code>pLock</code>	Pointer to a readers-writer lock object of type <code>OS_RWLOCK</code> .
<code>NumReaders</code>	Number of reader tasks. Maximum number is: $0 \leq \text{InitValue} \leq 2^{16} - 1 = 0xFFFF$ for 8/16-bit CPUs. $0 \leq \text{InitValue} \leq 2^{32} - 1 = 0xFFFFFFFF$ for 32-bit CPUs.

Additional information

If you use readers-writer lock from an unprivileged task you need not only access to the lock object itself but also to the semaphore and the mutex member. Please see `embOS-MPU` example below.

Error codes

With `embOS` debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
165: OS_ERR_INIT_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
258: OS_ERR_2USE_RWLOCK
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define NUM_READERS 2

static OS_RWLOCK Lock;

int main(void) {
    ...
    OS_RWLOCK_Create(&Lock, NUM_READERS);
    ...
    return 0;
}
```

Example using embOS-MPU

```
static OS_RWLOCK Lock;

static const OS_MPU_OBJ _aList[] = {
    {&Lock, OS_MPU_OBJTYPE_RWLOCK},
    {&Lock.Semaphore, OS_MPU_OBJTYPE_SEMA},
    {&Lock.Mutex, OS_MPU_OBJTYPE_MUTEX},
    {NULL, OS_MPU_OBJTYPE_INVALID}};

static void Task(void) {
    OS_MPU_SetAllowedObjects(&TCB, _aList);
    OS_MPU_SwitchToUnprivState();
    while (1) {
```

```
    OS_RWLOCK_RdLockBlocked(&Lock);  
    ReadData();  
    OS_RWLOCK_RdUnlock(&Lock);  
};  
}
```

9.2.2 OS_RWLOCK_Delete()

Description

Deletes the specified readers-writer lock.

Prototype

```
void OS_RWLOCK_Delete(OS_RWLOCK* pLock);
```

Parameters

Parameter	Description
<code>pLock</code>	Pointer to a readers-writer lock object of type <code>OS_RWLOCK</code> .

Additional information

Before deleting a readers-writer lock, make sure that no task is waiting for it.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
167: OS_ERR_CPU_STATE_ILLEGAL  
257: OS_ERR_RWLOCK_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_RWLOCK Lock;  
  
void Task(void) {  
    ...  
    OS_RWLOCK_Delete(&Lock);  
    ...  
}
```

9.2.3 OS_RWLOCK_RdLock()

Description

Claims the specified readers-writer lock and blocks it for writer tasks.

Prototype

```
OS_BOOL OS_RWLOCK_RdLock(OS_RWLOCK* pLock);
```

Parameters

Parameter	Description
<code>pLock</code>	Pointer to a readers-writer lock object of type <code>OS_RWLOCK</code> .

Return value

= 0 Failed, lock could not be claimed.
 ≠ 0 Success, lock was available.

Additional information

Reader tasks can still access the guarded object. `OS_RWLOCK_RdLock()` returns at once in any case.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

167: `OS_ERR_CPU_STATE_ILLEGAL`

257: `OS_ERR_RWLOCK_INVALID`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_RWLOCK Lock;

void Task(void) {
    OS_BOOL r;

    r = OS_RWLOCK_RdLock(&Lock);
    if (r != 0) {
        ReadSomeData();
        OS_RWLOCK_RdUnlock(&Lock);
    }
}
```

9.2.4 OS_RWLOCK_RdLockBlocked()

Description

Claims the specified readers-writer lock and blocks it for writer tasks.

Prototype

```
void OS_RWLOCK_RdLockBlocked(OS_RWLOCK* pLock);
```

Parameters

Parameter	Description
<code>pLock</code>	Pointer to a readers-writer lock object of type <code>OS_RWLOCK</code> .

Additional information

Reader tasks can still access the guarded object. `OS_RWLOCK_RdLockBlocked()` suspends the current task and returns once a read lock is available.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
257: OS_ERR_RWLOCK_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_RWLOCK Lock;

void Task(void) {
    OS_RWLOCK_RdLockBlocked(&Lock);
    ReadSomeData();
    OS_RWLOCK_RdUnlock(&Lock);
}
```

9.2.5 OS_RWLOCK_RdLockTimed()

Description

Claims the specified readers-writer lock if the lock is available within the specified timeout and blocks it for writer tasks.

Prototype

```
OS_BOOL OS_RWLOCK_RdLockTimed(OS_RWLOCK* pLock,
                               OS_TIME   Timeout);
```

Parameters

Parameter	Description
<code>pLock</code>	Pointer to a readers-writer lock object of type <code>OS_RWLOCK</code> .
<code>Timeout</code>	Maximum time in system ticks until the lock must be available. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Failed, lock could not be claimed within the timeout.
 ≠ 0 Success, lock was available.

Additional information

Reader tasks can still access the guarded object. `OS_RWLOCK_RdLockTimed()` suspends the current task and returns once a reader lock is available or the timeout has expired.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
257: OS_ERR_RWLOCK_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_RWLOCK Lock;

void Task(void) {
    OS_BOOL r;

    r = OS_RWLOCK_RdLockTimed(&Lock, 100);
    if (r != 0) {
        ReadSomeData();
        OS_RWLOCK_RdUnlock(&Lock);
    }
}
```

9.2.6 OS_RWLOCK_RdUnlock()

Description

Releases the specified readers-writer lock currently used by the reader task.

Prototype

```
void OS_RWLOCK_RdUnlock(OS_RWLOCK* pLock);
```

Parameters

Parameter	Description
<code>pLock</code>	Pointer to a readers-writer lock object of type <code>OS_RWLOCK</code> .

Additional information

After `OS_RWLOCK_RdUnlock()` the lock can be used by another reader or the writer task.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

167: OS_ERR_CPU_STATE_ILLEGAL

257: OS_ERR_RWLOCK_INVALID

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_RWLOCK Lock;

void Task(void) {
    OS_RWLOCK_RdLockBlocked(&Lock);
    ReadSomeData();
    OS_RWLOCK_RdUnlock(&Lock);
}
```

9.2.7 OS_RWLOCK_WrLock()

Description

Claims the specified readers-writer lock and blocks it for writer and reader tasks.

Prototype

```
OS_BOOL OS_RWLOCK_WrLock(OS_RWLOCK* pLock);
```

Parameters

Parameter	Description
<code>pLock</code>	Pointer to a readers-writer lock object of type <code>OS_RWLOCK</code> .

Return value

= 0 Failed, writer lock could not be claimed.
 ≠ 0 Success, writer lock was available.

Additional information

`OS_RWLOCK_WrLock()` returns at once in any case.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
257: OS_ERR_RWLOCK_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_RWLOCK Lock;

void Task(void) {
    OS_BOOL r;

    r = OS_RWLOCK_WrLock(&Lock);
    if (r != 0) {
        WriteSomeData();
        OS_RWLOCK_WrUnlock(&Lock);
    }
}
```

9.2.8 OS_RWLOCK_WrLockBlocked()

Description

Claims the specified readers-writer lock and blocks it for writer and reader tasks.

Prototype

```
void OS_RWLOCK_WrLockBlocked(OS_RWLOCK* pLock);
```

Parameters

Parameter	Description
<code>pLock</code>	Pointer to a readers-writer lock object of type <code>OS_RWLOCK</code> .

Additional information

It requires all readers to relinquish their locks before the writer lock can be acquired. `OS_RWLOCK_WrLockBlocked()` suspends the current task and returns once the write lock is available.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
167: OS_ERR_CPU_STATE_ILLEGAL  
226: OS_ERR_ILLEGAL_IN_MAIN  
257: OS_ERR_RWLOCK_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_RWLOCK Lock;  
  
void Task(void) {  
    OS_RWLOCK_WrLockBlocked(&Lock);  
    WriteSomeData();  
    OS_RWLOCK_WrUnlock(&Lock);  
}
```

9.2.9 OS_RWLOCK_WrLockTimed()

Description

Claims the specified readers-writer lock if the lock is available within the specified timeout and blocks it for writer and reader tasks.

Prototype

```
OS_BOOL OS_RWLOCK_WrLockTimed(OS_RWLOCK* pLock,
                               OS_TIME    Timeout);
```

Parameters

Parameter	Description
<code>pLock</code>	Pointer to a readers-writer lock object of type <code>OS_RWLOCK</code> .
<code>Timeout</code>	Maximum time in system ticks until the lock must be available. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Failed, lock could not be claimed.
 ≠ 0 Success, lock was available.

Additional information

It requires all readers to relinquish their locks before the writer lock can be acquired. `OS_RWLOCK_WrLockTimed()` suspends the current task and returns once the writer lock is available or the timeout has expired.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
257: OS_ERR_RWLOCK_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_RWLOCK Lock;

void Task(void) {
    OS_BOOL r;

    r = OS_RWLOCK_WrLockTimed(&Lock, 100);
    if (r != 0) {
        WriteSomeData();
        OS_RWLOCK_WrUnlock(&Lock);
    }
}
```

9.2.10 OS_RWLOCK_WrUnlock()

Description

Releases the specified readers-writer lock currently used by the writer task.

Prototype

```
void OS_RWLOCK_WrUnlock(OS_RWLOCK* pLock);
```

Parameters

Parameter	Description
<code>pLock</code>	Pointer to a readers-writer lock object of type <code>OS_RWLOCK</code> .

Additional information

After `OS_RWLOCK_WrUnlock()` the lock is available for other reader and writer tasks.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
167: OS_ERR_CPU_STATE_ILLEGAL  
257: OS_ERR_RWLOCK_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_RWLOCK Lock;  
  
void Task(void) {  
    OS_RWLOCK_WrLockBlocked(&Lock);  
    WriteSomeData();  
    OS_RWLOCK_WrUnlock(&Lock);  
}
```

Chapter 10

Mailbox

10.1 Introduction

In the preceding chapters, task synchronization by the use of semaphores was described. Unfortunately, semaphores cannot transfer data from one task to another. If we need to transfer data between tasks for example via a buffer, we could use a mutex every time we accessed the buffer. But doing so would make the program less efficient. Another major disadvantage would be that we could not access the buffer from an interrupt handler, because the interrupt handler is not allowed to wait for the mutex.

One solution would be the usage of global variables. In this case we would need to disable interrupts each time and in each place that we accessed these variables. This is possible, but it is a path full of pitfalls. It is also not easy for a task to wait for a character to be placed in a buffer without polling the global variable that contains the number of characters in the buffer. Again, there is solution -- the task could be notified by an event signaled to the task each time a character is placed in the buffer. This is why there is an easier way to do this with a real-time OS: The use of mailboxes.

A mailbox is a buffer that is managed by the RTOS. The buffer behaves like a normal buffer; you can deposit something (called a message) and retrieve it later. Mailboxes usually work as FIFO: first in, first out. So a message that is deposited first will usually be retrieved first. "Message" might sound abstract, but very simply it means "item of data". It will become clearer in the typical applications explained in the following section.

Limitations:

Both the number of mailboxes and buffers are limited only by the amount of available memory. However, the number of messages per mailbox, the message size per mailbox, and the buffer size per mailbox are limited by software design. The buffer must be big enough to hold the given number of messages of the specified size: Message size * Number of messages in bytes.

	8 or 16-bit CPUs	32-bit CPUs
Maximum number of messages	32,767	2,147,483,647
Maximum message size	32,767 bytes	32,767 bytes
Maximum buffer size	65,535 bytes	4,294,967,295 bytes

These limitations have been placed on mailboxes to guarantee efficient coding and also to ensure efficient management. These limitations are typically not a problem.

A mailbox can be used by more than one producer, but must be used by one consumer only. This means that more than one task or interrupt handler is allowed to deposit new data into the mailbox, but it does not make sense to retrieve messages by multiple tasks.

10.1.1 Single-byte mailbox functions

In many (if not the most) situations, mailboxes are used simply to hold and transfer single-byte messages. This is the case, for example, with a mailbox that takes the character received or sent via serial interface, or typically with a mailbox used as a keyboard buffer. In some of these cases, time is very critical, especially if a lot of data is transferred in short periods of time.

To minimize the overhead caused by the mailbox management of embOS, variations on some mailbox functions are available for single-byte mailboxes. The general functions `OS_MAILBOX_PutBlocked()`, `OS_MAILBOX_Put()`, `OS_MAILBOX_GetBlocked()`, and `OS_MAILBOX_Get()` can transfer messages of sizes between 1 and 32,767 bytes each.

Their single-byte equivalents `OS_MAILBOX_PutBlocked1()`, `OS_MAILBOX_Put1()`, `OS_MAILBOX_GetBlocked1()`, and `OS_MAILBOX_Get1()` work the same way with the exception that they execute much faster because management is simpler. It is recommended to use the single-byte versions if you transfer a lot of single-byte data via mailboxes.

The routines `OS_MAILBOX_PutBlocked1()`, `OS_MAILBOX_Put1()`, `OS_MAILBOX_GetBlocked1()`, and `OS_MAILBOX_Get1()` work exactly the same way as their universal equivalents. The only difference is that they must only be used for single-byte mailboxes.

Example

```
#define MAX_MSG_SIZE (9) // Max. number of bytes per message
#define MAX_MSG_NUM (2) // Max. number of messages per Mailbox

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK TCBHP, TCBLP; // Task control blocks
static OS_MAILBOX MyMailbox;
static char MyMailboxBuffer[MAX_MSG_SIZE * MAX_MSG_NUM];

static void HPTask(void) {
    char aData[MAX_MSG_SIZE];

    while (1) {
        OS_MAILBOX_GetBlocked(&MyMailbox, (void *)aData);
        OS_COM_SendString(aData);
    }
}

static void LPTask(void) {
    while (1) {
        OS_MAILBOX_PutBlocked(&MyMailbox, "Hello\0 ");
        OS_MAILBOX_PutBlocked(&MyMailbox, "World !\n");
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_Inithw(); // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_MAILBOX_Create(&MyMailbox, MAX_MSG_SIZE, MAX_MSG_NUM, &MyMailboxBuffer);
    OS_COM_SendString("embOS OS_Mailbox example");
    OS_COM_SendString("\n\nDemonstrating message passing\n");
    OS_Start(); // Start embOS
    return 0;
}
```

10.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_MAILBOX_Clear()</code>	Clears all messages in the specified mailbox.	•	•	•	•	•
<code>OS_MAILBOX_Create()</code>	Creates a mailbox.	•	•			
<code>OS_MAILBOX_Delete()</code>	Deletes the specified mailbox.	•	•			
<code>OS_MAILBOX_Get()</code>	Retrieves a new message of a predefined size from the specified mailbox if a message is available.	•	•	•	•	•
<code>OS_MAILBOX_Get1()</code>	Retrieves a new message of size 1 from the specified mailbox, if a message is available.	•	•	•	•	•
<code>OS_MAILBOX_GetBlocked()</code>	Retrieves a new message of a predefined size from the specified mailbox.		•	•		
<code>OS_MAILBOX_GetBlocked1()</code>	Retrieves a new message of size 1 from the specified mailbox.		•	•		
<code>OS_MAILBOX_GetMessageCnt()</code>	Returns the number of messages currently available in the specified mailbox.	•	•	•	•	•
<code>OS_MAILBOX_GetTimed()</code>	Retrieves a new message of a predefined size from the specified mailbox if a message is available within a given time.		•	•		
<code>OS_MAILBOX_GetTimed1()</code>	Retrieves a new message of size 1 from the specified mailbox if a message is available within a given time.		•	•		
<code>OS_MAILBOX_GetPtr()</code>	Retrieves a pointer to a new message of a predefined size from the specified mailbox if a message is available.	•	•	•	•	•
<code>OS_MAILBOX_GetPtrBlocked()</code>	Retrieves a pointer to a new message of a predefined size from the specified mailbox.		•	•		
<code>OS_MAILBOX_IsInUse()</code>	Returns whether the specified mailbox is currently in use.	•	•	•	•	•
<code>OS_MAILBOX_Peek()</code>	Peeks a message from the specified mailbox without removing the message.	•	•	•	•	•
<code>OS_MAILBOX_Purge()</code>	Deletes the last retrieved message in the specified mailbox.	•	•	•	•	•
<code>OS_MAILBOX_Put()</code>	Stores a new message of a predefined size in the specified mailbox if the mailbox is able to accept one more message.	•	•	•	•	•
<code>OS_MAILBOX_Put1()</code>	Stores a new message of size 1 in the specified mailbox if the mailbox is able to accept one more message.	•	•	•	•	•

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_MAILBOX_PutBlocked()</code>	Stores a new message of a predefined size in the specified mailbox.		•	•		
<code>OS_MAILBOX_PutBlocked1()</code>	Stores a new message of size 1 in the specified mailbox.		•	•		
<code>OS_MAILBOX_PutFront()</code>	Stores a new message of a predefined size into the specified mailbox in front of all other messages if the mailbox is able to accept one more message.	•	•	•	•	•
<code>OS_MAILBOX_PutFront1()</code>	Stores a new message of size 1 into the specified mailbox in front of all other messages if the mailbox is able to accept one more message.	•	•	•	•	•
<code>OS_MAILBOX_PutFrontBlocked()</code>	Stores a new message of a predefined size at the beginning of the specified mailbox in front of all other messages.		•	•		
<code>OS_MAILBOX_PutFrontBlocked1()</code>	Stores a new message of size 1 at the beginning of the specified mailbox in front of all other messages.		•	•		
<code>OS_MAILBOX_PutTimed()</code>	Stores a new message of a predefined size in the specified mailbox if the mailbox is able to accept one more message within a given time.		•	•		
<code>OS_MAILBOX_PutTimed1()</code>	Stores a new message of size 1 in the specified mailbox if the mailbox is able to accept one more message within a given time.		•	•		
<code>OS_MAILBOX_WaitBlocked()</code>	Waits until a message is available, but does not retrieve the message from the specified mailbox.		•	•		
<code>OS_MAILBOX_WaitTimed()</code>	Waits until a message is available or the timeout has expired, but does not retrieve the message from the specified mailbox.		•	•		

10.2.1 OS_MAILBOX_Clear()

Description

Clears all messages in the specified mailbox.

Prototype

```
void OS_MAILBOX_Clear(OS_MAILBOX* pMailbox);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .

Additional information

A mailbox must not be cleared when it is in use. In use means the application currently holds a pointer to a message in the mailbox. `OS_MAILBOX_Clear()` may cause a task switch.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`130: OS_ERR_INV_MAILBOX`

`147: OS_ERR_MAILBOX_INUSE`

`164: OS_ERR_OS_INT_ENTER_NOT_CALLED`

`167: OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBKey;

void ClearKeyBuffer(void) {
    OS_MAILBOX_Clear(&_MBKey);
}
```

10.2.2 OS_MAILBOX_Create()

Description

Creates a mailbox.

Prototype

```
void OS_MAILBOX_Create(OS_MAILBOX* pMailbox,
                      OS_U16      sizeofMsg,
                      OS_UINT     maxnofMsg,
                      void*       pBuffer);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>sizeofMsg</code>	Size of a message in bytes. Valid values are $1 \leq \text{sizeofMsg} \leq 32,767$.
<code>maxnofMsg</code>	Maximum number of messages. Valid values are $1 \leq \text{MaxnofMsg} \leq 32,767$ on 8 or 16-bit CPUs, or $1 \leq \text{MaxnofMsg} \leq 2,147,483,647$ on 32-bit CPUs.
<code>pBuffer</code>	Pointer to a memory area used as buffer. The buffer must be big enough to hold the given number of messages of the specified size: <code>sizeofMsg * maxnoMsg</code> bytes. For 8/16-bit CPUs the total buffer size for one mailbox is limited to 65,536 bytes. For 32-bit CPUs the total buffer size for one mailbox is limited to 4,294,967,295 bytes.

Additional information

Mailboxes created with `OS_MAILBOX_Create()` resume a waiting task for every new message.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
106: OS_ERR_MB_BUFFER_SIZE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
165: OS_ERR_INIT_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
172: OS_ERR_2USE_MAILBOX
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Mailbox used as keyboard buffer:

```
static OS_MAILBOX _MBKey;
char             MBKeyBuffer[6];

void InitKeyMan(void) {
    //
    // Create mailbox, functioning as type ahead buffer
    //
    OS_MAILBOX_Create(&_MBKey, 1, sizeof(MBKeyBuffer), &MBKeyBuffer);
}
```

Mailbox used for transferring complex commands from one task to another:

```
//  
// Example of mailbox used for transferring commands to a task  
// that controls a motor  
//  
typedef struct {  
    char Cmd;  
    int Speed[2];  
    int Position[2];  
} MOTORCMD;  
  
OS_MAILBOX MBMotor;  
  
#define NUM_MOTORCMDS 4  
  
char BufferMotor[sizeof(MOTORCMD) * NUM_MOTORCMDS];  
  
void MOTOR_Init(void) {  
    // Create mailbox that holds commands messages  
    OS_MAILBOX_Create(&MBMotor, sizeof(MOTORCMD), NUM_MOTORCMDS, &BufferMotor);  
}
```

10.2.3 OS_MAILBOX_Delete()

Description

Deletes the specified mailbox.

Prototype

```
void OS_MAILBOX_Delete(OS_MAILBOX* pMailbox);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .

Additional information

A mailbox must not be deleted when it is in use or a task is waiting for it. In use means the application currently holds a pointer to a message in the mailbox.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
136: OS_ERR_MAILBOX_DELETE
147: OS_ERR_MAILBOX_INUSE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBSerIn;

void Cleanup(void) {
    OS_MAILBOX_Delete(&_MBSerIn);
}
```

10.2.4 OS_MAILBOX_Get()

Description

Retrieves a new message of a predefined size from the specified mailbox if a message is available.

Prototype

```
char OS_MAILBOX_Get(OS_MAILBOX* pMailbox,
                   void* pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the memory area that the message should be stored at. Make sure that there is sufficient space for an entire message. The message size was defined when the mailbox was created.

Return value

= 0 Success; message retrieved.
 ≠ 0 Message could not be retrieved (mailbox is empty); destination remains unchanged.

Additional information

If the mailbox is empty, no message is retrieved and the memory area where `pMessage` points to remains unchanged, but the program execution continues. This function never suspends the calling task. It may therefore also be called from an interrupt routine.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
147: OS_ERR_MAILBOX_INUSE
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define MESSAGE_SIZE 4

static OS_MAILBOX _MBData;
static char _Buffer[MESSAGE_SIZE];

char GetData(void) {
    return OS_MAILBOX_Get(&_MBData, &_Buffer);
}
```

10.2.5 OS_MAILBOX_Get1()

Description

Retrieves a new message of size 1 from the specified mailbox, if a message is available.

Prototype

```
char OS_MAILBOX_Get1(OS_MAILBOX* pMailbox,
                    char* pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the memory area that the message should be stored at. Make sure that there is sufficient space for an entire message. The message size was defined when the mailbox was created.

Return value

= 0 Success; message retrieved.
 ≠ 0 Message could not be retrieved (mailbox is empty); destination remains unchanged.

Additional information

If the mailbox is empty, no message is retrieved and the memory area where `pMessage` points to remains unchanged, but the program execution continues. This function never suspends the calling task. It may therefore also be called from an interrupt routine.

See *Single-byte mailbox functions* on page 229 for differences between `OS_MAILBOX_Get()` and `OS_MAILBOX_Get1()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
135: OS_ERR_MAILBOX_NOT1
147: OS_ERR_MAILBOX_INUSE
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBKey;

//
// If a key has been pressed, it is taken out of the mailbox
// and returned to caller. Otherwise zero is returned.
//
char GetKey(void) {
    char c = 0;

    OS_MAILBOX_Get1(&_MBKey, &c);
    return c;
}
```

10.2.6 OS_MAILBOX_GetBlocked()

Description

Retrieves a new message of a predefined size from the specified mailbox.

Prototype

```
void OS_MAILBOX_GetBlocked(OS_MAILBOX* pMailbox,
                          void*       pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the memory area that the message should be stored at. Make sure that there is sufficient space for an entire message. The message size was defined when the mailbox was created.

Additional information

If the mailbox is empty, the task is suspended until the mailbox receives a new message. Because this routine might require a suspension, it must not be called from an interrupt routine. Use `OS_MAILBOX_Get()/OS_MAILBOX_Get1()` instead if you need to retrieve data from a mailbox from within an interrupt routine.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
147: OS_ERR_MAILBOX_INUSE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define MESSAGE_SIZE 4

static OS_MAILBOX _MBData;
static char      _Buffer[MESSAGE_SIZE];

char WaitData(void) {
    return OS_MAILBOX_GetBlocked(&_MBData, &_Buffer);
}
```

10.2.7 OS_MAILBOX_GetBlocked1()

Description

Retrieves a new message of size 1 from the specified mailbox.

Prototype

```
void OS_MAILBOX_GetBlocked1(OS_MAILBOX* pMailbox,
                           char*       pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the memory area that the message should be stored at. Make sure that there is sufficient space for an entire one byte message.

Additional information

If the mailbox is empty, the task is suspended until the mailbox receives a new message. Because this routine might require a suspension, it must not be called from an interrupt routine. Use `OS_MAILBOX_Get()/OS_MAILBOX_Get1()` instead if you need to retrieve data from a mailbox from within an interrupt routine.

See *Single-byte mailbox functions* on page 229 for differences between `OS_MAILBOX_GetBlocked()` and `OS_MAILBOX_GetBlocked1()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
135: OS_ERR_MAILBOX_NOT1
147: OS_ERR_MAILBOX_INUSE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBKey;

char WaitKey(void) {
    char c;

    OS_MAILBOX_GetBlocked1(&_MBKey, &c);
    return c;
}
```

10.2.8 OS_MAILBOX_GetMessageCnt()

Description

Returns the number of messages currently available in the specified mailbox.

Prototype

```
OS_UINT OS_MAILBOX_GetMessageCnt(OS_CONST_PTR OS_MAILBOX *pMailbox);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .

Return value

The number of messages currently available in the mailbox.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

130: OS_ERR_INV_MAILBOX

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBData;

void PrintAvailableMessages() {
    OS_UINT NumOfMsgs;

    NumOfMsgs = OS_MAILBOX_GetMessageCnt(&_MBData);
    printf("Mailbox contains %u messages.\n", NumOfMsgs);
}
```

10.2.9 OS_MAILBOX_GetTimed()

Description

Retrieves a new message of a predefined size from the specified mailbox if a message is available within a given time.

Prototype

```
char OS_MAILBOX_GetTimed(OS_MAILBOX* pMailbox,
                        void* pMessage,
                        OS_TIME Timeout);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the memory area that the message should be stored at. Make sure that it points to a valid memory area and that there is sufficient space for an entire message. The message size (in bytes) was defined when the mailbox was created.
<code>Timeout</code>	Maximum time in system ticks until the requested message must be available. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Success; message retrieved.
 ≠ 0 Message could not be retrieved (mailbox is empty); destination remains unchanged.

Additional information

If the mailbox is empty, no message is retrieved and the task is suspended for the given timeout. The task continues execution according to the rules of the scheduler as soon as a message is available within the given timeout, or after the timeout value has expired. If the timeout has expired and no message was available within the timeout the memory area where `pMessage` points to remains unchanged.

When the calling task is blocked by higher priority tasks for a period longer than the timeout value, it may happen that message becomes available after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the message was not available within the requested time. In this case, no message is retrieved from the mailbox.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
147: OS_ERR_MAILBOX_INUSE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define MESSAGE_SIZE 4

static OS_MAILBOX _MBData;
static char      _Buffer[MESSAGE_SIZE];

char WaitData(void) {
    //
    // Wait for up to 10 system ticks
    //
    return OS_MAILBOX_GetTimed(&_MBData, &_Buffer, 10);
}
```

10.2.10 OS_MAILBOX_GetTimed1()

Description

Retrieves a new message of size 1 from the specified mailbox if a message is available within a given time.

Prototype

```
char OS_MAILBOX_GetTimed1(OS_MAILBOX* pMailbox,
                           char*       pMessage,
                           OS_TIME     Timeout);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the memory area that the message should be stored at. Make sure that it points to a valid memory area and that there is sufficient space for an entire message. The message size (in bytes) was defined when the mailbox was created.
<code>Timeout</code>	Maximum time in system ticks until the requested message must be available. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Success; message retrieved.
 ≠ 0 Message could not be retrieved (mailbox is empty); destination remains unchanged.

Additional information

If the mailbox is empty, no message is retrieved and the task is suspended for the given timeout. The task continues execution according to the rules of the scheduler as soon as a message is available within the given timeout, or after the timeout value has expired. If the timeout has expired and no message was available within the timeout the memory area where `pMessage` points to remains unchanged.

When the calling task is blocked by higher priority tasks for a period longer than the timeout value, it may happen that message becomes available after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the message was not available within the requested time. In this case, no message is retrieved from the mailbox.

See *Single-byte mailbox functions* on page 229 for differences between `OS_MAILBOX_GetTimed()` and `OS_MAILBOX_GetTimed1()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
135: OS_ERR_MAILBOX_NOT1
147: OS_ERR_MAILBOX_INUSE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBKey;
//
// If a key has been pressed, it is taken out of the mailbox
// and returned to caller. Otherwise zero is returned.
//
char GetKey(void) {
    char c = 0;
    OS_MAILBOX_GetTimed1(&_MBKey, &c, 10); // Wait for 10 system ticks
    return c;
}
```

10.2.11 OS_MAILBOX_GetPtr()

Description

Retrieves a pointer to a new message of a predefined size from the specified mailbox if a message is available.

Prototype

```
char OS_MAILBOX_GetPtr(OS_MAILBOX* pMailbox,
                      void**      ppMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>ppMessage</code>	Address of the pointer which will be set to the address of the message.

Return value

= 0 Success; message retrieved.
 ≠ 0 Message could not be retrieved (mailbox is empty); destination remains unchanged.

Additional information

If the mailbox is empty, no message is retrieved and `ppMessage` remains unchanged, but the program execution continues. This function never suspends the calling task. It may therefore also be called from an interrupt routine or software timer.

The retrieved message is not removed from the mailbox, this must be done by a call of `OS_MAILBOX_Purge()` after the message was processed. Only one message can be processed at a time. As long as the message is not removed from the mailbox, the mailbox is marked "in use". Following calls of `OS_MAILBOX_Clear()`, `OS_MAILBOX_Delete()`, `OS_MAILBOX_GetBlocked*()` and `OS_MAILBOX_GetPtrBlocked*()` functions are not allowed until `OS_MAILBOX_Purge()` is called.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
147: OS_ERR_MAILBOX_INUSE
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBKey;

void PrintMessage(void) {
    char* p;
    char r;

    r = OS_MAILBOX_GetPtr(&_MBKey, (void**)&p);
    if (r == 0) {
        printf("%d\n", *p);
        OS_MAILBOX_Purge(&_MBKey);
    }
}
```

10.2.12 OS_MAILBOX_GetPtrBlocked()

Description

Retrieves a pointer to a new message of a predefined size from the specified mailbox.

Prototype

```
void OS_MAILBOX_GetPtrBlocked(OS_MAILBOX* pMailbox,
                             void**      ppMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>ppMessage</code>	Pointer to the memory area that a pointer to the message should be stored at. The message size (in bytes) was defined when the mailbox was created.

Additional information

If the mailbox is empty, the task is suspended until the mailbox receives a new message. Because this routine might require a suspension, it must not be called from an interrupt routine. Use `OS_MAILBOX_GetPtr()` instead if you need to retrieve data from a mailbox from within an interrupt routine.

The retrieved message is not removed from the mailbox, this must be done by a call of `OS_MAILBOX_Purge()` after the message was processed. Only one message can be processed at a time. As long as the message is not removed from the mailbox, the mailbox is marked "in use". Following calls of `OS_MAILBOX_Clear()`, `OS_MAILBOX_Delete()`, `OS_MAILBOX_GetBlocked*()` and `OS_MAILBOX_GetPtrBlocked*()` functions are not allowed until `OS_MAILBOX_Purge()` is called.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
147: OS_ERR_MAILBOX_INUSE
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBKey;

void PrintMessage(void) {
    char* p;

    OS_MAILBOX_GetPtrBlocked(&_MBKey, (void**)&p);
    printf("%d\n", *p);
    OS_MAILBOX_Purge(&_MBKey);
}
```

10.2.13 OS_MAILBOX_IsInUse()

Description

Returns whether the specified mailbox is currently in use.

Prototype

```
OS_BOOL OS_MAILBOX_IsInUse(OS_CONST_PTR OS_MAILBOX *pMailbox);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .

Return value

= 0 Mailbox is not in use.
 ≠ 0 Mailbox is in use and may not be deleted or cleared.

Additional information

A mailbox must not be cleared or deleted when it is in use. In use means a task or function currently holds a pointer to a message in the mailbox.

`OS_MAILBOX_IsInUse()` can be used to examine the state of the mailbox before it can be cleared or deleted, as these functions must not be performed as long as the mailbox is used.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBKey;

void PrintMessage(void) {
    OS_BOOL IsInUse;

    IsInUse = OS_MAILBOX_IsInUse(&_MBKey);
    if (IsInUse == 0u) {
        printf("Mailbox is not in use.\n");
        OS_MAILBOX_Clear(&_MBKey);
    } else {
        printf("Mailbox is in use.\n");
    }
}
```

10.2.14 OS_MAILBOX_Peek()

Description

Peeks a message from the specified mailbox without removing the message. The message is copied to `*pMessage` if one was available.

Prototype

```
char OS_MAILBOX_Peek(OS_CONST_PTR OS_MAILBOX *pMailbox,
                    void* pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to a buffer that should receive the message.

Return value

= 0 Success, message was available and is copied to `*pMessage`.
 ≠ 0 Mail could not be retrieved (mailbox is empty).

Additional information

This function is non-blocking and never suspends the calling task. It may therefore be called from an interrupt routine. If no message was available the memory area where `pMessage` points to remains unchanged.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define MESSAGE_SIZE 4

static OS_MAILBOX _MBData;
static char      _Buffer[MESSAGE_SIZE];

char PeekData(void) {
    return OS_MAILBOX_Peek(&_MBData, &_Buffer);
}
```

10.2.15 OS_MAILBOX_Purge()

Description

Deletes the last retrieved message in the specified mailbox.

Prototype

```
void OS_MAILBOX_Purge(OS_MAILBOX* pMailbox);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .

Additional information

This routine should be called by the task that retrieved the last message from the mailbox, after the message is processed.

Once a message was retrieved by a call of `OS_MAILBOX_GetPtrBlocked()` or `OS_MAILBOX_GetPtr()`, the message must be removed from the mailbox by a call of `OS_MAILBOX_Purge()` before a following message can be retrieved from the mailbox. Following calls of `OS_MAILBOX_Clear()`, `OS_MAILBOX_Delete()`, `OS_MAILBOX_GetBlocked*()` and `OS_MAILBOX_GetPtrBlocked*()` functions are not allowed until `OS_MAILBOX_Purge()` is called. You must call `OS_MAILBOX_Purge()` only once after retrieving a message from the mailbox.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
148: OS_ERR_MAILBOX_NOT_INUSE
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBKey;

void PrintMessage(void) {
    char* p;

    OS_MAILBOX_GetPtrBlocked(&_MBKey, (void**)&p);
    printf("%d\n", *p);
    OS_MAILBOX_Purge(&_MBKey);
}
```

10.2.16 OS_MAILBOX_Put()

Description

Stores a new message of a predefined size in the specified mailbox if the mailbox is able to accept one more message.

Prototype

```
char OS_MAILBOX_Put(OS_MAILBOX* pMailbox,
                   OS_CONST_PTR void *pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the message to store.

Return value

= 0 Success; message stored.
 ≠ 0 Message could not be stored (mailbox is full).

Additional information

If the mailbox is full, the message is not stored. This function never suspends the calling task. It may therefore be called from an interrupt routine.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBData;

void AddMessage(struct Data* pData) {
    char Result;

    Result = OS_MAILBOX_Put(&_MBData, pData);
    if (Result != 0) {
        printf("Was not able to add the message to the mailbox.\n");
    }
}
```

10.2.17 OS_MAILBOX_Put1()

Description

Stores a new message of size 1 in the specified mailbox if the mailbox is able to accept one more message.

Prototype

```
char OS_MAILBOX_Put1(OS_MAILBOX* pMailbox,
                    OS_CONST_PTR char *pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the message to store.

Return value

= 0 Success; message stored.
 ≠ 0 Message could not be stored (mailbox is full).

Additional information

If the mailbox is full, the message is not stored. This function never suspends the calling task. It may therefore be called from an interrupt routine.

See *Single-byte mailbox functions* on page 229 for differences between `OS_MAILBOX_Put()` and `OS_MAILBOX_Put1()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
135: OS_ERR_MAILBOX_NOT1
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBKey;
static char      _MBKeyBuffer[6];

char KEYMAN_StoreCond(char k) {
    return OS_MAILBOX_Put1(&_MBKey, &k); /* Store key if space in buffer */
}
```

This example can be used with the sample program shown earlier to handle a mailbox as keyboard buffer.

10.2.18 OS_MAILBOX_PutBlocked()

Description

Stores a new message of a predefined size in the specified mailbox.

Prototype

```
void OS_MAILBOX_PutBlocked(OS_MAILBOX* pMailbox,
                          OS_CONST_PTR void *pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the message to store.

Additional information

If the mailbox is full, the calling task is suspended. Because this routine might require a suspension, it must not be called from an interrupt routine. Use `OS_MAILBOX_Put()`/`OS_MAILBOX_Put1()` instead if you need to store data in a mailbox from within an interrupt routine.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBData;

void AddMessage(struct Data* pData) {
    OS_MAILBOX_PutBlocked(&_MBData, pData);
}
```

10.2.19 OS_MAILBOX_PutBlocked1()

Description

Stores a new message of size 1 in the specified mailbox.

Prototype

```
void OS_MAILBOX_PutBlocked1(OS_MAILBOX* pMailbox,
                           OS_CONST_PTR char *pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the message to store.

Additional information

If the mailbox is full, the calling task is suspended. Because this routine might require a suspension, it must not be called from an interrupt routine. Use `OS_MAILBOX_Put()`/`OS_MAILBOX_Put1()` instead if you need to store data in a mailbox from within an interrupt routine.

See *Single-byte mailbox functions* on page 229 for differences between `OS_MAILBOX_PutBlocked()` and `OS_MAILBOX_PutBlocked1()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
135: OS_ERR_MAILBOX_NOT1
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Single-byte mailbox as keyboard buffer:

```
static OS_MAILBOX _MBKey;
static char      MBKeyBuffer[6];

void KEYMAN_StoreKey(char k) {
    OS_MAILBOX_PutBlocked1(&_MBKey, &k); /* Store key, wait if no space in buffer
    */
}

void KEYMAN_Init(void) {
    /* Create mailbox functioning as type ahead buffer */
    OS_MAILBOX_Create(&_MBKey, 1, sizeof(MBKeyBuffer), &MBKeyBuffer);
}
```

10.2.20 OS_MAILBOX_PutFront()

Description

Stores a new message of a predefined size into the specified mailbox in front of all other messages if the mailbox is able to accept one more message. The new message will be retrieved first.

Prototype

```
char OS_MAILBOX_PutFront(OS_MAILBOX* pMailbox,
                        OS_CONST_PTR void *pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the message to store.

Return value

= 0 Success; message stored.
 ≠ 0 Message could not be stored (mailbox is full).

Additional information

If the mailbox is full, the message is not stored. This function never suspends the calling task. It may therefore be called from an interrupt routine. This function is useful to store "emergency" messages into a mailbox which must be handled quickly. It may also be used in general instead of `OS_MAILBOX_Put()` to change the FIFO structure of a mailbox into a LIFO structure.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBData;

void AddMessage(struct Data* pData) {
    char Result;

    Result = OS_MAILBOX_PutFront(&_MBData, pData);
    if (Result != 0) {
        printf("Was not able to add the message to the mailbox.\n");
    }
}
```

10.2.21 OS_MAILBOX_PutFront1()

Description

Stores a new message of size 1 into the specified mailbox in front of all other messages if the mailbox is able to accept one more message. The new message will be retrieved first.

Prototype

```
char OS_MAILBOX_PutFront1(OS_MAILBOX* pMailbox,
                          OS_CONST_PTR char *pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the message to store.

Return value

= 0 Success; message stored.
 ≠ 0 Message could not be stored (mailbox is full).

Additional information

If the mailbox is full, the message is not stored. This function never suspends the calling task. It may therefore be called from an interrupt routine. This function is useful to store "emergency" messages into a mailbox which must be handled quickly. It may also be used in general instead of `OS_MAILBOX_Put()` to change the FIFO structure of a mailbox into a LIFO structure.

See *Single-byte mailbox functions* on page 229 for differences between `OS_MAILBOX_PutFront()` and `OS_MAILBOX_PutFront1()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
135: OS_ERR_MAILBOX_NOT1
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBData;

void AddMessage(char c) {
    char Result;

    Result = OS_MAILBOX_PutFront1(&_MBData, &c);
    if (Result != 0) {
        printf("Was not able to add the message to the mailbox.\n");
    }
}
```

10.2.22 OS_MAILBOX_PutFrontBlocked()

Description

Stores a new message of a predefined size at the beginning of the specified mailbox in front of all other messages. This new message will be retrieved first.

Prototype

```
void OS_MAILBOX_PutFrontBlocked(OS_MAILBOX* pMailbox,
                               OS_CONST_PTR void *pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the message to store.

Additional information

If the mailbox is full, the calling task is suspended. Because this routine might require a suspension, it must not be called from an interrupt routine. Use `OS_MAILBOX_PutFront()`/`OS_MAILBOX_PutFront1()` instead if you need to store data in a mailbox from within an interrupt routine.

This function is useful to store “emergency” messages into a mailbox which must be handled quickly. It may also be used in general instead of `OS_MAILBOX_PutBlocked()` to change the FIFO structure of a mailbox into a LIFO structure.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBData;

void AddMessage(struct Data* pData) {
    OS_MAILBOX_PutFrontBlocked(&_MBData, pData);
}
```

10.2.23 OS_MAILBOX_PutFrontBlocked1()

Description

Stores a new message of size 1 at the beginning of the specified mailbox in front of all other messages. This new message will be retrieved first.

Prototype

```
void OS_MAILBOX_PutFrontBlocked1(OS_MAILBOX* pMailbox,
                                OS_CONST_PTR char *pMessage);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the message to store.

Additional information

If the mailbox is full, the calling task is suspended. Because this routine might require a suspension, it must not be called from an interrupt routine. Use `OS_MAILBOX_PutFront()`/`OS_MAILBOX_PutFront1()` instead if you need to store data in a mailbox from within an interrupt routine.

This function is useful to store “emergency” messages into a mailbox which must be handled quickly. It may also be used in general instead of `OS_MAILBOX_PutBlocked()` to change the FIFO structure of a mailbox into a LIFO structure.

See *Single-byte mailbox functions* on page 229 for differences between `OS_MAILBOX_PutFrontBlocked()` and `OS_MAILBOX_PutFrontBlocked1()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
135: OS_ERR_MAILBOX_NOT1
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Single-byte mailbox as keyboard buffer which will follow the LIFO pattern:

```
static OS_MAILBOX _MBCmd;
static char      _MBCmdBuffer[6];

void KEYMAN_StoreCommand(char k) {
    OS_MAILBOX_PutFrontBlocked1(&_MBCmd, &k);
    // Store command, wait if no space in buffer
}

void KEYMAN_Init(void) {
    /* Create mailbox for command buffer */
    OS_MAILBOX_Create(&_MBCmd, 1, sizeof(_MBCmdBuffer), &_MBCmdBuffer);
}
```

10.2.24 OS_MAILBOX_PutTimed()

Description

Stores a new message of a predefined size in the specified mailbox if the mailbox is able to accept one more message within a given time. Returns when a new message has been stored in the mailbox (mailbox not full) or a timeout occurred.

Prototype

```
OS_BOOL OS_MAILBOX_PutTimed(OS_MAILBOX* pMailbox,
                             OS_CONST_PTR void *pMessage,
                             OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the message to store.
<code>Timeout</code>	Maximum time in embOS system ticks until the given message must be stored. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Success; message stored.
 ≠ 0 Message could not be stored within the given timeout (mailbox is full). destination remains unchanged.

Additional information

If the mailbox is full, no message is stored and the task is suspended for the given timeout. The task continues execution according to the rules of the scheduler as soon as a new message is accepted within the given timeout, or after the timeout value has expired.

When the calling task is blocked by higher priority tasks for a period longer than the timeout value, it may happen that the mailbox accepts new messages after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the mailbox was not available within the requested time. In this case, no message is stored in the mailbox.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBData;

void AddMessage(char* pData) {
    OS_MAILBOX_PutTimed(&_MBData, pData, 10); // Wait maximum 10 system ticks
}
```

10.2.25 OS_MAILBOX_PutTimed1()

Description

Stores a new message of size 1 in the specified mailbox if the mailbox is able to accept one more message within a given time. Returns when a new message has been stored in the mailbox (mailbox not full) or a timeout occurred.

Prototype

```
OS_BOOL OS_MAILBOX_PutTimed1(OS_MAILBOX* pMailbox,
                             OS_CONST_PTR char *pMessage,
                             OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>pMessage</code>	Pointer to the message to store.
<code>Timeout</code>	Maximum time in embOS system ticks until the given message must be stored. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Success; message stored.
 ≠ 0 Message could not be stored within the given timeout (mailbox is full). destination remains unchanged.

Additional information

If the mailbox is full, no message is stored and the task is suspended for the given timeout. The task continues execution according to the rules of the scheduler as soon as a new message is accepted within the given timeout, or after the timeout value has expired.

When the calling task is blocked by higher priority tasks for a period longer than the timeout value, it may happen that the mailbox accepts new messages after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the mailbox was not available within the requested time. In this case, no message is stored in the mailbox.

See *Single-byte mailbox functions* on page 229 for differences between `OS_MAILBOX_PutTimed()` and `OS_MAILBOX_PutTimed1()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
135: OS_ERR_MAILBOX_NOT1
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBKey;
```

```
void SetKey(char c) {  
    OS_MAILBOX_PutTimed1(&_MBKey, &c, 10); // Wait maximum 10 system ticks  
}
```

10.2.26 OS_MAILBOX_WaitBlocked()

Description

Waits until a message is available, but does not retrieve the message from the specified mailbox.

Prototype

```
void OS_MAILBOX_WaitBlocked(OS_MAILBOX* pMailbox);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .

Additional information

If the mailbox is empty, the task is suspended until a message is available, otherwise the task continues. The task continues execution according to the rules of the scheduler as soon as a message is available, but the message is not retrieved from the mailbox.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBData;

void Task(void) {
    while (1) {
        OS_MAILBOX_WaitBlocked(&_MBData);
        ...
    }
}
```

10.2.27 OS_MAILBOX_WaitTimed()

Description

Waits until a message is available or the timeout has expired, but does not retrieve the message from the specified mailbox.

Prototype

```
char OS_MAILBOX_WaitTimed(OS_MAILBOX* pMailbox,
                          OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type <code>OS_MAILBOX</code> .
<code>Timeout</code>	Maximum time in embOS system ticks until the requested message must be available. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Success; message available.
 ≠ 0 `Timeout`; no message available within the given timeout time.

Additional information

If the mailbox is empty, the task is suspended for the given timeout. The task continues execution according to the rules of the scheduler as soon as a message is available within the given timeout, or after the timeout value has expired.

When the calling task is blocked by higher priority tasks for a period longer than the timeout value, it may happen that message becomes available after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the message was not available within the requested time.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
130: OS_ERR_INV_MAILBOX
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MAILBOX _MBData;

void Task(void) {
    char Result;

    Result = OS_MAILBOX_WaitTimed(&_MBData, 10);
    if (Result == 0) {
        // Compute message
    } else {
        // Timeout
    }
}
```

```
}
```

Chapter 11

Queue

11.1 Introduction

In the preceding chapter, inter-task communication using mailboxes was described. Mailboxes can handle small messages with fixed data size only. Queues enable inter-task communication with larger messages or with messages of differing lengths.

A queue consists of a data buffer and a control structure that is managed by the real-time operating system. The queue behaves like a normal buffer; you can deposit something (called a message) in the queue and retrieve it later. Queues work as FIFO: first in, first out. So a message that is deposited first will be retrieved first. There are three major differences between queues and mailboxes:

1. Queues accept messages of differing lengths. When depositing a message into a queue, the message size is passed as a parameter.
2. Retrieving a message from the queue does not copy the message, but returns a pointer to the message and its size. This enhances performance because the data is copied only when the message is written into the queue.
3. The retrieving function must delete every message after processing it.
4. A new message can only be retrieved from the queue when the previous message was deleted from the queue.

The queue data buffer contains the messages and some additional management information. Each message has a message header containing the message size. The define `OS_Q_SIZEOF_HEADER` defines the size of the message header. Additionally, the queue buffer will be aligned for those CPUs which need data alignment. Therefore the queue data buffer size must be bigger than the sum of all messages.

Limitations:

Both the number of queues and buffers are limited only by the amount of available memory. However, the individual message size and the buffer size per queue are limited by software design.

	8 or 16-bit CPUs	32-bit CPUs
Maximum message size	32,767 bytes	2,147,483,647 bytes
Maximum buffer size	65,535 bytes	4,294,967,295 bytes

These limitations have been placed on queues to guarantee efficient coding and also to ensure efficient management. These limitations are typically not a problem.

Similar to mailboxes, queues can be used by more than one producer, but must be used by one consumer only. This means that more than one task or interrupt handler is allowed to deposit new data into the queue, but it does not make sense to retrieve messages by multiple tasks.

Example

```

#define MESSAGE_ALIGNMENT      (4u) // Depends on core/compiler
#define MESSAGES_SIZE_HELLO   (7u + OS_Q_SIZEOF_HEADER + MESSAGE_ALIGNMENT)
#define MESSAGES_SIZE_WORLD   (9u + OS_Q_SIZEOF_HEADER + MESSAGE_ALIGNMENT)
#define QUEUE_SIZE             (MESSAGES_SIZE_HELLO + MESSAGES_SIZE_WORLD)

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task-control-blocks
static OS_QUEUE     MyQueue;
static char         MyQBuffer[QUEUE_SIZE];

static void HPTask(void) {
    char* pData;
    int   Len;

    while (1) {
        Len = OS_QUEUE_GetPtrBlocked(&MyQueue, (void**)&pData);
        OS_TASK_Delay(10);
        //
        // Evaluate Message
        //
        if (Len > 0) {
            OS_COM_SendString(pData);
            OS_QUEUE_Purge(&MyQueue);
        }
    }
}

static void LPTask(void) {
    while (1) {
        OS_QUEUE_Put(&MyQueue, "\nHello\n", 7);
        OS_QUEUE_Put(&MyQueue, "\nWorld !\n", 9);
        OS_TASK_Delay(500);
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_Inithw(); // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_QUEUE_Create(&MyQueue, &MyQBuffer, sizeof(MyQBuffer));
    OS_COM_SendString("embOS OS_Queue example");
    OS_COM_SendString("\n\nDemonstrating message passing\n");
    OS_Start(); // Start embOS
    return 0;
}

```

11.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_QUEUE_Clear()</code>	Clears all messages in the specified message queue.	•	•	•	•	•
<code>OS_QUEUE_Create()</code>	Creates a message queue.	•	•		•	•
<code>OS_QUEUE_Delete()</code>	Deletes the specific message queue.	•	•		•	•
<code>OS_QUEUE_GetMessageCnt()</code>	Returns the number of messages that are currently stored in the specified message queue.	•	•	•	•	•
<code>OS_QUEUE_GetMessageSize()</code>	Returns the size of the first message in the specified message queue.	•	•	•	•	•
<code>OS_QUEUE_GetPtr()</code>	Retrieves a pointer to a message from the specified message queue if available.	•	•	•	•	•
<code>OS_QUEUE_GetPtrBlocked()</code>	Retrieves a pointer to a message from the specified message queue.		•	•		
<code>OS_QUEUE_GetPtrTimed()</code>	Retrieves a pointer to a message from the specified message queue if available within the specified time.		•	•		
<code>OS_QUEUE_HasFreeSpace()</code>	Returns whether the message of size Size fits in the queue buffer.	•	•	•	•	•
<code>OS_QUEUE_IsInUse()</code>	Delivers information whether the specified message queue is currently in use.	•	•	•	•	•
<code>OS_QUEUE_PeekPtr()</code>	Retrieve the pointer to a message from the specified message queue.	•	•	•	•	•
<code>OS_QUEUE_Purge()</code>	Deletes the last retrieved message in the specified message queue.	•	•	•	•	•
<code>OS_QUEUE_Put()</code>	Stores a new message of given size in the specified message queue.	•	•	•	•	•
<code>OS_QUEUE_PutEx()</code>	Stores a new message, of which the distinct parts are distributed in memory as indicated by a <code>OS_QUEUE_SRLIST</code> structure, in the specified message queue.	•	•	•	•	•
<code>OS_QUEUE_PutBlocked()</code>	Stores a new message of given size in the specified message queue.		•	•		
<code>OS_QUEUE_PutBlockedEx()</code>	Stores a new message, of which the distinct parts are distributed in memory as indicated by a <code>OS_QUEUE_SRLIST</code> structure, in the specified message queue.		•	•		
<code>OS_QUEUE_PutTimed()</code>	Stores a new message of given size in the specified message queue if space is available within a given time.		•	•		
<code>OS_QUEUE_PutTimedEx()</code>	Stores a new message, of which the distinct parts are distributed in memory as indicated by a <code>OS_QUEUE_SRLIST</code>		•	•		

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
	CLIST structure, in the specified message queue.					

11.2.1 OS_QUEUE_Clear()

Description

Clears all messages in the specified message queue.

Prototype

```
void OS_QUEUE_Clear(OS_QUEUE* pQ);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .

Additional information

A queue must not be cleared when it is in use. In use means the application currently holds a pointer to a message in the queue. `OS_QUEUE_Clear()` may cause a task switch.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

145: OS_ERR_QUEUE_INVALID

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

143: OS_ERR_QUEUE_INUSE

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _Queue;

void ClearQueue() {
    OS_QUEUE_Clear(&_Queue);
}
```

11.2.2 OS_QUEUE_Create()

Description

Creates a message queue.

Prototype

```
void OS_QUEUE_Create(OS_QUEUE* pQ,
                    void*      pBuffer,
                    OS_UINT    Size);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>pBuffer</code>	Pointer to a memory area used as data buffer for the queue.
<code>Size</code>	<code>Size</code> in bytes of the data buffer.

Additional information

The define `OS_Q_SIZEOF_HEADER` can be used to calculate the additional management information bytes needed for each message in the queue data buffer. But it does not account for the additional space needed for data alignment. Thus the number of messages that can actually be stored in the queue buffer depends on the message sizes. The message size will be round up to the next multiple of the size of an integer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
165: OS_ERR_INIT_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
177: OS_ERR_2USE_QUEUE
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define MESSAGE_CNT 100
#define MESSAGE_SIZE 100
#define MEMORY_QSIZE (MESSAGE_CNT * (MESSAGE_SIZE + OS_Q_SIZEOF_HEADER))

static OS_QUEUE _MemoryQ;
static char      _acMemQBuffer[MEMORY_QSIZE];

void MEMORY_Init(void) {
    OS_QUEUE_Create(&_MemoryQ, &_acMemQBuffer, sizeof(_acMemQBuffer));
}
```

11.2.3 OS_QUEUE_Delete()

Description

Deletes the specific message queue.

Prototype

```
void OS_QUEUE_Delete(OS_QUEUE* pQ);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .

Additional information

A queue must not be deleted when it is in use or a task is waiting for it. In use means the application currently holds a pointer to a message in the mailbox.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
143: OS_ERR_QUEUE_INUSE
145: OS_ERR_QUEUE_INVALID
146: OS_ERR_QUEUE_DELETE
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _QSerIn;

void Cleanup(void) {
    OS_QUEUE_Delete(&_QSerIn);
}
```

11.2.4 OS_QUEUE_GetMessageCnt()

Description

Returns the number of messages that are currently stored in the specified message queue.

Prototype

```
int OS_QUEUE_GetMessageCnt(OS_CONST_PTR OS_QUEUE *pQ);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .

Return value

The number of messages in the queue.

Additional information

If `OS_QUEUE_GetMessageCnt()` returns zero the message queue is empty.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

145: OS_ERR_QUEUE_INVALID

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _Queue;

void PrintNumberOfMessages() {
    int Cnt;

    Cnt = OS_QUEUE_GetMessageCnt(&_Queue);
    printf("%d messages available.\n", Cnt);
}
```

11.2.5 OS_QUEUE_GetMessageSize()

Description

Returns the size of the first message in the specified message queue.

Prototype

```
int OS_QUEUE_GetMessageSize(OS_CONST_PTR OS_QUEUE *pQ);
```

Parameters

Parameter	Description
pQ	Pointer to a queue object of type OS_QUEUE.

Return value

= 0 No data available.
> 0 Size of message in bytes.

Additional information

If the queue is empty OS_QUEUE_GetMessageSize() returns zero. If a message is available OS_QUEUE_GetMessageSize() returns the size of that message. The message is not retrieved from the queue.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
145: OS_ERR_QUEUE_INVALID
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _MemoryQ;

static void _MemoryTask(void) {
    int Len;

    while (1) {
        Len = OS_QUEUE_GetMessageSize(&_MemoryQ); // Get message length
        if (Len > 0) {
            printf("Message with size %d retrieved\n", Len);
            OS_QUEUE_Purge(&_MemoryQ); // Delete message
        }
        OS_TASK_Delay(10);
    }
}
```

11.2.6 OS_QUEUE_GetPtr()

Description

Retrieves a pointer to a message from the specified message queue if available.

Prototype

```
int OS_QUEUE_GetPtr(OS_QUEUE* pQ,
                   void** ppMessage);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>ppMessage</code>	Address of the pointer which will be set to the address of the message.

Return value

- = 0 No message available in queue (queue is empty).
- > 0 Size of the message that was retrieved from the queue.

Additional information

If the queue is empty, `OS_QUEUE_GetPtr()` returns zero and `ppMessage` remains unchanged. This function never suspends the calling task. It may therefore be called from an interrupt routine or software timer.

The retrieved message is not removed from the queue, this must be done by a call of `OS_QUEUE_Purge()` after the message was processed. Only one message can be processed at a time. As long as the message is not removed from the queue, the queue is marked "in use". Following calls of `OS_QUEUE_Clear()`, `OS_QUEUE_Delete()`, `OS_QUEUE_GetPtr()`, `OS_QUEUE_GetPtrBlocked()` and `OS_QUEUE_GetPtrTimed()` functions are not allowed until `OS_QUEUE_Purge()` is called.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
143: OS_ERR_QUEUE_INUSE
145: OS_ERR_QUEUE_INVALID
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _MemoryQ;

static void _MemoryTask(void) {
    int Len;
    char* pData;

    while (1) {
        Len = OS_QUEUE_GetPtr(&_MemoryQ, &pData); // Check message
        if (Len > 0) {
            Memory_WritePacket(*(U32*)pData, Len); // Process message
            OS_QUEUE_Purge(&_MemoryQ); // Delete message
        } else {
            DoSomethingElse();
        }
    }
}
```

```
}
```

11.2.7 OS_QUEUE_GetPtrBlocked()

Description

Retrieves a pointer to a message from the specified message queue.

Prototype

```
int OS_QUEUE_GetPtrBlocked(OS_QUEUE* pQ,
                          void**  ppMessage);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>ppMessage</code>	Address of the pointer which will be set to the address of the message.

Return value

Size of the message in bytes.

Additional information

If the queue is empty, the calling task is suspended until the queue receives a new message. Because this routine might require a suspension, it must not be called from an interrupt routine or timer. Use `OS_QUEUE_GetPtr()` instead. The retrieved message is not removed from the queue, this must be done by a call of `OS_QUEUE_Purge()` after the message was processed. Only one message can be processed at a time. As long as the message is not removed from the queue, the queue is marked "in use".

Following calls of `OS_QUEUE_Clear()`, `OS_QUEUE_Delete()`, `OS_QUEUE_GetPtr()`, `OS_QUEUE_GetPtrBlocked()` and `OS_QUEUE_GetPtrTimed()` functions are not allowed until `OS_QUEUE_Purge()` is called.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
143: OS_ERR_QUEUE_INUSE
145: OS_ERR_QUEUE_INVALID
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _MemoryQ;

static void _MemoryTask(void) {
    int  Len;
    char* pData;

    while (1) {
        Len = OS_QUEUE_GetPtrBlocked(&_MemoryQ, &pData); // Get message
        Memory_WritePacket(*(U32*)pData, Len);           // Process message
        OS_QUEUE_Purge(&_MemoryQ);                       // Delete message
    }
}
```

11.2.8 OS_QUEUE_GetPtrTimed()

Description

Retrieves a pointer to a message from the specified message queue if available within the specified time.

Prototype

```
int OS_QUEUE_GetPtrTimed(OS_QUEUE* pQ,
                        void** ppMessage,
                        OS_TIME Timeout);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>ppMessage</code>	Address of the pointer which will be set to the address of the message.
<code>Timeout</code>	Maximum time in system ticks until the requested message must be available. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 No message available in queue.
 > 0 Size of the message that was retrieved from the queue.

Sets the pointer `ppMessage` to the message that should be retrieved.

Additional information

If the queue is empty no message is retrieved, the task is suspended for the given timeout. The task continues execution according to the rules of the scheduler as soon as a message is available within the given timeout, or after the timeout value has expired. If no message is retrieved within the timeout `ppMessage` will not be set.

When the calling task is blocked by higher priority tasks for a period longer than the timeout value, it may happen that a message becomes available after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the message was not available within the requested time. In this case the state of the queue is not modified by `OS_QUEUE_GetPtrTimed()` and a pointer to the message is not delivered. As long as a message was retrieved and the message is not removed from the queue, the queue is marked "in use".

Following calls of `OS_QUEUE_Clear()`, `OS_QUEUE_Delete()`, `OS_QUEUE_GetPtr()`, `OS_QUEUE_GetPtrBlocked()` and `OS_QUEUE_GetPtrTimed()` functions are not allowed until `OS_QUEUE_Purge()` is called.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
143: OS_ERR_QUEUE_INUSE
145: OS_ERR_QUEUE_INVALID
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _MemoryQ;

static void _MemoryTask(void) {
    int    Len;
    char*  pData;

    while (1) {
        Len = OS_QUEUE_GetPtrTimed(&_MemoryQ, &pData, 10); // Check message
        if (Len > 0) {
            Memory_WritePacket(*(U32*)pData, Len);           // Process message
            OS_QUEUE_Purge(&_MemoryQ);                       // Delete message
        } else {                                             // Timeout
            DoSomethingElse();
        }
    }
}
```

11.2.9 OS_QUEUE_HasFreeSpace()

Description

Returns whether the message of size `Size` fits in the queue buffer.

Prototype

```
OS_BOOL OS_QUEUE_HasFreeSpace(OS_CONST_PTR OS_QUEUE *pQ,
                               OS_UINT      Size);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>Size</code>	Message size.

Additional information

`OS_QUEUE_HasFreeSpace()` does not store any message in the queue.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
143: OS_ERR_QUEUE_INUSE
145: OS_ERR_QUEUE_INVALID
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(OS_QUEUE* pQ) {
    OS_BOOL r;

    OS_INT_IncDI();
    r = OS_QUEUE_HasFreeSpace(pQ, 42)
    if (r == 1u) {
        OS_QUEUE_Put(pQ, pMessage, 42);
    }
    OS_INT_DecRI();
}
```

11.2.10 OS_QUEUE_IsInUse()

Description

Delivers information whether the specified message queue is currently in use.

Prototype

```
OS_BOOL OS_QUEUE_IsInUse(OS_CONST_PTR OS_QUEUE *pQ);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .

Return value

= 0 Queue is not in use.
 ≠ 0 Queue is in use and may not be deleted or cleared.

Additional information

A queue must not be cleared or deleted when it is in use. In use means a task or function currently accesses the queue and holds a pointer to a message in the queue.

`OS_QUEUE_IsInUse()` can be used to examine the state of the queue before it can be cleared or deleted, as these functions must not be performed as long as the queue is used.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
145: OS_ERR_QUEUE_INVALID
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void DeleteQ(OS_QUEUE* pQ) {
    OS_INT_IncDI(); // Avoid state change of the queue by task or interrupt
    //
    // Wait until queue is not used
    //
    while (OS_QUEUE_IsInUse(pQ) != 0) {
        OS_TASK_Delay(1);
    }
    OS_QUEUE_Delete(pQ);
    OS_INT_DecRI();
}
```

11.2.11 OS_QUEUE_PeekPtr()

Description

Retrieve the pointer to a message from the specified message queue. The message must not be purged.

Prototype

```
int OS_QUEUE_PeekPtr(OS_CONST_PTR OS_QUEUE *pQ,
                    void**      ppMessage);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>ppMessage</code>	Address of the pointer which will be set to the address of the message.

Return value

= 0 No message available.
 ≠ 0 Size of message in bytes.

Additional information

Sets the pointer `ppMessage` to the message that should be retrieved. If no message is available `ppMessage` will not be set.

Note

Ensure the queues state is not altered as long as a message is processed. That is the reason for calling `OS_INT_IncDI()` in the sample. Ensure no cooperative task switch is performed, as this may also alter the queue state and buffer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
145: OS_ERR_QUEUE_INVALID
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _MemoryQ;
static void _MemoryTask(void) {
    int Len;
    char* pData;

    while (1) {
        // Avoid state changes of the queue by task or interrupt
        OS_INT_IncDI();
        Len = OS_QUEUE_PeekPtr(&_MemoryQ, &pData); // Get message
        if (Len > 0) {
            Memory_WritePacket(*(U32*)pData, Len); // Process message
        }
        OS_INT_DecRI();
    }
}
```

11.2.12 OS_QUEUE_Purge()

Description

Deletes the last retrieved message in the specified message queue.

Prototype

```
void OS_QUEUE_Purge(OS_QUEUE* pQ);
```

Parameters

Parameter	Description
pQ	Pointer to a queue object of type OS_QUEUE.

Additional information

This routine should be called by the task that retrieved the last message from the queue, after the message is processed.

Once a message was retrieved by a call of `OS_QUEUE_GetPtrBlocked()`, `OS_QUEUE_GetPtr()` or `OS_QUEUE_GetPtrTimed()`, the message must be removed from the queue by a call of `OS_QUEUE_Purge()` before a following message can be retrieved from the queue.

Consecutive calls of `OS_QUEUE_Purge()` or calling `OS_QUEUE_Purge()` without having retrieved a message from the queue are not allowed.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

143: `OS_ERR_QUEUE_INUSE`

145: `OS_ERR_QUEUE_INVALID`

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _MemoryQ;

static void _MemoryTask(void) {
    int    Len;
    char*  pData;

    while (1) {
        Len = OS_QUEUE_GetPtrBlocked(&_MemoryQ, &pData); // Get message
        Memory_WritePacket(*(U32*)pData, Len);           // Process message
        OS_QUEUE_Purge(&_MemoryQ);                       // Delete message
    }
}
```

11.2.13 OS_QUEUE_Put()

Description

Stores a new message of given size in the specified message queue.

Prototype

```
int OS_QUEUE_Put(OS_QUEUE*   pQ,
                 OS_CONST_PTR void *pMessage,
                 OS_UINT     Size);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>pMessage</code>	Pointer to the message to store.
<code>Size</code>	Size of the message to store. Valid values are: $1 \leq \text{Size} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Size} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Success, message stored.
 ≠ 0 Message could not be stored (queue is full).

Additional information

This routine never suspends the calling task and may therefore be called from an interrupt routine.

When the message is deposited into the queue, the entire message is copied into the queue buffer, not only the pointer to the data. Therefore the message content is protected and remains valid until it is retrieved and accessed by a task reading the message.

Note

The message is stored in the queue buffer with an additional message header. Additionally the message size will be round up to the next multiple of the size of an integer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
145: OS_ERR_QUEUE_INVALID
149: OS_ERR_MESSAGE_SIZE_ZERO
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _MemoryQ;

int MEMORY_Write(const char* pData, OS_UINT Len) {
    return OS_QUEUE_Put(&_MemoryQ, pData, Len);
}
```

11.2.14 OS_QUEUE_PutEx()

Description

Stores a new message, of which the distinct parts are distributed in memory as indicated by a `OS_QUEUE_SRCLIST` structure, in the specified message queue.

Prototype

```
int OS_QUEUE_PutEx(OS_QUEUE*    pQ,
                  OS_CONST_PTR OS_QUEUE_SRCLIST *pMessageList,
                  OS_UINT      NumMessages);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>pMessageList</code>	Pointer to an array of <code>OS_QUEUE_SRCLIST</code> structures which contain pointers to the data to store.
<code>NumMessages</code>	Number of <code>OS_QUEUE_SRCLIST</code> structures at <code>pMessageList</code> .

Return value

= 0 Success, message stored.
 ≠ 0 Message could not be stored (queue is full).

Additional information

This routine never suspends the calling task and may therefore be called from `main()`, an interrupt routine or a software timer.

When the message is deposited into the queue, the entire message is copied into the queue buffer, not only the pointer(s) to the data. Therefore the message content is protected and remains valid until it is retrieved and accessed by a task reading the message.

The `OS_QUEUE_SRCLIST` structure consists of two elements:

Parameter	Description
<code>pSrc</code>	Pointer to a part of the message to store.
<code>Size</code>	Size of the part of the message. Valid values are: $1 \leq \text{Size} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Size} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Note

The total size of all parts of the message must not exceed `0x7FFF` on 8/16-bit CPUs, or `0x7FFFFFFF` on 32-bit CPUs, respectively. The message is stored in the queue buffer with an additional message header. Additionally the message size will be round up to the next multiple of the size of an integer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
145: OS_ERR_QUEUE_INVALID
149: OS_ERR_MESSAGE_SIZE_ZERO
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
OS_CONST_PTR OS_QUEUE_SRCLIST aDataList[] = { { "Hello ", 6},
                                               { "World!", 6}
                                               };
OS_QUEUE_PutEx(&_amp;MemoryQ, aDataList, 2);
```

11.2.15 OS_QUEUE_PutBlocked()

Description

Stores a new message of given size in the specified message queue.

Prototype

```
void OS_QUEUE_PutBlocked(OS_QUEUE*   pQ,
                        OS_CONST_PTR void *pMessage,
                        OS_UINT      Size);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>pMessage</code>	Pointer to the message to store.
<code>Size</code>	<code>Size</code> of the message to store. Valid values are: $1 \leq \text{Size} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Size} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Additional information

If the queue is full, the calling task is suspended.

When the message is deposited into the queue, the entire message is copied into the queue buffer, not only the pointer(s) to the data. Therefore the message content is protected and remains valid until it is retrieved and accessed by a task reading the message.

Note

The message is stored in the queue buffer with an additional message header. Additionally the message size will be round up to the next multiple of the size of an integer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
145: OS_ERR_QUEUE_INVALID
149: OS_ERR_MESSAGE_SIZE_ZERO
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _MemoryQ;

void StoreMessage(const char* pData, OS_UINT Len)
{
    OS_QUEUE_PutBlocked(&_MemoryQ, pData, Len);
}
```

11.2.16 OS_QUEUE_PutBlockedEx()

Description

Stores a new message, of which the distinct parts are distributed in memory as indicated by a `OS_QUEUE_SRCLIST` structure, in the specified message queue. Blocks the calling task when queue is full.

Prototype

```
void OS_QUEUE_PutBlockedEx(OS_QUEUE* pQ,
                          OS_CONST_PTR OS_QUEUE_SRCLIST *pMessageList,
                          OS_UINT NumMessages);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>pMessageList</code>	Pointer to an array of <code>OS_QUEUE_SRCLIST</code> structures which contain pointers to the data to store.
<code>NumMessages</code>	Number of <code>OS_QUEUE_SRCLIST</code> structures at <code>pMessageList</code> .

Additional information

If the queue is full, the calling task is suspended.

When the message is deposited into the queue, the entire message is copied into the queue buffer, not only the pointer(s) to the data. Therefore the message content is protected and remains valid until it is retrieved and accessed by a task reading the message.

The `OS_QUEUE_SRCLIST` structure consists of two elements:

Parameter	Description
<code>pSrc</code>	Pointer to a part of the message to store.
<code>Size</code>	Size of the part of the message. Valid values are: $1 \leq \text{Size} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Size} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Note

The total size of all parts of the message must not exceed `0x7FFF` on 8/16-bit CPUs, or `0x7FFFFFFF` on 32-bit CPUs, respectively. The message is stored in the queue buffer with an additional message header. Additionally the message size will be round up to the next multiple of the size of an integer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
145: OS_ERR_QUEUE_INVALID
149: OS_ERR_MESSAGE_SIZE_ZERO
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
OS_CONST_PTR OS_QUEUE_SRCLIST aDataList[] = { { "Hello ", 6},  
                                              { "World!", 6}  
                                              };  
OS_QUEUE_PutEx(&_amp;MemoryQ, aDataList, 2);
```

11.2.17 OS_QUEUE_PutTimed()

Description

Stores a new message of given size in the specified message queue if space is available within a given time.

Prototype

```
char OS_QUEUE_PutTimed(OS_QUEUE*   pQ,
                      OS_CONST_PTR void *pMessage,
                      OS_UINT      Size,
                      OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>pMessage</code>	Pointer to the message to store.
<code>Size</code>	<code>Size</code> of the message to store. Valid values are: $1 \leq \text{Size} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Size} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.
<code>Timeout</code>	Maximum time in system ticks until the given message must be stored. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Success, message stored.
 ≠ 0 Message could not be stored within the specified time (insufficient space).

Additional information

If the queue holds insufficient space, the calling task is suspended until space for the message is available, or the specified timeout time has expired. If the message could be deposited into the queue within the specified time, the function returns zero.

When the message is deposited into the queue, the entire message is copied into the queue buffer, not only the pointer(s) to the data. Therefore the message content is protected and remains valid until it is retrieved and accessed by a task reading the message.

Note

The message is stored in the queue buffer with an additional message header. Additionally the message size will be round up to the next multiple of the size of an integer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
145: OS_ERR_QUEUE_INVALID
149: OS_ERR_MESSAGE_SIZE_ZERO
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_QUEUE _MemoryQ;

int MEMORY_WriteTimed(const char* pData, OS_UINT Len, OS_TIME Timeout) {
    return OS_QUEUE_PutTimed(&_MemoryQ, pData, Len, Timeout);
}
```

11.2.18 OS_QUEUE_PutTimedEx()

Description

Stores a new message, of which the distinct parts are distributed in memory as indicated by a `OS_QUEUE_SRCLIST` structure, in the specified message queue. Suspends the calling task for a given timeout when the queue is full.

Prototype

```
char OS_QUEUE_PutTimedEx(OS_QUEUE*    pQ,
                        OS_CONST_PTR OS_QUEUE_SRCLIST *pMessageList,
                        OS_UINT      NumMessages,
                        OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>pQ</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>pMessageList</code>	Pointer to an array of <code>OS_QUEUE_SRCLIST</code> structures which contain pointers to the data to store.
<code>NumMessages</code>	Number of <code>OS_QUEUE_SRCLIST</code> structures at <code>pMessageList</code> .
<code>Timeout</code>	Maximum time in system ticks until the given message must be stored. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= 0 Success, message stored.
 ≠ 0 Message could not be stored within the specified time (insufficient space).

Additional information

If the queue holds insufficient space, the calling task is suspended until space for the message is available or the specified timeout time has expired. If the message could be deposited into the queue within the specified time, the function returns zero.

When the message is deposited into the queue, the entire message is copied into the queue buffer, not only the pointer(s) to the data. Therefore the message content is protected and remains valid until it is retrieved and accessed by a task reading the message.

The `OS_QUEUE_SRCLIST` structure consists of two elements:

Parameter	Description
<code>pSrc</code>	Pointer to a part of the message to store.
<code>Size</code>	Size of the part of the message. Valid values are: $1 \leq \text{Size} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Size} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Note

The total size of all parts of the message must not exceed `0x7FFF` on 8/16-bit CPUs, or `0x7FFFFFFF` on 32-bit CPUs, respectively. The message is stored in the queue buffer with an additional message header. Additionally the message size will be round up to the next multiple of the size of an integer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
145: OS_ERR_QUEUE_INVALID
149: OS_ERR_MESSAGE_SIZE_ZERO
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
OS_CONST_PTR OS_QUEUE_SRCLIST aDataList[] = { {"Hello ", 6},
                                               {"World!", 6}
                                               };
OS_QUEUE_PutEx(&MemoryQ, aDataList, 2, 100);
```

Chapter 12

Multi Object Wait

12.1 Introduction

In the preceding chapter, inter-task communication and synchronization with RTOS objects were described. With the described API a task can block for one specific RTOS object and condition only.

However, there could be the requirement to concurrently wait for multiple conditions to be fulfilled. For example, a task could need to wait at the same time for new data in a queue and an event to get signaled.

The Multi Object Wait feature allows to wait concurrently for one or more conditions to be fulfilled. The Multi Object Wait API returns as soon as one of the conditions it is waiting for is fulfilled. It is possible for more than one condition to be fulfilled when the Multi Object Wait API returns. The caller must look at the state of all RTOS objects to figure out which ones were fulfilled and what actions to take. The Multi Object Wait API does not e.g. retrieve data from the queue. That must be done subsequently by the application.

For each RTOS object and condition, an embOS condition routine can be defined.

Multi Object Wait can be used with:

- Event object
- Semaphore
- Mailbox
- Queue
- Fixed Block Size Memory Pool

Example

```
#include "RTOS.h"

#define OS_COUNT_OF(a) (sizeof(a) / sizeof(a[0]))

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task control blocks
static OS_QUEUE     MyQueue;                      // Queue control block
static char         MyQBuffer[30];               // Queue buffer
static OS_EVENT     MyEvent;                     // Event control block
//
// List of RTOS objects to simultaneously wait for
//
static const OS_MULTIOBJ_COND aMyMultiObjCond[] = {
    { &MyQueue, OS_MULTIOBJ_IsMessageInQueue, (void*)2 },
    { &MyEvent, OS_MULTIOBJ_IsEventSignaled, (void*)NULL },
};
//
// Multiple object control blocks
//
static OS_MULTIOBJ aMyMultiObj[OS_COUNT_OF(aMyMultiObjCond)];

static void HPTask(void) {
    OS_INT  Index;
    char*   pData;
    int     MessageCnt;
    OS_BOOL Signaled;

    while (1) {
        Index = OS_MULTIOBJ_WaitBlocked(aMyMultiObj,
                                       aMyMultiObjCond,
                                       OS_COUNT_OF(aMyMultiObjCond));

        switch (Index) {
            case 0:
                MessageCnt = OS_QUEUE_GetMessageCnt(&MyQueue);
                if (MessageCnt == 2) {
                    OS_QUEUE_GetPtr(&MyQueue, (void**)&pData);
                    OS_COM_SendString(pData);
                    OS_QUEUE_Purge(&MyQueue);
                }
            }
        }
    }
}
```

```
    OS_QUEUE_GetPtr(&MyQueue, (void**)&pData);
    OS_COM_SendString(pData);
    OS_QUEUE_Purge(&MyQueue);
}
break;
case 1:
    Signaled = OS_EVENT_Get(&MyEvent);
    if (Signaled != 0u) {
        OS_COM_SendString("\nEvent received.");
        OS_EVENT_Reset(&MyEvent);
    }
    break;
}
}
}

static void LPTask(void) {
    while (1) {
        OS_QUEUE_Put(&MyQueue, "\nHello\n", 7);
        OS_QUEUE_Put(&MyQueue, "\nWorld !\n", 9);
        OS_EVENT_Set(&MyEvent);
        OS_TASK_Delay(2);
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_Inithw(); // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_QUEUE_Create(&MyQueue, MyQBuffer, sizeof(MyQBuffer));
    OS_EVENT_CreateEx(&MyEvent, OS_EVENT_RESET_MODE_MANUAL);
    OS_Start(); // Start embOS
    return 0;
}
```

12.2 Condition routines

The Multi Object Wait API routines expect an array of `OS_MULTIOBJ_COND` entries. A `OS_MULTIOBJ_COND` structure contains the RTOS object address, the condition routine address and an optional parameter that is passed to the condition routine.

The application can choose for each RTOS object a different condition routine. The same condition routine can be used with different parameters. A condition routine returns true or false. If `NULL` is passed as the condition routine address, the return value is false.

Prototype

```
static OS_BOOL _IsCondition(OS_CONST_PTR void* pObj, void* pParam)
```

Example

```
static const OS_MULTIOBJ_COND aMyMultiObjCond[] = {
    { &MyQueue, OS_MULTIOBJ_IsMessageInQueue, (void*)2 },
    { &MyEvent, OS_MULTIOBJ_IsEventSignaled, (void*)NULL},
};
```

12.2.1 API functions

Routine	Description
<code>OS_MULTIOBJ_IsEventSignaled()</code>	Returns whether the requested event object is in signaled state.
<code>OS_MULTIOBJ_IsTokenInSema()</code>	Returns whether the requested semaphore contains available tokens.
<code>OS_MULTIOBJ_IsMessageInMailbox()</code>	Returns whether the requested mailbox contains available messages.
<code>OS_MULTIOBJ_IsSpaceInMailbox()</code>	Returns whether the requested mailbox has free space for additional messages.
<code>OS_MULTIOBJ_IsMessageInQueue()</code>	Returns whether the requested queue contains available messages.
<code>OS_MULTIOBJ_IsSpaceInQueue()</code>	Returns whether there is space for a message of size <code>Size</code> .
<code>OS_MULTIOBJ_IsBlockInMemPool()</code>	Returns whether the requested Memory Pool contains available memory blocks.

12.2.1.1 OS_MULTIOBJ_IsEventSignaled()

Description

Returns whether the requested event object is in signaled state.

Prototype

```
OS_BOOL OS_MULTIOBJ_IsEventSignaled(OS_CONST_PTR void *pEvent,
                                     void*          pMask);
```

Parameters

Parameter	Description
<code>pEvent</code>	Pointer to an event object of type <code>OS_EVENT</code> .
<code>pMask</code>	If <code>pMask</code> equals <code>NULL</code> , the function evaluates the entire event object bits. Therefore, it behaves like <code>OS_EVENT_Get()</code> , <code>OS_EVENT_GetBlocked()</code> and <code>OS_EVENT_GetTimed()</code> without an event mask and must be used with none mask event object API only. Otherwise, the bit mask indicates the event bits that shall be evaluated and must be used with mask event object API like <code>OS_EVENT_SetMask()</code> only.

Return value

- = 0 Event object is not set to requested signal state.
- ≠ 0 Event object is set to requested signal state.

Additional information

The event bits are not consumed.

Note

`OS_MULTIOBJ_IsEventSignaled()` must be used with event object reset mode `OS_EVENT_RESET_MODE_MANUAL` only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
220: OS_ERR_EVENT_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example at the start of this sub-chapter.

12.2.1.2 OS_MULTIOBJ_IsTokenInSema()

Description

Returns whether the requested semaphore contains available tokens.

Prototype

```
OS_BOOL OS_MULTIOBJ_IsTokenInSema(OS_CONST_PTR void *pSemaphore,
                                  void*          pNumTokens);
```

Parameters

Parameter	Description
<code>pSemaphore</code>	Pointer to a semaphore object of type <code>OS_SEMAPHORE</code> .
<code>pNumTokens</code>	If <code>pNumTokens</code> equals <code>NULL</code> , the function returns whether any token is available. Otherwise, the function returns whether the given amount of tokens is available.

Return value

= 0 Requested amount of semaphore tokens is not available.
 ≠ 0 Requested amount of semaphore tokens is available.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

132: OS_ERR_INV_SEMAPHORE

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example at the start of this sub-chapter.

12.2.1.3 OS_MULTIOBJ_IsMessageInMailbox()

Description

Returns whether the requested mailbox contains available messages.

Prototype

```
OS_BOOL OS_MULTIOBJ_IsMessageInMailbox(OS_CONST_PTR void *pMailbox,
                                       void*          pNumMessages);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type OS_MAILBOX.
<code>pNumMessages</code>	If <code>pNumMessages</code> equals NULL, the function returns whether any message is available. Otherwise, the function returns whether the given number of messages is available.

Return value

- = 0 Requested number of messages is not available.
- ≠ 0 Requested number of messages is available.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

- 130: OS_ERR_INV_MAILBOX
- 147: OS_ERR_MAILBOX_INUSE

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example at the start of this sub-chapter.

12.2.1.4 OS_MULTIOBJ_IsSpaceInMailbox()

Description

Returns whether the requested mailbox has free space for additional messages.

Prototype

```
OS_BOOL OS_MULTIOBJ_IsSpaceInMailbox(OS_CONST_PTR void *pMailbox,
                                     void*          pNumMessages);
```

Parameters

Parameter	Description
<code>pMailbox</code>	Pointer to a mailbox object of type OS_MAILBOX.
<code>pNumMessages</code>	If <code>pNumMessages</code> equals NULL, the function returns whether the mailbox has sufficient space for any message. Otherwise, the function returns whether the mailbox has sufficient space for the given number of messages.

Return value

- = 0 Sufficient space for the requested number of messages is not available.
- ≠ 0 Sufficient space for the requested number of messages is available.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

- 130: OS_ERR_INV_MAILBOX
- 147: OS_ERR_MAILBOX_INUSE

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example at the start of this sub-chapter.

12.2.1.5 OS_MULTIOBJ_IsMessageInQueue()

Description

Returns whether the requested queue contains available messages.

Prototype

```
OS_BOOL OS_MULTIOBJ_IsMessageInQueue(OS_CONST_PTR void *pQueue,
                                     void*          pNumMessages);
```

Parameters

Parameter	Description
<code>pQueue</code>	Pointer to a queue object of type <code>OS_QUEUE</code> .
<code>pNumMessages</code>	If <code>pNumMessages</code> equals <code>NULL</code> , the function returns whether any message is available. Otherwise, the function returns whether the given amount of messages is available.

Return value

= 0 Requested amount of messages is not available, or the queue is in use.
 ≠ 0 Requested amount of messages is available.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
143: OS_ERR_QUEUE_INUSE
145: OS_ERR_QUEUE_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example at the start of this sub-chapter.

12.2.1.6 OS_MULTIOBJ_IsSpaceInQueue()

Description

Returns whether there is space for a message of size Size.

Prototype

```
OS_BOOL OS_MULTIOBJ_IsSpaceInQueue(OS_CONST_PTR void *pQueue,
                                   void*         pSize);
```

Parameters

Parameter	Description
pQ	Pointer to a queue object of type OS_QUEUE.
pSize	Message size in bytes.

Return value

= 0 Not sufficient space for the requested message.
 ≠ 0 Sufficient space for the requested message.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

145: OS_ERR_QUEUE_INVALID

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example at the start of this sub-chapter.

12.2.1.7 OS_MULTIOBJ_IsBlockInMemPool()

Description

Returns whether the requested Memory Pool contains available memory blocks.

Prototype

```
OS_BOOL OS_MULTIOBJ_IsBlockInMemPool(OS_CONST_PTR void *pMEMF,
                                     void*          pNumBlocks);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .
<code>pNumBlocks</code>	If <code>pNumBlocks</code> equals <code>NULL</code> , the function returns whether any memory block is available. Otherwise, the function returns whether the given amount of memory blocks is available.

Return value

= 0 Requested amount of memory blocks is not available.
 ≠ 0 Requested amount of memory blocks is available.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

190: OS_ERR_MEMF_INV

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example at the start of this sub-chapter.

12.3 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_MULTIOBJ_Wait()</code>	Returns whether one of the requested RTOS object conditions is true.	•	•	•	•	•
<code>OS_MULTIOBJ_WaitBlocked()</code>	Waits until one of the requested RTOS object conditions is true.		•	•		
<code>OS_MULTIOBJ_WaitTimed()</code>	Returns whether one of the requested RTOS objects conditions is true.		•	•		

12.3.1 OS_MULTIOBJ_Wait()

Description

Returns whether one of the requested RTOS object conditions is true.

Prototype

```
OS_INT OS_MULTIOBJ_Wait(OS_CONST_PTR OS_MULTIOBJ_COND *pMultiObjCond,
                       OS_UINT      NumObjects);
```

Parameters

Parameter	Description
<code>pMultiObjCond</code>	Pointer to an array of OS_MULTIOBJ_COND items.
<code>NumObjects</code>	Number of OS_MULTIOBJ entries.

Return value

= -1 No RTOS object conditions is true.
 ≥ 0 Index of the first RTOS object conditions that is true.

Additional information

OS_MULTIOBJ_Wait() allows a task to wait concurrently for multiple RTOS objects. OS_MULTIOBJ_Wait() returns immediately and does not block the task.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

167: OS_ERR_CPU_STATE_ILLEGAL
 265: OS_ERR_MULTIOBJ_INVALID_ROUTINE

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define OS_COUNT_OF(a) (sizeof(a) / sizeof(a[0]))

static const OS_MULTIOBJ_COND aMyMultiObjCond[] = {
    { &MyQueue, OS_MULTIOBJ_IsMessageInQueue, (void*)2 },
    { &MyEvent, OS_MULTIOBJ_IsEventSignaled, (void*)NULL },
};

static void HPTask(void) {
    OS_INT Index;

    while (1) {
        Index = OS_MULTIOBJ_Wait(aMyMultiObjCond, 2);
        ...
    }
}
```

12.3.2 OS_MULTIOBJ_WaitBlocked()

Description

Waits until one of the requested RTOS object conditions is true.

Prototype

```
OS_INT OS_MULTIOBJ_WaitBlocked(OS_MULTIOBJ* pMultiObj,
                               OS_CONST_PTR OS_MULTIOBJ_COND *pMultiObjCond,
                               OS_UINT      NumObjects);
```

Parameters

Parameter	Description
<code>pMultiObj</code>	Pointer to an array of OS_MULTIOBJ items.
<code>pMultiObjCond</code>	Pointer to an array of OS_MULTIOBJ_COND items.
<code>NumObjects</code>	Number of OS_MULTIOBJ entries.

Return value

Index of the first RTOS object conditions that is true.

Additional information

OS_MULTIOBJ_WaitBlocked() allows a task to wait concurrently for multiple RTOS objects. OS_MULTIOBJ_WaitBlocked() suspends the task until at least one condition is true.

Note

The `pMultiObj` and `pMultiObjCond` arrays must have the same number of items.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
265: OS_ERR_MULTIOBJ_INVALID_ROUTINE
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define OS_COUNT_OF(a) (sizeof(a) / sizeof(a[0]))

static const OS_MULTIOBJ_COND aMyMultiObjCond[] = {
    { &MyQueue, OS_MULTIOBJ_IsMessageInQueue, (void*)2 },
    { &MyEvent, OS_MULTIOBJ_IsEventSignaled, (void*)NULL },
};

static OS_MULTIOBJ aMyMultiObj[OS_COUNT_OF(aMyMultiObjCond)];

static void HPTask(void) {
    OS_INT Index;

    while (1) {
        Index = OS_MULTIOBJ_WaitBlocked(aMyMultiObj, aObjects, 2);
        ...
    }
}
```

```
}
```

12.3.3 OS_MULTIOBJ_WaitTimed()

Description

Returns whether one of the requested RTOS objects conditions is true.

Prototype

```
OS_INT OS_MULTIOBJ_WaitTimed(OS_MULTIOBJ* pMultiObj,
                             OS_CONST_PTR OS_MULTIOBJ_COND *pMultiObjCond,
                             OS_UINT      NumObjects,
                             OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>pMultiObj</code>	Pointer to an array of OS_MULTIOBJ items.
<code>pMultiObjCond</code>	Pointer to an array of OS_MULTIOBJ_COND items.
<code>NumObjects</code>	Number of OS_MULTIOBJ entries.
<code>Timeout</code>	Maximum time in system ticks until at least one condition must be true. The data type OS_TIME is defined as an integer, therefore valid values are: $1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= -1 No RTOS object conditions is true within the timeout.
 ≥ 0 Index of the first RTOS object conditions that is true.

Additional information

OS_MULTIOBJ_WaitTimed() allows a task to wait concurrently for multiple RTOS objects. OS_MULTIOBJ_WaitTimed() suspends the task until at least one condition is true or the timeout has expired.

Note

The `pMultiObj` and `pMultiObjCond` arrays must have the same number of items.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
265: OS_ERR_MULTIOBJ_INVALID_ROUTINE
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define OS_COUNT_OF(a) (sizeof(a) / sizeof(a[0]))

static const OS_MULTIOBJ_COND aMyMultiObjCond[] = {
    { &MyQueue, OS_MULTIOBJ_IsMessageInQueue, (void*)2 },
    { &MyEvent, OS_MULTIOBJ_IsEventSignaled, (void*)NULL },
};
```

```
static OS_MULTIOBJ aMyMultiObj[OS_COUNT_OF(aMyMultiObjCond)];

static void HPTask(void) {
    OS_INT Index;

    while (1) {
        Index = OS_MULTIOBJ_WaitTimed(aMyMultiObjMemj, aObjects, 2, 10);
        ...
    }
}
```

Chapter 13

Watchdog

13.1 Introduction

A watchdog timer is a hardware timer that is used to reset a microcontroller after a specified amount of time. During normal operation, the microcontroller application periodically restarts ("triggers" or "feeds") the watchdog timer to prevent it from timing out. In case of malfunction, however, the watchdog timer will eventually time out and subsequently reset the microcontroller. This allows to detect and recover from microcontroller malfunctions.

For example, in a system without an RTOS, the watchdog timer would be triggered periodically from a single point in the application. When the application does not run properly, the watchdog timer will not be triggered and thus the watchdog will cause a reset of the microcontroller.

In a system that includes an RTOS, on the other hand, multiple tasks run at the same time. It may happen that one or more of these tasks runs properly, while other tasks fail to run as intended. Hence it may be insufficient to trigger the watchdog from one of these tasks only. Therefore, embOS offers a watchdog support module that allows to automatically check if all tasks, software timers, or even interrupt routines are executing properly. The embOS watchdog support does not replace the hardware watchdog but makes it easier to use a hardware watchdog in an RTOS application.

Example

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128];
static OS_TASK      TCBHP, TCBLP;
static OS_WD        WatchdogHP, WatchdogLP;
static OS_TICK_HOOK Hook;

static void _TriggerWatchDog(void) {
    WD_REG = TRIGGER_WD;           // Trigger the hardware watchdog.
}

static void _Reset(OS_CONST_PTR OS_WD* pWD) {
    OS_USE_PARA(pWD);
    // Applications can use pWD to detect WD expiration cause.
    SYSTEM_CTRL_REG = PERFORM_RESET; // Reboot microcontroller.
}

static void HPTask(void) {
    OS_WD_Add(&WatchdogHP, 50);
    while (1) {
        OS_TASK_Delay(50);
        OS_WD_Trigger(&WatchdogHP);
    }
}

static void LPTask(void) {
    OS_WD_Add(&WatchdogLP, 200);
    while (1) {
        OS_TASK_Delay(200);
        OS_WD_Trigger(&WatchdogLP);
    }
}

int main(void) {
    OS_Init();           // Initialize embOS
    OS_Inithw();        // Initialize required hardware
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_WD_Config(&_TriggerWatchDog, &_Reset);
    OS_TICK_AddHook(&Hook, OS_WD_Check);
    OS_Start();         // Start embOS
    return 0;
}
```

13.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
OS_WD_Add()	Adds a software watchdog timer to the watchdog list.	•	•		•	•
OS_WD_Check()	Checks if a watchdog timer has expired.	•	•		•	•
OS_WD_Config()	Sets the watchdog callback functions.	•	•			
OS_WD_Remove()	Removes a watchdog timer from the watchdog list.	•	•		•	•
OS_WD_Trigger()	Triggers/Feeds a watchdog timer.	•	•	•	•	•

13.2.1 OS_WD_Add()

Description

Adds a software watchdog timer to the watchdog list.

Prototype

```
void OS_WD_Add(OS_WD* pWD,
               OS_TIME Timeout);
```

Parameters

Parameter	Description
<code>pWD</code>	Pointer to a watchdog object of type <code>OS_WD</code> .
<code>Timeout</code>	Watchdog timer timeout in system ticks. Must be within the following range: $0 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs $0 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs Please note that these are signed values.

Additional information

The watchdog timer will be checked by `OS_WD_Check()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
179: OS_ERR_2USE_WATCHDOG
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_WD _myWD;

void HPTask(void) {
    OS_WD_Add(&_myWD, 50);
    while (1) {
        OS_WD_Trigger(&_myWD);
        OS_TASK_Delay(50);
    }
}
```

13.2.2 OS_WD_Check()

Description

Checks if a watchdog timer has expired.

Prototype

```
void OS_WD_Check(void);
```

Additional information

If no watchdog timer has expired the hardware watchdog is triggered. If a watchdog timer has expired, the callback function is called. `OS_WD_Check()` must be called periodically. It is good practice to call it from the system tick handler.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void SysTick_Handler(void) {  
    OS_INT_Enter();  
    OS_Tick_Handle();  
    OS_WD_Check();  
    OS_INT_Leave();  
}
```

13.2.3 OS_WD_Config()

Description

Sets the watchdog callback functions.

Prototype

```
void OS_WD_Config(OS_ROUTINE_VOID*   pfTrigger,
                 OS_ROUTINE_WD_PTR* pfReset);
```

Parameters

Parameter	Description
<code>pfTrigger</code>	Pointer to the function of type <code>OS_ROUTINE_VOID</code> which shall be called to trigger (fed) the hardware watchdog.
<code>pfReset</code>	Pointer to the function of type <code>OS_ROUTINE_WD_PTR</code> which shall be called when an watchdog timer has expired.

Additional information

`pfReset` may be used to perform additional operations inside a callback routine prior to the reset of the microcontroller. For example, a message may be written to a log file. `pfReset` is optional and may be `NULL`. If `pfReset` is `NULL`, no callback routine gets executed, but the hardware watchdog will still cause a reset of the microcontroller.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void _TriggerWatchDog(void) {
    WD_REG = TRIGGER_WD;           // Trigger the hardware watchdog
}

static void _Reset(OS_CONST_PTR OS_WD* pWD) {
    //
    // Store information about expired watchdog prior to reset.
    //
    _WriteLogMessage(pWD);
    //
    // Reboot microcontroller
    //
    SYSTEM_CTRL_REG = PERFORM_RESET;
}

int main(void) {
    ...
    OS_WD_Config(&_TriggerWatchDog, &_Reset);
    OS_Start();
    return 0;
}
```

Note

In previous versions of embOS, `OS_WD_Config()` expected the parameter `pfResetFunc` to be of a different type.

Since embOS V4.40, instead of a callback of the type `voidRoutine*`, `OS_WD_Config()` expects a callback of type `OS_WD_RESET_CALLBACK*`. This allows for passing the relevant `OS_WD` structure to the routine, e.g. for further examination by the application.

13.2.4 OS_WD_Remove()

Description

Removes a watchdog timer from the watchdog list.

Prototype

```
void OS_WD_Remove(OS_CONST_PTR OS_WD *pWD);
```

Parameters

Parameter	Description
<code>pWD</code>	Pointer to a watchdog object of type <code>OS_WD</code> .

Additional information

A removed watchdog is no longer checked by `OS_WD_Check()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
int main(void) {  
    ...  
    OS_WD_Add(&_myWD);  
    OS_WD_Remove(&_myWD);  
    return 0;  
}
```

13.2.5 OS_WD_Trigger()

Description

Triggers/Feeds a watchdog timer.

Prototype

```
void OS_WD_Trigger(OS_WD* pWD);
```

Parameters

Parameter	Description
pWD	Pointer to a watchdog object of type OS_WD.

Additional information

Each software watchdog timer must be triggered (fed) periodically. If not, the timeout expires and OS_WD_Check() will no longer trigger the hardware watchdog timer, but will call the reset callback function (if any).

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_WD _myWD;

static void HPTask(void) {
    OS_WD_Add(&_myWD, 50);
    while (1) {
        OS_TASK_Delay(50);
        OS_WD_Trigger(&_myWD);
    }
}
```

Chapter 14

Multi-core Support

14.1 Introduction

embOS can be utilized on multi-core processors by running separate embOS instances on each individual core. For synchronization purposes and in order to exchange data between the cores, embOS includes a comprehensive spinlock API which can be used to control access to shared memory, peripherals, etc. This API is not available in embOS library mode `OS_LIBMODE_SAFE`.

Spinlocks

Spinlocks constitute a general purpose locking mechanism in which any process trying to acquire the lock is caused to actively wait until the lock becomes available. To do so, the process trying to acquire the lock remains active and repeatedly checks the availability of the lock in a loop. Effectively, the process will “spin” until it acquires the lock.

Once acquired by a process, spinlocks are usually held by that process until they are explicitly released. If held by one process for longer duration, spinlocks may severely impact the runtime behavior of other processes trying to acquire the same spinlock. Therefore, spinlocks should be held by one process for short periods of time only.

Usage of spinlocks with embOS

embOS spinlocks are intended for inter-core synchronization and communication. They are not intended for synchronization of individual tasks running on the same core, on which semaphores, queues and mailboxes should be used instead.

However, multitasking still has to be taken into consideration when using embOS spinlocks. Specifically, an embOS task holding a spinlock should not be preempted, for this would prevent that task from releasing the spinlock as fast as possible, which may in return impact the runtime behavior of other cores attempting to acquire the spinlock. Declaration of critical regions therefore is explicitly recommended while holding spinlocks.

embOS spinlocks are usually implemented using hardware instructions specific to one architecture, but a portable software implementation is provided in addition. If appropriate hardware instructions are unavailable for the specific architecture in use, the software implementation is provided exclusively.

Note

It is important to use matching implementations on each core of the multi-core processor that shall access the same spinlock.

For example, a core supporting a hardware implementation may use that implementation to access a spinlock that is shared with another core that supports the same hardware implementation. At the same time, that core may use the software implementation to access a different spinlock that is shared with a different core that does not support the same hardware implementation. However, in case all three cores in this example should share the same spinlock, each of them has to use the software implementation.

To know the spinlock’s location in memory, each core’s application must declare the appropriate `OS_SPINLOCK` variable (or `OS_SPINLOCK_SW`, respectively) at an identical memory address. Initialization of the spinlock, however, must be performed by one core only.

Example of using spinlocks

Two cores of a multi-core processor shall access an hardware peripheral, e.g. a LC display. To avoid situations in which both cores access the LCD simultaneously, access must be restricted through usage of a spinlock: Every time the LCD is used by one core, it must first claim the spinlock through the respective embOS API call. After the LCD has been written to, the spinlock is released by another embOS API call.

Data exchange between cores can be implemented analogously, e.g. through declaration of a buffer in shared memory: Here, every time a core shall write data to the buffer, it must acquire the spinlock first. After the data has been written to the buffer, the spinlock

is released. This ensures that neither core can interfere with the writing of data by the other core.

Core 0:

```
#include "RTOS.h"

static OS_STACKPTR int Stack[128]; // Task stack
static OS_TASK      TCB;           // Task-control-block
static OS_SPINLOCK  MySpinlock @ ".shared_mem";

static void Task(void) {
    while (1) {
        OS_TASK_EnterRegion(); // Inhibit preemptive task switches
        OS_SPINLOCK_Lock(&MySpinlock); // Acquire spinlock
        //
        // Perform critical operation
        //
        OS_SPINLOCK_Unlock(&MySpinlock); // Release spinlock
        OS_TASK_LeaveRegion(); // Re-allow preemptive task switches
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_Inithw(); // Initialize Hardware for OS
    OS_SPINLOCK_Create(&MySpinlock); // Initialize Spinlock
    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);
    OS_Start(); // Start multitasking
    return 0;
}
```

Core 1:

```
#include "RTOS.h"

static OS_STACKPTR int Stack[128]; // Task stack
static OS_TASK      TCB;           // Task-control-block
static OS_SPINLOCK  MySpinlock @ ".shared_mem";

static void Task(void) {
    while (1) {
        OS_TASK_EnterRegion(); // Inhibit preemptive task switches
        OS_SPINLOCK_Lock(&MySpinlock); // Acquire spinlock
        //
        // Perform critical operation
        //
        OS_SPINLOCK_Unlock(&MySpinlock); // Release spinlock
        OS_TASK_LeaveRegion(); // Re-allow preemptive task switches
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_Inithw(); // Initialize Hardware for OS
    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);
    OS_Start(); // Start multitasking
    return 0;
}
```

14.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_SPINLOCK_Create()</code>	Creates a hardware-specific spinlock.	•	•			
<code>OS_SPINLOCK_Lock()</code>	Acquires a hardware-specific spinlock. Busy waiting until the spinlock becomes available. This function is unavailable for some architectures.	•	•			
<code>OS_SPINLOCK_Unlock()</code>	Releases a hardware-specific spinlock. This function is unavailable for architectures that do not support an appropriate instruction set.	•	•			
<code>OS_SPINLOCK_SW_Create()</code>	Creates a software-implementation spinlock.	•	•			
<code>OS_SPINLOCK_SW_Lock()</code>	Acquires the specified software-implementation spinlock.	•	•			
<code>OS_SPINLOCK_SW_Unlock()</code>	Releases the specified software-implementation spinlock.	•	•			

14.2.1 OS_SPINLOCK_Create()

Description

Creates a hardware-specific spinlock.

Prototype

```
void OS_SPINLOCK_Create(OS_SPINLOCK* pSpinlock);
```

Parameters

Parameter	Description
<code>pSpinlock</code>	Pointer to a spinlock object of type <code>OS_SPINLOCK</code> . The variable must reside in shared memory.

Additional information

After creation, the spinlock is not locked.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

165: OS_ERR_INIT_NOT_CALLED

160: OS_ERR_ILLEGAL_IN_ISR

161: OS_ERR_ILLEGAL_IN_TIMER

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Multi-core Support* on page 320.

14.2.2 OS_SPINLOCK_Lock()

Description

`OS_SPINLOCK_Lock()` acquires a hardware-specific spinlock. If the spinlock is unavailable, the calling task will not be blocked, but will actively wait until the spinlock becomes available.

This function is unavailable for architectures that do not support an appropriate instruction set.

Prototype

```
void OS_SPINLOCK_Lock(OS_SPINLOCK* pSpinlock);
```

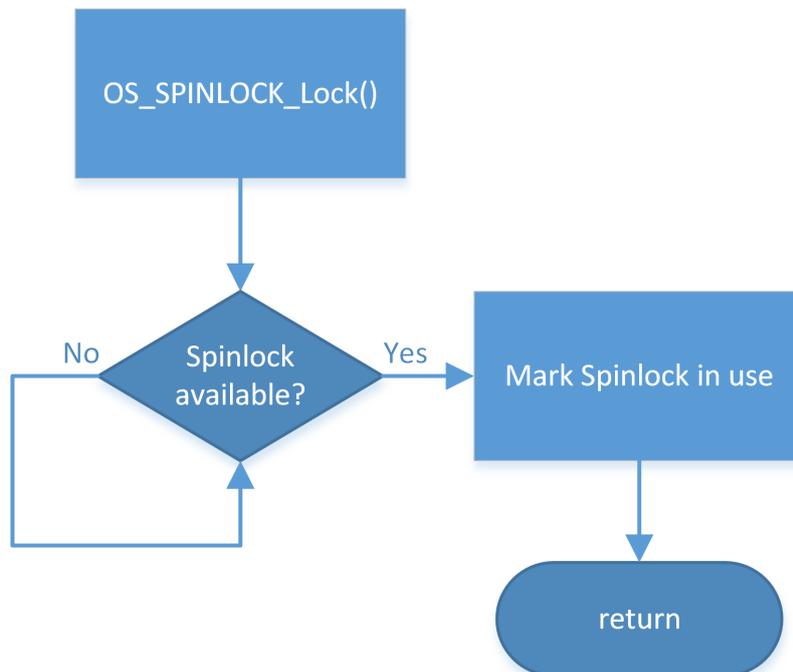
Parameters

Parameter	Description
<code>pSpinlock</code>	Pointer to a variable of type <code>OS_SPINLOCK</code> reserved for the management of the spinlock.

Additional information

A task that has acquired a spinlock must not call `OS_SPINLOCK_Lock()` for that spinlock again. The spinlock must first be released by a call to `OS_SPINLOCK_Unlock()`.

The following diagram illustrates how `OS_SPINLOCK_Lock()` works:



Example

Please refer to the example in the introduction of chapter *Multi-core Support* on page 320.

14.2.3 OS_SPINLOCK_Unlock()

Description

Releases a hardware-specific spinlock. This function is unavailable for architectures that do not support an appropriate instruction set.

Prototype

```
void OS_SPINLOCK_Unlock(OS_SPINLOCK* pSpinlock);
```

Parameters

Parameter	Description
<code>pSpinlock</code>	Pointer to a variable of type <code>OS_SPINLOCK</code> reserved for the management of the spinlock.

Example

Please refer to the example in the introduction of chapter *Multi-core Support* on page 320.

14.2.4 OS_SPINLOCK_SW_Create()

Description

Creates a software-implementation spinlock.

Prototype

```
void OS_SPINLOCK_SW_Create(OS_SPINLOCK_SW* pSpinlock);
```

Parameters

Parameter	Description
<code>pSpinlock</code>	Pointer to a spinlock object of type <code>OS_SPINLOCK_SW</code> . The variable must reside in shared memory.

Additional information

After creation, the spinlock is not locked.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

165: OS_ERR_INIT_NOT_CALLED

160: OS_ERR_ILLEGAL_IN_ISR

161: OS_ERR_ILLEGAL_IN_TIMER

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Multi-core Support* on page 320.

14.2.5 OS_SPINLOCK_SW_Lock()

Description

Acquires the specified software-implementation spinlock. If the spinlock is unavailable, the calling task will not be blocked, but will actively wait until the spinlock becomes available.

Prototype

```
void OS_SPINLOCK_SW_Lock(OS_SPINLOCK_SW* pSpinlock,
                        OS_UINT          Id);
```

Parameters

Parameter	Description
<code>pSpinlock</code>	Pointer to a spinlock object of type <code>OS_SPINLOCK_SW</code> .
<code>Id</code>	Unique identifier to specify the core accessing the spinlock. Valid values are $0 \leq Id < OS_SPINLOCK_MAX_CORES$. By default, <code>OS_SPINLOCK_MAX_CORES</code> is defined to 4 and may be changed when using source code.

Additional information

A task that has acquired a spinlock must not call `OS_SPINLOCK_SW_Lock()` for that spinlock again. The spinlock must first be released by a call to `OS_SPINLOCK_SW_Unlock()`.

`OS_SPINLOCK_SW_Lock()` implements Lamport's bakery algorithm, published by Leslie Lamport in "Communications of the Association for Computing Machinery", 1974, Volume 17, Number 8. An excerpt is publicly available at research.microsoft.com.

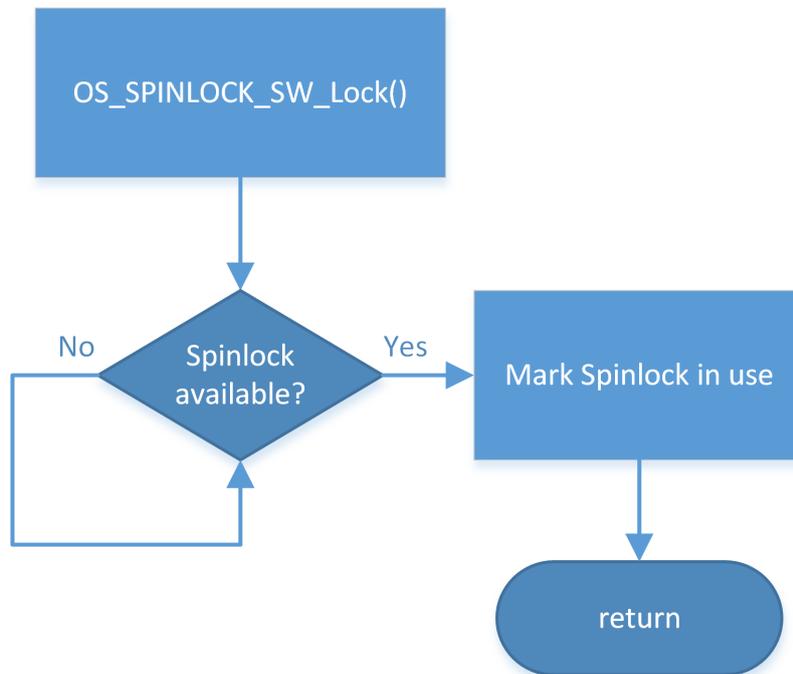
Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
185: OS_ERR_SPINLOCK_INV_CORE
```

For details, refer to the chapter *Runtime application errors* on page 455.

The following diagram illustrates how `OS_SPINLOCK_SW_Lock()` works:



Example

Please refer to the example in the introduction of chapter *Multi-core Support* on page 320.

14.2.6 OS_SPINLOCK_SW_Unlock()

Description

Releases the specified software-implementation spinlock.

Prototype

```
void OS_SPINLOCK_SW_Unlock(OS_SPINLOCK_SW* pSpinlock,
                           OS_UINT       Id);
```

Parameters

Parameter	Description
<code>pSpinlock</code>	Pointer to a spinlock object of type <code>OS_SPINLOCK_SW</code> .
<code>Id</code>	Unique identifier to specify the core accessing the spinlock. Valid values are $0 \leq \text{Id} < \text{OS_SPINLOCK_MAX_CORES}$. By default, <code>OS_SPINLOCK_MAX_CORES</code> is defined to 4 and may be changed when using source code.

Additional information

`OS_SPINLOCK_SW_Unlock()` must called after `OS_SPINLOCK_SW_Lock()` only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
185: OS_ERR_SPINLOCK_INV_CORE
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Multi-core Support* on page 320.

Chapter 15

Interrupts

15.1 What are interrupts?

This chapter explains how to use interrupt service routines (ISRs) in cooperation with embOS. Specific details for your CPU and compiler can be found in the CPU & Compiler Specifics manual of the embOS documentation.

Interrupts are interruptions of a program caused by hardware. When an interrupt occurs, the CPU saves its registers and executes a subroutine called an interrupt service routine, or ISR. After the ISR is completed, the program returns to the highest-priority task which is ready for execution. Normal interrupts are maskable. Maskable interrupts can occur at any time unless they are disabled. ISRs are also nestable -- they can be recognized and executed within other ISRs.

There are several good reasons for using interrupt routines. They can respond very quickly to external events such as the status change on an input, the expiration of a hardware timer, reception or completion of transmission of a character via serial interface, or other types of events. Interrupts effectively allow events to be processed as they occur.

15.2 Interrupt latency

Interrupt latency is the time between an interrupt request and the execution of the first instruction of the interrupt service routine. Every computer system has an interrupt latency. The latency depends on various factors and differs even on the same computer system. The value that one is typically interested in is the worst case interrupt latency. The interrupt latency is the sum of a number of individual smaller delays explained below.

Note

Interrupt latency caused by embOS can be avoided entirely when using zero latency interrupts, which are explained in chapter *Zero interrupt latency* on page 335.

15.2.1 Causes of interrupt latencies

- The first delay is typically in the hardware: The interrupt request signal needs to be synchronized to the CPU clock. Depending on the synchronization logic, typically up to three CPU cycles can be lost before the interrupt request reaches the CPU core.
- The CPU will typically complete the current instruction. This instruction can take multiple cycles to complete; on most systems, divide, push-multiple, or memory-copy instructions are the instructions which require most clock cycles. On top of the cycles required by the CPU, there are in most cases additional cycles required for memory access. In an ARM7 system, the instruction `STMDB SP!, {R0-R11, LR}`; typically is the worst case instruction. It stores thirteen 32-bit registers to the stack, which, in an ARM7 system, takes 15 clock cycles to complete.
- The memory system may require additional cycles for wait states.
- After the current instruction is completed, the CPU performs a mode switch or pushes registers (typically, PC and flag registers) to the stack. In general, modern CPUs (such as ARM) perform a mode switch, which requires fewer CPU cycles than saving registers.
- Pipeline fill
Most modern CPUs are pipelined. Execution of an instruction happens in various stages of the pipeline. An instruction is executed when it has reached its final stage of the pipeline. Because the mode switch flushes the pipeline, a few extra cycles are required to refill the pipeline.

15.2.2 Additional causes for interrupt latencies

There can be additional causes for interrupt latencies. These depend on the type of system used, but we list a few of them.

- Latencies caused by cache line fill. If the memory system has one or multiple caches, these may not contain the required data. In this case, not only the required data is loaded from memory, but in a lot of cases a complete line fill needs to be performed, reading multiple words from memory.
- Latencies caused by cache write back. A cache miss may cause a line to be replaced. If this line is marked as dirty, it needs to be written back to main memory, causing an additional delay.
- Latencies caused by MMU translation table walks. Translation table walks can take a considerable amount of time, especially as they involve potentially slow main memory accesses. In real-time interrupt handlers, translation table walks caused by the TLB not containing translations for the handler and/or the data it accesses can increase interrupt latency significantly.
- Application program. Of course, the application program can cause additional latencies by disabling interrupts. This can make sense in some situations, but of course causes additional latencies.
- Interrupt routines. On most systems, one interrupt disables further interrupts. Even if the interrupts are re-enabled in the ISR, this takes a few instructions, causing additional latency.
- An RTOS also needs to temporarily disable interrupts which can call API functions of the RTOS. Some RTOS disable all interrupts, effectively increasing interrupt latency for all

interrupts, others (like embOS) disable only low-priority interrupts and do thereby not affect the latency of high priority interrupts.

15.2.3 How to measure latency and detect its cause

It is sometimes desirable to detect the cause for high interrupt latency. High interrupt latency may occur if interrupts are disabled for extended periods of time, or if a low level interrupt handler is executed before the actual interrupt handler. In these regards, embOS related functions like `OS_INT_Enter()` add to interrupt latency as well.

To measure interrupt latency and detect its cause, a timer interrupt may be used. For example, if the hardware timer counts upwards starting from zero after each compare-match-interrupt, its current counter value may be read from within the interrupt service routine to evaluate how many timer cycles (and thus how much time) have lapsed between the interrupt's occurrence and the actual execution of the interrupt handler:

```
static int Latency = 0;

void TimerIntHandler(void) {
    OS_INT_Enter();
    Latency = TIMER_CNT_VALUE; // Get current timer value
    OS_INT_Leave();
}
```

If this measurement is repeated several times, different results will occur. This is for the reason that the interrupt will sometimes be asserted while interrupts have been disabled by the application, while at other times interrupts are enabled when this interrupt request occurs. Thus, an application may keep track of minimum and maximum latency as shown below:

```
static int Latency = 0;
static int MaxLatency = 0;
static int MinLatency = 0xFFFFFFFF;

void TimerIntHandler(void) {
    OS_INT_Enter();
    Latency = TIMER_CNT_VALUE; // Get current timer value
    MinLatency = (Latency < MinLatency) ? Latency : MinLatency;
    MaxLatency = (Latency > MaxLatency) ? Latency : MaxLatency;
    OS_INT_Leave();
}
```

Using this method, `MinLatency` will hold the latency that was caused by hardware (and any low-level interrupt handler, if applicable). On the other hand, `MaxLatency` will hold the latency caused both by hardware and interrupt-masking in software. Therefore, by subtracting `MaxLatency - MinLatency`, it is possible to calculate the exact latency that was caused by interrupt-masking (typically performed by the operating system).

Based on this information, a threshold may be defined to detect the cause of high interrupt latency. E.g., a breakpoint may be set for when the current timer value exceeds a pre-defined threshold as shown below:

```
static int Latency = 0;

void TimerIntHandler(void) {
    OS_INT_Enter();
    Latency = TIMER_CNT_VALUE; // Get current timer value
    if (Latency > LATENCY_THRESHOLD) {
        while (1); // Set a breakpoint here
    }
    OS_INT_Leave();
}
```

If code trace information is available upon hitting the breakpoint, the exact cause for the latency may be checked through a trace log.

Note

If the hardware timer interrupt is the only interrupt in the system, its priority may be chosen arbitrarily. Otherwise, in case other interrupts occur during measurement as well, the timer interrupt should be configured to match the specific priority for which to measure latency. This is important, for other (possibly non-nestable) interrupts will influence the results depending on their priority relative to the timer interrupt's priority, which may or may not be desired on a case-to-case basis. Also, in order to provide meaningful results, the interrupt should occur quite frequently. Hence, the timer reload value typically is configured for small periods of time, but must ensure that interrupt execution will not consume the entire CPU time.

15.2.4 Zero interrupt latency

Zero interrupt latency in the strict sense is not possible as explained above. What we mean when we say "Zero interrupt latency" is that the latency of high priority interrupts is not affected by the RTOS; a system using embOS will have the same worst case interrupt latency for high priority interrupts as a system running without embOS.

Why is Zero latency important?

In some systems, a maximum interrupt response time or latency can be clearly defined. This maximum latency can arise from requirements such as maximum reaction time for a protocol or a software UART implementation that requires very precise timing.

For example a UART receiving at up to 800 kHz in software using ARM FIQ on a 48 MHz ARM7. This would be impossible to do if FIQ were disabled even for short periods of time.

In many embedded systems, the quality of the product depends on event reaction time and therefore latency. Typical examples would be systems which periodically read a value from an A/D converter at high speed, where the accuracy depends on accurate timing. Less jitter means a better product.

Why can a zero latency ISR not use the embOS API?

embOS disables embOS interrupts when embOS data structures are modified. During this time zero latency ISRs are enabled. If they would call an embOS function, which also modifies embOS data, the embOS data structures would be corrupted.

How can a zero latency ISR communicate with a task?

The most common way is to use global variables, e.g. a periodical read from an ADC and the result is stored in a global variable.

Another way is to assert an interrupt request for an embOS interrupt from within the zero latency ISR, which may then communicate or wake up one or more tasks. This is helpful if you want to receive high amounts of data in your zero latency ISR. The embOS ISR may then store the data bytes e.g. in a message queue or in a mailbox.

15.2.5 High / low priority interrupts

Most CPUs support interrupts with different priorities. Different priorities have two effects:

- If different interrupts occur simultaneously, the interrupt with higher priority takes precedence and its ISR is executed first.
- Interrupts can never be interrupted by other interrupts of the same or lower priority.

The number of interrupt levels depends on the CPU and the interrupt controller. Details are explained in the CPU/MCU/SoC manuals and the CPU & Compiler Specifics manual of embOS. embOS distinguishes two different levels of interrupts: High and low priority in-

errupts. High priority interrupts are named **“Zero latency interrupts”** and low priority interrupts are named **“embOS interrupts”**. The embOS port-specific documentations explain which interrupts are considered high and which are considered low priority for that specific port. In general, the differences between those two are as follows:

embOS interrupts

- May call embOS API functions
- Latencies caused by embOS
- Also called “Low priority interrupts”

Zero latency interrupts

- May not call embOS API functions
- No latencies caused by embOS (Zero latency)
- Also called “High priority interrupts”

Example of different interrupt priority levels

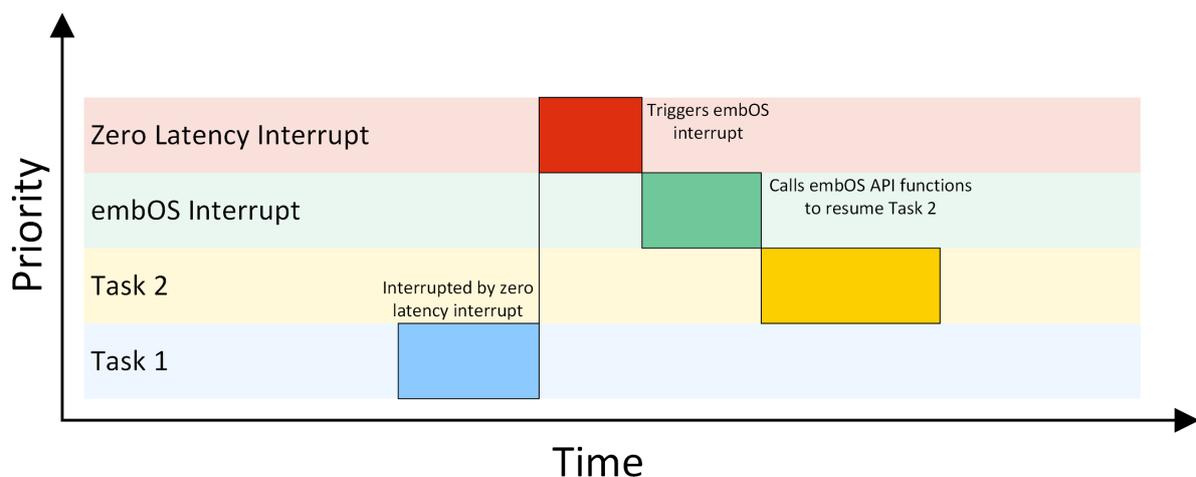
Let’s assume we have a CPU which supports eight interrupt priority levels. With embOS, the interrupt levels are divided per default equal in low priority and high priority interrupt levels. The four highest priority levels are considered “Zero latency interrupts” and the four lowest priority interrupts are considered as “embOS interrupts”. For ARM CPUs, which support regular interrupts (IRQ) and fast interrupt (FIQ), FIQ is considered as “Zero latency interrupt” when using embOS.

For most implementations the high-priority threshold is adjustable. For details, refer to the processor specific embOS manual.

15.2.5.1 Using embOS API from zero latency interrupts

Zero latency interrupts are prohibited from using embOS functions. This is a consequence of embOS’s zero latency design, according to which embOS never disables zero latency interrupts. This means that zero latency interrupts can interrupt the operating system at any time, even in critical sections such as the modification of RTOS-maintained linked lists. This design decision has been made because zero interrupt latencies for zero latency interrupts usually are more important than the ability to call OS functions.

However, zero latency interrupts may use embOS functions in an indirect manner: The zero latency interrupt triggers an embOS interrupt by setting the appropriate interrupt request flag. Subsequently, that embOS interrupt may call the OS functions that the zero latency interrupt was not allowed to use.



The task 1 is interrupted by a high priority interrupt. This zero latency interrupt is not allowed to call an embOS API function directly. Therefore the zero latency interrupt triggers an embOS interrupt, which is allowed to call embOS API functions. The embOS interrupt calls an embOS API function to resume task 2. How the embOS interrupt gets triggered is device specific and cannot be explained here in general. But with most devices and interrupt

controllers, it is possible to set a pending flag for an interrupt. This could for example be an unused peripheral interrupt like a hardware timer. Please refer to your core and/or device manual for more details.

15.3 Rules for interrupt handlers

15.3.1 General rules

There are some general rules for interrupt service routines (ISRs). These rules apply to both single-task programming as well as to multitask programming using embOS.

- ISR preserves all registers.
Interrupt handlers must restore the environment of a task completely. This environment normally consists of the registers only, so the ISR must make sure that all registers modified during interrupt execution are saved at the beginning and restored at the end of the interrupt routine
- Interrupt handlers must finish quickly.
Intensive calculations should be kept out of interrupt handlers. An interrupt handler should only be used for storing a received value or to trigger an operation in the regular program (task). It should not wait in any form or perform a polling operation.

15.3.2 Additional rules for preemptive multitasking

A preemptive multitasking system like embOS needs to know if the code that is executing is part of the current task or an interrupt handler. This is necessary because embOS cannot perform a task switch during the execution but only at the end of an ISR.

If a task switch was to occur during the execution of an ISR, the ISR would continue as soon as the interrupted task became the current task again. This is not a problem for interrupt handlers that do not allow further interruptions (which do not enable interrupts) and that do not call any embOS functions.

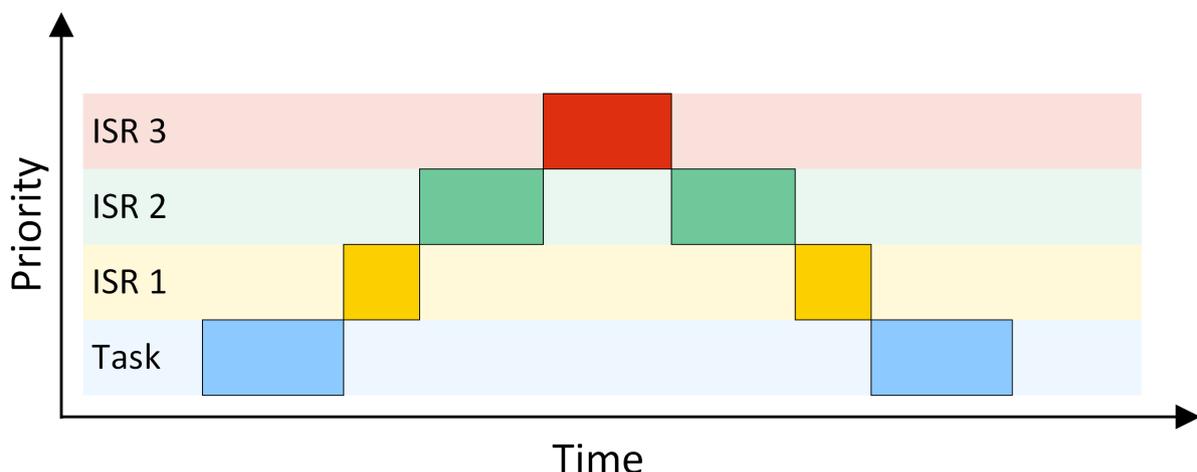
This leads us to the following rule:

- ISRs that re-enable interrupts or use any embOS function need to call `OS_INT_Enter()` at the beginning, before executing anything else, and call `OS_INT_Leave()` immediately before returning.

If a higher priority task is made ready by the ISR, the task switch may be performed in the routine `OS_INT_Leave()`. The end of the ISR is executed later on, when the interrupted task has been made ready again. Please consider this behavior if you debug an interrupt routine, this has proven to be the most efficient way of initiating a task switch from within an interrupt service routine.

15.3.3 Nesting interrupt routines

By default, interrupts are disabled in an ISR because most CPU disables interrupts with the execution of the interrupt handler. Re-enabling interrupts in an interrupt handler allows the execution of further interrupts with equal or higher priority than that of the current interrupt. These are known as nested interrupts, illustrated in the diagram below:



For applications requiring short interrupt latency, you may re-enable interrupts inside an ISR by using `OS_INT_EnterNestable()` and `OS_INT_LeaveNestable()` within the interrupt handler.

Nested interrupts can lead to problems that are difficult to debug; therefore it is not recommended to enable interrupts within an interrupt handler. As it is important that embOS keeps track of the status of the interrupt enable/disable flag, enabling and disabling of interrupts from within an ISR must be done using the functions that embOS offers for this purpose.

The routine `OS_INT_EnterNestable()` enables interrupts within an ISR and prevents further task switches; `OS_INT_LeaveNestable()` disables interrupts immediately before ending the interrupt routine, thus restoring the default condition. Re-enabling interrupts will make it possible for an embOS scheduler interrupt to interrupt this ISR. In this case, embOS needs to know that another ISR is still active and that it may not perform a task switch.

15.3.4 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_INT_Call()</code>	Entry function for use in an embOS interrupt handler.				•	
<code>OS_INT_CallNestable()</code>	Entry function for use in an embOS interrupt handler.				•	
<code>OS_INT_Enter()</code>	Informs embOS that interrupt code is executing.				•	
<code>OS_INT_EnterIntStack()</code>	Switches to another stack in interrupt routines.				•	
<code>OS_INT_EnterNestable()</code>	Informs embOS that interrupt code is executing and reenables interrupts.				•	
<code>OS_INT_InInterrupt()</code>	Checks if the calling function runs in an interrupt context.	•	•	•	•	•
<code>OS_INT_Leave()</code>	Informs embOS that the end of the interrupt routine has been reached; executes task switching within ISR.				•	
<code>OS_INT_LeaveIntStack()</code>	Switches back to the interrupt stack.				•	
<code>OS_INT_LeaveNestable()</code>	Informs embOS that the end of the interrupt routine has been reached; executes task switching within ISR.				•	

15.3.4.1 OS_INT_Call()

Description

Entry function for use in an embOS interrupt handler. Nestable interrupts are disabled.

Prototype

```
void OS_INT_Call(void ( *pfRoutine)());
```

Parameters

Parameter	Description
<code>pfRoutine</code>	Pointer to a routine that should run on interrupt.

Additional information

`OS_INT_Call()` can be used as an entry function in an embOS interrupt handler, when the corresponding interrupt should not be interrupted by another embOS interrupt.

`OS_INT_Call()` sets the interrupt priority of the CPU to the user definable 'fast' interrupt priority level, thus locking any other embOS interrupt. Fast interrupts are not disabled.

Note

For some specific CPUs `OS_INT_Call()` must be used to call an interrupt handler because `OS_INT_Enter()/OS_INT_Leave()` may not be available.

`OS_INT_Call()` must not be used when `OS_INT_Enter()/OS_INT_Leave()` is available. Please refer to the CPU/compiler specific embOS manual.

Example

```
#pragma interrupt
void SysTick_Handler(void) {
    OS_INT_Call(_IsrTickHandler);
}
```

15.3.4.2 OS_INT_CallNestable()

Description

Entry function for use in an embOS interrupt handler. Nestable interrupts are enabled.

Prototype

```
void OS_INT_CallNestable(void ( *pfRoutine)());
```

Parameters

Parameter	Description
<code>pfRoutine</code>	Pointer to a routine that should run on interrupt.

Additional information

`OS_INT_CallNestable()` can be used as an entry function in an embOS interrupt handler, when interruption by higher prioritized embOS interrupts should be allowed.

`OS_INT_CallNestable()` does not alter the interrupt priority of the CPU, thus keeping all interrupts with higher priority enabled.

Note

For some specific CPUs `OS_INT_CallNestable()` must be used to call an interrupt handler because `OS_INT_EnterNestable()/OS_INT_LeaveNestable()` may not be available.

`OS_INT_CallNestable()` must not be used when `OS_INT_EnterNestable()/OS_INT_LeaveNestable()` is available

Please refer to the CPU/compiler specific embOS manual.

Example

```
#pragma interrupt
void SysTick_Handler(void) {
    OS_INT_CallNestable(_IsrTickHandler);
}
```

15.3.4.3 OS_INT_Enter()

Description

Informs embOS that interrupt code is executing.

Prototype

```
void OS_INT_Enter(void);
```

Additional information

Note

This function is not available in all ports.

If `OS_INT_Enter()` is used, it should be the first function to be called in the interrupt handler. It must be paired with `OS_INT_Leave()` as the last function called. The use of this function has the following effects:

- disables task switches
- keeps interrupts in internal routines disabled.

Example

```
void ISR_Timer(void) {  
    OS_INT_Enter();  
    OS_TASKEVENT_Set(&Task, 1u); // Any functionality could be here  
    OS_INT_Leave();  
}
```

15.3.4.4 OS_INT_EnterIntStack()

Description

OS_INT_EnterIntStack() and OS_INT_LeaveIntStack() can be used to switch the stack pointer to another stack during execution of the interrupt routine.

Prototype

```
void OS_INT_EnterIntStack(void);
```

Additional information

The actual implementation is core and compiler dependent. Therefore, OS_INT_EnterIntStack() and OS_INT_LeaveIntStack() are not implemented in all embOS ports. In that case OS_INT_EnterIntStack() is defined for compatibility reasons to nothing. That simplifies the porting of an existing embOS application to another embOS port.

Note

Please be aware any variables that are declared while using the initial stack, will no longer be accessible after switching to the interrupt stack.

```
void ISR_Timer(void) {  
    //  
    // Accessible only before OS_INT_EnterIntStack() is called,  
    // and after OS_INT_LeaveIntStack() was called.  
    //  
    int localvar = 0;  
  
    OS_INT_Enter();  
    OS_INT_EnterIntStack();  
    OS_TASKEVENT_Set(&Task, Event);  
    OS_INT_LeaveIntStack();  
    OS_INT_Leave();  
}
```

15.3.4.5 OS_INT_EnterNestable()

Description

Re-enables interrupts and increments the embOS internal critical region counter, thus disabling further task switches.

Prototype

```
void OS_INT_EnterNestable(void);
```

Additional information

Note

This function is not available in all ports.

This function should be the first call inside an interrupt handler when nested interrupts are required. The function `OS_INT_EnterNestable()` is implemented as a macro and offers the same functionality as `OS_INT_Enter()` in combination with `OS_INT_DecRI()`, but is more efficient, resulting in smaller and faster code.

Example

```
_interrupt void ISR_Timer(void) {  
    OS_INT_EnterNestable();  
    OS_TASKEVENT_Set(&Task, 1); // Any functionality could be here  
    OS_INT_LeaveNestable();  
}
```

15.3.4.6 OS_INT_InInterrupt()

Description

This function can be called to examine if the calling function is running in an interrupt context. For application code, it may be useful to know if it is called from interrupt or task, because some functions must not be called from an interrupt-handler.

Prototype

```
OS_BOOL OS_INT_InInterrupt(void);
```

Return value

= 0 Code is not executed in an interrupt handler.
≠ 0 Code is executed in an interrupt handler.

Additional information

Note

This function is not available in all ports.

The function delivers the interrupt state by checking the according CPU registers. It is only implemented for those CPUs where it is possible to read the interrupt state from CPU registers. In case of doubt please contact the embOS support.

Example

```
void foo(void) {  
    if (OS_INT_InInterrupt() != 0) {  
        // Do something within the ISR  
    } else {  
        printf("No interrupt context.\n")  
    }  
}
```

15.3.4.7 OS_INT_Leave()

Description

Informs embOS that the end of the interrupt routine has been reached; executes task switching within ISR.

Prototype

```
void OS_INT_Leave(void);
```

Additional information

Note

This function is not available in all ports.

If `OS_INT_Leave()` is used, it should be the last function to be called in the interrupt handler. If the interrupt has caused a task switch, that switch is performed immediately (unless the program which was interrupted was in a critical region).

Example

```
void ISR_Timer(void) {  
    OS_INT_Enter();  
    OS_TASKEVENT_Set(&Task, 1); // Any functionality could be here  
    OS_INT_Leave();  
}
```

15.3.4.8 OS_INT_LeaveIntStack()

Description

OS_INT_EnterIntStack() and OS_INT_LeaveIntStack() can be used to switch the stack pointer to another stack during execution of the interrupt routine.

Prototype

```
void OS_INT_LeaveIntStack(void);
```

Additional information

The actual implementation is device and compiler dependent. Therefore OS_INT_EnterIntStack() and OS_INT_LeaveIntStack() are not implemented in all embOS ports. In that case OS_INT_EnterIntStack() is defined for compatibility reasons to nothing. That simplifies the porting of an existing embOS application to another embOS port.

Example

```
void ISR_Timer(void) {
    OS_INT_Enter();
    OS_INT_EnterIntStack();
    OS_TASKEVENT_Set(&Task, 1);
    OS_INT_LeaveIntStack();
    OS_INT_Leave();
}
```

15.3.4.9 OS_INT_LeaveNestable()

Description

Disables further interrupts, then decrements the embOS internal critical region count, thus re-enabling task switches if the counter has reached zero.

Prototype

```
void OS_INT_LeaveNestable(void);
```

Additional information

Note

This function is not available in all ports.

This function is the counterpart of `OS_INT_EnterNestable()`, and must be the last function call inside an interrupt handler when nested interrupts have been enabled by `OS_INT_EnterNestable()`.

The function `OS_INT_LeaveNestable()` is implemented as a macro and offers the same functionality as `OS_INT_Leave()` in combination with `OS_INT_IncDI()`, but is more efficient, resulting in smaller and faster code.

Example

```
_interrupt void ISR_Timer(void) {  
    OS_INT_EnterNestable();  
    OS_TASKEVENT_Set(&Task, 1); // Any functionality could be here  
    OS_INT_LeaveNestable();  
}
```

15.4 Interrupt control

15.4.1 Enabling / disabling interrupts

During the execution of a task, maskable interrupts are normally enabled. In certain sections of the program, however, it can be necessary to disable interrupts for short periods of time to make a section of the program an atomic operation that cannot be interrupted. We recommend disabling interrupts only for short periods of time, if possible (the longer interrupts are disabled, the higher is the interrupt latency).

An example would be the access to a global volatile variable of type long on an 8/16-bit CPU. To make sure that the value does not change between the two or more memory accesses that are needed, interrupts must be temporarily disabled:

Bad example:

```
volatile long lvar;

void IntHandler(void) {
    lvar++;
}

void Routine(void) {
    lvar++;
}
```

Good example:

```
volatile long lvar;

void IntHandler(void) {
    lvar++;
}

void Routine(void) {
    OS_INT_Disable();
    lvar++;
    OS_INT_Enable();
}
```

15.4.2 Nested interrupt disable and enable calls

`OS_INT_Disable()` does not use a counter on how often it was executed and `OS_INT_Enable()` does not evaluate any counter before enabling interrupts. The below sample fails when the application expects that interrupts are still disabled after the call to `bar()`.

```
void bar(void) {
    OS_INT_Disable();
    DoSomething();
    OS_INT_Enable();
}

void foo(void) {
    OS_INT_Disable();
    bar();
    DoSomethingElse();
    OS_INT_Enable();
}
```

To avoid this `bar()` must not enable embOS interrupts unconditionally but use `OS_INT_EnableConditional()` instead and the routine `foo()` must call `OS_INT_IncDI()`. `OS_INT_IncDI()` does not disable embOS interrupts only but also increments the interrupt disable counter `OS_Global.Counters.Cnt.DI`. `OS_INT_EnableConditional()` enables embOS interrupts only when the interrupt disable counter `OS_Global.Counters.Cnt.DI` is zero.

```
void bar(void) {
    OS_INT_Disable();
    DoSomething();
    OS_INT_EnableConditional();
}

void foo(void) {
    OS_INT_IncDI();
    foo();
    DoSomethingElse();
    OS_INT_DecRI();
}
```

Assuming the interrupt disable counter was zero it will be incremented with `OS_INT_IncDI()` to one and embOS interrupts will be disabled. `OS_INT_Disable()` will again disable embOS interrupts. `OS_INT_EnableConditional()` evaluates the interrupt disable counter and since it is unequal to zero embOS interrupts will not be enabled. `OS_INT_DecRI()` decrements the interrupt disable counter which sets it to zero. It also evaluates the interrupt disable counter and since it is equal to zero embOS interrupts will be re-enabled.

`OS_Global.Counters.Cnt.DI` is saved during the context switch in the task context and is restored when the task is activated again. Therefore, the interrupt disable counter is task specific. Disabling embOS interrupts in one task does not disable embOS interrupts for other tasks or the kernel.

`OS_INT_Disable()` and `OS_INT_Enable()` can be used when no embOS API functions are called which could enable interrupts before the actual call to `OS_INT_Enable()` and the interrupt disable counter `OS_Global.Counters.Cnt.DI` is zero. `OS_INT_Disable()` / `OS_INT_Enable()` are slightly more efficient than `OS_INT_IncDI()` / `OS_INT_DecRI()`. `OS_INT_EnableConditional()` should be used when the interrupt disable counter value is unknown.

`OS_INT_IncDI()` / `OS_INT_DecRI()` can be used to avoid that embOS interrupts could be enabled unconditionally. This could e.g. the case when you call an embOS API function which internally disables and enables embOS interrupts. embOS API functions always enable embOS interrupts conditionally with `OS_INT_EnableConditional()`.

```
void foo(void) {
    OS_INT_IncDI();
    OS_TASK_Delay(10);
    DoSomething() // embOS interrupts are still disabled
    OS_INT_DecRI();
}
```

15.4.3 Zero latency interrupt disable / enable

The embOS interrupt enable and disable functions enable and disable embOS interrupts only. Zero latency interrupts are never implicitly enabled or disabled by embOS. However, embOS provides additional API functions to enable and disable embOS interrupts and zero latency interrupts.

In an application it may be required to disable and enable all interrupts. These API functions have the suffix `All`. These functions affect the state of the CPU unconditionally and should be used with care.

It is CPU specific whether zero latency interrupts are supported. If not, API functions with the suffix `All` (like e.g. `OS_INT_DisableAll()`) behave exactly the same as the API functions without this suffix.

15.4.4 Non-maskable interrupts (NMIs)

embOS performs atomic operations by disabling interrupts. However, a non-maskable interrupt (NMI) cannot be disabled, meaning it can interrupt these atomic operations. Therefore, NMIs should be used with great care and are prohibited from calling any embOS routines.

15.4.5 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer	Idle
<code>OS_INT_DecrI()</code>	Decrements the embOS interrupt disable counter and enables interrupts if the counter reaches 0.	•	•		•	•	•
<code>OS_INT_Disable()</code>	Disables embOS interrupts. Does not change the interrupt disable counter.	•	•		•	•	•
<code>OS_INT_DisableAll()</code>	Disables all (embOS and zero latency) interrupts.	•	•		•	•	•
<code>OS_INT_Enable()</code>	Enables embOS interrupts unconditionally.	•	•		•	•	•
<code>OS_INT_EnableAll()</code>	Enables all (embOS and zero latency) interrupts unconditionally.	•	•		•	•	•
<code>OS_INT_EnableConditional()</code>	Enables embOS interrupts if the interrupt disable counter equals 0.	•	•		•	•	•
<code>OS_INT_IncDI()</code>	Increments the embOS interrupt disable counter and disables embOS interrupts.	•	•		•	•	•
<code>OS_INT_Preserve()</code>	Preserves the embOS interrupt state.	•	•		•	•	•
<code>OS_INT_PreserveAll()</code>	Preserves the embOS and zero latency interrupt state.	•	•		•	•	•
<code>OS_INT_PreserveAndDisable()</code>	Preserves the embOS interrupt state and then disables embOS interrupts.	•	•		•	•	•
<code>OS_INT_PreserveAndDisableAll()</code>	Preserves the embOS and zero latency interrupt state and then disables all (embOS and zero latency) interrupts.	•	•		•	•	•
<code>OS_INT_Restore()</code>	Restores the embOS interrupt state.	•	•		•	•	•
<code>OS_INT_RestoreAll()</code>	Restores the embOS and zero latency interrupt state.	•	•		•	•	•

15.4.5.1 OS_INT_DecRI()

Description

OS_INT_DecRI() decrements the embOS interrupt disable counter OS_Global.Counters.Cnt.DI and enables embOS interrupts if the interrupt disable counter reaches zero.

Prototype

```
void OS_INT_DecRI(void);
```

Additional information

Short for *Decrement and Restore Interrupts*. It is important that OS_INT_IncDI() and OS_INT_DecRI() are used as a pair.

OS_INT_IncDI() increments the interrupt disable counter. Interrupts will not be switched on within the running task before the matching OS_INT_DecRI() is executed.

The interrupt disable counter OS_Global.Counters.Cnt.DI is task specific. A task switch may change the value, so if interrupts are disabled in one task they could be enabled in the next task and vice versa.

You can safely call embOS API between OS_INT_IncDI() and OS_INT_DecRI(). embOS API functions will not change the counter value.

Example

```
volatile long lvar;  
  
void Routine(void) {  
    OS_INT_IncDI();  
    lvar++;  
    OS_INT_DecRI();  
}
```

15.4.5.2 OS_INT_Disable()

Description

OS_INT_Disable() disables embOS interrupts.

Prototype

```
void OS_INT_Disable(void);
```

Additional information

OS_INT_Disable() does not disable zero latency interrupts.

embOS interrupts can be re-enabled by calling OS_INT_Enable().

OS_INT_Disable() does not preserve the interrupt disable state. Please use OS_INT_Preserve() or OS_INT_PreserveAndDisable() instead when the interrupt disable state should be preserved.

OS_INT_Disable() does not increment the interrupt disable counter OS_Global.Counter-s.Cnt.DI. An embOS API function may re-enable embOS interrupts. Please use OS_INT_IncDI() and OS_INT_DecRI() instead.

```
void Task(void) {
    OS_INT_Disable();
    OS_TASK_Delay(10);
    DoSomething(); // embOS interrupts may be executed
}
```

Therefore, it is not recommend to call any embOS API function after OS_INT_Disable() except OS_INT_Enable().

Example

```
void Routine(void) {
    OS_INT_Disable(); // Disable embOS interrupts
    //
    // Execute any code that should be executed with interrupts disabled.
    // No embOS function should be called.
    //
    DoSomething();
    OS_INT_Enable(); // Re-enable embOS interrupts unconditionally.
}
```

15.4.5.3 OS_INT_DisableAll()

Description

OS_INT_DisableAll() disables embOS interrupts and zero latency interrupts.

Prototype

```
void OS_INT_DisableAll(void);
```

Additional information

embOS interrupts and zero latency interrupts can be re-enabled by calling OS_INT_EnableAll().

OS_INT_DisableAll() does not preserve the interrupt disable state. Please use OS_INT_PreserveAll() or OS_INT_PreserveAndDisableAll() instead when the interrupt disable state should be preserved.

OS_INT_DisableAll() does not increment the interrupt disable counter.

An embOS API function may re-enable embOS interrupts depending on the CPU. Please refer to the CPU and compiler specific manual for more details.

```
void Task(void) {
    OS_INT_DisableAll();
    OS_TASK_Delay(10);
    DoSomething(); // embOS interrupts may be executed
}
```

Therefore, it is not recommend to call any embOS API function after OS_INT_DisableAll() except OS_INT_EnableAll().

Example

```
void Routine(void) {
    OS_INT_DisableAll(); // Disable interrupts
    //
    // Execute any code that should be executed with interrupts disabled
    // No embOS function should be called.
    //
    DoSomething();
    OS_INT_EnableAll(); // Re-enable interrupts unconditionally
}
```

15.4.5.4 OS_INT_Enable()

Description

OS_INT_Enable() enables embOS interrupts.

Prototype

```
void OS_INT_Enable(void);
```

Additional information

OS_INT_Enable() does not enable zero latency interrupts.

OS_INT_Enable() does not decrement or check the interrupt disable counter OS_Global.Counters.Cnt.DI but enables interrupts unconditionally. If interrupts should be enabled only when the interrupt disable counter OS_Global.Counters.Cnt.DI is zero please use OS_INT_EnableConditional() instead.

Example

```
void Routine(void) {  
    OS_INT_Disable(); // Disable embOS interrupts  
    DoSomething();  
    OS_INT_Enable(); // Re-enable embOS interrupts unconditionally  
}
```

15.4.5.5 OS_INT_EnableAll()

Description

OS_INT_EnableAll() enables embOS interrupts and zero latency interrupts.

Prototype

```
void OS_INT_EnableAll(void);
```

Additional information

OS_INT_EnableAll() does not decrement or check the interrupt disable counter OS_Global.Counters.Cnt.DI but enables interrupts unconditionally.

Example

```
void Routine(void) {
    OS_INT_DisableAll(); // Disable interrupts
    DoSomething();
    OS_INT_EnableAll(); // Re-enable interrupts unconditionally
}
```

15.4.5.6 OS_INT_EnableConditional()

Description

OS_INT_EnableConditional() enables embOS interrupts conditionally depending on the interrupt disable counter OS_Global.Counters.Cnt.DI.

Prototype

```
void OS_INT_EnableConditional(void);
```

Additional information

OS_INT_EnableConditional() enables embOS interrupts only if the interrupt disable counter OS_Global.Counters.Cnt.DI is zero.

Example

```
void Routine (void) {  
    OS_INT_Disable();  
    DoSomething();  
    OS_INT_EnableConditional();  
}
```

15.4.5.7 OS_INT_IncDI()

Description

OS_INT_IncDI() disables embOS interrupts and increments the interrupt disable counter OS_Global.Counters.Cnt.DI.

Prototype

```
void OS_INT_IncDI(void);
```

Additional information

Short for Increment and Disable interrupts. OS_INT_IncDI() does not disable zero latency interrupts.

It is important that OS_INT_IncDI() and OS_INT_DecRI() are used as a pair. Interrupts will not be switched on within the running task before the matching OS_INT_DecRI() is executed.

The interrupt disable counter OS_Global.Counters.Cnt.DI is task specific. A task switch may change the value, so if interrupts are disabled they could be enabled in the next task and vice versa.

You can safely call embOS API between OS_INT_IncDI() and OS_INT_DecRI(). The embOS API will not enable interrupts.

Example

```
void Routine (void) {
    OS_INT_IncDI();
    DoSomething();
    OS_INT_DecRI();
}
```

15.4.5.8 OS_INT_Preserve()

Description

OS_INT_Preserve() preserves the current embOS interrupt disable state of the CPU.

Prototype

```
void OS_INT_Preserve(OS_U32* pState);
```

Parameters

Parameter	Description
pState	Pointer to an OS_U32 variable that receives the interrupt state.

Additional information

If the embOS interrupt disable state is not known and embOS interrupts should be disabled by a call of OS_INT_Disable(), the current embOS interrupt disable state can be preserved with OS_INT_Preserve() and restored later by a call of OS_INT_Restore().

OS_INT_Preserve() preserves the embOS interrupt disable state but not the zero latency interrupt state.

The pair of function calls OS_INT_Preserve() and OS_INT_Restore() can be nested, as long as the interrupt disable state is stored into an individual variable on each call of OS_INT_Preserve().

Example

```
void Sample(void) {
    OS_U32 IntState;

    OS_INT_Preserve(&IntState); // Remember the interrupt disable state.
    OS_INT_Disable();           // Disable embOS interrupts
    //
    // Execute any code that should be executed with embOS interrupts disabled.
    //
    DoSomething();
    OS_INT_Restore(&IntState); // Restore the interrupt disable state
}
```

15.4.5.9 OS_INT_PreserveAll()

Description

OS_INT_PreserveAll() preserves the current zero latency interrupt disable state of the CPU.

Prototype

```
void OS_INT_PreserveAll (OS_U32* pState);
```

Parameters

Parameter	Description
pState	Pointer to an OS_U32 variable that receives the interrupt state.

Additional information

If the zero latency interrupt disable state is not known and zero latency interrupts should be disabled by a call of OS_INT_DisableAll(), the current zero latency interrupt disable state can be preserved with OS_INT_PreserveAll() and restored later by a call of OS_INT_RestoreAll().

OS_INT_PreserveAll() preserves the zero latency interrupt disable state but not the em-bOS interrupt disable state.

The pair of function calls OS_INT_PreserveAll() and OS_INT_RestoreAll() can be nested, as long as the interrupt disable state is stored into an individual variable on each call of OS_INT_Preserve().

Example

```
void Sample(void) {
    OS_U32 IntState;

    // Remember the interrupt disable state.
    OS_INT_PreserveAll(&IntState);
    OS_INT_DisableAll(); // Disable interrupts
    //
    // Execute any code that should be executed with interrupts disabled
    //
    DoSomething();
    OS_INT_RestoreAll(&IntState); // Restore the interrupt disable state
}
```

15.4.5.10 OS_INT_PreserveAndDisable()

Description

OS_INT_PreserveAndDisable() preserves the current embOS interrupt disable state of the CPU and then disables embOS interrupts.

Prototype

```
void OS_INT_PreserveAndDisable (OS_U32* pState);
```

Parameters

Parameter	Description
pState	Pointer to an OS_U32 variable that receives the interrupt state.

Additional information

If the embOS interrupt disable state is not known and embOS interrupts should be disabled, the current embOS interrupt disable state can be preserved and embOS interrupts disabled with OS_INT_PreserveAndDisable() and restored later by a call of OS_INT_Restore().

OS_INT_PreserveAndDisable() preserves the embOS interrupt disable state but not the zero latency interrupt state.

The pair of function calls OS_INT_PreserveAndDisable() and OS_INT_Restore() can be nested, as long as the interrupt disable state is stored into an individual variable on each call of OS_INT_PreserveAndDisable().

Example

```
void Sample(void) {
    OS_U32 IntState;

    // Remember the interrupt disable state and disables interrupts.
    OS_INT_PreserveAndDisable(&IntState);
    //
    // Execute any code that should be executed with interrupts disabled
    //
    DoSomething();
    OS_INT_Restore(&IntState); // Restore the interrupt disable state
}
```

15.4.5.11 OS_INT_PreserveAndDisableAll()

Description

OS_INT_PreserveAndDisableAll() preserves the current zero latency interrupt disable state of the CPU and disables embOS interrupts and zero latency interrupts.

Prototype

```
void OS_INT_PreserveAndDisableAll (OS_U32* pState);
```

Parameters

Parameter	Description
pState	Pointer to an OS_U32 variable that receives the interrupt state.

Additional information

If the zero latency interrupt disable state is not known and interrupts should be disabled by a call of OS_INT_DisableAll(), the current interrupt disable state can be preserved and interrupts disabled with OS_INT_PreserveAll() and restored later by a call of OS_INT_RestoreAll().

OS_INT_PreserveAndDisableAll() preserves the zero latency interrupt disable state but not the embOS interrupt disable state.

An embOS API function may re-enable embOS interrupts depending on the CPU. Please refer to the CPU and compiler specific manual for more details.

The pair of function calls OS_INT_PreserveAndDisableAll() and OS_INT_RestoreAll() can be nested, as long as the interrupt disable state is stored into an individual variable on each call of OS_INT_PreserveAndDisableAll().

Example

```
void Sample(void) {
    OS_U32 IntState;

    // Remember the interrupt disable state and disables interrupts.
    OS_INT_PreserveAndDisableAll(&IntState);
    //
    // Execute any code that should be executed with interrupts disabled
    //
    DoSomething();
    OS_INT_RestoreAll(&IntState); // Restore the interrupt disable state
}
```

15.4.5.12 OS_INT_Restore()

Description

OS_INT_Restore() restores the embOS interrupt disable state.

Prototype

```
void OS_INT_Restore (OS_U32* pState);
```

Parameters

Parameter	Description
pState	Pointer to an OS_U32 variable that holds the interrupt disable state.

Additional information

OS_INT_Restore() restores the embOS interrupt disable state of the CPU which was saved before by a call of OS_INT_Preserve() or OS_INT_PreserveAndDisable().

Example

```
void Sample(void) {
    OS_U32 IntState;

    OS_INT_Preserve(&IntState); // Remember the interrupt disable state.
    OS_INT_Disable();           // Disable embOS interrupts
    //
    // Execute any code that should be executed with embOS interrupts disabled
    //
    DoSomething();
    OS_INT_Restore(&IntState); // Restore the interrupt disable state
}
```

15.4.5.13 OS_INT_RestoreAll()

Description

OS_INT_RestoreAll() restores the zero latency interrupt disable state.

Prototype

```
void OS_INT_RestoreAll (OS_U32* pState);
```

Parameters

Parameter	Description
pState	Pointer to an OS_U32 variable that holds the interrupt disable state.

Additional information

OS_INT_RestoreAll() restores the zero latency interrupt disable state of the CPU which was saved before by a call of OS_INT_PreserveAll() or OS_INT_PreserveAndDisableAll().

Example

```
void Sample(void) {
    OS_U32 IntState;

    // Remember the interrupt disable state.
    OS_INT_PreserveAll(&IntState);
    OS_INT_DisableAll(); // Disable interrupts
    //
    // Execute any code that should be executed with interrupts disabled
    // No embOS function should be called
    //
    DoSomething();
    OS_INT_RestoreAll(&IntState); // Restore the interrupt disable state
}
```

Chapter 16

Critical Region

16.1 Introduction

Critical regions are program sections which should not be interrupted by another task. A critical region can be used anywhere during execution of a task. Depending on the application, it can be necessary for some critical program sections to disable preemptive task switches and execution of software timers or even interrupts.

It depends on the application whether disabling task switches is sufficient or interrupts need to be disabled as well. Disabling interrupts can mean to disable embOS interrupts or even to also disable zero latency interrupts. Cooperative task switches are never affected and will be executed in critical regions. Interrupts, too, may still occur in critical regions.

They may also be used in software timers and interrupts. However, since those are executed as critical regions anyways, critical regions do not have any effect on them.

Critical regions can be nested; they will then be effective until the outermost region is left. If a task switch becomes pending during the execution of a critical region, it will be performed immediately once the region is left.

A typical example for critical regions is the execution of time-critical hardware accesses (for example, writing multiple bytes into an EEPROM where the bytes must be written in a certain amount of time), or writing to global variables that are accessed by different tasks and therefore must ensure that data is consistent.

Example

```
void HPTask(void) {
    OS_TASK_EnterRegion();
    DoSomething(); // This code will not be interrupted by other tasks
    OS_TASK_LeaveRegion();
}
```

Note

Cooperative task switches are still executed, although preemptive task switches are disabled in critical sections.

```
void HPTask(void) {
    OS_TASK_EnterRegion();
    OS_TASK_Delay(100); // OS_TASK_Delay() will cause a cooperative task switch
    OS_TASK_LeaveRegion();
}
```

16.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer	Idle
<code>OS_TASK_EnterRegion()</code>	Indicates the beginning of a critical region to embOS.	•	•		•	•	•
<code>OS_TASK_LeaveRegion()</code>	Indicates to embOS the end of a critical region.	•	•		•	•	•

16.2.1 OS_TASK_EnterRegion()

Description

Indicates the beginning of a critical region to embOS.

Prototype

```
void OS_TASK_EnterRegion(void);
```

Additional information

The critical region counter (`OS_Global.Counters.Cnt.Region`) is zero by default. It gets incremented upon calling `OS_TASK_EnterRegion()` and decremented upon calling `OS_TASK_LeaveRegion()`. Critical regions can be nested: the critical region ends when this counter reaches zero again. The counter is specific for all tasks, its value is saved and restored on any task switch.

Interrupts are not disabled in a critical region. However, preemptive task switches are. If any interrupt triggers a task switch, the task switch stays pending until the final call of `OS_TASK_LeaveRegion()`. When the counter reaches zero, a pending task switch is executed.

Cooperative task switches are not affected and will be executed in critical regions. When a task is running in a critical region and calls any blocking embOS function, the task will be suspended. When the task is resumed, the critical region counter is restored, the task continues to run in a critical region until `OS_TASK_LeaveRegion()` is called.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
157: OS_ERR_REGIONCNT
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Critical Regions* on page .

16.2.2 OS_TASK_LeaveRegion()

Description

Indicates to embOS the end of a critical region. Decrements the critical region counter and checks if a task switch is pending if the counter reaches 0.

Prototype

```
void OS_TASK_LeaveRegion(void);
```

Additional information

A critical region counter (`OS_Global.Counters.Cnt.Region`), which is zero by default, is decremented. If this counter reaches zero, the critical region ends. A task switch which became pending during a critical region will be executed in `OS_TASK_EnterRegion()` when the counter reaches zero.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

151: OS_ERR_LEAVEREGION_BEFORE_ENTERREGION

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Critical Regions* on page .

16.3 Disabling context transitions

The following table shows which context transitions may occur after calling appropriate embOS API:

	Cooperative task switch	Preemptive task switch	Software Timer	embOS interrupt	Zero latency interrupt
Regular execution	•	•	•	•	•
In critical region	•			•	•
With embOS interrupts disabled	•				•
With all interrupts disabled	•				

Example

In the following example `DoSomething()` in the `LPTask` cannot be interrupted by the `HPTask` or the software timer `SoftwareTimer`. But it can be interrupted by the interrupt routines `embOS_ISR` and `Zero_Latency_ISR`.

```
void Zero_Latency_ISR(void) {
    DoSomething();
}

void embOS_ISR(void) {
    OS_INT_Enter();
    DoSomething();
    OS_INT_Leave();
}

void SoftwareTimer(void) {
    DoSomething();
    OS_TIMER_Restart(&Timer);
}

void HPTask(void) {
    while (1) {
        DoSomething();
        OS_TASK_Delay(10);
    }
}

void LPTask(void) {
    while (1) {
        OS_TASK_EnterRegion();
        DoSomething();
        OS_TASK_LeaveRegion();
    }
}
```

In this example `DoSomething()` in the `LPTask` cannot be interrupted by the `HPTask`, the software timer `SoftwareTimer` or the embOS interrupt routine `embOS_ISR`. But it can be interrupted by the zero latency interrupt routine `Zero_Latency_ISR`.

```
void Zero_Latency_ISR(void) {
    DoSomething();
}

void embOS_ISR(void) {
    OS_INT_Enter();
    DoSomething();
    OS_INT_Leave();
}
```

```
void SoftwareTimer(void) {
    DoSomething();
    OS_TIMER_Restart(&Timer);
}

void HPTask(void) {
    while (1) {
        DoSomething();
        OS_TASK_Delay(10);
    }
}

void LPTask(void) {
    while (1) {
        OS_INT_Disable();
        DoSomething();
        OS_INT_Enable();
    }
}
```

In this last example, `DoSomething()` in the `LPTask` cannot be interrupt by any other function.

```
void Zero_Latency_ISR(void) {
    DoSomething();
}

void embOS_ISR(void) {
    OS_INT_Enter();
    DoSomething();
    OS_INT_Leave();
}

void SoftwareTimer(void) {
    DoSomething();
    OS_TIMER_Restart(&Timer);
}

void HPTask(void) {
    while (1) {
        DoSomething();
        OS_TASK_Delay(10);
    }
}

void LPTask(void) {
    while (1) {
        OS_INT_DisableAll();
        DoSomething();
        OS_INT_EnableAll();
    }
}
```

Chapter 17

Time Measurement

17.1 Introduction

embOS supports two basic types of run-time measurement which may be used e.g. for calculating the execution time of any section of user code. Low-resolution measurements are based on system ticks, while high-resolution measurements are based on a time unit called *cycle*. The length of a timer cycle depends on the timer clock frequency.

Example

The following sample demonstrates the use of low-resolution and high-resolution measurement to return the execution time of a section of code:

```
#include "RTOS.h"
#include <stdio.h>

static OS_STACKPTR int Stack[1000]; // Task stacks
static OS_TASK      TCB;           // Task-control-blocks
static volatile int  Dummy;

static void UserCode(void) {
    for (Dummy=0; Dummy < 11000; Dummy++); // Burn some time
}

//
// Measure the execution time with low resolution
//
static OS_TIME BenchmarkLoRes(void) {
    OS_TIME t;
    t = OS_TIME_GetTicks();
    UserCode(); // Execute the user code to be benchmarked
    t = OS_TIME_GetTicks() - t;
    return (int)t;
}

//
// Measure the execution time with high resolution
//
static OS_U32 BenchmarkHiRes(void) {
    OS_U32 t;

    OS_TIME_StartMeasurement(&t);
    UserCode(); // Execute the user code to be benchmarked
    OS_TIME_StopMeasurement(&t);
    return OS_TIME_GetResult_us(&t);
}

void Task(void) {
    int    tLow;
    OS_U32 tHigh;
    char   acBuffer[80];

    while (1) {
        tLow = BenchmarkLoRes();
        tHigh = BenchmarkHiRes();
        sprintf(acBuffer, "LoRes: %d system ticks\n", tLow);
        OS_COM_SendString(acBuffer);
        sprintf(acBuffer, "HiRes: %d microseconds\n", tHigh);
        OS_COM_SendString(acBuffer);
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_InitHW(); // Initialize hardware for embOS
    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);
    OS_Start(); // Start multitasking
    return 0;
}
```

The output of the sample is as follows:

```
LoRes: 7 system ticks
HiRes: 6641 microsecond
```

17.2 Low-resolution measurement

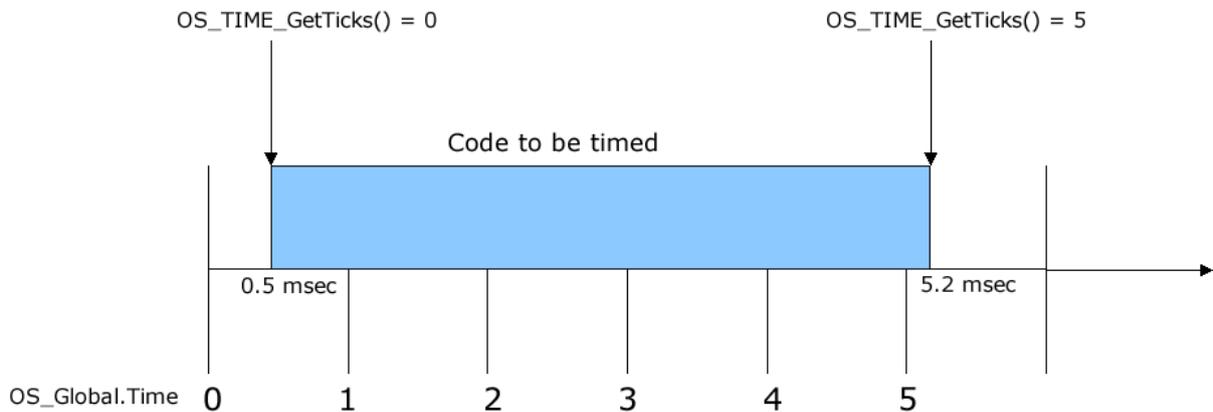
The global system time variable `OS_Global.Time` is measured in system ticks, which typically equal milliseconds. The low-resolution functions `OS_TIME_GetTicks()` and `OS_TIME_GetTicks32()` are used for returning the current contents of this variable. The basic concept behind low-resolution measurement is quite simple: The system time is returned once before the section of code to be timed and once after, and the first value is subtracted from the second to obtain the time it took for the code to execute.

The term low-resolution is used because the time values returned are measured in completed system ticks. Consider the following: The global variable `OS_Global.Time` is incremented with every system tick interrupt, with a default tick of one millisecond that means once each millisecond. This means that the actual system time can potentially be later than the low-resolution function returns (for example, if an interrupt actually occurs at 1.4 system ticks, the system will assume only one tick having elapsed). The problem even gets worse when concerning runtime measurement, because the system time must be measured twice. Since each measurement can, potentially, be up to one tick less than the actual time, the difference between two measurements could theoretically be inaccurate by up to one tick.

The following diagram illustrates how low-resolution measurement works. We can see that the section of code begins at 0.5 millisecond and ends at 5.2 millisecond, which means that its exact execution time is 5.2 millisecond - 0.5 millisecond = 4.7 millisecond. However, assuming one system tick per millisecond, the first call to `OS_TIME_GetTicks()` will return 0, while the second call will return 5. The measured execution time would therefore be returned as 5 system ticks - 0 system ticks = 5 system ticks.

```
void MeasureTask(void) {
    OS_TIME t0, t;

    t0 = OS_TIME_GetTicks32();
    CodeToBeTimed();
    t = OS_TIME_GetTicks32() - t0;
}
```



For many applications, low-resolution measurement is sufficient for most of all cases. In those cases, its ease of use as well as its faster computation time are clear benefits when compared to high-resolution measurement. Still, high-resolution measurement may be necessary when highly accurate measurements are mandatory.

Note

If disabling interrupts or execution of other interrupts prevent the system tick from being served, system ticks can be lost. It is the application's responsibility to ensure that each system tick is handled and no system tick gets lost. In order to prevent the application from losing system ticks, it has to ensure that the system tick handler is executed regularly.

17.2.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TIME_GetTicks()</code>	Returns the current system time in ticks as a native integer value.	•	•	•	•	•
<code>OS_TIME_GetTicks32()</code>	Returns the current system time in system ticks as a 32-bit signed integer value.	•	•	•	•	•
<code>OS_TIME_Convertms2Ticks()</code>	Converts milliseconds into system ticks.	•	•	•	•	•
<code>OS_TIME_ConvertTicks2ms()</code>	Converts system ticks into milliseconds.	•	•	•	•	•

17.2.1.1 OS_TIME_GetTicks()

Description

Returns the current system time in ticks as a native integer value.

Prototype

```
OS_TIME OS_TIME_GetTicks(void);
```

Return value

The system variable `OS_Global.Time` as a 16-bit integer value on 8/16-bit CPUs, and as a 32-bit integer value on 32-bit CPUs.

Additional information

The `OS_Global.Time` variable is a 32-bit integer value. Therefore, if the return value is 32 bit, it holds the entire contents of the `OS_Global.Time` variable. If the return value is 16 bit, it holds the lower 16 bits of the `OS_Global.Time` variable.

Note

You must not rely on `OS_TIME_GetTicks()` as a timestamp since reset. Please use `OS_TIME_Get_us64()` instead.

`OS_Global.Time` overflows from `0xFFFFFFFF` to `0x00000000` after 4294967296 system ticks. Assuming one system tick equals one millisecond this overflow happens after ~49 days. This overflow is no problem for all embOS timing API functions like `OS_TASK_Delay()` since embOS handles all timing calculations correctly. But this has to be taken into account when `OS_TIME_GetTicks()` should be used as an application timestamp.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintTask(void) {
    OS_TIME Time;

    Time = OS_TIME_GetTicks();
    printf("System Time: %d\n", Time);
}
```

17.2.1.2 OS_TIME_GetTicks32()

Description

Returns the current system time in system ticks as a 32-bit signed integer value.

Prototype

```
OS_I32 OS_TIME_GetTicks32(void);
```

Return value

The system variable `OS_Global.Time` as a 32-bit signed integer value.

Additional information

This function always returns the system time as a 32-bit value. Because the `OS_Global.Time` variable is also a 32-bit value, the return value is simply the entire contents of the `OS_Global.Time` variable.

Note

You must not rely on `OS_TIME_GetTicks32()` as a timestamp since reset. Please use `OS_TIME_Get_us64()` instead.

`OS_Global.Time` overflows from `0xFFFFFFFF` to `0x00000000` after 4294967296 system ticks. Assuming one system tick equals one millisecond this overflow happens after ~49 days. This overflow is no problem for all embOS timing API functions like `OS_TASK_Delay()` since embOS handles all timing calculations correctly. But this has to be taken into account when `OS_TIME_GetTicks32()` should be used as an application timestamp.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintTask(void) {
    OS_I32 Time;

    Time = OS_TIME_GetTicks32();
    printf("System Time: %d\n", Time);
}
```

17.2.1.3 OS_TIME_Convertms2Ticks()

Description

Converts milliseconds into system ticks.

Prototype

```
OS_U64 OS_TIME_Convertms2Ticks(OS_U32 ms);
```

Parameters

Parameter	Description
ms	Milliseconds.

Return value

The converted value in system ticks.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

OS_TIME_Convertms2Ticks() returns correct values only if OS_TIME_ConfigSysTimer() was called during initialization. All embOS board support packages already call OS_TIME_ConfigSysTimer().

OS_TIME_Convertms2Ticks() must be used with tick to interrupt ratio 1:1 only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

237: OS_ERR_FRACTIONAL_TICK

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Convert(void) {
    OS_U32 Ticks;

    Ticks = OS_TIME_Convertms2Ticks(100);
}
```

17.2.1.4 OS_TIME_ConvertTicks2ms()

Description

Converts system ticks into milliseconds.

Prototype

```
OS_U64 OS_TIME_ConvertTicks2ms(OS_U32 t);
```

Parameters

Parameter	Description
t	System ticks.

Return value

The converted value in milliseconds.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

OS_TIME_ConvertTicks2ms() returns correct values only if OS_TIME_ConfigSysTimer() was called during initialization. All embOS board support packages already call OS_TIME_ConfigSysTimer().

OS_TIME_ConvertTicks2ms() must be used with tick to interrupt ratio 1:1 only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

237: OS_ERR_FRACTIONAL_TICK

238: OS_ERR_ZERO_TIMER_INT_FREQ

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Convert(void) {
    OS_U32 milliseconds;

    milliseconds = OS_TIME_ConvertTicks2ms(100);
}
```

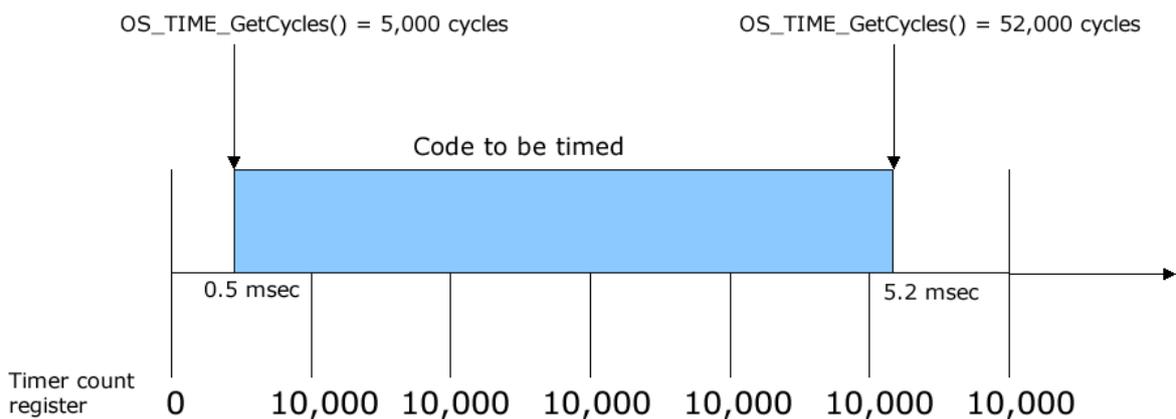
17.3 High-resolution measurement

Instead of measuring the number of completed system ticks at a given time, the hardware timer counter value is also taken into account. The hardware timer runs at a specific frequency which could be equal to the CPU frequency or a fraction of it. A timer cycle is the time for increment or decrement the timer count register at the timer frequency by one. Hence, the actual time resolution depends on the timer frequency. For example with a CPU and a hardware timer running at 100 MHz the cycle length is 10 nanoseconds. This is the minimal time which could be measured and therefore the measurement resolution.

Please refer to the illustration below, which measures the execution time of the same code that was used during the low-resolution calculation. For this example, we assume that the CPU has a timer running at 10 MHz and counts upwards. The number of cycles per system tick interrupt therefore equals $(10 \text{ MHz} / 1 \text{ kHz}) = 10,000$. This means that with each system tick interrupt, the timer restarts at zero and counts up to 10,000.

```
void MeasureTask(void) {
    OS_U64 t0, t;

    t0 = OS_TIME_Get_Cycles();
    CodeToBeTimed();
    t = OS_TIME_Get_Cycles() - t0;
}
```



The first call to `OS_TIME_Get_Cycles()` returns the starting value at 5,000 cycles, while the second call to `OS_TIME_Get_Cycles()` returns the ending value at 52,000 cycles. The measured execution time of the code in this example would therefore be $(52,000 \text{ cycles} - 5,000 \text{ cycles}) = 47,000 \text{ cycles}$, which exactly equals 4.7 millisecond.

Note

If disabling interrupts or execution of other interrupts prevent the system tick from being served, system ticks can be lost. It is the application's responsibility to ensure that each system tick is handled and no system tick gets lost. In order to prevent the application from losing system ticks, it has to ensure that the system tick handler is executed regularly.

17.3.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TIME_ConfigSysTimer()</code>	Configures the system time parameters for according timing functions, embOSView and profiling.	•				
<code>OS_TIME_ConvertCycles2ms()</code>	Converts timer cycles into milliseconds.	•	•	•	•	•
<code>OS_TIME_ConvertCycles2ns()</code>	Converts timer cycles into nanoseconds.	•	•	•	•	•
<code>OS_TIME_ConvertCycles2us()</code>	Converts timer cycles into microseconds.	•	•	•	•	•
<code>OS_TIME_Convertms2Cycles()</code>	Converts milliseconds into timer cycles.	•	•	•	•	•
<code>OS_TIME_Convertns2Cycles()</code>	Converts nanoseconds into timer cycles.	•	•	•	•	•
<code>OS_TIME_Convertus2Cycles()</code>	Converts microseconds into timer cycles.	•	•	•	•	•
<code>OS_TIME_Get_Cycles()</code>	Returns the current system time in timer cycles as a 64-bit value.	•	•	•	•	•
<code>OS_TIME_GetInts()</code>	Returns the number of system tick interrupts.	•	•	•	•	•
<code>OS_TIME_GetResult()</code>	Returns the execution time of the code between <code>OS_TIME_StartMeasurement()</code> and <code>OS_TIME_StopMeasurement()</code> in cycles.	•	•	•	•	•
<code>OS_TIME_GetResult_us()</code>	Returns the execution time of the code between <code>OS_TIME_StartMeasurement()</code> and <code>OS_TIME_StopMeasurement()</code> in microseconds.	•	•	•	•	•
<code>OS_TIME_Get_us()</code>	Returns the current system time in microseconds as a 32-bit value.	•	•	•	•	•
<code>OS_TIME_Get_us64()</code>	Returns the current system time in microseconds as a 64-bit value.	•	•	•	•	•
<code>OS_TIME_StartMeasurement()</code>	Marks the beginning of a code section to be timed.	•	•	•	•	•
<code>OS_TIME_StopMeasurement()</code>	Marks the end of a code section to be timed.	•	•	•	•	•

17.3.1.1 OS_TIME_ConfigSysTimer()

Description

Configures the system time parameters for according timing functions, embOSView and profiling.

Prototype

```
void OS_TIME_ConfigSysTimer(OS_CONST_PTR OS_SYSTIMER_CONFIG *pConfig);
```

Parameters

Parameter	Description
<code>pConfig</code>	Pointer to a data structure of type <code>OS_SYSTIMER_CONFIG</code> .

Additional information

This function is usually called once from `OS_InitHW()` (implemented in `RTOSInit.c`).

The OS_SYSTIMER_CONFIG struct

`OS_TIME_ConfigSysTimer()` uses the struct `OS_SYSTIMER_CONFIG`:

Member	Description
<code>TimerFreq</code>	System tick hardware timer frequency in Hz
<code>IntFreq</code>	System tick hardware timer interrupt frequency in Hz
<code>IsUpCounter</code>	<code>OS_TIMER_DOWNCOUNTING (0)</code> : for hardware timer which counts down <code>OS_TIMER_UPCOUNTING (1)</code> : for hardware timer which counts up
<code>pfGetTimerCycles</code>	Pointer to a function which returns the current hardware timer count value
<code>pfGetTimerIntPending</code>	Pointer to a function which indicates whether the hardware timer interrupt pending flag is set

pfGetTimerCycles()

Description

This callback function must be implemented by the user. It returns the current hardware timer count value.

Prototype

```
unsigned int (*pfGetTimerCycles)(void);
```

Return value

The current hardware timer count value.

pfGetTimerIntPending()

Description

This callback function must be implemented by the user. It returns a value unequal to zero if the hardware timer interrupt pending flag is set.

Prototype

```
unsigned int (*pfGetTimerIntPending)(void);
```

Return value

= 0 Hardware timer interrupt pending flag is not set.
≠ 0 The pending flag is set.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
167: OS_ERR_CPU_STATE_ILLEGAL  
228: OS_ERR_ILLEGAL_AFTER_OSSTART
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define OS_FSYS          72000000u      // 72 MHz CPU main clock  
#define OS_PCLK_TIMER   (OS_FSYS)      // HW timer runs at CPU speed  
#define OS_TICK_FREQ    1000u         // 1 KHz => 1 msc per system tick  
#define OS_INT_FREQ     (OS_TICK_FREQ) // 1 system tick per interrupt  
  
static unsigned int _OS_GetHWTimer_Cycles(void) {  
    return HW_TIMER_VALUE_REG;  
}  
  
static unsigned int _OS_GetHWTimer_IntPending(void) {  
    return HW_TIMER_INT_REG & (1uL << PENDING_BIT);  
}  
  
const OS_SYSTIMER_CONFIG Tick_Config = { OS_PCLK_TIMER,  
                                          OS_INT_FREQ,  
                                          OS_TIMER_DOWNCOUNTING,  
                                          _OS_GetHWTimer_Cycles  
                                          _OS_GetHWTimer_IntPending };  
  
void OS_InithW(void) {  
    OS_TIME_ConfigSysTimer(&Tick_Config);  
    ...  
}
```

17.3.1.2 OS_TIME_ConvertCycles2ms()

Description

Converts timer cycles into milliseconds.

Prototype

```
OS_U64 OS_TIME_ConvertCycles2ms(OS_U32 Cycles);
```

Parameters

Parameter	Description
Cycles	Timer cycles.

Return value

The converted value in milliseconds.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

OS_TIME_ConvertCycles2ms() returns correct values only if OS_TIME_ConfigSysTimer() was called during initialization. All embOS board support packages already call OS_TIME_ConfigSysTimer().

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

235: OS_ERR_NON_TIMERCYCLES_FUNC

236: OS_ERR_NON_TIMERINTPENDING_FUNC

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Convert(void) {
    OS_U64 ms;

    ms = OS_TIME_ConvertCycles2ms(2000);
}
```

17.3.1.3 OS_TIME_ConvertCycles2ns()

Description

Converts timer cycles into nanoseconds.

Prototype

```
OS_U64 OS_TIME_ConvertCycles2ns(OS_U32 Cycles);
```

Parameters

Parameter	Description
Cycles	Timer cycles.

Return value

The converted value in nanoseconds.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

OS_TIME_ConvertCycles2ns() returns correct values only if OS_TIME_ConfigSysTimer() was called during initialization. All embOS board support packages already call OS_TIME_ConfigSysTimer().

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

235: OS_ERR_NON_TIMERCYCLES_FUNC

236: OS_ERR_NON_TIMERINTPENDING_FUNC

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Convert(void) {
    OS_U64 ns;

    ns = OS_TIME_ConvertCycles2ns(2000);
}
```

17.3.1.4 OS_TIME_ConvertCycles2us()

Description

Converts timer cycles into microseconds.

Prototype

```
OS_U64 OS_TIME_ConvertCycles2us(OS_U32 Cycles);
```

Parameters

Parameter	Description
<code>Cycles</code>	Timer cycles.

Return value

The converted value in microseconds.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

`OS_TIME_ConvertCycles2us()` returns correct values only if `OS_TIME_ConfigSysTimer()` was called during initialization. All embOS board support packages already call `OS_TIME_ConfigSysTimer()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

235: OS_ERR_NON_TIMERCYCLES_FUNC

236: OS_ERR_NON_TIMERINTPENDING_FUNC

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Convert(void) {
    OS_U64 us;

    us = OS_TIME_ConvertCycles2us(2000);
}
```

17.3.1.5 OS_TIME_Convertms2Cycles()

Description

Converts milliseconds into timer cycles.

Prototype

```
OS_U64 OS_TIME_Convertms2Cycles(OS_U32 ms);
```

Parameters

Parameter	Description
<code>ms</code>	Milliseconds.

Return value

The converted value in timer cycles.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

`OS_TIME_Convertms2Cycles()` returns correct values only if `OS_TIME_ConfigSysTimer()` was called during initialization. All embOS board support packages already call `OS_TIME_ConfigSysTimer()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

235: OS_ERR_NON_TIMERCYCLES_FUNC

236: OS_ERR_NON_TIMERINTPENDING_FUNC

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Convert(void) {
    OS_U64 Cycles;

    Cycles = OS_TIME_Convertms2Cycles(100);
}
```

17.3.1.6 OS_TIME_Convertns2Cycles()

Description

Converts nanoseconds into timer cycles.

Prototype

```
OS_U64 OS_TIME_Convertns2Cycles(OS_U32 ns);
```

Parameters

Parameter	Description
<code>ns</code>	Nanoseconds.

Return value

The converted value in timer cycles.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

`OS_TIME_Convertns2Cycles()` returns correct values only if `OS_TIME_ConfigSysTimer()` was called during initialization. All embOS board support packages already call `OS_TIME_ConfigSysTimer()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

235: OS_ERR_NON_TIMERCYCLES_FUNC

236: OS_ERR_NON_TIMERINTPENDING_FUNC

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Convert(void) {
    OS_U64 Cycles;

    Cycles = OS_TIME_Convertns2Cycles(100);
}
```

17.3.1.7 OS_TIME_Convertus2Cycles()

Description

Converts microseconds into timer cycles.

Prototype

```
OS_U64 OS_TIME_Convertus2Cycles(OS_U32 us);
```

Parameters

Parameter	Description
<code>us</code>	Microseconds.

Return value

The converted value in timer cycles.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

`OS_TIME_Convertus2Cycles()` returns correct values only if `OS_TIME_ConfigSysTimer()` was called during initialization. All embOS board support packages already call `OS_TIME_ConfigSysTimer()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

235: OS_ERR_NON_TIMERCYCLES_FUNC

236: OS_ERR_NON_TIMERINTPENDING_FUNC

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Convert(void) {
    OS_U64 Cycles;

    Cycles = OS_TIME_Convertus2Cycles(100);
}
```

17.3.1.8 OS_TIME_Get_Cycles()

Description

Returns the current system time in timer cycles as a 64-bit value.

Prototype

```
OS_U64 OS_TIME_Get_Cycles(void);
```

Return value

The current system time in timer cycles as a 64-bit integer value.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

`OS_TIME_Get_Cycles()` returns correct values only if `OS_TIME_ConfigSysTimer()` was called during initialization. All embOS board support packages already call `OS_TIME_ConfigSysTimer()`. With this 64-bit value `OS_TIME_Get_Cycles()` can return up to $2^{64} - 1$ cycles = `0xFFFFFFFFFFFFFFFF` cycles (~18 quintillion cycles).

`OS_TIME_Get_Cycles()` presumes that the execution of the system tick handler and `OS_TICK_Handle()`, `OS_TICK_HandleEx()` or `OS_TICK_HandleNoHook()` are atomic. This is, for example, not true when `OS_TIME_Get_Cycles()` is called in the system tick handler routine before `OS_TICK_Handle()`, `OS_TICK_HandleEx()` or `OS_TICK_HandleNoHook()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Benchmark(void) {
    OS_U64 Cycles0, Cycles;

    Cycles0 = OS_TIME_Get_Cycles();
    DoSomething();
    Cycles = OS_TIME_Get_Cycles() - Cycles0;
}
```

17.3.1.9 OS_TIME_GetInts()

Description

Returns the number of system tick interrupts.

Prototype

```
OS_U64 OS_TIME_GetInts(void);
```

Return value

Number of system tick interrupts.

Additional information

In most cases the number of system tick interrupts is equal to the number of system ticks. But it may be different if `OS_TICK_Config()` was used before. This function should typically be used in board support packages only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void GetInts(void) {
    OS_U64 NumInts;

    NumInts = OS_TIME_GetInts();
}
```

17.3.1.10 OS_TIME_GetResult()

Description

Returns the execution time of the code between `OS_TIME_StartMeasurement()` and `OS_TIME_StopMeasurement()` in cycles.

Prototype

```
OS_U32 OS_TIME_GetResult(OS_CONST_PTR OS_U32 *pCycle);
```

Parameters

Parameter	Description
<code>pCycle</code>	Pointer to a <code>OS_U32</code> variable which will contain the result.

Return value

The execution time in timer cycles as a 32-bit integer value.

Additional information

Must be used with `OS_TIME_StartMeasurement()` and `OS_TIME_StopMeasurement()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void Benchmark(void) {
    OS_U32 t;
    OS_U32 r;

    OS_TIME_StartMeasurement(&t);
    UserCode();           // Execute the user code to be benchmarked
    OS_TIME_StopMeasurement(&t);
    r = OS_TIME_GetResult(&t);
    printf("Cycles=%u", r);
}
```

17.3.1.11 OS_TIME_GetResult_us()

Description

Returns the execution time of the code between `OS_TIME_StartMeasurement()` and `OS_TIME_StopMeasurement()` in microseconds.

Prototype

```
OS_U32 OS_TIME_GetResult_us(OS_CONST_PTR OS_U32 *pCycle);
```

Parameters

Parameter	Description
<code>pCycle</code>	Pointer to a <code>OS_U32</code> variable which will contain the result.

Return value

The execution time in microseconds as a 32-bit integer value.

Additional information

Must be used with `OS_TIME_StartMeasurement()` and `OS_TIME_StopMeasurement()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`164: OS_ERR_OS_INT_ENTER_NOT_CALLED`

`167: OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void Benchmark(void) {
    OS_U32 t;
    OS_U32 us;

    OS_TIME_StartMeasurement(&t);
    UserCode();           // Execute the user code to be benchmarked
    OS_TIME_StopMeasurement(&t);
    us = OS_TIME_GetResult_us(&t);
    printf("Time=%u", us);
}
```

17.3.1.12 OS_TIME_Get_us()

Description

Returns the current system time in microseconds as a 32-bit value.

Prototype

```
OS_U32 OS_TIME_Get_us(void);
```

Return value

The current system time in microseconds as a 32-bit integer value.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

`OS_TIME_Get_us()` returns correct values only if `OS_TIME_ConfigSysTimer()` was called during initialization. All embOS board support packages already call `OS_TIME_ConfigSysTimer()`. With this 32-bit value `OS_TIME_Get_us()` can return up to $2^{32} - 1$ microseconds = 0xFFFFFFFF microseconds (~71 minutes).

`OS_TIME_Get_us()` presumes that the execution of the system tick handler and `OS_TICK_Handle()`, `OS_TICK_HandleEx()` or `OS_TICK_HandleNoHook()` are atomic. This is, for example, not true when `OS_TIME_Get_us()` is called in the system tick handler routine before `OS_TICK_Handle()`, `OS_TICK_HandleEx()` or `OS_TICK_HandleNoHook()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintTime(void) {
    OS_U32 Time;

    Time = OS_TIME_Get_us();
    printf("System Time: %u microseconds\n", Time);
}
```

17.3.1.13 OS_TIME_Get_us64()

Description

Returns the current system time in microseconds as a 64-bit value.

Prototype

```
OS_U64 OS_TIME_Get_us64(void);
```

Return value

The current system time in microseconds as a 64-bit integer value.

Additional information

This function is unavailable for compilers that do not support a 64-bit data type (long long). This is the case only for very rare older compiler.

OS_TIME_Get_us64() returns correct values only if OS_TIME_ConfigSysTimer() was called during initialization. All embOS board support packages already call OS_TIME_ConfigSysTimer(). With this 64-bit value OS_TIME_Get_us64() can return up to $2^{64} - 1$ microseconds = 0xFFFFFFFFFFFFFFFF microseconds (~584,942 years).

OS_TIME_Get_us64() presumes that the execution of the system tick handler and OS_TICK_Handle(), OS_TICK_HandleEx() or OS_TICK_HandleNoHook() are atomic. This is, for example, not true when OS_TIME_Get_us64() is called in the system tick handler routine before OS_TICK_Handle(), OS_TICK_HandleEx() or OS_TICK_HandleNoHook().

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void MeasureTime(void) {
    OS_U64 t0, t1;
    OS_U32 delta;

    t0 = OS_TIME_Get_us64();
    DoSomething();
    t1 = OS_TIME_Get_us64();
    delta = (OS_U32)(t1 - t0);
    printf("Delta: %u microseconds\n", delta);
}
```

17.3.1.14 OS_TIME_StartMeasurement()

Description

Marks the beginning of a code section to be timed.

Prototype

```
void OS_TIME_StartMeasurement(OS_U32* pCycle);
```

Parameters

Parameter	Description
<code>pCycle</code>	Pointer to a <code>OS_U32</code> variable which will contain the result.

Additional information

This function must be used with `OS_TIME_StopMeasurement()`.

`OS_TIME_StartMeasurement()` presumes that the execution of the system tick handler and `OS_TICK_Handle()`, `OS_TICK_HandleEx()` or `OS_TICK_HandleNoHook()` are atomic. This is, for example, not true when `OS_TIME_StartMeasurement()` is called in the system tick handler routine before `OS_TICK_Handle()`, `OS_TICK_HandleEx()` or `OS_TICK_HandleNoHook()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
235: OS_ERR_NON_TIMERCYCLES_FUNC
236: OS_ERR_NON_TIMERINTPENDING_FUNC
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void Benchmark(void) {
    OS_U32 t;
    OS_U32 us;

    OS_TIME_StartMeasurement(&t);
    UserCode();           // Execute the user code to be benchmarked
    OS_TIME_StopMeasurement(&t);
    us = OS_TIME_GetResult_us(&t);
    printf("Time=%u", us);
}
```

17.3.1.15 OS_TIME_StopMeasurement()

Description

Marks the end of a code section to be timed.

Prototype

```
void OS_TIME_StopMeasurement(OS_U32* pCycle);
```

Parameters

Parameter	Description
<code>pCycle</code>	Pointer to a <code>OS_U32</code> variable which will contain the result.

Additional information

This function must be used with `OS_TIME_StartMeasurement()`.

`OS_TIME_StopMeasurement()` presumes that the execution of the system tick handler and `OS_TICK_Handle()`, `OS_TICK_HandleEx()` or `OS_TICK_HandleNoHook()` are atomic. This is, for example, not true when `OS_TIME_StopMeasurement()` is called in the system tick handler routine before `OS_TICK_Handle()`, `OS_TICK_HandleEx()` or `OS_TICK_HandleNoHook()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static void Benchmark(void) {
    OS_U32 t;
    OS_U32 us;

    OS_TIME_StartMeasurement(&t);
    UserCode();           // Execute the user code to be benchmarked
    OS_TIME_StopMeasurement(&t);
    us = OS_TIME_GetResult_us(&t);
    printf("Time=%u", us);
}
```

Chapter 18

Low Power Support

18.1 Introduction

embOS provides several means to control the power consumption of your target hardware. These include:

- The possibility to enter power save modes with the embOS function `OS_Idle()`.
- The embOS tickless support, allowing the microcontroller to remain in a power save mode for extended periods of time.
- The embOS peripheral power control module, which allows control of the power consumption of specific peripherals.

The following chapter explains each of these in more detail.

18.2 Starting power save modes in `OS_Idle()`

In case your controller supports some kind of power save mode, it is possible to use it with embOS. To enter that mode, you would usually implement the respective functionality in the function `OS_Idle()`, which is located inside the embOS source file `RTOSInit.c`.

`OS_Idle()` is executed whenever no task is ready for execution. With many embOS start projects it is preconfigured to activate a power save mode of the target CPU. Please note that the available power save modes are hardware-dependent. For example with Cortex-M CPUs, the `wfi` instruction is executed per default in `OS_Idle()` to put the CPU into a power save mode:

```
void OS_Idle(void) { // Idle loop: No task is ready to execute
    while (1) {
        __asm("wfi"); // Enter sleep mode
    }
}
```

For further information on `OS_Idle()`, please also refer to `OS_Idle()` on page 578.

18.3 Tickless support

18.3.1 Introduction

The embOS tickless support provides the needed functionality for users to stop the periodic system tick interrupt during idle periods and to adjust the embOS ticks again after the idle period has ended. Idle periods are periods of time when there are no tasks or software timers ready for execution. Stopping the system tick interrupt during idle periods allows the microcontroller to remain in a low power mode until a task or software timer needs to be executed again. It is the user's responsibility to ensure that the hardware timer(s) are (re-)configured properly and the system time is adjusted when the tickless mode is stopped and the normal operation is resumed.

The embOS tickless support comes with the functions `OS_TICKLESS_GetNumIdleTicks()`, `OS_TICKLESS_AdjustTime()`, `OS_TICKLESS_Start()`, `OS_TICKLESS_Stop()`, `OS_TICKLESS_GetPeriod()` and `OS_TICKLESS_IsExpired()`. These can be used to add tickless support to any embOS project.

Basic concept

Every time when there is no scheduled action, embOS calls the function `OS_Idle()`. The application will remain in `OS_Idle()` until an interrupt occurs. If an application arrives in `OS_Idle()`, it can therefore enter low power mode. `OS_TICKLESS_GetNumIdleTicks()` returns the amount of system ticks the application can stay in low power mode until the scheduler has to be executed again in order to schedule a task or execute a software timer.

Before entering low power mode, the application has to reconfigure the hardware timer to generate an interrupt at the time indicated by `OS_TICKLESS_GetNumIdleTicks()`. Furthermore, it needs to tell embOS how long the application will stay in tickless mode as well as which callback function embOS should call once the tickless mode ends. The time and the callback function are passed to embOS via `OS_TICKLESS_Start()`. Afterwards, the device can enter low power mode.

Now, if the hardware timer interrupt that denotes the end of the tickless period occurs, the interrupt handler simply has to call `OS_TICK_Handle()`, which activates the scheduler. The embOS scheduler then executes the callback function.

The user has to implement this callback function to adjust the system time by the time the device spent in tickless mode. It also needs to reconfigure the hardware timer to generate periodical system tick interrupts again. The amount of time that was passed to `OS_TICKLESS_Start()` can be retrieved via `OS_TICKLESS_GetPeriod()` and can be directly passed to `OS_TICKLESS_AdjustTime()`. From that point on, embOS resumes normal operation.

There might be situations in which a device may want to end the tickless mode earlier than expected. For instance, if an interrupt other than the timer interrupt occurs and makes a task ready for execution, the tickless mode can be canceled prematurely. To do so, the interrupt service routine has to call `OS_TICKLESS_Stop()`, which will execute the callback function directly. The callback can distinguish whether the tickless mode was stopped earlier than expected by using `OS_TICKLESS_IsExpired()`. If it returns zero, the callback cannot use `OS_TICKLESS_GetPeriod()` but has to read the timer counter and calculate how many ticks the device has spent in tickless mode. The calculated value can then be used to adjust the system tick accordingly.

Hardware timers

The embOS tickless support can be used with either one or two timers.

Using one hardware timer only, the timer would need to be reconfigured upon entering low power mode and upon leaving low power mode. If it isn't desired to use the same hardware timer during tickless and non-tickless operation (e.g. because it becomes non-functional during low power mode), a second hardware timer may be used. In this case, the non-tickless mode timer has to be disabled during each preparation for the tickless mode in `OS_Idle()`, and the tickless mode timer needs to be configured to generate an interrupt at the appropriate point in time. Once the tickless mode timer expires, its interrupt han-

der should then simply call `OS_TICK_Handle()` as if it was the non-tickless mode timer. Subsequently, the callback function should disable the tickless mode timer and activate the non-tickless mode timer again.

OS_Idle()

Usually, `OS_Idle()` just consists of an endless loop that optionally suspends the CPUs execution by entering a low power mode. The code for entering low power mode is contained in the loop to ensure that if an event like an interrupt wakes up the device, it will simply re-enter low power mode again. The implementation of such an `OS_Idle()` function may look like this:

```
void OS_Idle(void) {
    while (1) {
        _EnterLowPowerMode();
    }
}
```

Here, `_EnterLowPowerMode()` is a pseudo function which puts the device into low power mode.

Using such `OS_Idle()` implementation results in the device exiting and entering low power mode with each system tick interrupt, even if there is no task or software timer ready for execution. To avoid this, the embOS' tickless support may be used.

An implementation of `OS_Idle()` that uses the tickless support may look like this:

```
void OS_Idle(void) {
    OS_TIME IdleTicks;

    OS_INT_IncDI();
    IdleTicks = OS_TICKLESS_GetNumIdleTicks();
    if (IdleTicks > 1) {
        if (IdleTicks > TIMER1_MAX_TICKS) {
            IdleTicks = TIMER1_MAX_TICKS;
        }
        OS_TICKLESS_Start(IdleTicks, &_EndTicklessMode);
        _SetHWTimer(IdleTicks);
    }
    OS_INT_DecRI();
    while (1) {
        _EnterLowPowerMode();
    }
}
```

The following description explains the tickless `OS_Idle()` function step by step:

```
void OS_Idle(void) {
    OS_TIME IdleTicks;

    OS_INT_Disable();
```

Interrupts are disabled to avoid a timer interrupt.

```
IdleTicks = OS_TICKLESS_GetNumIdleTicks();
if (IdleTicks > 1) {
```

The `OS_Idle()` function evaluates the number system ticks that can be spent in tickless mode by calling `OS_TICKLESS_GetNumIdleTicks()`. Entering low mode for a small number of ticks only (e.g. 1 tick) may also be omitted when this would not conserve sufficient amounts of energy.

```
if ((OS_U32)IdleTicks > TIMER_MAX_TICKS) {
    IdleTicks = TIMER_MAX_TICKS;
```

```
}

```

If it is not possible to generate the timer interrupt at the specified time, e.g. due to hardware timer limitations, the idle period can be reduced to any lower value. For example, if `OS_TICKLESS_GetNumIdleTicks()` returns an idle period of 200 ticks, but the hardware timer's duration is limited to 100 ticks maximum, then the variable `IdleTicks` will initially be set to 100 system ticks. The system will then wake up after 100 system ticks, `OS_Idle()` will be executed once more and `OS_TICKLESS_GetNumIdleTicks()` now returns the remaining 100 idle systems ticks, for which tickless mode is entered once again. This means that the system wakes up two times for the entire 200 idle system ticks.

```
if (IdleTicks > 1) {
    ...
    OS_TICKLESS_Start(IdleTicks, &_EndTicklessMode);
    _SetHWTimer(IdleTicks);
}

```

`OS_TICKLESS_Start()` sets the idle system ticks and the callback function. `IdleTicks` is later used in the callback function, which is described in more detail below. `_SetHWTimer()` is a hardware-dependent function that must be implemented to reconfigure the hardware timer to generate a system tick interrupt at the time defined by `IdleTicks`. It is possible to reconfigure the current hardware timer or to stop it and to use another hardware timer in tickless mode. For simplicity, we assume the same hardware timer is used both in tickless mode and non-tickless mode.

```
OS_INT_Enable();
while (1) {
    _EnterLowPowerMode();
}
}

```

Interrupts are re-enabled and the CPU continually enters power save mode. `_EnterLowPowerMode()` is a hardware-dependent function that activates the low power mode.

Callback Function

The callback function calculates how long the processor spent in tickless mode and corrects the system time accordingly. embOS API functions that are allowed to be called within the callback function are limited to `OS_TICKLESS_AdjustTime()`, `OS_TICKLESS_GetPeriod()` and `OS_TICKLESS_IsExpired()`.

```
static void _EndTicklessMode(void) {
    OS_U32 NumTicks;

    if (OS_TICKLESS_IsExpired() != 0) {
        OS_TICKLESS_AdjustTime(OS_TICKLESS_GetPeriod());
    } else {
        NumTicks = _GetLowPowerTicks();
        OS_TICKLESS_AdjustTime(NumTicks);
    }
    _SetHWTimer(OS_TIMER_RELOAD);
}

```

The following description explains the callback function step by step:

```
static void _EndTicklessMode(void) {
    OS_U32 NumTicks;

    if (OS_TICKLESS_IsExpired() != 0) {
        OS_TICKLESS_AdjustTime(OS_TICKLESS_GetPeriod());
    }
}
```

`OS_TICKLESS_IsExpired()` returns whether the hardware timer expired and the system tick interrupt was executed. This can be used to determine if the device spent the whole idle period in tickless mode and terminated the tickless mode by a call to `OS_TICK_Handle()`, or if the tickless mode was terminated by an interrupt calling `OS_TICKLESS_Stop()`. If `OS_TICKLESS_IsExpired()` does not return zero, we can simply use the value returned by `OS_TICKLESS_GetPeriod()` to adjust the system time. This value is the period that was previously passed to `OS_TICKLESS_Start()` with the `IdleTicks` variable.

```
    } else {
        NumTicks = _GetLowPowerTicks();
        OS_TICKLESS_AdjustTime(NumTicks);
    }
}
```

If `OS_TICKLESS_IsExpired()` returns zero, the time spent in tickless mode needs to be computed. Here, `_GetLowPowerTicks()` is a hardware-dependent function which returns the amount of system ticks already spent in tickless mode. We use that value to adjust the system time.

```
    _SetHWTimer(OS_TIMER_RELOAD);
}
```

`_SetHWTimer()` is a hardware-dependent function which reconfigures the hardware timer to generate a periodic system tick again.

18.3.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer	Idle
<code>OS_TICKLESS_AdjustTime()</code>	Adjusts the embOS internal time variable by the specified amount of system ticks.	•	•		•	•	
<code>OS_TICKLESS_GetNumIdleTicks()</code>	Retrieves the number of embOS system ticks until the next time-scheduled action will be started.						•
<code>OS_TICKLESS_GetPeriod()</code>	Returns the configured tickless period in system ticks that was passed to <code>OS_TICKLESS_Start()</code> .		•	•	•	•	
<code>OS_TICKLESS_IsExpired()</code>	Returns whether the tickless period has expired.		•	•	•	•	
<code>OS_TICKLESS_Start()</code>	Start the tickless mode.						•
<code>OS_TICKLESS_Stop()</code>	Prematurely stops the tickless mode.				•		

18.3.2.1 OS_TICKLESS_AdjustTime()

Description

Adjusts the embOS internal time variable by the specified amount of system ticks.

Prototype

```
void OS_TICKLESS_AdjustTime(OS_TIME Time);
```

Parameters

Parameter	Description
Time	The amount of time which should be added to the embOS internal time variable.

Additional information

The function may be useful when the embOS system timer was halted for any interval of time with a well-known duration. When the embOS timer is subsequently re-started, the internal time variable must be adjusted to that duration in order to guarantee time-scheduled actions are performed accurately.

This function must not be used when the tick-to-interrupt-ratio is not x:1, which means it can be used for ratios where x system ticks equal one timer interrupt. A debug build of embOS will call `OS_Error()` in case the ratio is not x:1. The hardware timer may need to be programmed accordingly to take into account that not the complete time until the next timer interrupt has expired.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example described in `OS_Idle()` on page 578.

18.3.2.2 OS_TICKLESS_GetNumIdleTicks()

Description

Retrieves the number of embOS system ticks until the next time-scheduled action will be started.

Prototype

```
OS_TIME OS_TICKLESS_GetNumIdleTicks(void);
```

Return value

- > 0 Number of system ticks until next time scheduled action.
- = 0 A time scheduled action is pending.

Additional information

The function may be useful when the embOS timer and CPU shall be halted by the application and restarted after the idle time to save power. This works when the application has its own time base and a special interrupt that can wake up the CPU.

When the embOS timer is started again the internal time must be adjusted to guarantee time-scheduled actions to be executed. This can be done by a call of `OS_TICKLESS_AdjustTime()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
167: OS_ERR_CPU_STATE_ILLEGAL  
226: OS_ERR_ILLEGAL_IN_MAIN  
227: OS_ERR_ILLEGAL_IN_TASK
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example described in `OS_Idle()` on page 578.

18.3.2.3 OS_TICKLESS_GetPeriod()

Description

Returns the configured tickless period in system ticks that was passed to `OS_TICKLESS_Start()`. This value can be used to adjust the system tick if `OS_TICKLESS_IsExpired()` does not return zero.

Prototype

```
OS_TIME OS_TICKLESS_GetPeriod(void);
```

Return value

Configured tickless period in system ticks.

Additional information

This is the period the application should stay in tickless mode until it ends and which was passed to `OS_TICKLESS_Start()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void TimerISR(void) {
    OS_BOOL r;
    OS_TIME t;

    OS_INT_Enter();
    r = OS_TICKLESS_IsExpired();
    if (r != 0) {
        t = OS_TICKLESS_GetPeriod();
        OS_TICKLESS_AdjustTime(t);
    }
    OS_INT_Leave();
}
```

18.3.2.4 OS_TICKLESS_IsExpired()

Description

Returns whether the tickless period has expired.

Prototype

```
OS_BOOL OS_TICKLESS_IsExpired(void);
```

Return value

= 0 Tickless period has not expired.
≠ 0 Tickless period has expired.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void TimerISR(void) {  
    OS_BOOL r;  
    OS_TIME t;  
  
    OS_INT_Enter();  
    r = OS_TICKLESS_IsExpired();  
    if (r != 0) {  
        t = OS_TICKLESS_GetPeriod();  
        OS_TICKLESS_AdjustTime(t);  
    }  
    OS_INT_Leave();  
}
```

18.3.2.5 OS_TICKLESS_Start()

Description

Start the tickless mode.

Prototype

```
void OS_TICKLESS_Start(OS_TIME          Period,
                      OS_ROUTINE_VOID* pfEndTicklessMode);
```

Parameters

Parameter	Description
<code>Period</code>	Time in ticks which will be spent in power save mode.
<code>pfEndTicklessMode</code>	Pointer to the routine of type <code>OS_ROUTINE_VOID</code> which shall be called when the tickless mode has ended.

Additional information

`OS_TICKLESS_Start()` must be called before the CPU enters a low power mode. It sets the sleep time and the user callback routine which is called from the scheduler after wake-up from power save mode.

The callback routine must stop the tickless mode. It must calculate how many system ticks are actually spent in lower power mode and adjust the system time by calling `OS_TICKLESS_AdjustTime()`. It also must re-program and/or start the system tick timer to its default tick period.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example described in `OS_Idle()` on page 578.

18.3.2.6 OS_TICKLESS_Stop()

Description

Prematurely stops the tickless mode.

Prototype

```
void OS_TICKLESS_Stop(void);
```

Additional information

The tickless mode is stopped immediately even when no time-scheduled action is due. `OS_TICKLESS_Stop()` calls the callback function registered when tickless mode was enabled. `OS_TICKLESS_Stop()` is mainly used when the tickless mode is not stopped by the system tick but by another event which causes an interrupt.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
161: OS_ERR_ILLEGAL_IN_TIMER  
167: OS_ERR_CPU_STATE_ILLEGAL  
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void ButtonISR(void) {  
    OS_INT_Enter();  
    OS_TICKLESS_Stop();  
    OS_TASKEVENT_Set(&TCB0, EVENT_KEYPRESSED);  
    OS_INT_Leave();  
}
```

18.3.3 Frequently Asked Questions

Q: Can I use embOS without tickless support?

A: you can use embOS without tickless support. No changes to your project are required.

Q: What hardware-dependent functions must be implemented and where?

A: `OS_Idle()` must be modified and the callback function must be implemented. `OS_Idle()` is part of the `RTOSInit.c` file. We suggest to implement the callback function in the same file. The callback function is not called by the user directly, but is instead passed as an argument to the `OS_TICKLESS_Start()` API function.

Q: What triggers the callback function?

A: The callback function is executed once from the scheduler when the tickless operation ends and normal operation resumes. Alternatively, the callback may also be executed by calling `OS_TICKLESS_Stop()` from any other interrupt than the system tick interrupt, and executes the callback function immediately.

18.4 Peripheral power control

18.4.1 Introduction

The embOS peripheral power control is used to determine if a peripheral's clock or its power supply can be switched off to save power.

It includes three functions: `OS_POWER_GetMask()`, `OS_POWER_UsageInc()` and `OS_POWER_UsageDec()`. These functions can be used to add peripheral power control to any embOS start project.

If a peripheral gets initialized a call to `OS_POWER_UsageInc()` increments a specific entry in the power management counter to signal that it is in use. When a peripheral is no longer in use, a call to `OS_POWER_UsageDec()` decrements this counter. Within `OS_Idle()` a call of `OS_POWER_GetMask()` generates a bit mask which describes which clock or power supply is in use, and which is not and may therefore be switched off.

This is an example for the peripheral power control. As it depends on the used hardware, its implementation is fictional: A, B and C are used to represent arbitrary peripherals.

```
#define OS_POWER_USE_A    (1 << 0) // peripheral "A"
#define OS_POWER_USE_B    (1 << 1) // peripheral "B"
#define OS_POWER_USE_C    (1 << 2) // peripheral "C"
#define OS_POWER_USE_ALL (OS_POWER_USE_A | OS_POWER_USE_B | OS_POWER_USE_C)
```

In the following function the peripherals A and C have been initialized and were marked in-use by a call to `OS_POWER_UsageInc()`:

```
void _InitAC(void) {
    ...
    OS_POWER_UsageInc(OS_POWER_USE_A); // Mark "A" as used
    OS_POWER_UsageInc(OS_POWER_USE_C); // Mark "C" as used
    ...
}
```

After some time, C will not be used any more and can therefore be marked as unused by a call to `OS_POWER_UsageDec()`:

```
void _WorkDone(void) {
    ...
    OS_POWER_UsageDec(OS_POWER_USE_C); // Mark "C" as unused
    ...
}
```

While in `OS_Idle()`, a call to `OS_POWER_GetMask()` retrieves a bit mask from the power management counter. That bit mask subsequently is used to modify the corresponding bits of a control register, leaving only those bits set that represent a peripheral which is in-use.

```
void OS_Idle(void) { // Idle loop: No task is ready to execute
    OS_UINT PowerMask;
    OS_U16 ClkControl;
    //
    // Initially disable interrupts
    //
    OS_INT_IncDI();
    //
    // Examine which peripherals may be switched off
    //
    PowerMask = OS_POWER_GetMask();
    //
    // Store the content of CTRLREG and clear all OS_POWER_USE related bits
    //
    ClkControl = CTRLREG & ~OS_POWER_USE_ALL;
    //
}
```

```
// Set only bits for used peripherals and write them to the specific register
// In this case only "A" is marked as used, so "C" gets switched off
//
CTRLREG      = ClkControl | PowerMask;
//
// Re-enable interrupts
//
OS_INT_DecRI();
for (;;) {
    _do_nothing();
};
}
```

18.4.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer	Idle
OS_POWER_GetMask()	Retrieves the power management counter.	•	•	•	•	•	•
OS_POWER_UsageDec()	Decrements the power management counter(s).	•	•	•	•	•	•
OS_POWER_UsageInc()	Increments the power management counter(s).	•	•	•	•	•	•

18.4.2.1 OS_POWER_GetMask()

Description

Retrieves the power management counter.

Prototype

```
OS_UINT OS_POWER_GetMask(void);
```

Return value

A bit mask which describes whether a peripheral is in use or not.

Additional information

This function generates a bit mask from the power management counter it retrieves. The bit mask describes which peripheral is in use and which one can be turned off. Switching off a peripheral can be done by writing this mask into the specific register. Please refer to the Example for additional information.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Peripheral power control* on page 415.

18.4.2.2 OS_POWER_UsageDec()

Description

Decrements the power management counter(s).

Prototype

```
void OS_POWER_UsageDec(OS_UINT Index);
```

Parameters

Parameter	Description
Index	Contains a mask with bits set for those counters which should be updated. (Bit 0 => Counter 0) The debug version checks for underflow, overflow and undefined counter number.

Additional information

When a peripheral is no longer in use this function is called to mark the peripheral as unused and signal that it can be switched off.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

123: OS_ERR_POWER_UNDER

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Peripheral power control* on page 415.

18.4.2.3 OS_POWER_UsageInc()

Description

Increments the power management counter(s).

Prototype

```
void OS_POWER_UsageInc(OS_UINT Index);
```

Parameters

Parameter	Description
<code>Index</code>	Contains a mask with bits set for those counters which should be updated. (Bit 0 => Counter 0) The debug version checks for underflow, overflow and undefined counter number.

Additional information

When a peripheral is in use this function is called to mark the peripheral as in use.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`122: OS_ERR_POWER_OVER`

`164: OS_ERR_OS_INT_ENTER_NOT_CALLED`

`167: OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Peripheral power control* on page 415.

Chapter 19

Heap Type Memory Management

19.1 Introduction

ANSI C offers some basic dynamic memory management functions. These are e.g. `malloc()`, `free()`, and `realloc()`. Unfortunately, these routines are not thread-safe, unless a special thread-safe implementation exists in the compiler runtime libraries; they can only be used from one task or by multiple tasks if they are called sequentially. Therefore, embOS offer thread safe variants of these routines. These variants have the same names as their ANSI counterparts, but are prefixed `OS_HEAP_`; they are called `OS_HEAP_malloc()`, `OS_HEAP_free()`, `OS_HEAP_realloc()`. The thread-safe variants that embOS offers use the standard ANSI routines, but they guarantee that the calls are serialized using a mutex.

If heap memory management is not supported by the standard C libraries, embOS heap memory management is not implemented.

This API is not available in embOS library mode `OS_LIBMODE_SAFE`.

Note

Many modern toolchain standard libraries can be made thread-safe with hook functions which are implemented by embOS. With it functions like `malloc()`, `free()` and `realloc()` are thread-safe and is not necessary to use `OS_HEAP_malloc()`, `OS_HEAP_free()` and `OS_HEAP_realloc()`. Please have a look in the core/compiler specific embOS manual for more details.

Example

```
void HPTask(void) {
    OS_U32* p;

    while (1) {
        p = (OS_U32*)OS_HEAP_malloc(4);
        *p = 42;
        OS_HEAP_free(p);
    }
}

void LPTask(void) {
    OS_U16* p;

    while (1) {
        p = (OS_U16*)OS_HEAP_malloc(2);
        *p = 0;
        OS_HEAP_free(p);
    }
}
```

19.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_HEAP_free()</code>	Frees a block of memory previously allocated.	•	•			
<code>OS_HEAP_malloc()</code>	Allocates a block of memory on the heap.	•	•			
<code>OS_HEAP_realloc()</code>	Changes the allocation size.	•	•			

19.2.1 OS_HEAP_free()

Description

Frees a block of memory previously allocated. This is the thread safe free() variant.

Prototype

```
void OS_HEAP_free(void* pMemBlock);
```

Parameters

Parameter	Description
<code>pMemBlock</code>	Pointer to a memory block previously allocated with <code>OS_HEAP_malloc()</code> .

Additional information

`OS_HEAP_free()` calls `free()` from the standard library. An embOS mutex is used internally to make `OS_HEAP_free()` thread safe.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

160: `OS_ERR_ILLEGAL_IN_ISR`

161: `OS_ERR_ILLEGAL_IN_TIMER`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void UseHeapMem(void) {
    char* sText;

    sText = (char*)OS_HEAP_malloc(20);
    strcpy(sText, "Hello World");
    printf(sText);
    OS_HEAP_free(sText);
}
```

19.2.2 OS_HEAP_malloc()

Description

Allocates a block of memory on the heap. This is the thread safe malloc() variant.

Prototype

```
void *OS_HEAP_malloc(unsigned int Size);
```

Parameters

Parameter	Description
Size	Size of the requested memory block in bytes.

Return value

Upon successful completion with size not equal zero, OS_HEAP_malloc() returns a pointer to the allocated space. Otherwise, it returns a NULL pointer.

Additional information

OS_HEAP_malloc() calls malloc() from the standard library. An embOS mutex is used internally to make OS_HEAP_malloc() thread safe.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void UseHeapMem(void) {  
    char* sText;  
  
    sText = (char*)OS_HEAP_malloc(20);  
    strcpy(sText, "Hello World");  
    printf(sText);  
    OS_HEAP_free(sText);  
}
```

19.2.3 OS_HEAP_realloc()

Description

Changes the allocation size. This is the thread safe realloc() variant.

Prototype

```
void *OS_HEAP_realloc(void* pMemBlock,
                     unsigned int Size);
```

Parameters

Parameter	Description
<code>pMemBlock</code>	Pointer to a memory block previously allocated with <code>OS_HEAP_malloc()</code> .
<code>Size</code>	New size for the memory block in bytes.

Return value

Upon successful completion, `OS_HEAP_realloc()` returns a pointer to the reallocated memory block. Otherwise, it returns a `NULL` pointer.

Additional information

`OS_HEAP_realloc()` calls `realloc()` from the standard library. An embOS mutex is used internally to make `OS_HEAP_realloc()` thread safe.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void UseHeapMem(void) {
    char* sText;

    sText = (char*)OS_HEAP_malloc(10);
    strcpy(sText, "Hello");
    printf(sText);
    sText = (char*)OS_HEAP_realloc(sText, 20);
    strcpy(sText, "Hello World");
    printf(sText);
    OS_HEAP_free(sText);
}
```

Chapter 20

Fixed Block Size Memory Pool

20.1 Introduction

Fixed block size memory pools contain a specific number of fixed-size blocks of memory. The location in memory of the pool, the size of each block, and the number of blocks are set at runtime by the application via a call to the `OS_MEMPOOL_Create()` function. The advantage of fixed memory pools is that a block of memory can be allocated from within any task in a very short, determined period of time.

Example

```
#include "RTOS.h"
#include <string.h>
#include <stdio.h>

#define BLOCK_SIZE          (16)
#define NUM_BLOCKS         (16)
#define POOL_SIZE          (NUM_BLOCKS * BLOCK_SIZE)

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                // Task-control-blocks
static OS_MEMPOOL   MEMF;
static OS_U8        aPool[POOL_SIZE];

static void HPTask(void) {
    char* a;

    while (1) {
        //
        // Request one memory block
        //
        a = OS_MEMPOOL_AllocBlocked(&MEMF);
        //
        // Work with memory block
        //
        strcpy(a, "Hello World\n");
        printf(a);
        OS_MEMPOOL_FreeEx(&MEMF, a); // Release memory block
        OS_TASK_Delay (10);
    }
}

static void LPTask(void) {
    char* b;

    while (1) {
        //
        // Request one memory block when available in max. next 10 system ticks
        //
        b = OS_MEMPOOL_AllocTimed(&MEMF, 10);
        if (b != 0) {
            //
            // Work with memory block
            //
            b[0] = 0x12;
            b[1] = 0x34;
            //
            // Release memory block
            //
            OS_MEMPOOL_FreeEx(&MEMF, b);
        }
        OS_TASK_Delay (50);
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
}
```

```
OS_InitHW(); // Initialize hardware for embOS
OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
//
// Create [NUM_BLOCKS] blocks with a size of [BLOCK_SIZE] each
//
OS_MEMPOOL_Create(&MEMF, aPool, NUM_BLOCKS, BLOCK_SIZE);
OS_Start(); // Start multitasking
return 0;
}
```

20.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_MEMPOOL_Alloc()</code>	Allocates a memory block from the specified memory pool.	•	•	•	•	•
<code>OS_MEMPOOL_AllocBlocked()</code>	Allocates a memory block from the specified memory pool.		•	•		
<code>OS_MEMPOOL_AllocTimed()</code>	Allocates a memory block from the specified memory pool.		•	•		
<code>OS_MEMPOOL_Create()</code>	Creates and initializes a fixed block size memory pool.	•	•			
<code>OS_MEMPOOL_Delete()</code>	Deletes the specified memory pool.	•	•			
<code>OS_MEMPOOL_Free()</code>	Releases the specified memory block.	•	•	•	•	•
<code>OS_MEMPOOL_FreeEx()</code>	Releases the specified memory block.	•	•	•	•	•
<code>OS_MEMPOOL_GetBlockSize()</code>	Returns the size of a single memory block in the specified memory pool.	•	•	•	•	•
<code>OS_MEMPOOL_GetMaxUsed()</code>	Returns the maximum number of memory blocks in the specified pool that have been used simultaneously since creation of the pool.	•	•	•	•	•
<code>OS_MEMPOOL_GetNumBlocks()</code>	Returns the total number of memory blocks in the specified memory pool.	•	•	•	•	•
<code>OS_MEMPOOL_GetNumFreeBlocks()</code>	Returns the number of free memory blocks in the specified memory pool.	•	•	•	•	•
<code>OS_MEMPOOL_IsInPool()</code>	Returns whether the specified memory block belongs to the specified memory pool.	•	•	•	•	•

20.2.1 OS_MEMPOOL_Alloc()

Description

Allocates a memory block from the specified memory pool.

Prototype

```
void *OS_MEMPOOL_Alloc(OS_MEMPOOL* pMEMF);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .

Return value

≠ NULL Pointer to the allocated block.
 = NULL If no block has been allocated.

Additional information

The calling task is never suspended by calling `OS_MEMPOOL_Alloc()`. The returned pointer must be passed as a parameter to `OS_MEMPOOL_Free()` or `OS_MEMPOOL_FreeEx()` to free the memory block. The pointer must not be modified.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
190: OS_ERR_MEMF_INV
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MEMPOOL _MemPool;

void Task(void) {
    void* pData;

    pData = OS_MEMPOOL_Alloc(&_MemPool);
    if (pData != NULL) {
        // Success: Work with the allocated memory.
    } else {
        // Failed: Do something else.
    }
}
```

20.2.2 OS_MEMPOOL_AllocBlocked()

Description

Allocates a memory block from the specified memory pool.

Prototype

```
void *OS_MEMPOOL_AllocBlocked(OS_MEMPOOL* pMEMF);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .

Return value

Pointer to the allocated memory block.

Additional information

If there is no free memory block in the pool, the calling task is suspended until a memory block becomes available. The returned pointer must be passed as a parameter to `OS_MEMPOOL_Free()` or `OS_MEMPOOL_FreeEx()` to free the memory block. The pointer must not be modified.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
190: OS_ERR_MEMF_INV
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Fixed Block Size Memory Pools* on page .

20.2.3 OS_MEMPOOL_AllocTimed()

Description

Allocates a memory block from the specified memory pool.

Prototype

```
void *OS_MEMPOOL_AllocTimed(OS_MEMPOOL* pMEMF,
                            OS_TIME      Timeout);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .
<code>Timeout</code>	Maximum time in system ticks until the memory block must be available. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are: $0 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $0 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.

Return value

= NULL No memory block could be allocated within the specified time.
 ≠ NULL Pointer to the allocated memory block.

Additional information

If there is no free memory block in the pool, the calling task is suspended until a memory block becomes available or the timeout has expired. The returned pointer must be passed as a parameter to `OS_MEMPOOL_Free()` or `OS_MEMPOOL_FreeEx()` to free the memory block. The pointer must not be modified. When the calling task is blocked by higher priority tasks for a period longer than the timeout value, it may happen that the memory block becomes available after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the memory block was not available within the requested time.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
190: OS_ERR_MEMF_INV
226: OS_ERR_ILLEGAL_IN_MAIN
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MEMPOOL _MemPool;

void Task(void) {
    void* pData;

    pData = OS_MEMPOOL_AllocTimed(&_MemPool, 20);
    if (pData != NULL) {
        // Success: Work with the allocated memory.
    } else {
        // Failed: Do something else.
    }
}
```

```
}
```

20.2.4 OS_MEMPOOL_Create()

Description

Creates and initializes a fixed block size memory pool.

Prototype

```
void OS_MEMPOOL_Create(OS_MEMPOOL* pMEMF,
                      void* pPool,
                      OS_UINT NumBlocks,
                      OS_UINT BlockSize);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .
<code>pPool</code>	Pointer to memory to be used for the memory pool. Required size is: <code>NumBlocks * BlockSize</code> .
<code>NumBlocks</code>	Number of blocks in the pool. $1 \leq \text{NumBlocks} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs $1 \leq \text{NumBlocks} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs
<code>BlockSize</code>	Size in bytes of one block. $1 \leq \text{BlockSize} \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs $1 \leq \text{BlockSize} \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs

Additional information

The buffer must be big enough to hold the given number of memory blocks of the specified block size: `NumBlocks * BlockSize` bytes.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
165: OS_ERR_INIT_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
190: OS_ERR_MEMF_INV
194: OS_ERR_MEMF_POOLADDR
195: OS_ERR_MEMF_BLOCKSIZE
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
#define NUM_BLOCKS (16)
#define BLOCK_SIZE (16)
#define POOL_SIZE (NUM_BLOCKS * BLOCK_SIZE)

static OS_U8 _aPool[POOL_SIZE];
static OS_MEMPOOL _MyMEMF;

void Init(void) {
    // Create 16 Blocks with size of 16 bytes
    OS_MEMPOOL_Create(&_MyMEMF, _aPool, NUM_BLOCKS, BLOCK_SIZE);
}
```

20.2.5 OS_MEMPOOL_Delete()

Description

Deletes the specified memory pool.

Prototype

```
void OS_MEMPOOL_Delete(OS_MEMPOOL* pMEMF);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .

Additional information

After deletion, the memory pool and memory blocks inside this pool can no longer be used. `pMEMF` must address a valid memory pool. Before deleting a memory pool, make sure that no task is waiting for it.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
190: OS_ERR_MEMF_INV
196: OS_ERR_MEMF_DELETE
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MEMPOOL MyMEMF;

int main(void) {
    ...
    //
    // Delete memory pool
    //
    OS_MEMPOOL_Delete(&MyMEMF);
    ...
    return 0;
}
```

20.2.6 OS_MEMPOOL_Free()

Description

Releases the specified memory block. The memory pool does not need to be denoted.

Prototype

```
void OS_MEMPOOL_Free(void* pMemBlock);
```

Parameters

Parameter	Description
<code>pMemBlock</code>	Pointer to the memory block.

Additional information

This function may be used instead of `OS_MEMPOOL_FreeEx()`. It has the advantage that only one parameter is needed since embOS will automatically determine the associated memory pool. The memory block becomes available for other tasks waiting for a memory block from the associated pool, which may cause a subsequent task switch.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

190: `OS_ERR_MEMF_INV`

191: `OS_ERR_MEMF_INV_PTR`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MEMPOOL _MemPool;

void Task(void) {
    void* pData;

    pData = OS_MEMPOOL_Alloc(&_amp;_MemPool); // Allocate memory
    ... // Work with allocated memory
    OS_MEMPOOL_Free(pData); // Free allocated memory
}
```

20.2.7 OS_MEMPOOL_FreeEx()

Description

Releases the specified memory block.

Prototype

```
void OS_MEMPOOL_FreeEx(OS_MEMPOOL* pMEMF,
                      void* pMemBlock);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .
<code>pMemBlock</code>	Pointer to memory block to free.

Additional information

The memory block becomes available for other tasks waiting for a memory block from the associated pool, which may cause a subsequent task switch.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
190: OS_ERR_MEMF_INV
191: OS_ERR_MEMF_INV_PTR
192: OS_ERR_MEMF_PTR_FREE
193: OS_ERR_MEMF_RELEASE
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Fixed Block Size Memory Pools* on page .

20.2.8 OS_MEMPOOL_GetBlockSize()

Description

Returns the size of a single memory block in the specified memory pool.

Prototype

```
int OS_MEMPOOL_GetBlockSize(OS_CONST_PTR OS_MEMPOOL *pMEMF);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .

Return value

Size in bytes of a single memory block in the specified memory pool.

Additional information

The return value is the value of the parameter when the memory pool was created.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

190: `OS_ERR_MEMF_INV`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MEMPOOL _MemPool;

void PrintBlockSize(void) {
    int Size;

    Size = OS_MEMPOOL_GetBlockSize(&_MemPool);
    printf("Block Size: %d\n", Size);
}
```

20.2.9 OS_MEMPOOL_GetMaxUsed()

Description

Returns the maximum number of memory blocks in the specified pool that have been used simultaneously since creation of the pool.

Prototype

```
int OS_MEMPOOL_GetMaxUsed(OS_CONST_PTR OS_MEMPOOL *pMEMF);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .

Return value

Maximum number of blocks in the specified memory pool that were used simultaneously since the pool was created.

Additional information

`OS_MEMPOOL_GetMaxUsed()` can be used during development to optimize the memory pool size which helps to save RAM.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
190: OS_ERR_MEMF_INV
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MEMPOOL _MemPool;

void PrintMemoryUsagePeak(void) {
    int    BlockCnt, UsedBlocks;
    void*  pData;

    pData = OS_MEMPOOL_AllocBlocked(&_MemPool);

    BlockCnt  = OS_MEMPOOL_GetNumBlocks(&_MemPool);
    UsedBlocks = OS_MEMPOOL_GetMaxUsed(&_MemPool);
    if (UsedBlocks != 0) {
        printf("Max used Memory: %d%%\n", (int)
            ((float)UsedBlocks / BlockCnt) * 100);
    } else {
        printf("Max used Memory: 0%%");
    }
}
```

20.2.10 OS_MEMPOOL_GetNumBlocks()

Description

Returns the total number of memory blocks in the specified memory pool.

Prototype

```
int OS_MEMPOOL_GetNumBlocks(OS_CONST_PTR OS_MEMPOOL *pMEMF);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .

Return value

Returns the number of blocks in the specified memory pool.

Additional information

The return value is the value that was given as parameter during creation of the memory pool.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

190: OS_ERR_MEMF_INV

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example of `OS_MEMPOOL_GetMaxUsed()` or `OS_MEMPOOL_GetNumFreeBlocks()`.

20.2.11 OS_MEMPOOL_GetNumFreeBlocks()

Description

Returns the number of free memory blocks in the specified memory pool.

Prototype

```
int OS_MEMPOOL_GetNumFreeBlocks(OS_CONST_PTR OS_MEMPOOL *pMEMF);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .

Return value

The number of free blocks currently available in the specified memory pool.

Additional information

If `OS_MEMPOOL_GetNumFreeBlocks()` returns the same as `OS_MEMPOOL_GetNumBlocks()`, no memory block is currently allocated.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

190: `OS_ERR_MEMF_INV`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MEMPOOL _MemPool;

void PrintMemoryUsage(void) {
    int    BlockCnt;
    int    UnusedBlocks;
    void*  pData;

    pData = OS_MEMPOOL_AllocBlocked(&_MemPool);

    BlockCnt      = OS_MEMPOOL_GetNumBlocks(&_MemPool);
    UnusedBlocks  = OS_MEMPOOL_GetNumFreeBlocks(&_MemPool);
    if (UnusedBlocks != 0) {
        printf("Used Memory: %d%%\n", 100 - (int)
            (((float)UnusedBlocks / BlockCnt) * 100));
    } else {
        printf("Used Memory: 0%%");
    }
}
```

20.2.12 OS_MEMPOOL_IsInPool()

Description

Returns whether the specified memory block belongs to the specified memory pool.

Prototype

```
OS_BOOL OS_MEMPOOL_IsInPool(OS_CONST_PTR OS_MEMPOOL *pMEMF,
                             OS_CONST_PTR void *pMemBlock);
```

Parameters

Parameter	Description
<code>pMEMF</code>	Pointer to a memory pool object of type <code>OS_MEMPOOL</code> .
<code>pMemBlock</code>	Pointer to a memory block that should be checked.

Return value

= 0 Pointer does not belong to the specified memory pool.
 ≠ 0 Pointer belongs to the specified memory pool.

Additional information

`OS_MEMPOOL_IsInPool()` checks if the memory block is located in the range of the memory pool buffer.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
190: OS_ERR_MEMF_INV
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_MEMPOOL _MemPool;

void CheckPointerLocation(OS_MEMPOOL* pMEMF, void* Pointer) {
    if (OS_MEMPOOL_IsInPool(pMEMF, Pointer) == 0) {
        printf("Pointer doesn't belong to the specified memory pool.\n");
    } else {
        printf("Pointer belongs to the specified memory pool.\n");
    }
}
```

Chapter 21

System Tick

21.1 Introduction

This chapter explains the concept of the system tick, which is used as a time base for embOS.

Typically, a hardware timer is used to generate periodic interrupts which are then utilized as a time base for embOS. To do so, the timer's according interrupt service routine must call one of the embOS tick handlers.

embOS offers different tick handlers with different functionality, and also provides the means to optionally call a user-defined hook function from within these tick handlers.

The used hardware timer usually is initialized within `OS_InitHW()`, which is delivered with the respective embOS start project's `RTOSInit.c`. This also includes the interrupt handler that is called by the hardware timer interrupt. Modifications to this initialization and the respective interrupt handler are required when a different hardware timer should be used (see *Board Support Packages* on page 574).

Tick handler

The interrupt service routine used as a time base must call one of the embOS tick handlers. The reason why there are different tick handlers is simple: They differ in capabilities, code size and execution speed. Most applications use the standard tick handler `OS_TICK_Handle()`, which increments the tick count by one each time it is called. This tick handler is small and efficient, but it cannot handle situations in which the interrupt rate differs from the tick rate. `OS_TICK_HandleEx()` is capable of handling even fractional interrupt rates, such as 1.6 interrupts per tick. If no tick hooks are used, `OS_TICK_HandleNoHook()` can be used instead of `OS_TICK_Handle()` in order to improve the tick handling performance even more.

Note

It is the application's responsibility to ensure that each system tick is handled and no system tick gets lost. In order to prevent the application from losing system ticks, it has to ensure that the system tick handler is executed regularly. If disabling interrupts or execution of other interrupts prevent the system tick from being served, system ticks can be lost.

Disabling the system tick

With many MCUs, power consumption may be reduced by using the embOS tickless support. Please refer to *Tickless support* on page 403 for further information.

21.1.1 Hooking into the system tick

There are various situations in which it can be desirable to call a function from the tick handler. Some examples are:

- Watchdog update
- Periodic status check
- Periodic I/O update

Advantage of using a hook function

Using a hook function is more efficient than performing a task switch or activating a software timer because the hook function is directly called from the embOS system tick interrupt handler and does not cause a context switch.

21.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TICK_AddHook()</code>	Adds the specified tick hook handler.	•	•			
<code>OS_TICK_Config()</code>	Configures the tick to interrupt ratio.	•	•			
<code>OS_TICK_Handle()</code>	Default embOS timer tick handler.				•	
<code>OS_TICK_HandleEx()</code>	Alternate tick handler that may be used instead of the default tick handler.				•	
<code>OS_TICK_HandleNoHook()</code>	Speed-optimized embOS timer tick handler without hook functionality.				•	
<code>OS_TICK_RemoveHook()</code>	Removes the specified tick hook handler.	•	•			

21.2.1 OS_TICK_AddHook()

Description

Adds the specified tick hook handler.

Prototype

```
void OS_TICK_AddHook(OS_TICK_HOOK*   pHook,
                    OS_ROUTINE_VOID* pfRoutine);
```

Parameters

Parameter	Description
<code>pHook</code>	Pointer to a structure of <code>OS_TICK_HOOK</code> .
<code>pfRoutine</code>	Pointer to the hook routine of type <code>OS_ROUTINE_VOID</code> which shall be called from the tick handler.

Additional information

The hook function is called directly from the interrupt handler. The function therefore should execute as quickly as possible. The function called by the tick hook must not re-enable interrupts.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
223: OS_ERR_TICKHOOK_INVALID
224: OS_ERR_TICKHOOK_FUNC_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TICK_HOOK _Hook;

void HookRoutine(void) {
    char c;
    c = GetADValue();
    OS_MAILBOX_Put1(&_MBData, &c);
}

int main(void) {
    ...
    OS_TICK_AddHook(&_Hook, HookRoutine);
    ...
}
```

21.2.2 OS_TICK_Config()

Description

Configures the tick to interrupt ratio. The default tick handler, `OS_TICK_Handle()`, assumes a 1:1 ratio, meaning one interrupt increments the tick count (`OS_Global.Time`) by one. When `OS_TICK_Config()` is not called the ratio is set per default to 1:1. For other ratios, `OS_TICK_HandleEx()` must to be used instead of the default handler and the tick to interrupt ratio must be configured through a call to `OS_TICK_Config()`. Since this must be done before the embOS timer is started, it is suggested to call `OS_TICK_Config()` during `OS_InitHW()`. The ratio can easily be set by the system tick and timer interrupt frequencies. embOS calculates the actual ratio automatically.

Prototype

```
void OS_TICK_Config(OS_U32 TickFreq,
                  OS_U32 IntFreq);
```

Parameters

Parameter	Description
<code>TickFreq</code>	System tick frequency in Hz
<code>IntFreq</code>	Timer interrupt frequency in Hz

Additional information

Fractional values are supported. For example, a 1 kHz system tick can be used even when an interrupt is generated every 1.6 millisecond only (625 Hz). In that case, `TickFreq` and `IntFreq` must be:

```
TickFreq = 1000;
IntFreq = 625;
```

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
//
// The system tick occurs every 1 milliseconds (1 kHz) but
// the timer interrupt occurs every 500 microseconds (2 kHz).
//
#define OS_TIMER_FREQ 168000000u // 168 MHz
#define OS_TICK_FREQ 1000u // 1 kHz
#define OS_INT_FREQ 2000u // 2 kHz

void OS_InitHW(void) {
    ...
    OS_SYSTIMER_CONFIG SysTimerConfig = {OS_TIMER_FREQ,
                                          OS_INT_FREQ,
                                          OS_TIMER_DOWNCOUNTING,
                                          _OS_GetHWTimerCycles,
                                          _OS_GetHWTimer_IntPending};
    OS_TIME_ConfigSysTimer(&SysTimerConfig);
    ...
    OS_TICK_Config(OS_TICK_FREQ, OS_INT_FREQ);
}
```

```
//  
// Configure hardware timer to OS_INT_FREQ  
//  
_SetHWTimer(OS_INT_FREQ);  
}
```

21.2.3 OS_TICK_Handle()

Description

Default embOS timer tick handler.

Prototype

```
void OS_TICK_Handle(void);
```

Additional information

The embOS tick handler must not be called by the application, but must be called from the hardware timer interrupt handler. `OS_INT_Enter()` or `OS_INT_EnterNestable()` must be called before calling the embOS tick handler.

If any tick hook functions have been added by the application (see *Hooking into the system tick* on page 445), these will be called by `OS_TICK_Handle()`.

`OS_TICK_Handle()` assumes a 1:1 tick to interrupt ratio, i.e. one interrupt increments the tick count by one. If any other ratio is configured, `OS_TICK_HandleEx()` must be used instead.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
162: OS_ERR_ILLEGAL_OUT_ISR  
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL  
256: OS_ERR_TICKHANDLE_WITH_FRACTIONAL_TICK
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
__interrupt void SysTick_Handler(void) {  
    OS_INT_EnterNestable();  
    OS_TICK_Handle();  
    OS_INT_LeaveNestable();  
}
```

21.2.4 OS_TICK_HandleEx()

Description

Alternate tick handler that may be used instead of the default tick handler. It may be used in situations in which the interrupt rate differs from the tick rate.

Prototype

```
void OS_TICK_HandleEx(void);
```

Additional information

The embOS tick handler must not be called by the application, but must be called from the hardware timer interrupt handler. `OS_INT_Enter()` or `OS_INT_EnterNestable()` must be called before calling the embOS tick handler.

If any tick hook functions have been added by the application (see *Hooking into the system tick* on page 445), these will be called by `OS_TICK_HandleEx()`.

Refer to `OS_TICK_Config()` for information on how to configure the tick to interrupt ratio for `OS_TICK_HandleEx()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
162: OS_ERR_ILLEGAL_OUT_ISR  
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
__interrupt void SysTick_Handler(void) {  
    OS_INT_EnterNestable();  
    OS_TICK_HandleEx();  
    OS_INT_LeaveNestable();  
}
```

21.2.5 OS_TICK_HandleNoHook()

Description

Speed-optimized embOS timer tick handler without hook functionality.

Prototype

```
void OS_TICK_HandleNoHook(void);
```

Additional information

The embOS tick handler must not be called by the application, it is only called from the system tick interrupt handler. `OS_INT_Enter()` or `OS_INT_EnterNestable()` must be called before calling the embOS tick handler.

`OS_TICK_HandleNoHook()` will not call any tick hook functions that may have been added by the application (see *Hooking into the system tick* on page 445).

`OS_TICK_HandleNoHook()` assumes a 1:1 tick to interrupt ratio, i.e. one interrupt increments the tick count by one. If any other ratio is configured, `OS_TICK_HandleEx()` must be used instead.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
162: OS_ERR_ILLEGAL_OUT_ISR  
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL  
256: OS_ERR_TICKHANDLE_WITH_FRACTIONAL_TICK
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
__interrupt void SysTick_Handler(void) {  
    OS_INT_EnterNestable();  
    OS_TICK_HandleNoHook();  
    OS_INT_LeaveNestable();  
}
```

21.2.6 OS_TICK_RemoveHook()

Description

Removes the specified tick hook handler.

Prototype

```
void OS_TICK_RemoveHook(OS_CONST_PTR OS_TICK_HOOK *pHook);
```

Parameters

Parameter	Description
<code>pHook</code>	Pointer to a structure of <code>OS_TICK_HOOK</code> .

Additional information

The function may be called to dynamically remove a tick hook function installed by a call to `OS_TICK_AddHook()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
167: OS_ERR_CPU_STATE_ILLEGAL  
223: OS_ERR_TICKHOOK_INVALID  
224: OS_ERR_TICKHOOK_FUNC_INVALID
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_TICK_HOOK _Hook;  
  
void Task(void) {  
    ...  
    OS_TICK_RemoveHook(&_Hook);  
    ...  
}
```

Chapter 22

Debugging

22.1 Runtime application errors

Many application errors can be detected during runtime. These are for example:

- Invalid usage of embOS API
- Usage of uninitialized embOS data structures
- Invalid pointers
- Stack overflow

Which runtime errors can be detected depends on how many checks are performed. Unfortunately, additional checks cost memory and performance (it is not that significant, but there is a difference). Not all embOS library modes include the debug and stack check code. For example `OS_LIBMODE_DP` includes the debug and stack check, whereas `OS_LIBMODE_R` does not contain any debug or stack check code.

Note

If an application error is detected and `OS_Error()` is called, do not switch to another embOS library mode which does not contain the debug checks. While doing so avoids calls to `OS_Error()` but it does not fix the original application error and subsequent behavior is unpredictable.

When embOS detects a runtime error, it calls the following routine:

```
void OS_Error(OS_STATUS ErrCode);
```

This routine is shipped as source code as part of the module `OS_Error.c`. Although this function is named `OS_Error()`, it does not show embOS errors but application errors. It is called with disabled preemptive task switches and, after re-enabling interrupts, loops forever as follows:

Example

```
void OS_Error(OS_STATUS ErrCode) {
    //
    // Disabling preemptive task switches avoids that other higher priority
    // tasks preempt OS_Error() which makes debugging easier.
    //
    OS_TASK_EnterRegion();
    //
    // Enable interrupts for embOSView communication.
    //
    OS_Global.Counters.Cnt.DI = 0u;
    OS_INT_Enable();
    //
    // OS_Global.Status will be shown in e.g. embOSView and IDE plugins.
    // It is available in debug and stack check builds only.
    //
    #if (OS_DEBUG != 0) || (OS_SUPPORT_STACKCHECK != 0)
        OS_Global.Status = ErrCode;
    #endif
    //
    // Endless loop may be left by setting ErrCode to OS_OK (0).
    //
    while (ErrCode != OS_OK) {
    }
}
```

If you are using embOSView, you can see the value and meaning of `OS_Global.Status` in the system variable window.

When using a debugger, you should set a breakpoint at the beginning of this routine or simply stop the program after a failure. The error code is passed to the function as a parameter. You should add `OS_Global.Status` to your watch window.

Your call stack window shows where the error occurred. If a call stack windows is not available you can (as described below) step back to the program sequence causing the problem.

You can modify the routine to accommodate to your own hardware; this could mean that your target hardware sets an error-indicating LED or shows a small message on the display.

Note

When modifying the `OS_Error()` routine, the last statement needs to be the infinite loop.

If you look at the `OS_Error()` routine, you will see that it is more complicated than necessary. The actual error code is passed to `OS_Error()` to the argument `ErrCode`. The program then waits for this variable to be reset. Simply reset this variable to 0 using your debugger, and you can easily step back to the program sequence causing the problem. Most of the time, looking at this part of the program will make the problem clear.

22.1.1 List of error codes

Value	enum value	Explanation
0	<code>OS_OK</code>	No error, everything okay.
100	<code>OS_ERR_ISR_INDEX</code>	Index value out of bounds during interrupt controller initialization or interrupt installation.
101	<code>OS_ERR_ISR_VECTOR</code>	Default interrupt handler called, but interrupt vector not initialized.
102	<code>OS_ERR_ISR_PRIO</code>	Wrong interrupt priority.
103	<code>OS_ERR_WRONG_STACK</code>	Wrong stack used before <code>main()</code> .
104	<code>OS_ERR_ISR_NO_HANDLER</code>	No interrupt handler was defined for this interrupt.
105	<code>OS_ERR_TLS_INIT</code>	<code>OS_TLS_Init()</code> called multiple times from one task.
106	<code>OS_ERR_MB_BUFFER_SIZE</code>	For 16-bit CPUs, the maximum buffer size for a mailbox (65,535 bytes) exceeded.
115	<code>OS_ERR_TASK_STACK_SIZE_INSUFFICIENT</code>	Task stack size too small.
116	<code>OS_ERR_EXTEND_CONTEXT</code>	<code>OS_TASK_SetContextExtension()</code> called multiple times from one task.
119	<code>OS_ERR_IDLE_RETURNS</code>	<code>OS_Idle()</code> must not return.
120	<code>OS_ERR_TASK_STACK_OVERFLOW</code>	Task stack overflow.
121	<code>OS_ERR_SEMAPHORE_OVERFLOW</code>	Semaphore value overflow.
122	<code>OS_ERR_POWER_OVER</code>	Counter overflows when calling <code>OS_POWER_UsageInc()</code> .
123	<code>OS_ERR_POWER_UNDER</code>	Counter underflows when calling <code>OS_POWER_UsageDec()</code> .
125	<code>OS_ERR_SYSTEM_STACK_OVERFLOW</code>	System stack overflow.
126	<code>OS_ERR_INTERRUPT_STACK_OVERFLOW</code>	Interrupt stack overflow.

Value	enum value	Explanation
128	OS_ERR_INV_TASK	Task control block invalid, not initialized or overwritten.
129	OS_ERR_INV_TIMER	Timer control block invalid, not initialized or overwritten.
130	OS_ERR_INV_MAILBOX	Mailbox control block invalid, not initialized or overwritten.
132	OS_ERR_INV_SEMAPHORE	Control block for semaphore invalid, not initialized or overwritten.
133	OS_ERR_INV_MUTEX	Control block for mutex invalid, not initialized or overwritten.
135	OS_ERR_MAILBOX_NOT1	One of the following 1-byte mailbox functions has been used on a multi-byte mailbox: OS_MAILBOX_Get1(), OS_MAILBOX_GetBlocked1(), OS_MAILBOX_GetTimed1(), OS_MAILBOX_Put1(), OS_MAILBOX_PutBlocked1(), OS_MAILBOX_PutFront1(), OS_MAILBOX_PutFrontBlocked1() or OS_MAILBOX_PutTimed1().
136	OS_ERR_MAILBOX_DELETE	OS_MAILBOX_Delete() was called on a mailbox with waiting tasks.
137	OS_ERR_SEMAPHORE_DELETE	OS_SEMAPHORE_Delete() was called on a semaphore with waiting tasks.
138	OS_ERR_MUTEX_DELETE	OS_MUTEX_Delete() was called on a mutex which is claimed by a task.
143	OS_ERR_QUEUE_INUSE	Queue in use.
144	OS_ERR_QUEUE_NOT_INUSE	Queue not in use.
145	OS_ERR_QUEUE_INVALID	Queue invalid.
146	OS_ERR_QUEUE_DELETE	A queue was deleted by a call of OS_QUEUE_Delete() while tasks are waiting at the queue.
147	OS_ERR_MAILBOX_INUSE	Mailbox in use.
148	OS_ERR_MAILBOX_NOT_INUSE	Mailbox not in use.
149	OS_ERR_MESSAGE_SIZE_ZERO	Attempt to store a message with size of zero.
150	OS_ERR_UNUSE_BEFORE_USE	OS_MUTEX_Unlock() has been called on a mutex that hasn't been locked before.
151	OS_ERR_LEAVEREGION_BEFORE_ENTERREGION	OS_TASK_LeaveRegion() has been called before OS_TASK_EnterRegion().
152	OS_ERR_LEAVEINT	OS_INT_Leave()/OS_INT_LeaveNestable() called without OS_INT_Enter()/OS_INT_EnterNestable().
153	OS_ERR_DICNT_OVERFLOW	The interrupt disable counter (OS_Global.Counters.Cnt.DI) is out of range (0-15). The counter is affected by the API calls OS_INT_IncDI(), OS_INT_DecRI(), OS_INT_Enter() and OS_INT_Leave().
154	OS_ERR_INTERRUPT_DISABLED	OS_TASK_Delay() or OS_TASK_DelayUntil() called from inside a critical region with interrupts disabled.
155	OS_ERR_TASK_ENDS_WITHOUT_TERMINATE	Task routine returns without OS_TASK_Terminate().

Value	enum value	Explanation
156	OS_ERR_MUTEX_OWNER	OS_MUTEX_Unlock() has been called from a task which does not own the mutex.
157	OS_ERR_REGIONCNT	The region counter overflows (>255).
158	OS_ERR_DELAYUS_INTERRUPT_DISABLED	OS_TASK_Delay_us() called with interrupts disabled.
159	OS_ERR_MUTEX_OVERFLOW	OS_MUTEX_Lock(), OS_MUTEX_LockBlocked() or OS_MUTEX_LockTimed() has been called too often from the same task.
160	OS_ERR_ILLEGAL_IN_ISR	Illegal function call in an interrupt service routine: A routine that must not be called from within an ISR has been called from within an ISR.
161	OS_ERR_ILLEGAL_IN_TIMER	Illegal function call in a software timer: A routine that must not be called from within a software timer has been called from within a timer.
162	OS_ERR_ILLEGAL_OUT_ISR	Not a legal API outside interrupt.
163	OS_ERR_OS_INT_ENTER_CALLED	OS_INT_Enter() has been called, but CPU is not in ISR state.
164	OS_ERR_OS_INT_ENTER_NOT_CALLED	OS_INT_Enter() has not been called, but CPU is in ISR state.
165	OS_ERR_INIT_NOT_CALLED	OS_Init() was not called.
166	OS_ERR_ISR_PRIORITY_INVALID	embOS API called from ISR with an invalid priority.
167	OS_ERR_CPU_STATE_ILLEGAL	CPU runs in illegal mode.
169	OS_ERR_TICKLESS_WITH_FRACTIONAL_TICK	OS_TICKLESS_AdjustTime() was called despite OS_TICK_Config() has been called before.
170	OS_ERR_2USE_TASK	Task control block has been initialized by calling a create function twice.
171	OS_ERR_2USE_TIMER	Timer control block has been initialized by calling a create function twice.
172	OS_ERR_2USE_MAILBOX	Mailbox control block has been initialized by calling a create function twice.
174	OS_ERR_2USE_SEMAPHORE	Semaphore has been initialized by calling a create function twice.
175	OS_ERR_2USE_MUTEX	Mutex has been initialized by calling a create function twice.
176	OS_ERR_2USE_MEMF	Fixed size memory pool has been initialized by calling a create function twice.
177	OS_ERR_2USE_QUEUE	Queue has been initialized by calling a create function twice.
178	OS_ERR_2USE_EVENT	Event object has been initialized by calling a create function twice.
179	OS_ERR_2USE_WATCHDOG	Watchdog has been initialized by calling a create function twice.
180	OS_ERR_NESTED_RX_INT	OS_COM_OnRx() interrupt handler for embOSView is nested. Disable nestable interrupts.
181	OS_ERR_ISR_ENTRY_FUNC_INVALID	Invalid function pointer for ISR entry callback.

Value	enum value	Explanation
185	OS_ERR_SPINLOCK_INV_CORE	Invalid core ID specified for accessing a OS_SPINLOCK_SW struct.
190	OS_ERR_MEMF_INV	Fixed size memory block control structure not created before use.
191	OS_ERR_MEMF_INV_PTR	Pointer to memory block does not belong to memory pool on Release.
192	OS_ERR_MEMF_PTR_FREE	Pointer to memory block is already free when calling OS_MEMPOOL_Free() or OS_MEMPOOL_FreeEx(). Possibly, same pointer was released twice.
193	OS_ERR_MEMF_RELEASE	OS_MEMPOOL_Free() or OS_MEMPOOL_FreeEx() was called for a memory pool, that had no memory block allocated (all available blocks were already free before).
194	OS_ERR_MEMF_POOLADDR	OS_MEMPOOL_Create() was called with a memory pool base address which is not located at a word aligned base address.
195	OS_ERR_MEMF_BLOCKSIZE	OS_MEMPOOL_Create() was called with a data block size which is not a multiple of processors word size.
196	OS_ERR_MEMF_DELETE	OS_MEMPOOL_Delete() was called on a memory pool with waiting tasks.
200	OS_ERR_SUSPEND_TOO_OFTEN	Number of nested calls to OS_TASK_Suspend() exceeded 3.
201	OS_ERR_RESUME_BEFORE_SUSPEND	OS_TASK_Resume() called on a task that was not suspended.
202	OS_ERR_TASK_PRIORITY	OS_TASK_Create() was called with a task priority which is already assigned to another task. This error can only occur when embOS was compiled without round-robin support.
203	OS_ERR_TASK_PRIORITY_INVALID	The value 0 was used as task priority.
205	OS_ERR_TIMER_PERIOD_INVALID	The value 0 was used as timer period.
210	OS_ERR_EVENT_INVALID	An OS_EVENT object was used before it was created.
212	OS_ERR_EVENT_DELETE	An OS_EVENT object was deleted with waiting tasks.
213	OS_ERR_EVENTOBJ_RESETMODE	OS_MULTIOBJ_IsEventSignaled() used with reset mode other than OS_EVENT_RESET_MODE_MANUAL.
220	OS_ERR_WAITLIST_RING	This error should not occur. Please contact the support.
221	OS_ERR_WAITLIST_PREV	This error should not occur. Please contact the support.
222	OS_ERR_WAITLIST_NEXT	This error should not occur. Please contact the support.
223	OS_ERR_TICKHOOK_INVALID	Invalid tick hook.
224	OS_ERR_TICKHOOK_FUNC_INVALID	Invalid tick hook function.
225	OS_ERR_NOT_IN_REGION	A function was called without declaring the necessary critical region.
226	OS_ERR_ILLEGAL_IN_MAIN	Not a legal API call from main().

Value	enum value	Explanation
227	OS_ERR_ILLEGAL_IN_TASK	Not a legal API after OS_Start().
228	OS_ERR_ILLEGAL_AFTER_OSSTART	Not a legal API after OS_Start().
229	OS_ERR_ILLEGAL_IN_IDLE	Not a legal API call from OS_Idle().
230	OS_ERR_NON_ALIGNED_INVALIDATE	Cache invalidation needs to be cache line aligned.
234	OS_ERR_HW_NOT_AVAILABLE	Hardware unit is not implemented or enabled.
235	OS_ERR_NON_TIMERCYCLES_FUNC	OS_TIME_ConfigSysTimer() not called or called with NULL pointer for function to read timer counter value.
236	OS_ERR_NON_TIMERINT-PENDING_FUNC	OS_TIME_ConfigSysTimer() not called or called with NULL pointer for function to read timer interrupt pending.
237	OS_ERR_FRACTIONAL_TICK	embOS API function called with fractional tick to interrupt ratio.
238	OS_ERR_ZERO_TIMER_INT_FREQ	OS_TIME_ConfigSysTimer() not called or called with an interrupt frequency of 0.
239	OS_ERR_COUNTER_FREQ_ZERO	OS_TIME_ConfigSysTimer() not called or called with a counter frequency of 0.
240	OS_ERR_MPU_NOT_PRESENT	MPU unit not present in the device.
241	OS_ERR_MPU_INVALID_REGION	Invalid MPU region index number.
242	OS_ERR_MPU_INVALID_SIZE	Invalid MPU region size.
243	OS_ERR_MPU_INVALID_PERMISSION	Invalid MPU region permission.
244	OS_ERR_MPU_INVALID_ALIGNMENT	Invalid MPU region alignment.
245	OS_ERR_MPU_INVALID_OBJECT	OS object is directly accessible from the task which is not allowed.
246	OS_ERR_MPU_PRIVSTATE_INVALID	Invalid call from a privileged task.
247	OS_ERR_MPU_NOINIT	OS_MPU_Init() not called.
248	OS_ERR_MPU_DEVICE_INDEX	Invalid device driver index.
249	OS_ERR_MPU_INV_DEVICE_LIST	Invalid device driver list.
250	OS_ERR_CONFIG_OSSTOP	OS_Stop() is called without using OS_ConfigStop() before.
251	OS_ERR_OSSTOP_BUFFER	Buffer is too small to hold a copy of the main() stack.
253	OS_ERR_VERSION_MISMATCH	OS library and RTOS.h have different version numbers. Please ensure both are from the same embOS shipment.
254	OS_ERR_LIB_INCOMPATIBLE	Incompatible OS library is used.
255	OS_ERR_INV_PARAMETER_VALUE	An invalid value was passed to the called function (see call stack). Check the API description for valid values.
256	OS_ERR_TICKHANDLE_WITH_FRACTIONAL_TICK	OS_TICK_Handle() or OS_TICK_HandleNoHook() was called after OS_TICK_Config() was used for an interrupt to tick ratio other than 1:1.
257	OS_ERR_RWLOCK_INVALID	Readers-Writer Lock control block invalid, not initialized or overwritten.
258	OS_ERR_2USE_RWLOCK	Readers-Writer Lock has been initialized by calling a create function twice.

Value	enum value	Explanation
260	OS_ERR_UNALIGNED_IRQ_STACK	Unaligned IRQ stack.
261	OS_ERR_UNALIGNED_MAIN_STACK	Unaligned main stack.
262	OS_ERR_FPU_NOT_ENABLED	FPU was not enabled before embOS gets initialized.
263	OS_ERR_MPU_SANITY_BUFFER_NOT_SET	MPU sanity check buffer not set with OS_MPU_SetSanityCheckBuffer().
264	OS_ERR_MPU_INVALID_ATTR_INDEX	MPU Invalid index used with OS_ARMv8M_SetMPUAttribute().
265	OS_ERR_MULTIOBJ_INVALID_ROUTINE	Invalid condition routine used with OS_MULTIOBJ_Wait(), OS_MULTIOBJ_WaitBlocked() or OS_MULTIOBJ_WaitTimed().

22.1.2 Application defined error codes

The embOS error codes begin at 100. The range 1 - 99 can be used for application defined error codes. With it you can call `OS_Error()` with your own defined error code from your application.

Example

```
#define OS_ERR_APPL (0x02u)

void UserAppFunc(void) {
    int r;
    r = DoSomething()
    if (r == 0) {
        OS_Error(OS_ERR_APPL)
    }
}
```

22.1.3 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_DEBUG_GetError()</code>	Returns the system status.	•	•	•	•	•

22.1.3.1 OS_DEBUG_GetError()

Description

Returns the system status.

Prototype

```
OS_STATUS OS_DEBUG_GetError(void);
```

Return value

= 0 OS_OK, No application error occurred.
≠ 0 Application error occurred.

Additional information

The system status codes are described in the embOS manual in chapter *List of error codes* on page 456. OS_DEBUG_GetError() always returns OS_OK when no debug or stack check code is included.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintOSStatus(void) {  
    OS_STATUS s;  
  
    s = OS_DEBUG_GetError();  
    printf("embOS status: %u\n", s);  
}
```

22.2 Human readable object identifiers

embOS objects like mailbox or semaphore are handled via separate control structures. Each OS object is identified by the address of the according control structure. For debugging purpose this address is displayed in external tools like embOSView or IDE RTOS plugins.

Tasks always have a human readable task name (except in embOS library mode `OS_LIBMODE_XR`). The task name is set at task creation. It can be helpful to have human readable identifiers for other OS objects, as well. With the following API functions human readable identifiers to an unlimited amount of OS objects can easily be added. Human readable object identifiers are not supported in embOS library mode `OS_LIBMODE_XR`.

Example

```
#include "RTOS.h"
#include <stdio.h>

static OS_STACKPTR int _Stack[128];
static OS_TASK      _TCB;
static OS_MAILBOX   _Mailbox;
static OS_OBJNAME   _MailboxName;
static char         _acBuffer[100];

static void _Task(void) {
    const char* s;
    s = OS_DEBUG_GetObjName(&_Mailbox);
    printf(s);
    //
    // Set another name for the mailbox
    //
    OS_DEBUG_RemoveObjName(&_MailboxName);
    OS_DEBUG_SetObjName(&_MailboxName, &_Mailbox, "My new Mailbox");
    while (1) {
        OS_TASK_Delay(50);
    }
}

int main(void) {
    OS_Init();           // Initialize embOS
    OS_Inithw();        // Initialize required hardware
    OS_TASK_CREATE(&_TCB, "Task", 100, _Task, _Stack);
    OS_MAILBOX_Create(&_Mailbox, 10, 10, &_acBuffer);
    OS_DEBUG_SetObjName(&_MailboxName, &_Mailbox, "My Mailbox");
    OS_Start();         // Start embOS
    return 0;
}
```

22.2.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
OS_DEBUG_GetObjName()	Returns the name of an RTOS object.	•	•	•		
OS_DEBUG_RemoveObjName()	Removes an RTOS object name.	•	•			
OS_DEBUG_SetObjName()	Sets an RTOS object name.	•	•			

22.2.1.1 OS_DEBUG_GetObjName()

Description

Returns the name of an RTOS object.

Prototype

```
char *OS_DEBUG_GetObjName(OS_CONST_PTR void *pOSObj);
```

Parameters

Parameter	Description
<code>pOSObj</code>	Pointer to the RTOS object.

Return value

= NULL Name was not set for this object.
 ≠ NULL Pointer to the RTOS object name.

Additional information

`OS_DEBUG_GetObjName()` returns the object name which was set before with `OS_DEBUG_SetObjName()`. The return value is valid only when using an embOS build with object name support. When using an embOS build without object name support, `OS_DEBUG_GetObjName()` returns "n/a" in any case. The embOS `OS_LIBMODE_XR` library mode does not support object names.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please find an example at *Human readable object identifiers* on page 464.

22.2.1.2 OS_DEBUG_RemoveObjName()

Description

Removes an RTOS object name.

Prototype

```
void OS_DEBUG_RemoveObjName(OS_CONST_PTR OS_OBJNAME *pObjName);
```

Parameters

Parameter	Description
<code>pObjName</code>	Pointer to a OS_OBJNAME control structure.

Additional information

OS_DEBUG_RemoveObjName() removes the object name which was set before with OS_DEBUG_SetObjName(). When using an embOS build without object name support, OS_DEBUG_RemoveObjName() has no effect. The embOS OS_LIBMODE_XR library mode does not support object names.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please find an example at *Human readable object identifiers* on page 464.

22.2.1.3 OS_DEBUG_SetObjName()

Description

Sets an RTOS object name.

Prototype

```
void OS_DEBUG_SetObjName(OS_OBJNAME* pObjName,
                        OS_CONST_PTR void *pOSObj,
                        OS_CONST_PTR char *sName);
```

Parameters

Parameter	Description
<code>pObjName</code>	Pointer to a OS_OBJNAME control structure.
<code>pOSObj</code>	Pointer to the RTOS object.
<code>sName</code>	Name of the RTOS object. embOS does not copy the RTOS object name, but uses the pointer exclusively.

Additional information

With `OS_DEBUG_SetObjName()` every RTOS object like mailbox can have a name. This name can be shown in debug tools like IDE RTOS plug-ins. Every object name needs a control structure of type `OS_OBJNAME`. When using an embOS build without object name support, `OS_DEBUG_SetObjName()` does not set an object name. The embOS `OS_LIBMODE_XR` library mode does not support object names.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please find an example at *Human readable object identifiers* on page 464.

22.3 embOS API trace

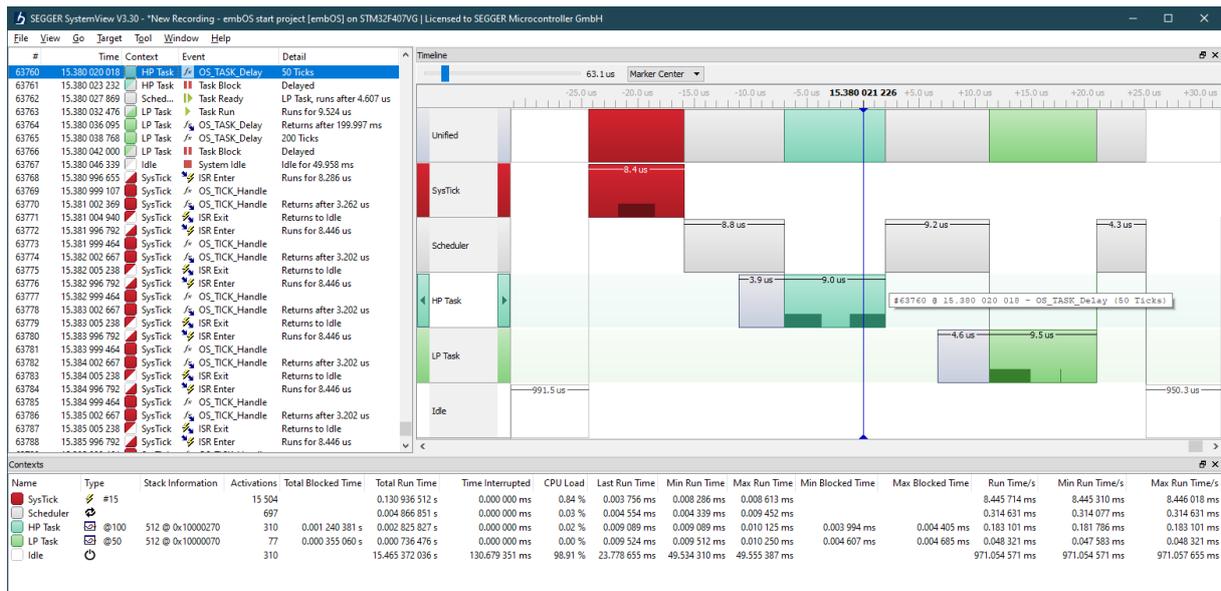
embOS supports API trace in two different ways:

- embOS API trace with embOSView (refer to *embOSView* on page 491).
- embOS API trace with any other tool (e.g. SystemView).

To do so, the embOS API functions call specific routines which store trace events to a given memory location. With embOSView, these routines are called directly inside the embOS API functions. To enable the use of embOS API trace with other tools than embOSView, however, a structure containing various function pointers is used to store trace events in memory. That structure may be configured to point at specific routines for the desired tool via `OS_TRACE_SetAPI()`, which are then called from the embOS API functions when API trace is enabled. These specific routines must be provided as part of the application and are shipped for example with the SystemView target sources.

Example

```
void SEGGER_SYSVIEW_Conf(void) {
    ..
    //
    // Configure embOS to use SystemView
    //
    OS_TRACE_SetAPI (&embOS_TraceAPI_SYSVIEW);
    ..
}
```



22.3.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TRACE_SetAPI()</code>	OS_TRACE_SetAPI() sets the pointer to the API trace function table.	•	•			

22.3.1.1 OS_TRACE_SetAPI()

Description

OS_TRACE_SetAPI() sets the pointer to the API trace function table.

Prototype

```
void OS_TRACE_SetAPI(OS_CONST_PTR OS_TRACE_API *pTraceAPI);
```

Parameters

Parameter	Description
pTraceAPI	Pointer to API trace function table or NULL to disable API trace.

Additional information

The function pointers are defined as shown below.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Definition of OS_TRACE_API

```
typedef struct {
    //
    // OS specific Trace Events
    //
    void (*pfRecordEnterISR)          (void);
    void (*pfRecordExitISR)           (void);
    void (*pfRecordExitISRToScheduler)(void);
    void (*pfRecordTaskInfo)          (const OS_TASK* pTask);
    void (*pfRecordTaskCreate)        (OS_U32 TaskId);
    void (*pfRecordTaskStartExec)     (OS_U32 TaskId);
    void (*pfRecordTaskStopExec)      (void);
    void (*pfRecordTaskStartReady)    (OS_U32 TaskId);
    void (*pfRecordTaskStopReady)     (OS_U32 TaskId, unsigned int Reason);
    void (*pfRecordIdle)              (void);
    //
    // Generic Trace Event logging (used by OS API)
    //
    void (*pfRecordVoid)              (unsigned int Id);
    void (*pfRecordU32)                (unsigned int Id, OS_U32 Para0);
    void (*pfRecordU32x2)              (unsigned int Id, OS_U32 Para0,
                                       OS_U32 Para1);
    void (*pfRecordU32x3)              (unsigned int Id, OS_U32 Para0,
                                       OS_U32 Para1, OS_U32 Para2);
    void (*pfRecordU32x4)              (unsigned int Id, OS_U32 Para0,
                                       OS_U32 Para1, OS_U32 Para2, OS_U32 Para3);
    OS_U32 (*pfPtrToId)                (OS_U32 Ptr);
    //
    // Additional Trace Event logging
    //
    void (*pfRecordEnterTimer)         (OS_U32 TimerID);
    void (*pfRecordExitTimer)         (void);
    void (*pfRecordEndCall)            (unsigned int Id);
    void (*pfRecordEndCallU32)        (unsigned int Id, OS_U32 Para0);
};
```

```
void (*pfRecordTaskTerminate) (OS_U32 TaskId);
void (*pfRecordU32x5) (unsigned int Id, OS_U32 Para0,
                     OS_U32 Para1, OS_U32 Para2,
                     OS_U32 Para3, OS_U32 Para4);
void (*pfRecordObjName) (OS_U32 Id, OS_CONST_PTR char* Para0);
} OS_TRACE_API;
```

Example

```
void SEGGER_SYSVIEW_Conf(void) {
    // Configure embOS to use SystemView.
    OS_TRACE_SetAPI(&embOS_TraceAPI_SYSVIEW);
}
```

Chapter 23

Profiling

23.1 Task execution

embOS profiling counts the number of task activations and preemptions. For example the number of task activations can be seen in embOSView and in IDE RTOS plugins but can also be requested in the application with the following API functions.

23.1.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_STAT_GetNumActivations()</code>	Return the number of task activations.	•	•	•	•	•
<code>OS_STAT_GetNumPreemptions()</code>	Return the number of task preemptions.	•	•	•	•	•

23.1.1.1 OS_STAT_GetNumActivations()

Description

Return the number of task activations.

Prototype

```
OS_U32 OS_STAT_GetNumActivations(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

Number of task activations.

Additional information

This API function is available only when task statistic information are enabled. This is the default in `OS_LIBMODE_DT`, `OS_LIBMODE_DP`, `OS_LIBMODE_D`, and `OS_LIBMODE_SP`. It is not available in `OS_LIBMODE_SAFE`.

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintActivations(OS_TASK* pTask) {
    OS_U32 NumActivations;

    NumActivations = OS_STAT_GetNumActivations();
    printf("Task has been activated %u times\n", NumActivations);
}
```

23.1.1.2 OS_STAT_GetNumPreemptions()

Description

Return the number of task preemptions.

Prototype

```
OS_U32 OS_STAT_GetNumPreemptions(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

Number of task preemptions.

Additional information

This API function is available only when task statistic information are enabled. This is the default in `OS_LIBMODE_DT`, `OS_LIBMODE_DP`, `OS_LIBMODE_D`, and `OS_LIBMODE_SP`. It is not available in `OS_LIBMODE_SAFE`.

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintPreemptions(OS_TASK* pTask) {
    OS_U32 NumPreemptions;

    NumPreemptions = OS_STAT_GetNumPreemptions();
    printf("Task has been preempted %u times\n", NumPreemptions);
}
```

23.2 Task specific CPU load measurement

embOS profiling measures the task execution time and calculates the task specific CPU load. This information can be seen in embOSView in the task list CPU load column and in the CPU load window. The application can use the following profiling API routines to receive the same information.

The screenshot displays the SEGGER embOSView V5.16.1.2 interface. The 'Task List' window shows the following data:

Prio	ID	Name	Status	Timeout	Stack	CPU Load	Run Count	Time Slice
100	0x100007D0	HP Task	Delayed	0 (39398)	144 / 512 @ 0x10000480	51.30%	39398	0 / 2
50	0x1000077C	LP Task	Delayed	0 (39428)	144 / 512 @ 0x10000280	26.42%	39428	0 / 2
1	0x10000728	Sample Task	Delayed	542 (40000)	420 / 512 @ 0x10000080	0.00%	40	0 / 2

The 'System information' window shows the following details:

Name	Value
OS Version	5.16.1.0
CPU	Cortex M4F/M7F
Library mode	v7vLDP
OS Time	39332
Number of Tasks	3
Status	O.K.
Active task	0x0000
Current task	0x0000
System stack	228 / 1024 @ 0x2001FC00
Interrupt stack	N/A
Trace buffer	0 / 0 (off)
CPU load	77.72%

The 'Terminal' window is empty. The status bar at the bottom indicates 'Running' with Tx: 14490/1449 Rx: 39123/1449, J-Link MEM 4000 kHz SWD Cortex-M4.

Example

```
#include "RTOS.h"
#include <stdio.h>

static OS_STACKPTR int StackHP[128], StackLP[128], StackSample[128];
static OS_TASK      TCBHP, TCBLP, TCBSample;

static void HPTask(void) {
    while (1) {
        OS_TASK_Delay_us(500); // Do something.
        OS_TASK_Delay(1);     // Give other tasks a chance to run.
    }
}

static void LPTask(void) {
    while (1) {
        OS_TASK_Delay_us(250); // Do something.
        OS_TASK_Delay(1);     // Give other tasks a chance to run.
    }
}

static void SampleTask(void) {
    while (1) {
```

```
OS_STAT_Sample();           // Calculate CPU load.
printf("CPU usage of HP Task: %d\n", OS_STAT_GetLoad(&TCBHP));
printf("CPU usage of LP Task: %d\n\n", OS_STAT_GetLoad(&TCBLP));
OS_TASK_Delay(1000);       // Wait for at least 1 second before next sampling.
}
}

int main(void) {
    OS_Init();               // Initialize embOS
    OS_Inithw();            // Initialize the hardware
    OS_TASK_CREATE(&TCBHP,   "HP Task",    100, HPTask,    StackHP);
    OS_TASK_CREATE(&TCBLP,   "LP Task",    50, LPTask,    StackLP);
    OS_TASK_CREATE(&TCBSample, "Sample Task", 1, SampleTask, StackSample);
    OS_Start();             // Start multitasking
    return 0;
}
```

Output

```
CPU usage of HP Task: 520
CPU usage of LP Task: 268
```

Note

For embOS V5.06 and later `OS_TIME_ConfigSysTimer()` must be called before using profiling.

23.2.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_STAT_Disable()</code>	Disables the kernel profiling.	•	•		•	•
<code>OS_STAT_Enable()</code>	Enables the kernel profiling (for an indefinite time).	•	•		•	•
<code>OS_STAT_GetExecTime()</code>	Returns the total task execution time.	•	•		•	•
<code>OS_STAT_GetLoad()</code>	Calculates the current task's CPU load in permille.	•	•		•	•
<code>OS_STAT_Sample()</code>	Starts the kernel profiling and calculates the absolute task run time for all tasks since the last call to <code>OS_STAT_Sample()</code> .	•	•		•	•

23.2.1.1 OS_STAT_Disable()

Description

Disables the kernel profiling.

Prototype

```
void OS_STAT_Disable(void);
```

Additional information

The function `OS_STAT_Enable()` may be used to start profiling.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void StopProfiling(void) {  
    OS_STAT_Disable();  
}
```

23.2.1.2 OS_STAT_Enable()

Description

Enables the kernel profiling (for an indefinite time).

Prototype

```
void OS_STAT_Enable(void);
```

Additional information

The function OS_STAT_Disable() may be used to stop profiling.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void StartProfiling(void) {  
    OS_STAT_Enable();  
}
```

23.2.1.3 OS_STAT_GetExecTime()

Description

Returns the total task execution time.

Prototype

```
OS_U32 OS_STAT_GetExecTime(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

The total task execution time in timer cycles.

Additional information

This function only returns valid values when profiling was enabled before by a call to `OS_STAT_Enable()`. If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

The task execution time is counted internally as a 32-bit value. This counter could overflow depending on the actual task execution time and timer frequency. For example the counter overflows after ~43 seconds if the task runs at 100% CPU load and the system tick hardware timer runs at 100 MHz.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
OS_U32 ExecTime;

void MyTask(void) {
    OS_STAT_Enable();
    while (1) {
        ExecTime = OS_STAT_GetExecTime(NULL);
        OS_TASK_Delay(100);
    }
}
```

23.2.1.4 OS_STAT_GetLoad()

Description

Calculates the current task's CPU load in permille.

Prototype

```
int OS_STAT_GetLoad(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block.

Return value

The current task's CPU load in permille.

Additional information

`OS_STAT_GetLoad()` requires `OS_STAT_Sample()` to be periodically called.

`OS_STAT_GetLoad()` cannot be used from multiple contexts simultaneously because it utilizes a global variable. It must e.g. not be called from a task and an ISR simultaneously.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`164: OS_ERR_OS_INT_ENTER_NOT_CALLED`

`167: OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Profiling* on page 473.

23.2.1.5 OS_STAT_Sample()

Description

Starts the kernel profiling and calculates the absolute task run time for all tasks since the last call to `OS_STAT_Sample()`.

Prototype

```
void OS_STAT_Sample(void);
```

Additional information

Unless profiling has been activated before by a call to `OS_STAT_Enable()`, `OS_STAT_Sample()` enables profiling for 5000 consecutive system ticks. The next call to `OS_STAT_Sample()` must be performed within this period. To retrieve the calculated CPU load in permille, use the embOS function `OS_STAT_GetLoad()`.

`OS_STAT_Sample()` cannot be used from multiple contexts simultaneously because it utilizes a global variable. It must e.g. not be called from a task and an ISR simultaneously.

The sample period is counted internally in hardware timer cycles as a 32-bit value. This counter could overflow depending on the timer frequency. For example the counter overflows after ~43 seconds if the system tick hardware timer runs at 100 MHz. The next call to `OS_STAT_Sample()` must be performed within this period.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
```

```
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *Profiling* on page 473.

23.3 CPU load measurement

The CPU load measurement routines serve as an addition to the above profiling API. With it the total CPU load can easily be measured by the application. This can be helpful in situations where other profiling tools are not available.

Example

```
#include "RTOS.h"
#include <stdio.h>

static OS_STACKPTR int StackPrint[128], StackWasteCPUtime[128];
static OS_TASK      TCBPrint, TCBWasteCPUtime;

static void PrintTask(void) {
    unsigned int CPUload;

    while (1) {
        CPUload = OS_STAT_GetLoadMeasurement();
        printf("CPU usage: %d%%\n", CPUload);
        OS_TASK_Delay(500); // Wait for next sample to print
    }
}

static void WasteCPUtimeTask(void) {
    while (1) {
        //
        // Cause ~50% CPU load
        //
        OS_TASK_Delay_us(1000);
        OS_TASK_Delay(1);
    }
}

/*****
 *
 *     main()
 */
int main(void) {
    OS_Init(); // Initialize embOS
    OS_InitHW(); // Initialize the hardware
    OS_STAT_AddLoadMeasurement(500, 1, 0);
    OS_TASK_CREATE(&TCBWasteCPUtime, "Waste CPU time task", 10,
        WasteCPUtimeTask, StackWasteCPUtime);
    OS_TASK_CREATE(&TCBPrint, "Print task", 20,
        PrintTask, StackPrint);
    OS_Start(); // Start multitasking
    return 0;
}
```

```
void OS_Idle(void) {
    while (1) {
        OS_INC_IDLE_CNT();
    }
}
```

23.3.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_STAT_AddLoadMeasurement()</code>	Initializes the periodic CPU load measurement.	•	•			
<code>OS_STAT_AddLoadMeasurementEx()</code>	Initializes the periodic CPU load measurement.	•	•			
<code>OS_STAT_GetLoadMeasurement()</code>	Returns the result of the CPU load measurement.	•	•		•	•

23.3.1.1 OS_STAT_AddLoadMeasurement()

Description

Initializes the periodic CPU load measurement.

Prototype

```
void OS_STAT_AddLoadMeasurement(OS_TIME Period,
                                OS_U8   AutoAdjust,
                                OS_I32   DefaultMaxValue);
```

Parameters

Parameter	Description
Period	Measurement period in embOS system ticks.
AutoAdjust	If not zero, the measurement is auto-adjusted once initially.
DefaultMaxValue	May be used to set a default counter value when AutoAdjust is not used. (See additional information)

Additional information

May be used to start the calculation of the total CPU load of an application.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Additional information

This function is not available in `OS_LIBMODE_SAFE`.

The CPU load is the percentage of CPU time that was not spent in `OS_Idle()`. To measure it, `OS_STAT_AddLoadMeasurement()` creates a task running at highest priority. This task periodically suspends itself by calling `OS_TASK_Delay(Period)`. Each time it is resumed, it calculates the CPU load through comparison of two counter values.

For this calculation, it is required that `OS_Idle()` gets executed and increments a counter by calling `OS_INC_IDLE_CNT()`. Furthermore, the calculation will fail if `OS_Idle()` starts a power save mode of the CPU. `OS_Idle()` must therefore be similar to:

```
void OS_Idle(void) {
    while (1) {
        OS_INC_IDLE_CNT();
    }
}
```

The maximum value of the idle counter is stored once at the beginning and is subsequently used for comparison with the current value of the counter each time the measurement task gets activated. For this comparison, it is assumed that the maximum value of the counter represents a CPU load of 0%, whereas a value of zero represents a CPU load of 100%. The maximum value of the counter can either be examined automatically, or may else be set manually. When [AutoAdjust](#) is non-zero, the task will examine the maximum value of the counter automatically. To do so, it will initially suspend all other tasks for the [Period](#)-time and will subsequently call `OS_TASK_Delay(Period)`. This way, the entire period is spent in `OS_Idle()` and the counter incremented in `OS_Idle()` reaches its maximum value, which is then saved and used for comparisons. Especially when the initial suspension of all tasks

for the `Period`-time is not desired, the maximum counter value may also be configured manually via the parameter `DefaultMaxValue` when `AutoAdjust` is zero.

The global variable `OS_IdleCnt` holds the counter value used for CPU load measurement. It may be helpful when examining the appropriate `DefaultMaxValue` for the manual configuration of `OS_STAT_AddLoadMeasurement()`.

The global variable `OS_IdleCnt` is implemented as a 32-bit signed integer value. embOS internally calculates the delta between two `OS_IdleCnt` values. The delta must not exceed `0x7FFFFFFF`. If your CPU increments `OS_IdleCnt` too fast, you might need to choose a shorter time period with `OS_STAT_AddLoadMeasurement()/OS_STAT_AddLoadMeasurementEx()`.

```
volatile OS_I32 OS_IdleCnt;
```

The appropriate `DefaultMaxValue` may, for example, be examined prior to creating any other task, similar to the given sample below:

```
void MainTask(void) {
    OS_I32 DefaultMax;
    OS_TASK_Delay(100);
    DefaultMax = OS_IdleCnt; /* This value can be used as DefaultMaxValue. */
    /* Now other tasks can be created and started. */
}
```

Note

`OS_STAT_AddLoadMeasurement()` starts a CPU load task with a predefined task stack size of 64 integers. The stack size is sufficient for most applications. However, in some situations more task stack may be required. In that case please use `OS_STAT_AddLoadMeasurementEx()` which allows to use an application defined stack size.

Example

Please refer to the example in the introduction of chapter *CPU load measurement* on page 485.

23.3.1.2 OS_STAT_AddLoadMeasurementEx()

Description

Initializes the periodic CPU load measurement.

Prototype

```
void OS_STAT_AddLoadMeasurementEx(OS_TIME Period,
                                  OS_U8   AutoAdjust,
                                  OS_I32   DefaultMaxValue,
                                  void     OS_STACKPTR *pStack,
                                  OS_UINT  StackSize);
```

Parameters

Parameter	Description
<code>Period</code>	Measurement period in embOS system ticks.
<code>AutoAdjust</code>	If not zero, the measurement is auto-adjusted once initially.
<code>DefaultMaxValue</code>	May be used to set a default counter value when <code>AutoAdjust</code> is not used. (See additional information)
<code>pStack</code>	Pointer to the stack.
<code>StackSize</code>	Size of the stack.

Additional information

May be used to start the calculation of the total CPU load of an application.

`OS_STAT_AddLoadMeasurementEx()` allows to define the stack location and size for the CPU load task which is started automatically by `OS_STAT_AddLoadMeasurementEx()`

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Additional information

Please refer to the description of `OS_STAT_AddLoadMeasurement()` for more details. This function is not available in `OS_LIBMODE_SAFE`.

Example

```
static OS_STACKPTR int TaskStack[128], MeasureStack[128];

int main(void) {
    OS_Init();    // Initialize embOS
    OS_Inithw(); // Initialize hardware for embOS
    OS_STAT_AddLoadMeasurementEx(1000, 1, 0, MeasureStack, 128);
    OS_TASK_CREATE(&TCB, "Task", 100, Task, TaskStack);
    OS_Start();
    return 0;
}
```

23.3.1.3 OS_STAT_GetLoadMeasurement()

Description

Returns the result of the CPU load measurement.

Prototype

```
int OS_STAT_GetLoadMeasurement(void);
```

Return value

The total CPU load in percent.

Additional information

OS_STAT_GetLoadMeasurement() delivers correct results if

- the CPU load measurement was started before by calling OS_STAT_AddLoadMeasurement() with auto-adjustment or else with a correct default value, and
- OS_Idle() updates the measurement by calling OS_INC_IDLE_CNT().

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
```

```
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

The global variable OS_CPU_Load holds the total CPU load as a percentage. It may prove helpful to monitor the variable in a debugger with live-watch capability during development.

```
volatile OS_UINT OS_CPU_Load;
```

This variable will not contain correct results unless the CPU load measurement was started by a call to OS_STAT_AddLoadMeasurement(). This function is not available in OS_LIBMODE_SAFE.

Example

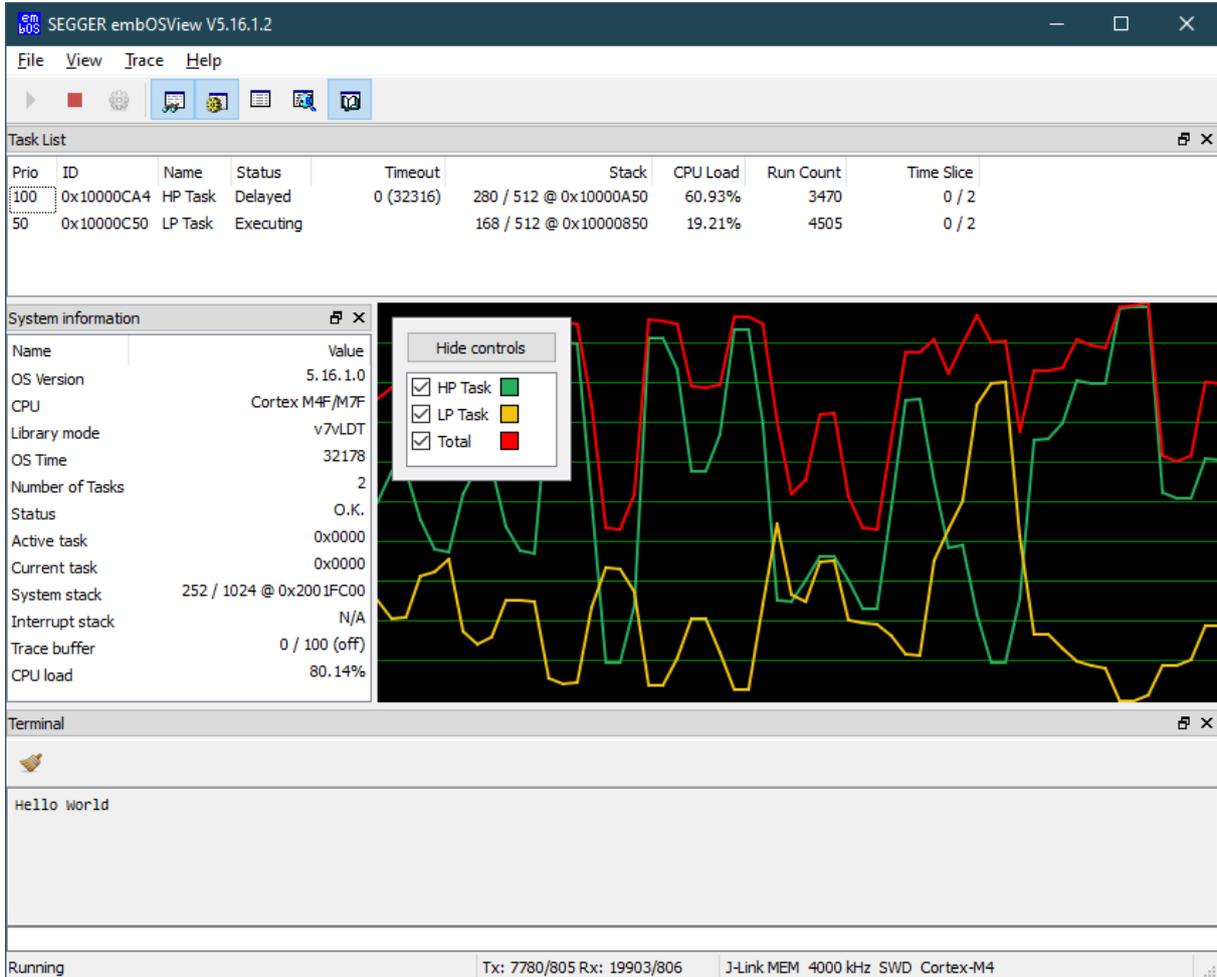
Please refer to the example in the introduction of chapter *CPU load measurement* on page 485.

Chapter 24

embOSView

24.1 Introduction

The embOSView utility is a helpful tool for analyzing the running target application. It is shipped with embOS as `embOSView.exe` and runs on Windows.



Most often, a serial interface (UART) is used for the communication with the target hardware. Alternative communication channels include Ethernet, memory read/write for Cortex-M and RX CPUs, as well as DCC for ARM7/9 and Cortex-A/R CPUs. The hardware dependent routines and defines available for communication with embOSView are implemented in the source file `RTOSInit.c`. Details on how to modify this file are also given in chapter *Setup target for communication* on page 498.

The communication API is not available in the embOS library mode `OS_LIBMODE_SAFE`.

Note

For embOS V5.06 and later, `OS_TIME_ConfigSysTimer()` must be called before using embOSView.

Note

The embOS target communication buffer per default is set to 200 bytes which limits the amount of displayed tasks in embOSView. If you use more than 48 tasks please modify `OS_COM_OUT_BUFFER_SIZE` accordingly. There is no such limitation in embOSView.

24.1.1 Task list window

embOSView shows the state of every task created by the target application in the Task list window. The information shown depends on the library mode that is used in your application.

Item	Description	Builds
Prio	Current priority of task.	All
Id	Task ID, which is the address of the task control block.	All
Name	Name assigned during creation.	All
Status	Current state of task (ready, executing, delay, reason for suspension).	All
Data	Depends on status.	All
Timeout	Time in ticks until the task is ready for execution again. The value in parenthesis shows the absolute point in time at which the timeout expires.	All
Stack	Used stack size/max. stack size/stack location.	S, SP, D, DP, DT
CPUload	Percentage CPU load caused by task.	SP, DP, DT
Run Count	Number of activations since reset.	SP, DP, DT
Time slice	Round-robin time slice	All

The **Task list window** is helpful in analyzing the stack usage and CPU load for every running task.

24.1.2 System variables window

embOSView shows the state of major system variables in the System variables window. The information shown depends on the library mode that is used by your application:

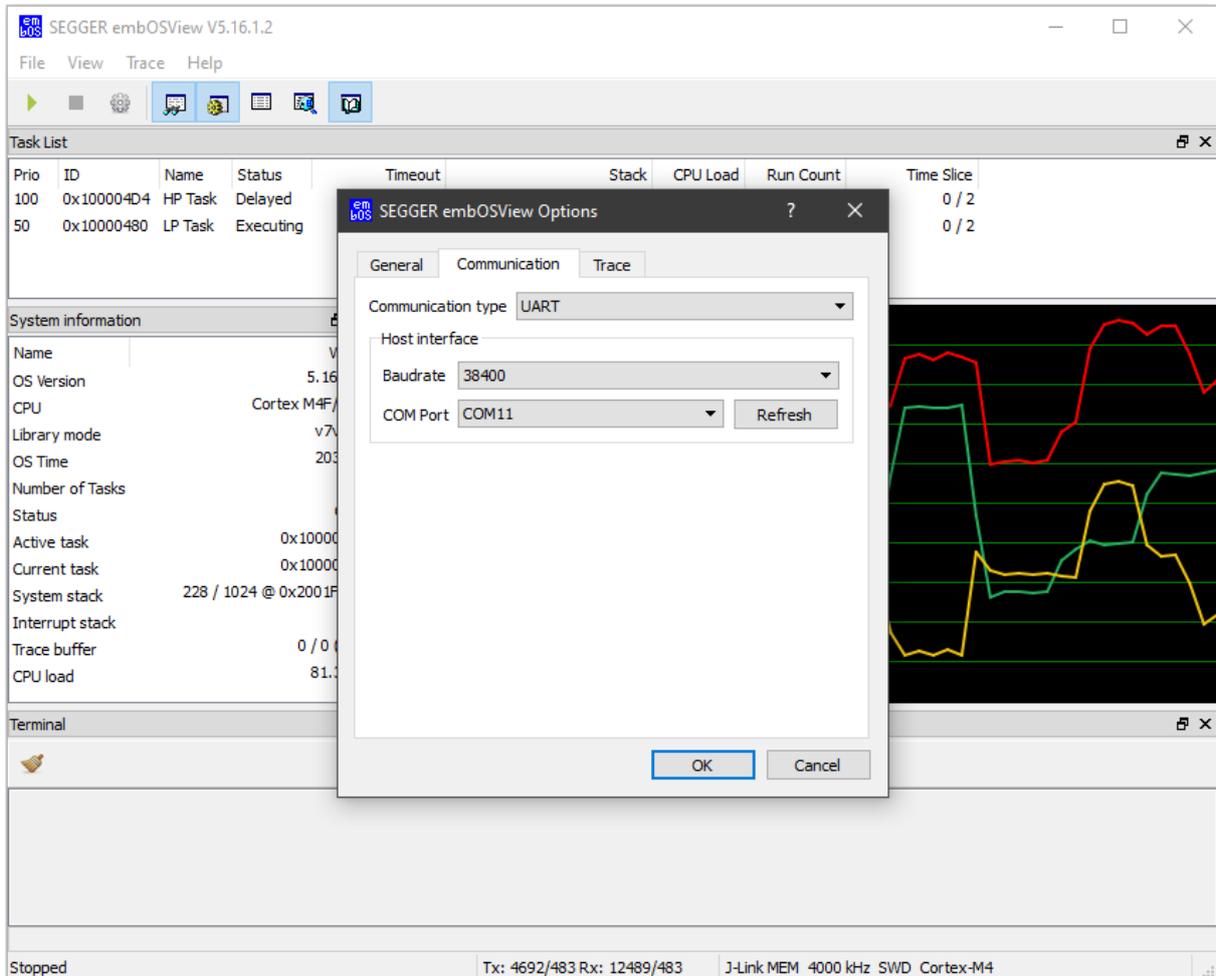
Item	Description	Library mode
OS_VERSION	Current version of embOS.	All
CPU	Target CPU and compiler.	All
LibMode	Library mode used for target application.	All
OS_Time	Current system time in system ticks.	All
OS_NumTasks	Current number of defined tasks.	All
OS_Global.Status	Current error code (or O.K.).	All
OS_pActiveTask	Active task that should be running.	SP, D, DP, DT
OS_pCurrentTask	Actual currently running task.	SP, D, DP, DT
SysStack	Used size/max. size/location of system stack.	SP, DP, DT
IntStack	Used size/max. size/location of interrupt stack.	SP, DP, DT
TraceBuffer	Current count/maximum size and current state of trace buffer.	DT
CPU load	Current CPU total load.	SP, DP, DT

24.2 Setup embOSView for communication

When the communication to embOSView is enabled in the target application, embOSView may be used to analyze the running application. The communication channel of embOSView must be setup according to the communication channel which was selected in the project.

24.2.1 Select a UART for communication

Start embOSView and open the Setup menu:



In the Communication tab, choose "UART" in the Type selection list box.

In the Host interface box, select the desired baud rate for communication and the COM port of the PC that should be connected to the target board. The default baud rate of all projects shipped with embOS is 38,400. The ComPort list box lists all currently available COM ports for the PC that embOSView is executed on.

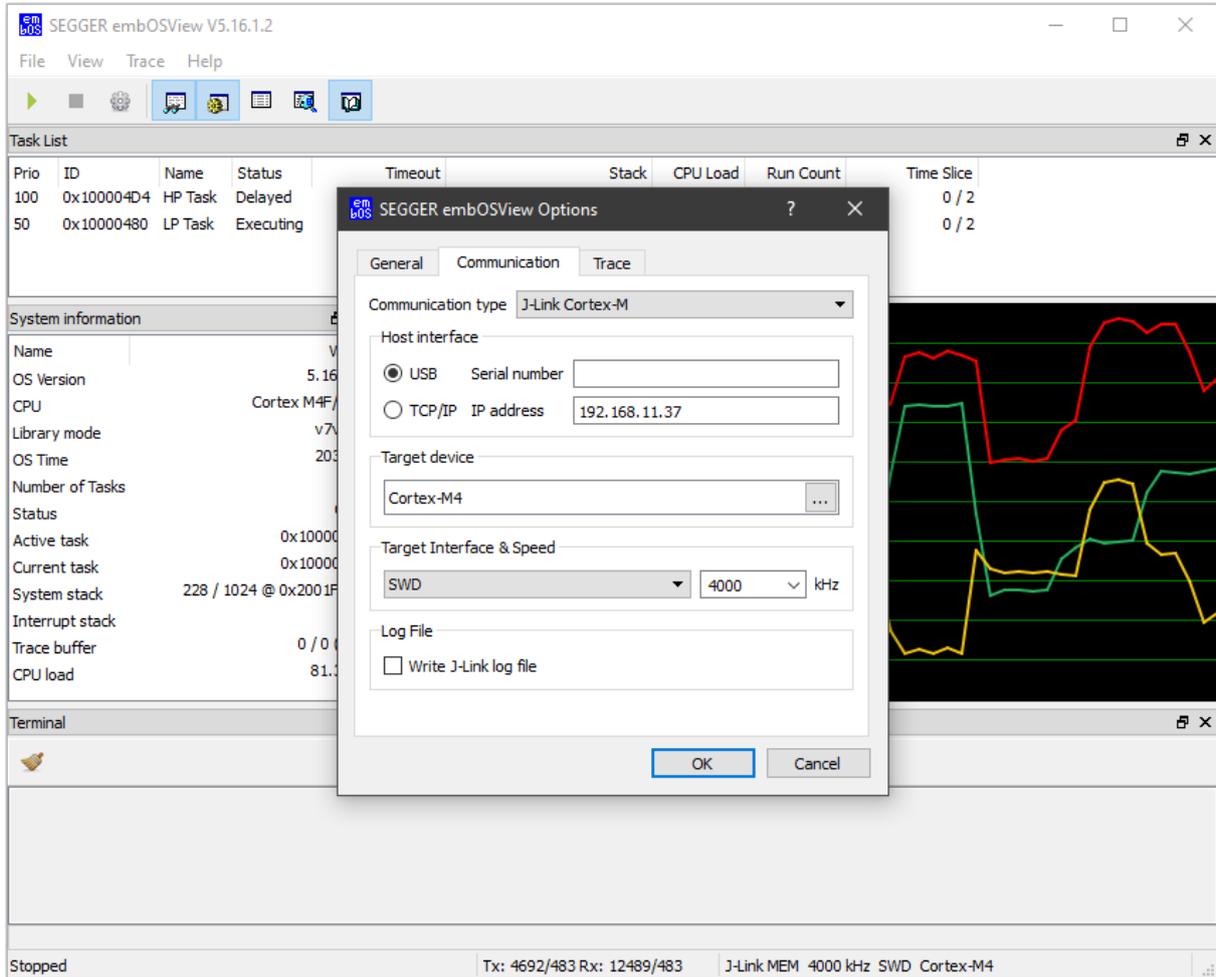
The serial communication will work when the target is running stand-alone, or during a debug session when the target is connected to a debugger.

The serial connection can be used when the target board has a spare UART port and the UART functions are included in the application.

24.2.2 Select J-Link for communication

embOS supports a communication channel to embOSView which uses J-Link to communicate with the running application. embOSView V3.82g or higher and a J-Link DLL is required to use J-Link for communication.

To select this communication channel, start embOSView and open the Setup menu:



In the Communication tab, choose "J-Link Cortex-M (memory access)", "J-Link RX (memory access)" or "J-Link ARM7/9/11 (DCC)" in the `Type` selection list box.

In the `Host interface` box, select the USB or TCP/IP channel to be used to communicate with your J-Link.

In the `Target interface` box, select the communication speed of the target interface and the physical target connection (i.e. JTAG, SWD, or FINE).

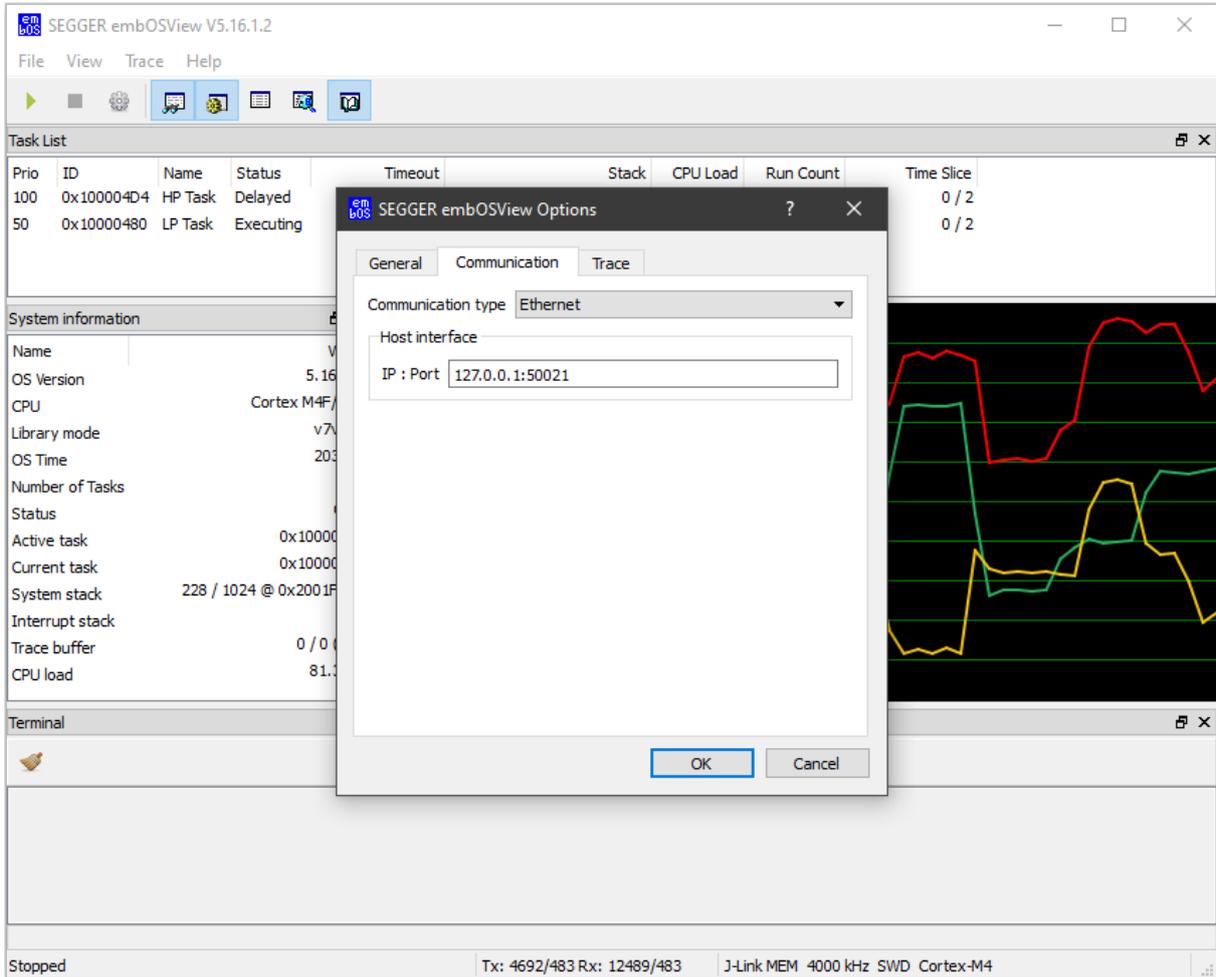
In the `Log File` box, choose whether a log file should be created and define its file name and location.

The `JTAG Chain` box allows the selection of any specific device in a JTAG scan chain with multiple devices. Currently, up to eight devices in the scan chain are supported. Two values must be configured: the position of the target device in the scan chain, and the total number of bits in the instruction registers of all devices before the target device (IR len). Target position is numbered in descending order, which means the target that is closest to J-Link's TDI is in the highest position (max. 7), while the target closest to J-Link's TDO is in the lowest position (always 0). Upon selecting the position, the according IR len is determined automatically, which should succeed for most of all target devices. IR len may also be written manually, which is mandatory in case automatic detection was not successful. For further information, please refer to the J-Link / J-Trace User Guide (UM08001, chapter "JTAG interface").

24.2.3 Select Ethernet for communication

embOS supports a communication channel to embOSView which uses Ethernet to communicate with the running application. A TCP/IP stack, for example SEGGER's emNET stack, is required to use Ethernet for communication.

To select this communication channel, start embOSView and open the Setup menu:



In the Communication tab, choose "Ethernet" in the Type selection list box.

In the Host interface box, configure the IP address of your target and the port number 50021.

24.2.4 Use J-Link for communication and debugging in parallel

J-Link can be used to communicate with embOSView during a running debug session that uses the same J-Link as debug probe. To avoid potential incompatibilities, the target interface settings for J-Link should be identical in both the debugger settings and embOSView target interface settings.

To use embOSView during a debug session, proceed as follows:

- Examine the target interface settings in the debugger settings of the project.
- Before starting the debugger, start embOSView and configure the same target interface settings as found in the debugger settings.
- Close embOSView.
- Start the debugger.
- Restart embOSView.

J-Link will now communicate with the debugger and embOSView will simultaneously communicate with embOS via J-Link.

24.3 Setup target for communication

The communication to embOSView can be enabled by setting the compile time switch `OS_VIEW_IFSELECT` to an interface define, e.g. inside the project settings or in the configuration file `OS_Config.h`. If `OS_VIEW_IFSELECT` is defined to `OS_VIEW_DISABLED`, the communication is disabled. In the `RTOSInit.c` files, the `OS_VIEW_IFSELECT` switch is set to a specific interface unless overwritten by project options.

By default, the `OS_Config.h` file sets the compile time switch `OS_VIEW_IFSELECT` to `OS_VIEW_DISABLED` when `DEBUG=1` is not defined. Therefore, in the embOS start projects, the communication is enabled per default for Debug configurations, while it is disabled for Release configurations.

<code>OS_VIEW_IFSELECT</code>	Communication interface
<code>OS_VIEW_DISABLED</code>	Disabled
<code>OS_VIEW_IF_UART</code>	UART
<code>OS_VIEW_IF_JLINK</code>	J-Link
<code>OS_VIEW_IF_ETHERNET</code>	Ethernet

24.3.1 Select a UART for communication

Set the compile time switch `OS_VIEW_IFSELECT` to `OS_VIEW_IF_UART` by project option/compiler preprocessor or in `RTOSInit.c` to enable the communication via UART. This assumes the necessary UART routines are implemented. Usually these are implemented in a file called `BSP_UART.c`. If the file `BSP_UART.c` does not exist in your board support package, you can create it on your own or contact the embOS support. Please have a look in the chapter *Board Support Packages* on page 574 for more details.

24.3.2 Select J-Link for communication

Per default, J-Link is selected as communication device in most embOS start projects, if available.

The compile time switch `OS_VIEW_IFSELECT` is predefined to `OS_VIEW_IF_JLINK` in the CPU specific `RTOSInit.c` files, thus J-Link communication is selected per default unless overwritten by project / compiler preprocessor options.

24.3.3 Select Ethernet for communication

Set the compile time switch `OS_VIEW_IFSELECT` to `OS_VIEW_IF_ETHERNET` by project / compiler preprocessor options or in `RTOSInit.c` to switch the communication to Ethernet.

This communication mode is only available when emNET or a different TCP/IP stack is included with the project. Also, the file `UDP_Process.c` must be added to your project and the file `UDPCOM.h` to your `Start\Inc` folder. These files are not shipped with embOS by default, but are available on request. Using a different TCP/IP stack than emNet requires modifications to `UDP_Process.c`. Subsequently, the `RTOSInit.c` needs to be modified to include the below section:

```
#elif (OS_VIEW_IFSELECT == OS_VIEW_IF_ETHERNET)
#include "UDPCOM.h"

/*****
 *
 *      OS_COM_Send1()
 *
 *  Function description
 *      Sends one character via UDP
 */
void OS_COM_Send1(OS_U8 c) {
    UDP_Process_Send1(c);
}
```

```
/******  
*  
*     OS_COM_Init()  
*  
*  Function description  
*     Initializes UDP communication for embOSView  
*/  
void OS_COM_Init(void) {  
    UDP_Process_Init();  
}  
#endif
```

24.3.4 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_COM_ClearTxActive()</code>	Resets the embOS communication variables.		•		•	
<code>OS_COM_GetNextChar()</code>	This routine is used to retrieve the next character to be sent.		•		•	
<code>OS_COM_OnRx()</code>	<code>OS_COM_OnRx()</code> informs embOS about a received byte from <code>embOSView</code> .		•		•	
<code>OS_COM_OnTx()</code>	<code>OS_COM_OnTx()</code> returns whether there are more bytes to send.		•		•	

24.3.4.1 OS_COM_ClearTxActive()

Description

Resets the embOS communication variables.

Prototype

```
void OS_COM_ClearTxActive(void);
```

Additional information

OS_COM_ClearTxActive() is used to reset the embOS communication variables. OS_COM_ClearTxActive can e.g. be called after a communication issue. OS_COM_ClearTxActive() is usually not called by the application but from the embOSView communication routines which are part of the board support package.

Example

```
void ResetCom(void) {  
    OS_COM_ClearTxActive();  
}
```

24.3.4.2 OS_COM_GetNextChar()

Description

This routine is used to retrieve the next character to be sent. It may be called for communication by a non interrupt driven implementation. The user should be aware, that the function may enable interrupts and may cause a task switch.

Prototype

```
OS_INT OS_COM_GetNextChar(void);
```

Return value

- ≥ 0 The character to be sent.
- < 0 Buffer empty, no more bytes to be sent.

Example

```
void OS_ISR_Tx(void) {  
    if (OS_COM_GetNextChar() >= 0u) {  
        SendByte(c);  
    }  
}
```

24.3.4.3 OS_COM_OnRx()

Description

OS_COM_OnRx() informs embOS about a received byte from embOSView.

Prototype

```
void OS_COM_OnRx(OS_U8 Data);
```

Parameters

Parameter	Description
Data	Received byte.

Additional information

This routine is normally called from the rx interrupt service handler when a character was received.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

180: OS_ERR_NESTED_RX_INT

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void OS_ISR_Rx(void) {
    OS_U8 c;
    c = UART_RX_REGISTER;
    OS_COM_OnRx(c);
}
```

24.3.4.4 OS_COM_OnTx()

Description

OS_COM_OnTx() returns whether there are more bytes to send. This routine is normally called from the transmitter buffer empty interrupt service handler. In case there are more bytes to send, OS_COM_OnTx() calls OS_COM_Send1() to send the next byte.

Prototype

```
OS_U8 OS_COM_OnTx(void);
```

Return value

= 0 There are more bytes to be sent.
≠ 0 Buffer empty, no more bytes to be sent.

Example

```
void OS_ISR_Tx(void) {  
    if (OS_COM_OnTx() != 0u) {  
        UART_TX_INT_ENABLE_REGISTER = 0;  
    }  
}
```

24.4 Sharing the SIO for terminal I/O

The serial input/output (SIO) used by embOSView may also be used by the application at the same time for both input and output. Terminal input is often used as keyboard input, where terminal output may be used for outputting debug messages. Input and output is done via the **Terminal window**, which can be shown by selecting **View/Terminal** from the menu.

To ensure communication via the **Terminal window** in parallel with the viewer functions, the application uses the function `OS_COM_SendString()` for sending a string to the Terminal window and the function `OS_COM_SetRxCallback()` to hook a reception-routine that receives one byte.

24.4.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_COM_SendString()</code>	Sends a string to the embOSView terminal window.	•	•	•		
<code>OS_COM_SetRxCallback()</code>	Sets a callback routine for receiving one character from embOSView.	•	•			•

24.4.1.1 OS_COM_SendString()

Description

Sends a string to the embOSView terminal window.

Prototype

```
void OS_COM_SendString(const char* s);
```

Parameters

Parameter	Description
s	Pointer to a null-terminated string that should be sent to the terminal window.

Additional information

This function utilizes the target-specific function `OS_COM_Send1()`. `OS_COM_SendString()` deactivates the current active task until the string is sent.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

160: `OS_ERR_ILLEGAL_IN_ISR`

161: `OS_ERR_ILLEGAL_IN_TIMER`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_COM_SendString("Task started.\n");
    while (1) {
        OS_TASK_Delay(100);
    }
}
```

24.4.1.2 OS_COM_SetRxCallback()

Description

Sets a callback routine for receiving one character from embOSView.

Prototype

```
OS_ROUTINE_CHAR *OS_COM_SetRxCallback(OS_ROUTINE_CHAR* pFRXCallback);
```

Parameters

Parameter	Description
<code>pFRXCallback</code>	Pointer to the routine of type <code>OS_ROUTINE_CHAR</code> that shall be called when one character is received over the serial interface.

Return value

This is the pointer to the callback routine that was hooked before the call.

Additional information

The user callback routine is called from embOS. The received character is passed as a parameter.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

160: OS_ERR_ILLEGAL_IN_ISR

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

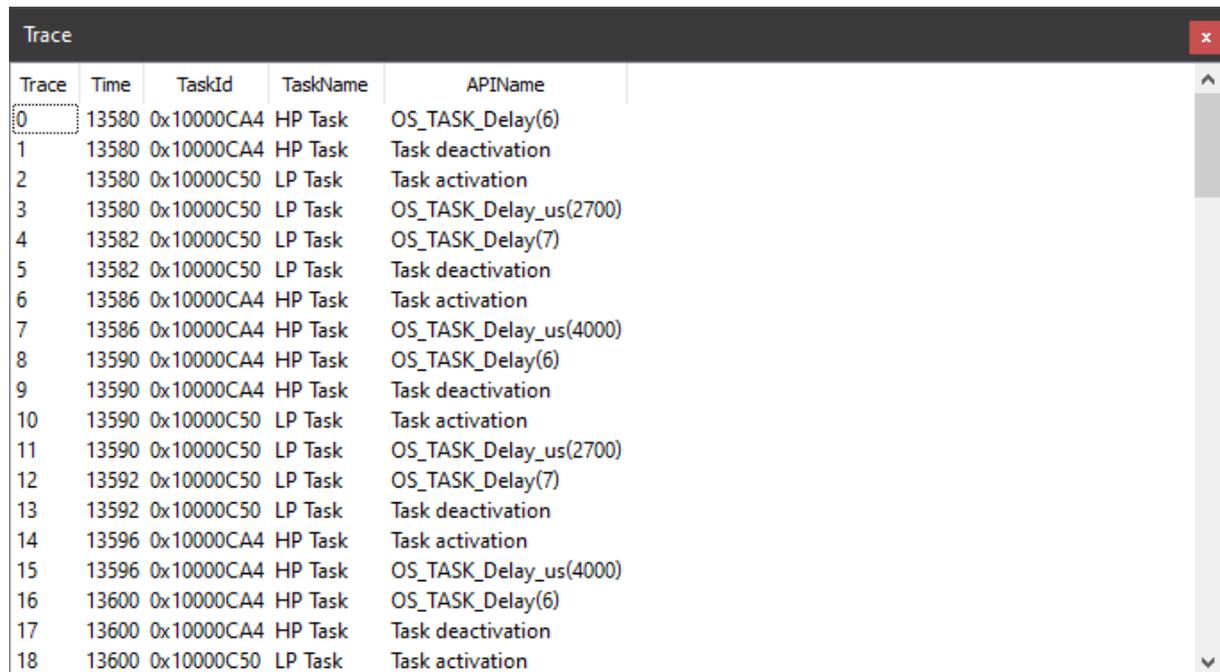
```
//
// Routine to be called from Rx-interrupt
//
static void _OnRx(OS_U8 Data) {
    DisplayChar(Data);
}

int main(void) {
    ...
    OS_COM_SetRxCallback(&_OnRx);
    ...
    return 0;
}
```

24.5 embOSView API trace

embOS contains a trace feature for API calls. This requires the use of the trace build libraries in the target application.

The trace build libraries implement a buffer for 100 trace entries. Tracing of API calls can be started and stopped from embOSView via the **Trace menu**, or from within the application by using the functions `OS_TRACE_Enable()` and `OS_TRACE_Disable()`. Individual filters may be defined to determine which API calls should be traced for different tasks or from within interrupt or timer routines. Once the trace is started, the API calls are recorded in the trace buffer, which is periodically read by embOSView. The result is shown in the **Trace window**:



Trace	Time	TaskId	TaskName	APIName
0	13580	0x10000CA4	HP Task	OS_TASK_Delay(6)
1	13580	0x10000CA4	HP Task	Task deactivation
2	13580	0x10000C50	LP Task	Task activation
3	13580	0x10000C50	LP Task	OS_TASK_Delay_us(2700)
4	13582	0x10000C50	LP Task	OS_TASK_Delay(7)
5	13582	0x10000C50	LP Task	Task deactivation
6	13586	0x10000CA4	HP Task	Task activation
7	13586	0x10000CA4	HP Task	OS_TASK_Delay_us(4000)
8	13590	0x10000CA4	HP Task	OS_TASK_Delay(6)
9	13590	0x10000CA4	HP Task	Task deactivation
10	13590	0x10000C50	LP Task	Task activation
11	13590	0x10000C50	LP Task	OS_TASK_Delay_us(2700)
12	13592	0x10000C50	LP Task	OS_TASK_Delay(7)
13	13592	0x10000C50	LP Task	Task deactivation
14	13596	0x10000CA4	HP Task	Task activation
15	13596	0x10000CA4	HP Task	OS_TASK_Delay_us(4000)
16	13600	0x10000CA4	HP Task	OS_TASK_Delay(6)
17	13600	0x10000CA4	HP Task	Task deactivation
18	13600	0x10000C50	LP Task	Task activation

Every entry in the **Trace list** is recorded with the actual system time. In case of calls or events from tasks, the task ID (**TaskId**) and task name (**TaskName**) (limited to 15 characters) are also recorded. Parameters of API calls are recorded if possible, and are shown as part of the **APIName** column. In the example above, this can be seen with `OS_TASK_Delay(6)`. Once the trace buffer is full, trace is automatically stopped. The **Trace list** and buffer can be cleared from embOSView.

Example

```
#define MY_TRACE_ID 100

void Task(void) {
    OS_TASK_Delay(100);
    OS_TRACE_Void(MY_TRACE_ID);
    OS_TRACE_DisableAll();
    while (1) {
        OS_TASK_Delay(100);
    }
}

int main(void) {
    OS_Init();
    OS_Inithw();
    OS_TRACE_EnableAll();
    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);
    OS_Start();
    return 0;
}
```

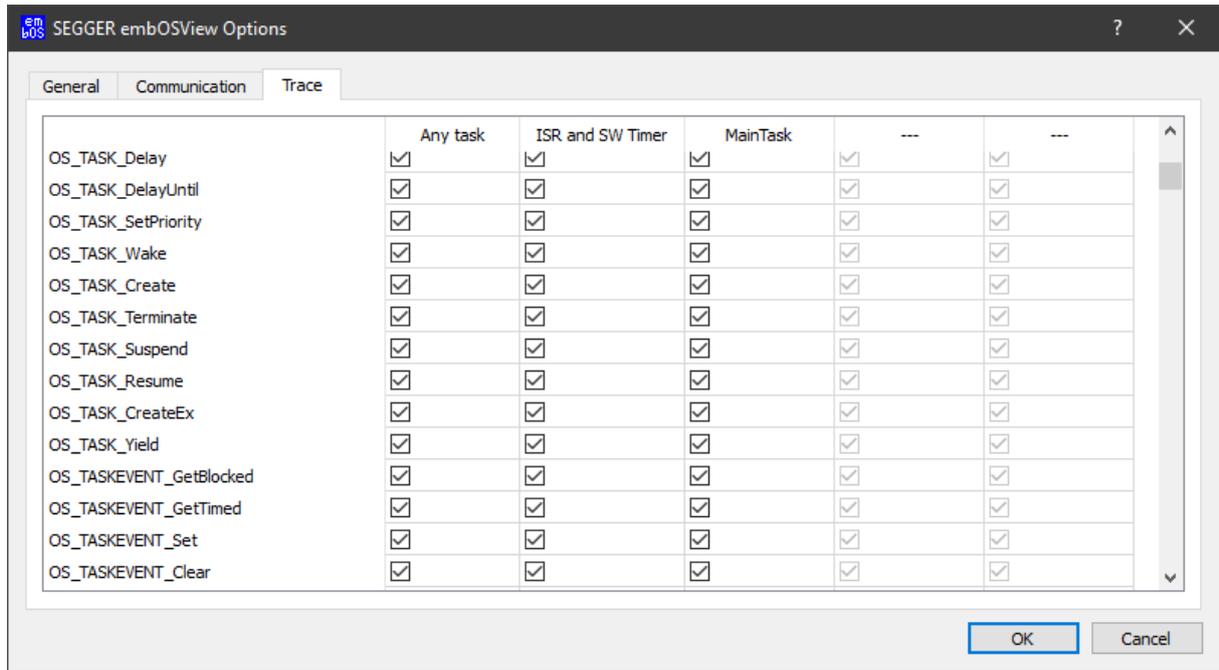
24.5.1 Setup API trace from embOSView

Three different kinds of trace filters are defined for tracing. These filters can be set up from embOSView via the menu **Options/Setup/Trace**.

Filter “Any task” is not task-specific and records all specified events regardless of the task. As the Idle loop is not a task, calls from within the idle loop are not traced.

Filter “ISR and SW Timer” is specific for interrupt service routines, software timers and all calls that occur outside a running task. These calls may come from the idle loop or during startup when no task is running.

Three further **Custom filters** allow trace of API calls from named tasks.



To enable or disable a filter, simply check or uncheck the corresponding checkboxes. For any of these five filters, individual API functions can be enabled or disabled by checking or unchecking the corresponding checkboxes in the list.

Custom filters allow tracing of task-specific API calls. A task name can therefore be specified for each of these filters. In the example above, a custom filter is configured to trace calls from the task called `MainTask`. After the settings are saved, the new settings are sent to the target application.

24.5.2 Trace filter setup API

Tracing of API or user function calls can be started or stopped from embOSView. By default, trace is initially disabled in an application program. It may be helpful to control recording of trace events directly from the application, using the following functions.

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TRACE_Enable()</code>	Enables tracing of filtered API calls.	•	•		•	•
<code>OS_TRACE_EnableAll()</code>	Sets up Filter 0 (any task), enables tracing of all API calls and then enables the trace function.	•	•		•	•
<code>OS_TRACE_EnableId()</code>	Sets the specified ID value in Filter 0 (any task), thus enabling trace of the specified function, but does not start trace.	•	•		•	•
<code>OS_TRACE_EnableFilterId()</code>	Sets the specified ID value in the specified trace filter, thus enabling trace of the specified function, but does not start trace.	•	•		•	•
<code>OS_TRACE_Disable()</code>	Disables tracing of filtered API and user function calls.	•	•		•	•
<code>OS_TRACE_DisableAll()</code>	Sets up Filter 0 (any task), disables tracing of all API calls and also disables trace.	•	•		•	•
<code>OS_TRACE_DisableId()</code>	Resets the specified ID value in Filter 0 (any task), thus disabling trace of the specified function, but does not stop trace.	•	•		•	•
<code>OS_TRACE_DisableFilterId()</code>	Resets the specified ID value in the specified trace filter, thus disabling trace of the specified function, but does not stop trace.	•	•		•	•

24.5.2.1 OS_TRACE_Enable()

Description

Enables tracing of filtered API calls.

Prototype

```
void OS_TRACE_Enable(void);
```

Additional information

The trace filter conditions must be set up before calling this function. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the pre-processor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
```

```
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
int main(void) {
    OS_Init();           // Initialize embOS
    OS_Inithw();        // Initialize hardware for embOS
    OS_TRACE_EnableId(OS_TRACE_ID_TASK_DELAY); // Enable trace for OS_TASK_DELAY()
    OS_TRACE_Enable();  // Enable tracing
    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);
    OS_Start();
    return 0;
}
```

24.5.2.2 OS_TRACE_EnableAll()

Description

Sets up Filter 0 (any task), enables tracing of all API calls and then enables the trace function.

Prototype

```
void OS_TRACE_EnableAll(void);
```

Additional information

The trace filter conditions of all the other trace filters are not affected. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the pre-processor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
int main(void) {
    OS_Init();           // Initialize embOS
    OS_Inithw();        // Initialize hardware for embOS
    OS_TRACE_EnableAll(void); // Enable trace
    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);
    OS_Start();
    return 0;
}
```

24.5.2.3 OS_TRACE_EnableId()

Description

Sets the specified ID value in Filter 0 (any task), thus enabling trace of the specified function, but does not start trace.

Prototype

```
void OS_TRACE_EnableId(OS_U8 id);
```

Parameters

Parameter	Description
<code>id</code>	ID value of API call that should be enabled for trace: $0 \leq id \leq 255$ Values from 0 to 99 and 128 to 255 are reserved for embOS.

Additional information

To enable trace of a specific embOS API function, you must use the correct Id value. These values are defined as symbolic constants in `RTOS.h`. This function may also enable trace of your own functions. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example of `OS_TRACE_Enable()`.

24.5.2.4 OS_TRACE_EnableFilterId()

Description

Sets the specified ID value in the specified trace filter, thus enabling trace of the specified function, but does not start trace.

Prototype

```
void OS_TRACE_EnableFilterId(OS_U8 FilterIndex,
                             OS_U8 id);
```

Parameters

Parameter	Description
<code>FilterIndex</code>	Index of the filter that should be affected: $0 \leq \text{FilterIndex} \leq 4$ 0 affects Filter 0 (any task) and so on.
<code>id</code>	ID value of API call that should be enabled for trace: $0 \leq \text{id} \leq 255$ Values from 0 to 99 and 128 to 255 are reserved for embOS.

Additional information

To enable trace of a specific embOS API function, you must use the correct Id value. These values are defined as symbolic constants in `RTOS.h`. This function may also be used for enabling trace of your own functions. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
int main(void) {
    OS_Init();           // Initialize embOS
    OS_Inithw();        // Initialize hardware for embOS
    OS_TRACE_EnableFilterId(1, OS_TRACE_ID_TASK_DELAY);
    OS_TRACE_Enable();  // Enable tracing
    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);
    OS_Start();
    return 0;
}
```

24.5.2.5 OS_TRACE_Disable()

Description

Disables tracing of filtered API and user function calls.

Prototype

```
void OS_TRACE_Disable(void);
```

Additional information

This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void StopTracing(void) {  
    OS_TRACE_Disable();  
}
```

24.5.2.6 OS_TRACE_DisableAll()

Description

Sets up Filter 0 (any task), disables tracing of all API calls and also disables trace.

Prototype

```
void OS_TRACE_DisableAll(void);
```

Additional information

The trace filter conditions of all the other trace filters are not affected, but tracing is stopped. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void StopTracing(void) {  
    OS_TRACE_DisableAll();  
}
```

24.5.2.7 OS_TRACE_DisableId()

Description

Resets the specified ID value in Filter 0 (any task), thus disabling trace of the specified function, but does not stop trace.

Prototype

```
void OS_TRACE_DisableId(OS_U8 id);
```

Parameters

Parameter	Description
<code>id</code>	ID value of API call that should be enabled for trace: $0 \leq id \leq 255$ Values from 0 to 99 and 128 to 255 are reserved for embOS.

Additional information

To disable trace of a specific embOS API function, you must use the correct Id value. These values are defined as symbolic constants in `RTOS.h`. This function may also be used for disabling trace of your own functions. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void StopTracing(void) {
    OS_TRACE_DisableId(OS_TRACE_ID_TASK_DELAY);
}
```

24.5.2.8 OS_TRACE_DisableFilterId()

Description

Resets the specified ID value in the specified trace filter, thus disabling trace of the specified function, but does not stop trace.

Prototype

```
void OS_TRACE_DisableFilterId(OS_U8 FilterIndex,
                             OS_U8 id);
```

Parameters

Parameter	Description
<code>FilterIndex</code>	Index of the filter that should be affected: $0 \leq \text{FilterIndex} \leq 4$ 0 affects Filter 0 (any task) and so on.
<code>id</code>	ID value of API call that should be enabled for trace: $0 \leq \text{id} \leq 255$ Values from 0 to 99 and 128 to 255 are reserved for embOS.

Additional information

To disable trace of a specific embOS API function, you must use the correct Id value. These values are defined as symbolic constants in `RTOS.h`. This function may also be used for disabling trace of your own functions. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`164: OS_ERR_OS_INT_ENTER_NOT_CALLED`

`167: OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void StopTracing(void) {
    OS_TRACE_DisableFilterId(1, OS_TRACE_ID_TASK_DELAY);
}
```

24.5.3 Trace record API

The following functions write data into the trace buffer. As long as only embOS API calls should be recorded, these functions are used internally by the trace build libraries. If, for some reason, you want to trace your own functions with your own parameters, you may call one of these routines.

All of these functions have the following points in common:

- To record data, trace must be enabled.
- An ID value in the range 100 to 127 must be used as the ID parameter. ID values from 0 to 99 and 128 to 255 are internally reserved for embOS.
- The events specified as ID must be enabled in trace filters.
- Active system time and the current task are automatically recorded together with the specified event.

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TRACE_Data()</code>	Writes an entry with ID and an integer as parameter into the trace buffer.	•	•		•	•
<code>OS_TRACE_DataPtr()</code>	Writes an entry with ID, an integer, and a pointer as parameter into the trace buffer.	•	•		•	•
<code>OS_TRACE_Ptr()</code>	Writes an entry with ID and a pointer as parameter into the trace buffer.	•	•		•	•
<code>OS_TRACE_PtrU32()</code>	Writes an entry with ID, a pointer, and a 32-bit unsigned integer as parameter into the trace buffer.	•	•		•	•
<code>OS_TRACE_U32Ptr()</code>	Writes an entry with ID, a 32-bit unsigned integer, and a pointer as parameter into the trace buffer.	•	•		•	•
<code>OS_TRACE_Void()</code>	Writes an entry identified only by its ID into the trace buffer.	•	•		•	•

Example

```
#define MY_TRACE_ID 100

void Task(void) {
    OS_TRACE_Data(MY_TRACE_ID, 42);
    OS_TRACE_DataPtr(MY_TRACE_ID, 42, OS_TASK_GetID());
    OS_TRACE_Ptr(MY_TRACE_ID, OS_TASK_GetID());
    OS_TRACE_U32Ptr(MY_TRACE_ID, 42, OS_TASK_GetID());
    OS_TRACE_Void(MY_TRACE_ID)
    while (1) {
        OS_Task_Delay(100);
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_InitHW(); // Initialize hardware for embOS
    OS_TRACE_EnableId(MY_TRACE_ID); // Enable trace for MY_TRACE_ID
    OS_TRACE_Enable(); // Enable tracing
    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);
    OS_Start();
    return 0;
}
```

24.5.3.1 OS_TRACE_Data()

Description

Writes an entry with ID and an integer as parameter into the trace buffer.

Prototype

```
void OS_TRACE_Data(OS_U8 id,
                  int v);
```

Parameters

Parameter	Description
<code>id</code>	ID value of API call that should be enabled for trace: $0 \leq id \leq 255$ Values from 0 to 99 and 128 to 255 are reserved for embOS.
<code>v</code>	Any integer value that should be recorded as parameter.

Additional information

The value passed as parameter will be displayed in the trace list window of embOSView. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in chapter *Trace record API* on page 519.

24.5.3.2 OS_TRACE_DataPtr()

Description

Writes an entry with ID, an integer, and a pointer as parameter into the trace buffer.

Prototype

```
void OS_TRACE_DataPtr(          OS_U8      id,
                              int         v,
                              volatile OS_CONST_PTR void *p);
```

Parameters

Parameter	Description
<code>id</code>	ID value of API call that should be enabled for trace: $0 \leq id \leq 255$ Values from 0 to 99 and 128 to 255 are reserved for embOS.
<code>v</code>	Any integer value that should be recorded as parameter.
<code>p</code>	Any void pointer that should be recorded as parameter.

Additional information

The values passed as parameters will be displayed in the trace list window of embOSView. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`164: OS_ERR_OS_INT_ENTER_NOT_CALLED`

`167: OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in chapter *Trace record API* on page 519.

24.5.3.3 OS_TRACE_Ptr()

Description

Writes an entry with ID and a pointer as parameter into the trace buffer.

Prototype

```
void OS_TRACE_Ptr(          OS_U8          id,
                       volatile OS_CONST_PTR void *p);
```

Parameters

Parameter	Description
<code>id</code>	ID value of API call that should be enabled for trace: $0 \leq id \leq 255$ Values from 0 to 99 and 128 to 255 are reserved for embOS.
<code>p</code>	Any void pointer that should be recorded as parameter.

Additional information

The pointer passed as parameter will be displayed in the trace list window of embOSView. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in chapter *Trace record API* on page 519.

24.5.3.4 OS_TRACE_PtrU32()

Description

Writes an entry with ID, a pointer, and a 32-bit unsigned integer as parameter into the trace buffer.

Prototype

```
void OS_TRACE_PtrU32(      OS_U8      id,
                          volatile OS_CONST_PTR void *p0,
                          OS_U32      p1);
```

Parameters

Parameter	Description
<code>id</code>	ID value of API call that should be enabled for trace: $0 \leq id \leq 255$ Values from 0 to 99 and 128 to 255 are reserved for embOS.
<code>p0</code>	Any void pointer that should be recorded as parameter.
<code>p1</code>	Any unsigned 32-bit value that should be recorded as parameter.

Additional information

This function may be used for recording two pointers. The values passed as parameters will be displayed in the trace list window of embOSView. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`164: OS_ERR_OS_INT_ENTER_NOT_CALLED`

`167: OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in chapter *Trace record API* on page 519.

24.5.3.5 OS_TRACE_U32Ptr()

Description

Writes an entry with ID, a 32-bit unsigned integer, and a pointer as parameter into the trace buffer.

Prototype

```
void OS_TRACE_U32Ptr(          OS_U8      id,
                             OS_U32      p0,
                             volatile OS_CONST_PTR void *p1);
```

Parameters

Parameter	Description
<code>id</code>	ID value of API call that should be enabled for trace: $0 \leq id \leq 255$ Values from 0 to 99 and 128 to 255 are reserved for embOS.
<code>p0</code>	Any unsigned 32-bit value that should be recorded as parameter.
<code>p1</code>	Any void pointer that should be recorded as parameter.

Additional information

This function may be used for recording two pointers. The values passed as parameters will be displayed in the trace list window of embOSView. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`164: OS_ERR_OS_INT_ENTER_NOT_CALLED`

`167: OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in chapter *Trace record API* on page 519.

24.5.3.6 OS_TRACE_Void()

Description

Writes an entry identified only by its ID into the trace buffer.

Prototype

```
void OS_TRACE_Void(OS_U8 id);
```

Parameters

Parameter	Description
<code>id</code>	ID value of API call that should be enabled for trace: $0 \leq id \leq 255$ Values from 0 to 99 and 128 to 255 are reserved for embOS.

Additional information

This functionality is available in trace builds only, and the API call is not removed by the preprocessor.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in chapter *Trace record API* on page 519.

24.5.4 Application-controlled trace example

As described in the previous section, the user application can enable and set up the trace conditions without a connection or command from embOSView. The trace record functions can also be called from any user function to write data into the trace buffer, using ID numbers from 100 to 127.

Controlling trace from the application can be useful for tracing API and user functions just after starting the application, when the communication to embOSView is not yet available or when the embOSView setup is not complete.

The example below shows how a trace filter can be set up by the application. The function `OS_TRACE_EnableID()` sets trace filter 0 which affects calls from any running task. Therefore, the first call to `SetState()` in the example would not be traced because there is no task running at that moment. The additional filter setup routine `OS_TRACE_EnableFilterId()` is called with filter 1, which results in tracing calls from outside running tasks.

Example code

```
#include "RTOS.h"

#define APP_TRACE_ID_SETSTATE 100 // Application specific trace id

char MainState;

void SetState(char* pState, char Value) {
    #if (OS_SUPPORT_TRACE != 0)
        OS_TRACE_DataPtr(APP_TRACE_ID_SETSTATE, Value, pState);
    #endif
    *pState = Value;
}

int main(void) {
    OS_Init();
    OS_Inithw();
    #if (OS_SUPPORT_TRACE != 0)
        OS_TRACE_DisableAll(); // Disable all API trace calls
        OS_TRACE_EnableId(APP_TRACE_ID_SETSTATE); // User trace
        OS_TRACE_EnableFilterId(0, APP_TRACE_ID_SETSTATE); // User trace
        OS_TRACE_Enable();
    #endif
    SetState(&MainState, 1);
    OS_TASK_CREATE(&TCBMain, "MainTask", 100, MainTask, MainStack);
    OS_Start(); // Start multitasking
    return 0;
}
```

By default, embOSView lists all user function traces in the trace list window as Routine, followed by the specified ID and two parameters as hexadecimal values. The example above would result in the following:

```
Routine100(0xabcd, 0x01)
```

where `0xabcd` is the pointer address and `0x01` is the parameter recorded from `OS_TRACE_DataPtr()`.

24.5.5 User-defined functions

To use the built-in trace (available in trace builds of embOS) for application program user functions, embOSView can be customized. This customization is done in the setup file `embOS.ini`.

This setup file is parsed at the startup of embOSView. It is optional; you will not see an error message if it cannot be found.

To enable trace setup for user functions, embOSView needs to know an ID number, the function name and the type of two optional parameters that can be traced. The format is explained in the following sample `embOS.ini` file:

Example code

```
# File: embOS.ini
#
# embOSView Setup file
#
# embOSView loads this file at startup. It must reside in the same
# directory as the executable itself.
#
# Note: The file is not required to run embOSView. You will not get
# an error message if it is not found. However, you will get an error message
# if the contents of the file are invalid.
#
# Define add. API functions.
# Syntax: API( <Index>, <Routinename> [parameters])
# Index: Integer, between 100 and 127
# Routinename: Identifier for the routine. Should be no more than 32 characters
# parameters: Optional parameters. A max. of 2 parameters can be specified.
#           Valid parameters are:
#               int
#               ptr
#           Every parameter must be placed after a colon.
#
API( 100, "Routine100")
API( 101, "Routine101", int)
API( 102, "Routine102", int, ptr)
```

Chapter 25

MPU - Memory Protection

25.1 Introduction

embOS-MPU is a separate product and adds task sandboxing support. A sandboxed task has limited access to the hardware and the RTOS but cannot have any harmful impact on other tasks or the RTOS itself. Even if the sandboxed task crashes, all other tasks and the RTOS itself run unaffectedly.

The task sandboxing mechanism is based on:

- Hardware memory protection unit (MPU) or memory management unit (MMU)
- Privileged and unprivileged CPU state
- RTOS software checks

embOS-MPU is available only for CPUs with a memory protection unit (MPU) or memory management unit (MMU). Memory protection is a way to control memory access rights, and is a part of most modern processor architectures and operating systems. The main purpose of memory protection is to prevent a task from accessing memory that has not been allocated to it. This prevents a bug or malware within a task from affecting other tasks, or the operating system itself.

An unprivileged CPU state limits the access to specific CPU features. Which CPU features are affected depends on the specific CPU. For example with Cortex-M an access from the unprivileged CPU state to the BASEPRI register is ignored.

While with embOS all tasks are privileged, embOS-MPU adds support for switching to an unprivileged task state. After the transition the task runs in an unprivileged CPU state and is called unprivileged task. With embOS-MPU memory protection is enabled for unprivileged tasks only. Privileged tasks and the RTOS itself run in the privileged CPU state and do not use the hardware memory protection.

An unprivileged task is not allowed to call all embOS API routines and needs explicit access permissions to RTOS objects. Such checks cannot be performed in hardware but must be implemented in software.

If a task violates a rule, embOS-MPU will terminate the task automatically.

For example:

- An unprivileged task violates the MPU/MMU memory permissions
- A task violates memory access restrictions (e.g. access to non-existing memory)
- An unprivileged task calls invalid embOS API
- An unprivileged task accesses invalid RTOS objects
- An unprivileged task uses an invalid address parameter to an embOS API routine
- Any other invalid operation which causes a fault exception like e.g. division by zero

Example

```

#include "RTOS.h"
#include "BSP.h"

extern unsigned int __FLASH_segment_start__;
extern unsigned int __FLASH_segment_size__;
extern unsigned int __RAM_segment_start__;
extern unsigned int __RAM_segment_size__;
extern unsigned int __ostext_start__;
extern unsigned int __ostext_size__;

static OS_TASK TCBHP, TCBLP;
static OS_STACKPTR int StackHP[128];
static OS_STACKPTR int StackLP[256] __attribute__((aligned(1024)));

static void _HPTask(void) {
    while (1) {
        BSP_ToggleLED(0);
        OS_TASK_Delay(200);
    }
}

static void _Recursive(unsigned int i) {
    volatile int k;
    k = i + 1;
    _Recursive(k);
}

static void _LPTask(void) {
    OS_MPU_ExtendTaskContext();
    OS_MPU_SetAllowedObjects(&TCBLP, _aObjList);
    OS_MPU_SwitchToUnprivState();
    _Recursive(1u);
}

static void _ErrorCallback(OS_CONST_PTR OS_TASK* pTask, OS_MPU_ERRORCODE ErrorCode) {
    while (1) {
    }
}

int main(void) {
    OS_Init();
    OS_MPU_Init(&OS_ARMv7M_MPU_API);
    //
    // Setup memory information, must be done before first task is created
    //
    OS_MPU_ConfigMem(&__FLASH_segment_start__, (OS_U32)&__FLASH_segment_size__,
                    &__RAM_segment_start__, (OS_U32)&__RAM_segment_size__,
                    &__ostext_start__, (OS_U32)&__ostext_size__);
    OS_MPU_SetErrorCallback(&_ErrorCallback);
    OS_Inithw();
    BSP_Init();
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, _HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, _LPTask, StackLP);
    OS_Start();
    return 0;
}

```

25.1.1 Privilege states

Application tasks which may affect other tasks or the OS itself must not have the permission to access the whole memory, special function registers or embOS control structures. Such application code could be e.g. unreliable software from a third party vendor.

Therefore, those application tasks do not run on the same privileged state like the OS. The OS runs in privileged state which means that it has full access to all memory, peripherals and CPU features. Application tasks, on the other hand, run in unprivileged state and have restricted access only to the memory. To access peripherals and memory from unprivileged tasks, additional API and specific device drivers may be used.

State	Description
Privileged	Full access to memory, peripheral and CPU features
Unprivileged	Only restricted access to memory, no direct access to peripherals, no access to some CPU features

25.1.2 Code organization

embOS-MPU assumes that the application code is divided into two parts. The first part runs in privileged state: it initializes the MPU settings and includes the device driver. It contains critical code and must be verified for full reliability by the responsible developers. Usually, this code consists of only a few simple functions which may be located in one single C file.

The second part is the application itself which doesn't need to or in some cases cannot be verified for full reliability. As it runs in unprivileged state, it cannot affect the remaining system. Usually, this code is organized in several C files. This can e.g. simplify a certification.

Part	Description
1st part	Task and MPU initialization Device drivers
2nd part	Application code from e.g. third party vendor

25.2 Memory Access permissions

All privileged tasks have full access to the whole memory. An unprivileged task, however, can have access to several memory regions with different access permissions. Access permissions for RAM and ROM can be used combined, e.g. a ROM region could be readable and code execution could be allowed. In that case the permission defines would be used as `OS_MPU_READONLY | OS_MPU_EXECUTION_ALLOWED`.

The following memory access permissions exist:

Permission	Description
<code>OS_MPU_NOACCESS</code>	No access to a memory region
<code>OS_MPU_READONLY</code>	Read only access to a memory region
<code>OS_MPU_READWRITE</code>	Read and write access to a memory region

Permission	Description
<code>OS_MPU_EXECUTION_ALLOWED</code>	Code execution is allowed
<code>OS_MPU_EXECUTION_DISALLOWED</code>	Code execution is not allowed

25.2.1 Default memory access permissions

A newly created unprivileged task has per default only access to the following memory regions:

Region	Permissions
ROM	<code>OS_MPU_READONLY, OS_MPU_EXECUTION_ALLOWED</code>
RAM	<code>OS_MPU_READONLY, OS_MPU_EXECUTION_ALLOWED</code>
Task stack	<code>OS_MPU_READWRITE, OS_MPU_EXECUTION_ALLOWED</code>

An unprivileged task can read and execute the whole RAM and ROM. Write access is restricted to its own task stack. More access rights can be added by embOS API calls.

25.2.2 Interrupts

Interrupts are always privileged and can access the whole memory.

25.2.3 Access to additional memory regions

An unprivileged task can have access to additional memory regions. This could be necessary e.g. when a task needs to write LCD data to a frame buffer in RAM. Using a device driver could be too inefficient. Additional memory regions can be added with the API function `OS_MPU_AddRegion()`. It is CPU specific if the region has to be aligned. Please refer to the according CPU/ compiler specific embOS manual for more details.

25.2.4 Access to OS objects

An unprivileged task has no direct write access to embOS objects. Per default, it also has no access via embOS API functions. Access to OS objects via embOS API functions can be granted with `OS_MPU_SetAllowedObjects()`. This does not grant direct write access to the RTOS object, but only via embOS API functions. Even more, there must be no MPU region for the unprivileged task which grants write access to the memory location containing the RTOS object. Ideally, the object list should be located in ROM memory so it can not be modified by the application by mistake. The RTOS object must be created in the privileged part of the application.

25.3 ROM placement of embOS

embOS must be placed in one memory section. Usually this section is called `.ostext`. embOS-MPU requires this information to e.g. check that supervisor calls are made from embOS API functions exclusively. The address and the size of this section must be passed to embOS with `OS_MPU_ConfigMem()`. `__os_start__` and `__os_size__` are linker symbols which are defined in the linker file.

Example This example is for the GCC linker.

Linker file:

```
.ostext :
{
  __ostext_start__ = .;
  *(.ostext*)
  __ostext_end__ = .;
} > FLASH
__ostext_size__ = __ostext_end__ - __ostext_start__;
```

C Code:

```
extern unsigned int __ROM_start__;
extern unsigned int __ROM_size__;
extern unsigned int __RAM_start__;
extern unsigned int __RAM_size__;
extern unsigned int __ostext_start__;
extern unsigned int __ostext_size__;

int main(void) {
  OS_Init();
  OS_MPU_Init();
  OS_MPU_ConfigMem(__ROM_start__, __ROM_size__, // ROM
                  __RAM_start__, __RAM_size__, // RAM
                  __ostext_start__, __ostext_size__); // OS
  ..
  OS_Start();
  return 0;
}
```

25.4 Allowed embOS API in unprivileged tasks

Not all embOS API routines are allowed to be called from an unprivileged task. embOS terminates the task when this rule gets violated. If an API routine is allowed to be called from an unprivileged task a dot is placed in the column "Unpriv Task" in the according API table.

Example

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_TASK_Delay()</code>	Suspends the calling task for a specified amount of system ticks, or waits actively when called from <code>main()</code> .	•	•	•		

It is allowed to call `OS_TASK_Delay()` from `main()`, privileged tasks and unprivileged tasks.

25.5 Device driver

Unprivileged tasks run on an unprivileged CPU state with enabled MPU, which restricts the access to specific CPU features and memory. For example unprivileged tasks have no direct access to peripherals. A device driver can be used to access peripherals like e.g. UART, SPI or port pins. embOS executes the device driver routine in the privileged CPU state which does not limit the memory access. With it, embOS offers an explicit and safe way to execute privileged code from an unprivileged task.

`OS_MPU_CallDeviceDriver()` and `OS_MPU_CallDeviceDriverEx()` are used to call the device driver routine. The device driver must be in the device driver list set by `OS_MPU_SetDeviceDriverList()`. An optional parameter can be passed to the device driver routine.

Note

You must not call any embOS API from a device driver.

Example

A device driver for a LED should be developed. The LED driver can toggle a LED with a given index number. The function `BSP_Toggle_LED()` can be called from the unprivileged task and calls `OS_MPU_CallDeviceDriver()` or `OS_MPU_CallDeviceDriverEx()`.

```
typedef enum {
    BSP_LED_SET,
    BSP_LED_CLR,
    BSP_LED_TOGGLE
} BSP_LED_DRIVER_API;

typedef struct BSP_LED_PARAM_STRUCT {
    BSP_LED_DRIVER_API Action;
    OS_U32 Index;
} BSP_LED_PARAM;

void BSP_ToggleLED(int LEDIndex) {
    BSP_LED_PARAM p;
    p.Action = BSP_LED_TOGGLE;
    p.Index = LEDIndex;
    OS_MPU_CallDeviceDriver(0u, &p);
}
```

All device driver addresses are stored in one list which is passed to embOS with `OS_MPU_SetDeviceDriverList()`.

```
static OS_ROUTINE_VOID_PTR* const _DeviceDriverList[] = {
    BSP_LED_DeviceDriver,
    SPI_DeviceDriver,
    UART_DeviceDriver,
    NULL // Last item must be NULL
};

void BSP_Init(void) {
    OS_MPU_SetDeviceDriverList(_DeviceDriverList);
}
```

Alternatively the device driver can be called directly with `OS_MPU_CallDeviceDriverEx()`:

```
void BSP_ToggleLED(int LEDIndex) {
    BSP_LED_PARAM p;
    p.Action = BSP_LED_TOGGLE;
    p.Index = LEDIndex;
    OS_MPU_CallDeviceDriverEx(BSP_LED_DeviceDriver, &p);
}
```

The device driver itself runs in privilege CPU state without memory access restrictions. Therefore, it can access the LED port pins.

```
void BSP_LED_DeviceDriver(void* Param) {
    BSP_LED_PARAM* p;
    p = (BSP_LED_PARAM*)Param;
    switch (p->Action) {
        case BSP_LED_SET:
            SetPortPin(p->Index);
            break;
        case BSP_LED_CLR:
            ClearPortPin(p->Index);
            break;
        case BSP_LED_TOGGLE:
            TogglePortPin(p->Index);
            break;
        default:
            break;
    }
}
```

25.6 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_MPU_AddRegion()</code>	Grants permission to access an additional memory region.	•	•			
<code>OS_MPU_CallDeviceDriver()</code>	Calls a device driver.	•	•	•	•	•
<code>OS_MPU_CallDeviceDriverEx()</code>	Calls a device driver.	•	•	•	•	•
<code>OS_MPU_ConfigMem()</code>	Configures basic memory information.	•				
<code>OS_MPU_ExtendTaskContext()</code>	Extends the task context for the MPU registers.		•			
<code>OS_MPU_GetThreadState()</code>	Returns the current tasks privilege state.		•	•		
<code>OS_MPU_Init()</code>	Initializes the MPU hardware with the specified MPU API list.	•				
<code>OS_MPU_SetAllowedObjects()</code>	Grants permission to access a list of RTOS objects via embOS API.	•	•			
<code>OS_MPU_SetDeviceDriverList()</code>	Sets the device driver list.	•	•			
<code>OS_MPU_SetErrorCallback()</code>	Sets the MPU error callback routine.	•	•			
<code>OS_MPU_SwitchToUnprivState()</code>	Switches a task to unprivileged state.		•			
<code>OS_MPU_SetSanityCheckBuffer()</code>	Sets the pointer in the task control block to a buffer which holds a copy of the MPU register for sanity check.	•	•			
<code>OS_MPU_SanityCheck()</code>	Performs an MPU sanity check which checks if the MPU register still have the correct value.			•		

25.6.1 OS_MPU_AddRegion()

Description

Grants permission to access an additional memory region.

Prototype

```
void OS_MPU_AddRegion(OS_TASK* pTask,
                     void* BaseAddr,
                     OS_U32 Size,
                     OS_U32 Permissions,
                     OS_U32 Attributes);
```

Parameters

Parameter	Description
pTask	Pointer to a task control block of type OS_TASK or NULL for current task.
BaseAddr	Region base address.
Size	Region size.
Permissions	Access permissions.
Attributes	Additional core specific memory attributes.

Additional information

OS_MPU_AddRegion() can be used if a task needs access to additional memory regions. This memory region can be e.g. a LCD frame buffer or a queue data buffer. If OS_MPU_AddRegion() should be used for a running unprivileged task, the MPU task context extension is mandatory. The amount of MPU regions and whether the region has to be aligned is CPU specific. Please refer to the according CPU and compiler specific embOS manual for more details.

Note

OS_MPU_AddRegion() expected until embOS V5.8.2 [BaseAddr](#) as a OS_U32 value. From embOS V5.10.0 this parameter is a void pointer. Existing applications which call OS_MPU_AddRegion() needs to be updated accordingly.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
241: OS_ERR_MPU_INVALID_REGION
247: OS_ERR_MPU_NOINIT
```

For details, refer to the chapter *Runtime application errors* on page 455.

A memory region can have the following access permissions:

Permission	Description
OS_MPU_NOACCESS	No access to memory region
OS_MPU_READONLY	Read only access to memory region
OS_MPU_READWRITE	Read and write access to memory region
OS_MPU_EXECUTION_ALLOWED	Code execution is allowed

Permission	Description
<code>OS_MPU_EXECUTION_DISALLOWED</code>	Code execution is not allowed

Access permissions for data and code execution can be jointly set for one region. A region can for example be set to read only and code execution can be disabled (`OS_MPU_READONLY | OS_MPU_EXECUTION_DISALLOWED`). Per default an unprivileged task has only access to the following memory regions:

Region	Permission
ROM	Read and execution access for complete ROM
RAM	Read only and and execution access for complete RAM
Task stack	Read and write and execution access to the task stack

Note

`OS_MPU_AddRegion()` does take affect only when it is called before `OS_MPU_SwitchToUnprivState()`.

Example

```
static void HPTask(void) {
    OS_MPU_AddRegion(&TCBHP, (OS_U32)MyQBuffer, 512, OS_MPU_READWRITE, 0u);
}
```

25.6.2 OS_MPU_CallDeviceDriver()

Description

Calls a device driver.

Prototype

```
void OS_MPU_CallDeviceDriver(OS_U32 Index,
                             void* pParam);
```

Parameters

Parameter	Description
<code>Index</code>	<code>Index</code> of device driver function.
<code>pParam</code>	Parameter passed to the device driver routine.

Additional information

Unprivileged tasks run on an unprivileged CPU state with enabled MPU, which restricts the access to specific CPU features and memory. For example unprivileged tasks have no direct access to peripherals.

A device driver can be used to execute the desired action in a privileged context. embOS executes the device driver routine in the privileged CPU state which does not limit the memory access.

An optional parameter can be passed to the device driver routine. Using a void pointer gives the flexibility to pass any kind of data to the device driver routine.

`OS_MPU_CallDeviceDriver()` presumes that the device driver routine addresses are stored in a list which was set with `OS_MPU_SetDeviceDriverList()`. The index parameter denotes the routine position in that list.

If called from an unprivileged task with an invalid index, the task gets terminated.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
167: OS_ERR_CPU_STATE_ILLEGAL
247: OS_ERR_MPU_NOINIT
248: OS_ERR_MPU_DEVICE_INDEX
249: OS_ERR_MPU_INV_DEVICE_LIST
```

For details, refer to the chapter *Runtime application errors* on page 455.

Note

You must not call any embOS API from a device driver.

Example

```
typedef enum {
    BSP_LED_SET,
    BSP_LED_CLR,
    BSP_LED_TOGGLE
} BSP_LED_DRIVER_API;

typedef struct BSP_LED_PARAM_STRUCT {
    BSP_LED_DRIVER_API Action;
    OS_U32 Index;
} BSP_LED_PARAM;
```

```
static OS_ROUTINE_VOID_PTR* const _DeviceDriverList[] = {
    BSP_LED_DeviceDriver,
    NULL // Last item must be NULL
};

void BSP_LED_DeviceDriver(void* Param) {
    BSP_LED_PARAM* p;

    p = (BSP_LED_PARAM*)Param;
    switch (p->Action) {
        case BSP_LED_SET:
            BSP_SetLED_SVC(p->Index);
            break;
        case BSP_LED_CLR:
            BSP_ClrLED_SVC(p->Index);
            break;
        case BSP_LED_TOGGLE:
            BSP_ToggleLED_SVC(p->Index);
            break;
        default:
            break;
    }
}

void BSP_ToggleLED(int Index) {
    BSP_LED_PARAM p;

    p.Action = BSP_LED_TOGGLE;
    p.Index = Index;
    OS_MPU_CallDeviceDriver(0u, &p);
}
```

25.6.3 OS_MPU_CallDeviceDriverEx()

Description

Calls a device driver.

Prototype

```
void OS_MPU_CallDeviceDriverEx(OS_ROUTINE_VOID_PTR* pfRoutine,
                              void*                pParam);
```

Parameters

Parameter	Description
<code>pfRoutine</code>	Pointer to the device driver routine of type <code>OS_ROUTINE_VOID_PTR</code> .
<code>pParam</code>	Parameter passed to the device driver routine.

Additional information

Unprivileged tasks run on an unprivileged CPU state with enabled MPU, which restricts the access to specific CPU features and memory. For example unprivileged tasks have no direct access to peripherals.

A device driver can be used to execute the desired action in a privileged context. embOS executes the device driver routine in the privileged CPU state which does not limit the memory access.

An optional parameter can be passed to the device driver routine. Using a void pointer gives the flexibility to pass any kind of data to the device driver routine.

`OS_MPU_CallDeviceDriverEx()` presumes that the device driver routine addresses are stored in a list which was set with `OS_MPU_SetDeviceDriverList()`.

If called from an unprivileged task with an invalid device driver routine, the task gets terminated.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

167: `OS_ERR_CPU_STATE_ILLEGAL`

247: `OS_ERR_MPU_NOINIT`

249: `OS_ERR_MPU_INV_DEVICE_LIST`

For details, refer to the chapter *Runtime application errors* on page 455.

Note

You must not call any embOS API from a device driver.

Example

```
typedef enum {
    BSP_LED_SET,
    BSP_LED_CLR,
    BSP_LED_TOGGLE
} BSP_LED_DRIVER_API;

\code{c}
typedef struct BSP_LED_PARAM_STRUCT {
    BSP_LED_DRIVER_API Action;
    OS_U32                Index;
}
```

```
} BSP_LED_PARAM;

void BSP_LED_DeviceDriver(void* Param) {
    BSP_LED_PARAM* p;

    p = (BSP_LED_PARAM*)Param;
    switch (p->Action) {
        case BSP_LED_SET:
            BSP_SetLED_SVC(p->Index);
            break;
        case BSP_LED_CLR:
            BSP_ClrLED_SVC(p->Index);
            break;
        case BSP_LED_TOGGLE:
            BSP_ToggleLED_SVC(p->Index);
            break;
        default:
            break;
    }
}

void BSP_ToggleLED(int Index) {
    BSP_LED_PARAM p;

    p.Action = BSP_LED_TOGGLE;
    p.Index = Index;
    OS_MPU_CallDeviceDriverEx(BSP_LED_DeviceDriver, &p);
}
```

25.6.4 OS_MPU_ConfigMem()

Description

Configures basic memory information.

Prototype

```
void OS_MPU_ConfigMem(void* ROM_BaseAddr,
                      OS_U32 ROM_Size,
                      void* RAM_BaseAddr,
                      OS_U32 RAM_Size,
                      void* OS_BaseAddr,
                      OS_U32 OS_Size);
```

Parameters

Parameter	Description
ROM_BaseAddr	ROM base address
ROM_Size	ROM size.
RAM_BaseAddr	RAM base address
RAM_Size	RAM size.
OS_BaseAddr	embOS ROM region base address.
OS_Size	embOS ROM region size.

Additional information

`OS_MPU_ConfigMem()` tells embOS where ROM, RAM and the embOS code is located in memory. This information is used to setup the default task regions at task creation.

Note

With embOS-MPU `OS_MPU_ConfigMem()` must be called before creating any task.

Note

`OS_MPU_ConfigMem()` expected until embOS V5.8.2 [ROM_BaseAddr](#), [RAM_BaseAddr](#) and [OS_BaseAddr](#) as a `OS_U32` value. From embOS V5.10.0 these parameters are void pointer. Existing applications which call `OS_MPU_ConfigMem()` need to be updated accordingly.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
165: OS_ERR_INIT_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
228: OS_ERR_ILLEGAL_AFTER_OSSTART
247: OS_ERR_MPU_NOINIT
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

Please refer to the example in the introduction of chapter *MPU - Memory Protection* on page 528.

25.6.5 OS_MPU_ExtendTaskContext()

Description

Extends the task context for the MPU registers.

Prototype

```
void OS_MPU_ExtendTaskContext(void);
```

Additional information

It is device dependent how many MPU regions are available. This function makes it possible to use all MPU regions for every single task. Otherwise the tasks would have to share the MPU regions. To do so the MPU register must be saved and restored with every context switch.

This function allows the user to extend the task context for the MPU registers. A major advantage is that the task extension is task-specific. This means that the additional MPU register needs to be saved only by tasks that actually use these registers. The advantage is that the task switching time of other tasks is not affected. The same is true for the required stack space: Additional stack space is required only for the tasks which actually save the additional MPU registers. The task context can be extended only once per task. The function must not be called multiple times for one task.

`OS_MPU_ExtendTaskContext()` is not available when no task context extension is available. The embOS `OS_LIBMODE_XR` library mode does not support task context extension.

`OS_SetDefaultContextExtension(&OS_MPU_ContextExtension)` can be used to automatically add MPU register to the task context of every newly created task.

The task context can also be extended using `OS_TASK_AddContextExtension(&OS_MPU_ContextExtension)` or `OS_TASK_SetContextExtension(&OS_MPU_ContextExtension)`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

`247: OS_ERR_MPU_NOINIT`

For details, refer to the chapter *Runtime application errors* on page 455.

Note

If you run more than one unprivileged task you must use `OS_MPU_ExtendTaskContext()` in order to save and restore the MPU register for each unprivileged task. With specific MPUs it might be necessary to use `OS_MPU_ExtendTaskContext()` even with one unprivileged task. Please refer to the CPU/compiler specific embOS manual for more details.

Example

Please refer to the example in the introduction of chapter *MPU - Memory Protection* on page 528.

25.6.6 OS_MPU_GetThreadState()

Description

Returns the current tasks privilege state.

Prototype

```
OS_MPU_THREAD_STATE OS_MPU_GetThreadState(void);
```

Return value

= 0 Privileged state (OS_MPU_THREAD_STATE_PRIVILEGED).
≠ 0 Unprivileged state (OS_MPU_THREAD_STATE_UNPRIVILEGED).

Additional information

A new created task has the task state OS_MPU_THREAD_STATE_PRIVILEGED. It can be set to OS_MPU_THREAD_STATE_UNPRIVILEGED with the API function OS_MPU_SwitchToUnprivState(). A task can never set itself back to the privileged state OS_MPU_THREAD_STATE_PRIVILEGED.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK  
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
164: OS_ERR_OS_INT_ENTER_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL  
247: OS_ERR_MPU_NOINIT
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintMPUState(void) {  
    if (OS_MPU_GetThreadState() == OS_MPU_THREAD_STATE_PRIVILEGED) {  
        printf("Task is in privileged state");  
    } else {  
        printf("Task is in unprivileged state");  
    }  
}
```

25.6.7 OS_MPU_Init()

Description

Initializes the MPU hardware with the specified MPU API list.

Prototype

```
void OS_MPU_Init(OS_CONST_PTR OS_MPU_API_LIST *pAPIList);
```

Parameters

Parameter	Description
<code>pAPIList</code>	Pointer to core specific MPU API list.

Additional information

This function must be called before any embOS-MPU related function is used or any task is created.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
165: OS_ERR_INIT_NOT_CALLED  
167: OS_ERR_CPU_STATE_ILLEGAL  
228: OS_ERR_ILLEGAL_AFTER_OSSTART
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
int main(void) {  
    ...  
    OS_MPU_Init(&OS_ARMv7M_MPU_API);  
    ...  
    return 0;  
}
```

25.6.8 OS_MPU_SetAllowedObjects()

Description

Grants permission to access a list of RTOS objects via embOS API.

Prototype

```
void OS_MPU_SetAllowedObjects(OS_TASK* pTask,
                             OS_CONST_PTR OS_MPU_OBJ *pObjList);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.
<code>pObjList</code>	Pointer to a list of allowed objects.

Additional information

`OS_MPU_SetAllowedObjects()` sets a task specific list of RTOS objects, to which the task has access via embOS API functions.

Per default, a task has neither direct nor indirect write access, via embOS API functions to any embOS object, like a task control block. Even if the object is in the list of allowed objects a direct write access to an RTOS object is still not possible. But if an object is in the list, the task can access the RTOS object via embOS API functions. Even more, an embOS debug build will check within `OS_MPU_SetAllowedObject()`, if there is a MPU region which grants write access to the memory location containing the RTOS object, and call `OS_Error()` with an appropriate error code. This ensures, that the unprivileged task is unable to modify the RTOS object by mistake.

The OS object can be e.g. the own task control block, a mailbox control structure which is mutual used by different task or even the task control block of another task. It is the developer responsibility to only add objects which are necessary for the unprivileged task. The list is null-terminated, which means the last entry must always be: `{NULL, OS_MPU_OBJTYPE_INVALID}`. If the list of allowed objects should be updated at run-time, `OS_MPU_SetAllowedObjects()` may be called repeatedly or the array may be modified (if not defined as const).

The following object types exist:

```
OS_MPU_OBJTYPE_TASK
OS_MPU_OBJTYPE_MUTEX
OS_MPU_OBJTYPE_SEMA
OS_MPU_OBJTYPE_RWLOCK
OS_MPU_OBJTYPE_EVENT
OS_MPU_OBJTYPE_QUEUE
OS_MPU_OBJTYPE_MAILBOX
OS_MPU_OBJTYPE_SWTIMER
OS_MPU_OBJTYPE_MEMPOOL
OS_MPU_OBJTYPE_WATCHDOG
```

embOS task events are handled in the task control block. When task events should be used from an unprivileged task the according task control block needs be added to this object list.

Note

Up to embOS V5.8.2, `OS_MPU_SetAllowedObjects()` expected the first value in `pObjList` as a `OS_U32` value. Starting from embOS V5.10.0, this parameter is a void pointer. Existing applications which call `OS_MPU_SetAllowedObjects()` need to be updated accordingly.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
245: OS_ERR_MPU_INVALID_OBJECT
247: OS_ERR_MPU_NOINIT
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static const OS_MPU_OBJ_ObjList[] = {{&TCBHP, OS_MPU_OBJTYPE_TASK},
                                     {NULL, OS_MPU_OBJTYPE_INVALID}};

static void _Unpriv(void) {
    OS_TASK_SetName(&TCBHP, "Segger");
    while (1) {
        OS_TASK_Delay(10);
    }
}

static void HPTask(void) {
    OS_MPU_ExtendTaskContext();
    OS_MPU_SetAllowedObjects(&TCBHP, _ObjList);
    OS_MPU_SwitchToUnprivState();
    _Unpriv();
}
```

25.6.9 OS_MPU_SetDeviceDriverList()

Description

Sets the device driver list.

Prototype

```
void OS_MPU_SetDeviceDriverList(OS_ROUTINE_VOID_PTR* OS_CONST_PTR *pList);
```

Parameters

Parameter	Description
<code>pList</code>	Pointer to the list of device driver routine addresses.

Additional information

All device driver routine addresses are stored in one list. The last item must be NULL. A device driver is called with the according index to this list. `OS_MPU_SetDeviceDriverList()` must be called before `OS_Start()` only.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
247: OS_ERR_MPU_NOINIT
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_ROUTINE_VOID_PTR* const _DeviceDriverList[] = {
    BSP_LED_DeviceDriver,
    SPI_DeviceDriver,
    UART_DeviceDriver,
    NULL // Last item must be NULL
};

void BSP_Init(void) {
    OS_MPU_SetDeviceDriverList(_DeviceDriverList);
}
```

25.6.10 OS_MPU_SetErrorCallback()

Description

Sets the MPU error callback routine.

Prototype

```
void OS_MPU_SetErrorCallback(OS_ROUTINE_TASK_PTR_ERRORCODE* pfRoutine);
```

Parameters

Parameter	Description
<code>pfRoutine</code>	Pointer to the routine of type <code>OS_ROUTINE_TASK_PTR_ERRORCODE</code> which shall be called when a task gets terminated due to an illegal operation.

Additional information

embOS terminates any task that violates its MPU permissions or causes an exception by other means. embOS calls the user callback routine in order to inform the application. The application can e.g. turn on an error LED or write the fault into a log file.

Note

The error callback function must not call any embOS API function.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
247: OS_ERR_MPU_NOINIT
```

For details, refer to the chapter *Runtime application errors* on page 455.

embOS-MPU error callback routine

The callback function is called with the following parameter:

Parameter type	Description
<code>OS_CONST_PTR OS_TASK*</code>	Pointer to task control block of the unprivileged task which caused the MPU error.
<code>OS_MPU_ERRORCODE</code>	Error code which describes the cause for the MPU error.

Note

Since embOS V5.20.0 the first parameter is a pointer to a constant task control block.

embOS-MPU error codes

Define	Explanation
<code>OS_MPU_ERROR_INVALID_REGION</code>	Indicates that the unprivileged task tried to access an disallowed memory region via embOS API.
<code>OS_MPU_ERROR_INVALID_OBJECT</code>	Indicates that the unprivileged task tried to access an disallowed this RTOS object.

Define	Explanation
OS_MPU_ERROR_INVALID_API	Indicates that the unprivileged task tried to call an disallowed embOS API function. For example, unprivileged tasks must not call OS_TASK_EnterRegion().
OS_MPU_ERROR_HARDFAULT	Indicates that the task caused a hard fault.
OS_MPU_ERROR_MEMFAULT	Indicates that the task caused a memory fault due to an illegal memory access.
OS_MPU_ERROR_BUSFAULT	Indicates that the task caused a bus fault.
OS_MPU_ERROR_USAGEFAULT	Indicates that the task caused a usage fault.
OS_MPU_ERROR_SVC	Indicates that a supervisor call was made outside an embOS API function. This is not allowed.
OS_MPU_ERROR_INVALID_CONDITION_ROUTINE	Indicates that an invalid condition routine was used.

Example

Please refer to the example in the introduction of chapter *MPU - Memory Protection* on page 528.

25.6.11 OS_MPU_SwitchToUnprivState()

Description

Switches a task to unprivileged state.

Prototype

```
void OS_MPU_SwitchToUnprivState(void);
```

Additional information

The task code must be split into two parts. The first part runs in privileged state and initializes the embOS MPU settings. The second part runs in unprivileged state and is called after the privileged part switched to the unprivileged state with `OS_MPU_SwitchToUnprivState()`.

If this function is called from an invalid context, debug builds of embOS will call `OS_Error()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
167: OS_ERR_CPU_STATE_ILLEGAL  
226: OS_ERR_ILLEGAL_IN_MAIN  
247: OS_ERR_MPU_NOINIT
```

For details, refer to the chapter *Runtime application errors* on page 455.

Note

If you run more than one unprivileged task you must use `OS_MPU_ExtendTaskContext()` in order to save and restore the MPU register for each unprivileged task.

Example

Please refer to the example in the introduction of chapter *MPU - Memory Protection* on page 528.

25.6.12 OS_MPU_SetSanityCheckBuffer()

Description

Sets the pointer in the task control block to a buffer which holds a copy of the MPU register for sanity check. The buffer size needs to be the size of all MPU register.

Prototype

```
void OS_MPU_SetSanityCheckBuffer(OS_TASK* pTask,
                                void*     p);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.
<code>p</code>	Pointer to the MPU register buffer.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used.

`OS_MPU_SetSanityCheckBuffer()` is only available in `OS_LIBMODE_SAFE`. Due to e.g. a hardware failure, a MPU register content could change. A copy of all relevant MPU register is held in the buffer. `OS_MPU_SanityCheck()` compares this copy to the actual MPU register and returns whether the register still have the same value.

`OS_MPU_SetSanityCheckBuffer()` must be called from and for a privileged task only. It must be called before `OS_MPU_SanityCheck()` is used for the first time. The size of the buffer depends on the used hardware MPU. Appropriate defines are provided, e.g. `OS_ARM_V7M_MPU_REGS_SIZE`. These defines are documented in the CPU and compiler specific manual.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
247: OS_ERR_MPU_NOINIT
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_U8 HPBuffer[OS_ARM_V7M_MPU_REGS_SIZE];

static void HPTask(void) {
    OS_BOOL r;

    OS_MPU_SetSanityCheckBuffer(NULL, HPBuffer);
    OS_MPU_ExtendTaskContext();
    OS_MPU_SwitchToUnprivState();
    while (1) {
        r = OS_MPU_SanityCheck();
        if (r == 0) {
            while (1) { // MPU register value invalid
            }
        }
    }
}
```

25.6.13 OS_MPU_SanityCheck()

Description

Performs an MPU sanity check which checks if the MPU register still have the correct value.

Prototype

```
OS_BOOL OS_MPU_SanityCheck(void);
```

Return value

= 0 Failure, at least one register has not the correct value.
 ≠ 0 Success, all registers have the correct value.

Additional information

OS_MPU_SanityCheck() is only available in OS_LIBMODE_SAFE which is used in the certified embOS-MPU. Due to e.g. a hardware failure, an MPU register content could change. A copy of all relevant MPU register is held in a buffer and a pointer to this buffer is stored in the according task control block. OS_MPU_SanityCheck() compares this copy to the actual MPU register and returns whether the register still have the same value.

OS_MPU_SanityCheck() must be called in unprivileged tasks after the call to OS_MPU_SwitchToUnprivState() only.

OS_MPU_SetSanityCheckBuffer() must be called before OS_MPU_SanityCheck() is used for the first time.

Error codes

With embOS debug checks enabled erroneous calls to this function result in OS_Error() being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
246: OS_ERR_MPU_PRIVSTATE_INVALID
247: OS_ERR_MPU_NOINIT
263: OS_ERR_MPU_SANITY_BUFFER_NOT_SET
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
static OS_U8 HPBuffer[OS_ARM_V7M_MPU_REGS_SIZE];

static void HPTask(void) {
    OS_BOOL r;

    OS_MPU_SetSanityCheckBuffer(&TCBHP, HPBuffer);
    OS_MPU_ExtendTaskContext();
    OS_MPU_SwitchToUnprivState();
    while (1) {
        r = OS_MPU_SanityCheck();
        if (r == 0) {
            while (1) { // MPU register value invalid
            }
        }
    }
}
```

Chapter 26

Stacks

26.1 Introduction

The stack is the memory area used for storing the return address of function calls, parameters, and local variables, as well as for temporary storage. Interrupt routines also use the stack to save the return address and flag registers, except in cases where the CPU has a separate stack for interrupt functions. Refer to the CPU & Compiler Specifics manual of embOS documentation for details on your processor's stack.

With most CPUs the stacks grow from high-memory to low-memory. A stack pointer points to the current stack position. Depending on the CPU you can have more than one stack pointer, for example separate stack pointers for the system and interrupt stack. The embOS scheduler programs the stack pointer where necessary, for instance to point to the task stack of the next running task.

System stack

Before embOS takes control (before the call to `OS_Start()`), a program uses the so called system stack. This is the same stack that a non-embOS program for this CPU would use. After transferring control to the embOS scheduler by calling `OS_Start()`, the system stack is used for the following (when no task is executing):

- embOS scheduler
- embOS software timers (and the callback).

For details regarding required size of your system stack, refer to the CPU & Compiler specific embOS manual.

Interrupt stack

To reduce stack size in a multitasking environment, some processors use a specific stack area for interrupt service routines (called a hardware interrupt stack). If there is no interrupt stack, you will need to add stack requirements of your interrupt service routines to each task stack.

Even if the CPU does not support a hardware interrupt stack, embOS may support a separate stack for interrupts by calling the function `OS_INT_EnterIntStack()` at beginning of an interrupt service routine and `OS_INT_LeaveIntStack()` at its very end. In case the CPU already supports hardware interrupt stacks or if a separate interrupt stack is not supported at all, these function calls are implemented as empty macros.

We recommend using `OS_INT_EnterIntStack()` and `OS_INT_LeaveIntStack()` even if there is currently no additional benefit for your specific CPU, because code that uses them might reduce stack size on another CPU or a new version of embOS with support for an interrupt stack for your CPU. For details about interrupt stacks, see the CPU & Compiler specific embOS manual.

Task stack

In a multitasking system, every task must have its own stack. The location and size of this stack is defined when creating the task. The size of a task stack depends on the CPU, compiler and the task routine. For example if you use more local variables in the task routine, you will need more task stack. For details, see the CPU & Compiler Specifics manual of embOS documentation.

It is possible to calculate the task stack size but this assumes the exact path of the code is known at all time and you know the stack consumption of all called routine. This might be very difficult. It is good practice to initially start with a large task stack space, run the application, and use a tool like embOSView to monitor the actual stack usage. The task stack size can then be adjusted accordingly. A stack that has been defined larger than necessary does no harm; even though it is a waste of memory.

You can define the task stack for example as an array of integers. This automatically ensures the stack is integer aligned, if necessary.

```
static OS_STACKPTR int Stack[128],

void Task(void) {
    int array[16]; // Will be located on the task stack

    while (1) {
        OS_TASK_Delay(100);
    }
}

int main(void) {
    OS_Init(); // Initialize embOS

    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);

    OS_Start();
    return 0;
}
```

Stack overflow

The stack needs to have a minimum size, which is determined by the sum of the stack usage of the routines in the worst-case nesting. If the stack is too small, a section of the memory that is not reserved for the stack will be overwritten, and a serious program failure is most likely to occur. With most CPUs the stacks grow from high-memory to low-memory, meaning data, which is located before this stack in memory, will be overwritten. This could e.g. be another task stack, a task control block or even application data.

Stack-check

The stack-check builds of embOS monitor the stack end of the system stack, all task stacks and interrupt stack (if available) and call `OS_Error()` if they detect stack overflows.

To detect a stack overflow, the stack is filled with pattern bytes, thereby allowing for a check on these characters every time a task is deactivated. This pattern byte is `0xCD`. However, embOS can not guarantee to reliably detect all stack overflows. The task routine could have stored data on the task stack which has overwritten bytes on the task stack before and after the last byte, but not the last byte itself. In this case, embOS is unable to detect the stack overflow. This can be done by hardware only, e.g. with a memory protection unit.

Stack size calculation

embOS includes stack size calculation routines. embOS fills the task stacks and also the system stack and the interrupt stack with a pattern byte. If requested by according API functions or `embOSView`, embOS checks how many bytes at the end of the stack still include the pattern byte. With this information, the amount of used and unused stack can be estimated. However, accuracy is not guaranteed at all times, since the task routine could have stored data on the task stack which has overwritten bytes on the task stack before and after the last byte(s), but not the last byte(s) themselves.

26.2 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_STACK_GetIntStackBase()</code>	Returns the base address of the interrupt stack.	•	•	•	•	•
<code>OS_STACK_GetIntStackSize()</code>	Returns the size of the interrupt stack.	•	•	•	•	•
<code>OS_STACK_GetIntStackSpace()</code>	Returns the amount of interrupt stack which was never used (Free interrupt stack space).	•	•	•	•	•
<code>OS_STACK_GetIntStackUsed()</code>	Returns the maximal amount of interrupt stack which has been used.	•	•	•	•	•
<code>OS_STACK_GetTaskStackBase()</code>	Returns a pointer to the base of a task stack.	•	•	•	•	•
<code>OS_STACK_GetTaskStackSize()</code>	Returns the total size of a task stack.	•	•	•	•	•
<code>OS_STACK_GetTaskStackSpace()</code>	Returns the amount of task stack which was never used by the task (Free task stack space).	•	•	•	•	•
<code>OS_STACK_GetTaskStackUsed()</code>	Returns the maximal amount of task stack which has been used.	•	•	•	•	•
<code>OS_STACK_GetSysStackBase()</code>	Returns the base address of the system stack.	•	•	•	•	•
<code>OS_STACK_GetSysStackSize()</code>	Returns the size of the system stack.	•	•	•	•	•
<code>OS_STACK_GetSysStackSpace()</code>	Returns the amount of system stack which was never used (Free system stack space).	•	•	•	•	•
<code>OS_STACK_GetSysStackUsed()</code>	Returns the maximal amount of system stack which has been used.	•	•	•	•	•
<code>OS_STACK_GetCheckLimit()</code>	Returns the stack check limit in percent.	•	•	•		
<code>OS_STACK_SetCheckLimit()</code>	Sets the stack check limit to a percentaged value of the stack size.	•	•			

26.2.1 OS_STACK_GetIntStackBase()

Description

Returns a pointer to the base of the interrupt stack.

Prototype

```
void* OS_STACK_GetIntStackBase(void);
```

Return value

The pointer to the base address of the interrupt stack.

Additional information

The return value of this function is valid only when an interrupt stack exists. In all other builds it will be 0.

Example

```
void CheckIntStackBase(void) {  
    printf("Addr Interrupt Stack %p", OS_STACK_GetIntStackBase());  
}
```

26.2.2 OS_STACK_GetIntStackSize()

Description

Returns the size of the interrupt stack.

Prototype

```
unsigned int OS_STACK_GetIntStackSize(void);
```

Return value

The size of the interrupt stack in bytes.

Additional information

The return value of this function is valid only when an interrupt stack exists. In all other builds it will be 0.

Example

```
void CheckIntStackSize(void) {  
    printf("Size Interrupt Stack %u", OS_STACK_GetIntStackSize());  
}
```

26.2.3 OS_STACK_GetIntStackSize()

Description

Returns the amount of interrupt stack which was never used (Free interrupt stack space).

Prototype

```
unsigned int OS_STACK_GetIntStackSize(void);
```

Return value

Amount of interrupt stack which was never used in bytes.

Additional information

The return value of this function is valid only in embOS stack check builds and when an interrupt stack exists. In all other builds it will be 0.

Note

This routine does not reliably detect the amount of stack space left, because it can only detect modified bytes on the stack. Unfortunately, space used for register storage or local variables is not always modified. In most cases, this routine will detect the correct amount of stack bytes, but in case of doubt, be generous with your stack space or use other means to verify that the allocated stack space is sufficient.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
```

```
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void CheckIntStackSize(void) {  
    printf("Unused Interrupt Stack %u", OS_STACK_GetIntStackSize());  
}
```

26.2.4 OS_STACK_GetIntStackUsed()

Description

Returns the maximal amount of interrupt stack which has been used.

Prototype

```
unsigned int OS_STACK_GetIntStackUsed(void);
```

Return value

Amount of interrupt stack which has been used in bytes.

Additional information

The return value of this function is valid only in embOS stack check builds and when an interrupt stack exists. In all other builds it will be 0.

Note

This routine does not reliably detect the amount of stack space used, because it can only detect modified bytes on the stack. Unfortunately, space used for register storage or local variables is not always modified. In most cases, this routine will detect the correct amount of stack bytes, but in case of doubt, be generous with your stack space or use other means to verify that the allocated stack space is sufficient.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
```

```
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void CheckIntStackUsed(void) {  
    printf("Used Interrupt Stack %u", OS_STACK_GetIntStackUsed());  
}
```

26.2.5 OS_STACK_GetTaskStackBase()

Description

Returns a pointer to the base of a task stack. If `pTask` is `NULL`, the currently executed task is checked.

Prototype

```
void OS_STACKPTR *OS_STACK_GetTaskStackBase(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

Pointer to the base address of the task stack.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

The return value of this function is valid only in embOS stack check builds. In all other builds it will be 0.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void CheckStackBase(void) {
    printf("Addr Stack[0] %p", OS_STACK_GetTaskStackBase(&TCB[0]));
    OS_TASK_Delay(1000);
    printf("Addr Stack[1] %p", OS_STACK_GetTaskStackBase(&TCB[1]));
    OS_TASK_Delay(1000);
}
```

26.2.6 OS_STACK_GetTaskStackSize()

Description

Returns the total size of a task stack.

Prototype

```
unsigned int OS_STACK_GetTaskStackSize(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

Total size of the task stack in bytes.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

The return value of this function is valid only in embOS stack check builds. In all other builds it will be 0.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void CheckStackSize(void) {
    printf("Size Stack[0] %u", OS_STACK_GetTaskStackSize(&TCB[0]));
    OS_TASK_Delay(1000);
    printf("Size Stack[1] %u", OS_STACK_GetTaskStackSize(&TCB[1]));
    OS_TASK_Delay(1000);
}
```

26.2.7 OS_STACK_GetTaskStackSize()

Description

Returns the amount of task stack which was never used by the task (Free task stack space).

Prototype

```
unsigned int OS_STACK_GetTaskStackSize(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

Amount of task stack which was never used by the task in bytes.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

In most cases, the stack size required by a task cannot be easily calculated because it takes quite some time to calculate the worst-case nesting and the calculation itself is difficult.

However, the required stack size can be calculated using the function `OS_STACK_GetTaskStackSize()`, which returns the number of unused bytes on the stack. If there is a lot of space left, you can reduce the size of this stack. The return value of this function is valid only in embOS stack check builds. In all other builds it will be 0.

Note

This routine does not reliably detect the amount of stack space left, because it can only detect modified bytes on the stack. Unfortunately, space used for register storage or local variables is not always modified. In most cases, this routine will detect the correct amount of stack bytes, but in case of doubt, be generous with your stack space or use other means to verify that the allocated stack space is sufficient.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void CheckStackSize(void) {
    printf("Unused Stack[0] %u", OS_STACK_GetTaskStackSize(&TCB[0]));
    OS_TASK_Delay(1000);
    printf("Unused Stack[1] %u", OS_STACK_GetTaskStackSize(&TCB[1]));
    OS_TASK_Delay(1000);
}
```

26.2.8 OS_STACK_GetTaskStackUsed()

Description

Returns the maximal amount of task stack which has been used.

Prototype

```
unsigned int OS_STACK_GetTaskStackUsed(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

Parameter	Description
<code>pTask</code>	Pointer to a task control block of type <code>OS_TASK</code> or <code>NULL</code> for the current task.

Return value

Amount of task stack which has been used by the task in bytes.

Additional information

If `NULL` is passed for `pTask`, the currently running task is used. If this function is not called from a task context, no task might currently be running and there is no valid task. A debug build of embOS will call `OS_Error()` in this case. We suggest to call this function from a context other than the task context with a pointer to a valid task control block only.

In most cases, the stack size required by a task cannot be easily calculated, because it takes quite some time to calculate the worst-case nesting and the calculation itself is difficult.

However, the required stack size can be calculated using the function `OS_STACK_GetTaskStackUsed()`, which returns the number of used bytes on the stack. If there is a lot of space left, you can reduce the size of this stack. The return value of this function is valid only in embOS stack check builds. In all other builds it will be 0.

Note

This routine does not reliably detect the amount of stack space used, because it can only detect modified bytes on the stack. Unfortunately, space used for register storage or local variables is not always modified. In most cases, this routine will detect the correct amount of stack bytes, but in case of doubt, be generous with your stack space or use other means to verify that the allocated stack space is sufficient.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
128: OS_ERR_INV_TASK
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void CheckStackUsed(void) {
    printf("Used Stack[0] %u", OS_STACK_GetTaskStackUsed(&TCB[0]));
    OS_TASK_Delay(1000);
    printf("Used Stack[1] %u", OS_STACK_GetTaskStackUsed(&TCB[1]));
    OS_TASK_Delay(1000);
}
```

26.2.9 OS_STACK_GetSysStackBase()

Description

Returns a pointer to the base of the system stack.

Prototype

```
void* OS_STACK_GetSysStackBase(void);
```

Return value

The pointer to the base address of the system stack. The return value of this function is valid only when an system stack exists. In all other builds it will be 0.

Example

```
void CheckSysStackBase(void) {  
    printf("Addr System Stack %p", OS_STACK_GetSysStackBase());  
}
```

26.2.10 OS_STACK_GetSysStackSize()

Description

Returns the size of the system stack.

Prototype

```
unsigned int OS_STACK_GetSysStackSize(void);
```

Return value

The size of the system stack in bytes. The return value of this function is valid only when an system stack exists. In all other builds it will be 0.

Example

```
void CheckSysStackSize(void) {  
    printf("Size System Stack %u", OS_STACK_GetSysStackSize());  
}
```

26.2.11 OS_STACK_GetSysStackSize()

Description

Returns the amount of system stack which was never used (Free system stack space).

Prototype

```
unsigned int OS_STACK_GetSysStackSize(void);
```

Return value

Amount of unused system stack, in bytes.

Additional information

The return value of this function is valid only in embOS stack check builds and when an system stack exists. In all other builds it will be 0.

Note

This routine does not reliably detect the amount of stack space left, because it can only detect modified bytes on the stack. Unfortunately, space used for register storage or local variables is not always modified. In most cases, this routine will detect the correct amount of stack bytes, but in case of doubt, be generous with your stack space or use other means to verify that the allocated stack space is sufficient.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void CheckSysStackSize(void) {  
    printf("Unused System Stack %u", OS_STACK_GetSysStackSize());  
}
```

26.2.12 OS_STACK_GetSysStackUsed()

Description

Returns the maximal amount of system stack which has been used.

Prototype

```
unsigned int OS_STACK_GetSysStackUsed(void);
```

Return value

Amount of used system stack, in bytes.

Additional information

The return value of this function is valid only in embOS stack check builds and when an system stack exists. In all other builds it will be 0.

Note

This routine does not reliably detect the amount of stack space used, because it can only detect modified bytes on the stack. Unfortunately, space used for register storage or local variables is not always modified. In most cases, this routine will detect the correct amount of stack bytes, but in case of doubt, be generous with your stack space or use other means to verify that the allocated stack space is sufficient.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
164: OS_ERR_OS_INT_ENTER_NOT_CALLED
```

```
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void CheckSysStackUsed(void) {  
    printf("Used System Stack %u", OS_STACK_GetSysStackUsed());  
}
```

26.2.13 OS_STACK_GetCheckLimit()

Description

Returns the stack check limit in percent.

Prototype

```
OS_U8 OS_STACK_GetCheckLimit(void);
```

Return value

The stack check limit as a percentaged value of the stack size.

Additional information

This function is only available when the embOS compile time switch `OS_SUPPORT_STACKCHECK` is set to 2. This is e.g. the default in safety builds of embOS (`OS_LIBMODE_SAFE`). In all other embOS builds the stack check limit is fixed at 100%.

Note

This setting is jointly used for the system stack, the interrupt stack and all task stacks.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR  
161: OS_ERR_ILLEGAL_IN_TIMER  
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {  
    OS_U8 Limit;  
  
    Limit = OS_STACK_GetCheckLimit()  
    printf("Limit: %u\n", Limit);  
}
```

26.2.14 OS_STACK_SetCheckLimit()

Description

Sets the stack check limit to a percentaged value of the stack size.

Prototype

```
void OS_STACK_SetCheckLimit(OS_U8 Limit);
```

Parameters

Parameter	Description
Limit	Stack check limit in percent. Valid values are 0..100%. Values above 100% are trimmed to 100%.

Additional information

This function is only available when the embOS compile time switch `OS_SUPPORT_STACKCHECK` is set to 2. This is e.g. the default in safety builds of embOS (`OS_LIBMODE_SAFE`). In all other embOS builds the stack check limit is fixed at 100%. It can be used to set the stack check limit to a value which triggers the error condition before the stack is filled completely. With the safety build of embOS the application can react before the stack actually overflows.

Note

This routine must only be called from `main()` or privileged tasks. This setting is jointly used for the system stack, the interrupt stack and all task stacks. The best practice is to call it in `main()` before `OS_Start()`.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

```
160: OS_ERR_ILLEGAL_IN_ISR
161: OS_ERR_ILLEGAL_IN_TIMER
167: OS_ERR_CPU_STATE_ILLEGAL
```

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
int main(void) {
    OS_Init();
    OS_Inithw();
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_STACK_SetCheckLimit(70); // Set the stack check limit to 70%
    OS_Start();
}
```

Chapter 27

Board Support Packages

27.1 Introduction

This chapter explains the target system specific parts of embOS, called BSP (board support package).

In general, no modifications to the board support package are required to get started with embOS. The board support packages supplied with your embOS shipment will execute out of the box on the dedicated board. Small modifications to the configuration might be necessary at a later point, for example to use a different hardware timer for the system tick or in order to enable a UART for the optional communication with embOSView.

All mandatory hardware-specific routines that may require modifications are located in the file `RTOSInit.c`. The file `RTOSInit.c` is provided in source code in every board support package. Furthermore, the `BSP.c` as well as optional `BSP_*.c` files are provided in source code. The `BSP.c` contains routines to initialize and control LEDs. Hence, it is not vital for embOS but may be used in some embOS sample applications. The `BSP_*.c` files contain initialization for other hardware like UARTs for embOSView communication or external memory.

Some board support packages include additional files for e.g. clock and PLL initialization. Further details on these are available with the CPU & Compiler specifics manual of the embOS documentation.

27.2 How to create a new board support package

If none of the shipped board support packages matches your target hardware it might be necessary to create a new board support package. This can be done on your own or can be requested to be done by SEGGER. Please contact us for further information on the latter. Good practice is to make a copy of an existing board support package and modify it according to the target hardware. The following chapters explain which routines need to be updated.

27.3 Example

This `RTOSInit.c` serves as a template and shows the basic structure.

```
#include "RTOS.h"

#define OS_TIMER_FREQ (168000000u) // Peripheral clock for timer
#define OS_TICK_FREQ (1000u) // System tick frequency
#define OS_INT_FREQ (OS_TICK_FREQ) // Timer interrupt frequency

static unsigned int _OS_GetHWTimerCycles(void) {
    return TIMER_COUNT_REGISTER;
}

static unsigned int _OS_GetHWTimer_IntPending(void) {
    return TIMER_PENDING_FLAG;
}

void SysTick_Handler(void) {
    OS_INT_EnterNestable();
    OS_TICK_Handle();
    OS_INT_LeaveNestable();
}

static const OS_SYSTIMER_CONFIG SysTimerConfig = {OS_TIMER_FREQ,
                                                  OS_INT_FREQ,
                                                  OS_TIMER_DOWNCOUNTING,
                                                  _OS_GetHWTimerCycles,
                                                  _OS_GetHWTimer_IntPending};

void OS_InitHW(void) {
    OS_INT_IncDI();
    //
    // Initialize timer for embOS
    //

    //
    // Setup timing information
    //
    OS_TIME_ConfigSysTimer(&SysTimerConfig);
    //
    // Initialize communication for embOSView
    //

    OS_INT_DecRI();
}

void OS_Idle(void) { // Idle loop: No task is ready to execute
    while (1) { // Nothing to do ... wait for interrupt
    }
}

void OS_COM_Send1(OS_U8 c) {
}
```

27.4 Mandatory routines

The following routines are not exposed as user API, but are instead required by embOS for internal usage. They are shipped as source code to allow for modifications to match your actual target hardware. However, unless explicitly stated otherwise, these functions must not be called from your application. Usually they are implemented in a file named `RTOSInit.c`.

Routine	Description
Mandatory for embOS	
<code>_OS_GetHWTimerCycles()</code>	Returns the current hardware timer count value.
<code>_OS_GetHWTimerIntPending()</code>	Returns if the hardware timer interrupt pending flag is set.
<code>OS_Idle()</code>	The idle loop is executed whenever no task is ready for execution.
Mandatory for embOSView	
<code>OS_COM_Send1()</code>	Sends one character towards embOSView.

`_OS_GetHWTimerCycles()` and `_OS_GetHWTimer_IntPending()` are passed as function pointers to `OS_TIME_ConfigSysTimer()`. Therefore the actual names are not vital. `OS_TIME_ConfigSysTimer()` must be called before `OS_Start()`. Usually it is called in `OS_InitHW()`. For details and samples, refer to the chapter *Time Measurement* on page 374.

27.4.1 OS_Idle()

Description

The function `OS_Idle()` is called when no task, software timer or ISR is ready for execution. Usually, `OS_Idle()` is implemented as an endless loop without any content. However, it may be used e.g. to activate a power save mode of the target CPU.

Prototype

```
void OS_Idle(void);
```

Additional information

`OS_Idle()` is not a task: it neither has a task context nor a dedicated stack. Instead, it runs on the system's C stack, which is used by the kernel as well. Exceptions and interrupts occurring during `OS_Idle()` will return to `OS_Idle()` unless they trigger a task switch. When returning to `OS_Idle()`, execution continues from where it was interrupted. However, in case a task switch occurs during execution of `OS_Idle()`, the function is abandoned and execution will start from the beginning when it is activated again. Hence, no functionality should be implemented that relies on the stack contents to be preserved. If this is required, please consider implementing a custom idle task (*Creating a custom Idle task* on page 579).

Peripheral power control and Tickless support API functions may be called from `OS_Idle()` to save power consumption.

Calling `OS_TASK_EnterRegion()` and `OS_TASK_LeaveRegion()` from `OS_Idle()` allows to inhibit task switches during the execution of `OS_Idle()`. Running in a critical region does not block interrupts, but disables task switches until `OS_TASK_LeaveRegion()` is called. Using a critical region during `OS_Idle()` will therefore affect task activation time, but will not affect interrupt latency.

Calling interrupt enable and disable functions like `OS_INT_Enable()` and `OS_INT_Disable()` from `OS_Idle()` allows to inhibit interrupts during the execution of `OS_Idle()`. Disabling interrupts during `OS_Idle()` will therefore affect interrupt latency and task activation time.

You must not call any other embOS API from within `OS_Idle()`.

Example

```
void OS_Idle(void) { // Idle loop: No task is ready to execute
    while (1) {
    }
}
```

27.4.1.1 Creating a custom Idle task

As an alternative to `OS_Idle()`, it is also possible to create a custom "idle task". This task must run as an endless loop at the lowest task priority within the system. If no blocking function is called from that task, the system will effectively never enter `OS_Idle()`, but will execute this task instead whenever no other task, software timer or ISR is ready for execution.

Example

```
#include "RTOS.h"
#include "BSP.h"

static OS_STACKPTR int StackHP[128], StackLP[128], StackIdle[128];
static OS_TASK      TCBHP, TCBLP, TCBIde;

static void HPTask(void) {
    while (1) {
        BSP_ToggleLED(0);
        OS_TASK_Delay(50);
    }
}

static void LPTask(void) {
    while (1) {
        BSP_ToggleLED(1);
        OS_TASK_Delay(200);
    }
}

static void IdleTask(void) {
    while (1) {
        //
        // Perform idle duty, e.g.
        // - Switch off clocks for unused peripherals.
        // - Free resources that are no longer used by any task.
        // - Enter power save mode.
        //
    }
}

int main(void) {
    OS_Init();           // Initialize embOS
    OS_Inithw();        // Initialize hardware for embOS
    BSP_Init();         // Initialize LED ports
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_TASK_CREATE(&TCBIde, "Idle Task", 1, IdleTask, StackIdle);
    OS_Start();         // Start multitasking
    return 0;
}
```

27.4.2 OS_COM_Send1()

Description

Sends one character towards embOSView via the configured interface.

Prototype

```
void OS_COM_Send1(OS_U8 c);
```

Parameters

Parameter	Description
c	The character to send towards embOSView.

Additional information

This function is required for `OS_COM_SendString()` and the embOSView communication.

You must modify this routine according to your communication interface. To select a communications interface other than UART, refer to *Setup target for communication* on page 498.

27.5 Optional routines

The following routines are not called internally from embOS but usually included in the board support package. They are shipped as source code to allow for modifications to match your actual target hardware. The routine names are not vital and just an example although we suggest to use the name `OS_InitHW()` since this routine is called from all embOS example applications.

Routine	Description
<code>SystemTick_Handler()</code>	The embOS system tick timer interrupt handler.
<code>OS_InitHW()</code>	Initializes the hardware required for embOS to run.

27.5.1 SysTick_Handler()

Description

The embOS system timer tick interrupt handler.

Prototype

```
void SysTick_Handler(void);
```

Additional information

With specific embOS start projects, this handler may be implemented using a device specific interrupt name. When using a different timer, always check the specified interrupt vector.

Example

```
void SysTick_Handler(void) {  
    OS_INT_EnterNestable();  
    OS_TICK_Handle();  
    OS_INT_LeaveNestable();  
}
```

27.5.2 OS_InitHW()

Description

Initializes the hardware required for embOS to run. embOS needs a timer interrupt to determine when to activate tasks that wait for the expiration of a delay, when to call a software timer, and to keep the time variable up-to-date.

This function must be called once during `main()`.

Prototype

```
void OS_InitHW(void);
```

Additional information

You must modify this routine when a different hardware timer should be used.

With most embOS start projects, this routine may also call further, optional configuration functions, e.g. for

- Configuration of the embOS system time parameters (see `OS_TIME_ConfigSysTimer()`)
- Initialization of the communication interface to be used with `embOSView`.

Example

```
void OS_InitHW(void) {  
    //  
    // Initialize hardware timer  
    //  
  
    //  
    // Setup timing information  
    //  
    OS_TIME_ConfigSysTimer(&SysTimerConfig);  
}
```

27.6 Settings

The following defines are used in the `RTOSInit.c`.

Define	Description
<code>OS_TIMER_FREQ</code>	System tick hardware timer frequency in Hz
<code>OS_TICK_FREQ</code>	System tick frequency in Hz
<code>OS_INT_FREQ</code>	System tick interrupt frequency in Hz
Optional for embOSView	
<code>OS_VIEW_IFSELECT</code>	embOSView communication interface
<code>OS_UART</code>	Used UART
<code>OS_BAUDRATE</code>	UART baudrate

27.6.1 System tick setting

The actual CPU frequency depends on the hardware and clock/PLL initialization. embOS does not need to know the actual CPU frequency but the system tick timer frequency `OS_TIMER_FREQ`. The system tick frequency `OS_TICK_FREQ` is set per default to 1 kHz. Hence, the default time unit is 1 millisecond. Usually the system tick frequency `OS_TICK_FREQ` and the system tick interrupt frequency `OS_INT_FREQ` are equal but for some targets it might be necessary to have a different interrupt to tick ratio.

The basic time unit does not need to be 1 millisecond; it might just as well be 100 microseconds or 10 millisecond or any other value. For most applications, however, 1 milliseconds is an appropriate value.

27.6.2 Using a different hardware timer

Relevant defines

- `OS_TIMER_FREQ`
- `OS_TICK_FREQ`
- `OS_INT_FREQ`

Relevant routines

- `OS_InitHW()`

embOS usually generates one interrupt per millisecond, making the timer interrupt, or tick, normally equal to 1 millisecond. This is done by a timer initialized in the routine `OS_InitHW()`. If you want to use a different timer for your application, you must modify `OS_InitHW()` to initialize the appropriate timer.

27.6.3 Using a different UART or baud rate for embOSView

Relevant defines

- `OS_UART` (Selection of UART to be used with embOSView)
- `OS_BAUDRATE` (Selection of baud rate for communication with embOSView)

Relevant routines:

- `OS_COM_Send1()`

In some cases, this may be done by simply changing the define `OS_UART` or `OS_BAUDRATE`. Refer to the contents of the `BSP_UART.c` file for more information about which UARTs are supported on your target hardware.

27.7 UART routines for embOSView

If the board support package does not come with the necessary UART routines for embOSView you can easily implement them by yourself. The UART communication is handled via the UART interrupt. With some devices there are separate interrupt handler for Rx and Tx but the following example code assumes there is one UART interrupt handler only. You will need to initialize the UART to the same settings as configured in embOSView. Per default this is 38400 baud and 8N1.

```
void UART_IRQHandler(void) {
    OS_U8 c;
    //
    // If this is a Rx interrupt read character
    // from receive register and call OS_COM_OnRx().
    //
    if (UART_STATUS_REGISTER & UART_RX) {
        c = UART_RX_REGISTER;
        OS_COM_OnRx(c);
    }
    //
    // If this is a Tx interrupt call OS_COM_OnTx().
    // Disable further Tx interrupts when there are no more
    // bytes to sent.
    //
    if (UART_STATUS_REGISTER & UART_TX) {
        if (OS_COM_OnTx() != 0u) {
            UART_CONTROL_REGISTER &= ~UART_TX_INT_ENABLE;
        }
    }
}

void OS_COM_Send1(OS_U8 c) {
    //
    // Write character to UART transmit register
    // and enable Tx interrupt
    //
    UART_CONTROL_REGISTER |= UART_TX_INT_ENABLE;
    UART_TX_REGISTER = c;
}

void UART_Init(void) {
    //
    // Initialize UART register for 38400 baud and 8N1.
    // Enable UART Rx interrupt (but not Tx interrupt).
    //
    UART_CONTROL_REGISTER |= UART_RX_INT_ENABLE;
}
```

Chapter 28

System Variables

28.1 Introduction

The system variables are described here for a deeper understanding of how embOS works and to make debugging easier.

Not all embOS internal variables are explained here as they are not required to use embOS. Your application should not rely on any of the internal variables. Only the documented API functions are guaranteed to remain unchanged in future versions of embOS.

These variables are accessible, for instance using an IDEs watch feature, but they should only be altered by embOS. However, some of these variables can be very useful, especially the time variables.

Note

Do not alter any system variables or OS object structures!

Example

```
static OS_MUTEX Mutex;
static int      c;
static OS_TIME  t;

void foo(void) {
    Mutex.UseCnt = 0;           // Invalid
    c = Mutex.UseCnt;         // Ok, but not recommended
    c = OS_MUTEX_GetValue(&Mutex); // Ok

    OS_Global.Time = 1;       // Invalid
    t = OS_Global.Time;      // Ok, but not recommended
    t = OS_TIME_GetTicks();  // Ok
}
```

28.2 OS_Global

`OS_Global` is a structure which includes embOS internal variables. It contains information about the current state of embOS, its tasks and software timers. The members of `OS_Global` may vary depending on the embOS port and used library mode.

28.2.1 OS_Global.pCurrentTask

A pointer to the currently active task.

This information can be retrieved using the embOS API `OS_TASK_GetID()`.

28.2.2 OS_Global.pTask

This is a pointer to the first object in a linked list of all existing tasks, ordered by their priority. Each task control block contains a member `OS_TASK.pNext` which points to the next task in the list.

This information can be retrieved by passing the value zero to the embOS API `OS_TASK_Index2Ptr()`.

28.2.3 OS_Global.pTimer

This is a pointer to the first object in a linked list of all running software timers, ordered by their expiration time. Each software timer control block contains a member `OS_TIMER.pNext` which points to the next software timer in the list.

28.2.4 OS_Global.Time

This is the time variable which contains the current system time in embOS system ticks (typically equivalent to millisecond).

The time variable has a resolution of one time unit, which is normally 1/1000 sec (1 millisecond) and is normally the time between two successive calls to the embOS timer interrupt handler. Instead of accessing this variable directly, use `OS_TIME_GetTicks()` or `OS_TIME_GetTicks32()` as explained in the Chapter *Time Measurement* on page 374.

28.2.5 OS_Global.TimeDex

For internal use only. Contains the time at which the next task switch or timer activation is due. If $((\text{int})(\text{OS_Global.Time} - \text{OS_Global.TimeDex})) \neq 0$, the task list and timer list will be checked for a task or timer to activate. After activation, `OS_Global.TimeDex` will be assigned the time stamp of the next task or timer to be activated.

Note that the value of `OS_Global.TimeDex` may be invalid during task execution. It contains correct values during execution of `OS_Idle()` and when used internally in the embOS scheduler. The value of `OS_Global.TimeDex` should not be used by the application.

If you need any information about the next time-scheduled action from embOS, the function `OS_TICKLESS_GetNumIdleTicks()` can be used to get the number of system ticks spent idle.

28.3 OS information routines

28.3.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
<code>OS_INFO_GetCPU()</code>	Returns the CPU name.	•	•	•	•	•
<code>OS_INFO_GetLibMode()</code>	Returns the library mode.	•	•	•	•	•
<code>OS_INFO_GetLibName()</code>	Returns the library name.	•	•	•	•	•
<code>OS_INFO_GetModel()</code>	Returns the memory model name.	•	•	•	•	•
<code>OS_INFO_GetTimerFreq()</code>	Returns the system tick hardware timer frequency in hertz.	•	•	•	•	•
<code>OS_INFO_GetVersion()</code>	Returns the embOS version number.	•	•	•	•	•

28.3.1.1 OS_INFO_GetCPU()

Description

Returns the CPU name.

Prototype

```
char *OS_INFO_GetCPU(void);
```

Return value

Char pointer to a null-terminated string containing the CPU name.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: OS_ERR_OS_INT_ENTER_NOT_CALLED

167: OS_ERR_CPU_STATE_ILLEGAL

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintCPUName(void) {
    char* Name;

    Name = OS_INFO_GetCPU();
    printf("CPU: %s\n", Name);
}
```

28.3.1.2 OS_INFO_GetLibMode()

Description

Returns the library mode.

Prototype

```
char *OS_INFO_GetLibMode(void);
```

Return value

Char pointer to a null-terminated string containing the embOS library mode, e.g. "DP", "R" or "SP".

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintLibMode(void) {
    char* Mode;

    Mode = OS_INFO_GetLibMode();
    printf("Library Mode: %s\n", Mode);
}
```

28.3.1.3 OS_INFO_GetLibName()

Description

Returns the library name.

Prototype

```
char *OS_INFO_GetLibName(void);
```

Return value

Char pointer to a null-terminated string containing the complete embOS library name, memory model and library mode, e.g. "v7vLDP".

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintLibName(void) {
    char* LibName;

    LibName = OS_INFO_GetLibName();
    printf("Full Library Name: %s\n", LibName);
}
```

28.3.1.4 OS_INFO_GetModel()

Description

Returns the memory model name.

Prototype

```
char *OS_INFO_GetModel(void);
```

Return value

Char pointer to a null-terminated string containing the embOS memory model string, e.g. "v7vL".

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintMemModel(void) {
    char* Model;

    Model = OS_INFO_GetModel();
    printf("Memory Model: %s\n", Model);
}
```

28.3.1.5 OS_INFO_GetTimerFreq()

Description

Returns the system tick hardware timer frequency in hertz.

Prototype

```
OS_U32 OS_INFO_GetTimerFreq(void);
```

Return value

The system tick hardware timer frequency in hertz as a 32-bit value.

Additional information

`OS_INFO_GetTimerFreq()` returns the timer frequency which was set with `OS_TIME_ConfigSysTimer()`. This value is the frequency at which the hardware timer counts and is the amount of timer cycles per second. It is not the system timer tick frequency nor the system timer interrupt frequency. Hence, it can be used to calculate the actual time in seconds, microseconds or nanoseconds from timer cycles.

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void Task(void) {
    OS_U64 t, t0;
    OS_U32 TimerFreq;
    OS_U64 Result;

    TimerFreq = OS_INFO_GetTimerFreq();
    t0 = OS_TIME_Get_Cycles();
    DoSomething();
    t = OS_TIME_Get_Cycles() - t0;
    // Result in nanoseconds
    Result = (t * 1000000000) / TimerFreq;
}
```

28.3.1.6 OS_INFO_GetVersion()

Description

Returns the embOS version number.

Prototype

```
OS_UINT OS_INFO_GetVersion(void);
```

Return value

Returns the embOS version number, e.g. "50801" for embOS version 5.8.1. The version number is defined as: $\text{Version} = (\text{Major} * 10000) + (\text{Minor} * 100) + \text{Patch} + (\text{Revision} * 25)$

Error codes

With embOS debug checks enabled erroneous calls to this function result in `OS_Error()` being called with one of the following application error IDs:

164: `OS_ERR_OS_INT_ENTER_NOT_CALLED`

167: `OS_ERR_CPU_STATE_ILLEGAL`

For details, refer to the chapter *Runtime application errors* on page 455.

Example

```
void PrintOSVersion(void) {
    OS_U16 Version;
    OS_U8  Major;
    OS_U8  Minor;
    OS_U8  Patch;
    OS_U8  Revision;

    Version = OS_INFO_GetVersion();
    Major   = Version / 10000u;
    Minor   = (Version / 100u) % 100u;
    Patch   = (Version % 100u) % 25u;
    Revision = (Version % 100u) / 25u;

    printf("embOS V%u.%u.%u.%u\n", Major, Minor, Patch, Revision);
}
```

Chapter 29

Source Code

29.1 Introduction

embOS is available in two packages:

- Object code package: embOS object code libraries.
- Source code package: embOS object code libraries + embOS source code.

Both packages come with board support packages including hardware initialization as source code.

The object code package offers the full functionality of embOS including all supported memory models of the compiler. However, the object code package does not allow source-level debugging of the library routines and the kernel.

The full source code package gives you complete flexibility:

- All embOS features can be individually enabled and disabled.
- Different compile options give you full control, e.g. to use the most suitable optimization settings for a specific application.
- Use potential optimizations across C files which cannot be done with object code libraries.
- Debug the entire system by using the embOS source code in your project which gives the ability to step through all embOS API functions and kernel code.

The source code distribution of embOS contains the following additional files:

- The `CPU` folder contains all CPU and compiler-specific source code and header files used for building the embOS libraries. Generally, you should not modify any of the files in the `CPU` folder.
- The `GenOSSrc` folder contains all generic embOS sources.
- The embOS libraries can be rebuilt with the additional batch files in the root folder. All of them are described in the following section.

Note

You must not modify the embOS sources as we guarantee faultless operation with unmodified embOS sources only. If source modifications are necessary please contact the embOS support first.

29.2 Building embOS libraries

The embOS libraries can only be built if you have licensed a source code package of embOS.

In the root path of embOS, you will find a DOS batch file `Prep.bat`, which needs to be modified to match the installation directory of your C compiler. Once this is done, you can call the batch file `M.bat` to build all embOS libraries and `RTOS.h` for your CPU.

The build process should run without any error or warning message. If the build process reports any problem, check the following:

- Are you using the same compiler version as mentioned in the file `Release.html`?
- Can you compile a simple test file after running `Prep.bat` and does it really use the compiler version you have specified?
- Is there anything mentioned about possible compiler warnings in the `Release.html`?

If you still have a problem, let us know.

The whole build process is controlled with a small number of batch files which are located in the root directory of your source code package:

- `ASM.bat`: This batch file calls the assembler and is used for assembling the assembly part of embOS which contains the task switch functionality. This file is called from the embOS internal batch file `CC_OS.bat` and cannot be called directly.
- `ASM_CPU.bat`: This batch file is used to compile additional assembler files in the `CPU/OSSrcCPU` folder. `ASM_CPU.bat` cannot be called directly.
- `CC.bat`: This batch file calls the compiler and is used for compiling one embOS source file without debug information output. Most compiler options are defined in this file and generally should not be modified. For your purposes, you might activate debug output and may also modify the optimization level. All modifications should be done with care. This file is called from the embOS internal batch file `CC_OS.bat` and cannot be called directly.
- `CC_CPU.bat`: This batch file is used to compile additional C files in the `CPU/OSSrcCPU` folder. `CC_CPU.bat` cannot be called directly.
- `CCD.bat`: This batch file calls the compiler and is used for compiling `OS_Global.c` which contains all global variables. All compiler settings are identical to those used in `CC.bat`, except debug output is activated to enable debugging of global variables when using embOS libraries. This file is called from the embOS internal batch file `CC_OS.bat` and cannot be called directly.
- `Clean.bat`: Deletes the entire output of the embOS library build process. It is called during the build process, before new libraries are generated. It deletes the `Start` folder. Therefore, be careful not to call this batch file accidentally. This file is called initially by `M.bat` during the build process of all libraries.
- `M.bat`: This batch file must be called to rebuild all embOS libraries and `RTOS.h`. It initially calls `Clean.bat` and therefore deletes the previous libraries and `RTOS.h`.
- `M1.bat`: This batch file is called from `M.bat` and is used for building one specific embOS library, it cannot be called directly.
- `MakeH.bat`: Builds the embOS header file `RTOS.h` which is composed from the `CPU/compiler-specific` part `OSCHIP.h` and the generic part `OS_RAW.h`. `RTOS.h` is output in the subdirectory `Start\Inc`.
- `Prep.bat`: Sets up the environment for the compiler, assembler, and linker. Ensure that this file sets the path and additional include directories which are needed for your compiler. This batch file is the only one which might require modifications to build the embOS libraries. This file is called from `M.bat` during the build process of all libraries.

29.3 Compile time switches

Many features of embOS may be modified using compile-time switches. With each embOS distribution, these switches are preset to appropriate values for each embOS library mode. In case a configuration set is desired that was not covered by the shipped embOS libraries, the compile-time switches may be modified accordingly to create customized configurations on your own authority. The embOS source code is necessary in order to do so.

Note

You must not use these compile time switches with the embOS object code package.

According modifications must not be done to `OS_RAW.h` or `RTOS.h`, instead compile-time switches must be added to `OS_Config.h` or configured as preprocessor definitions. Subsequently, the embOS sources must be recompiled to reflect the modified switches. In case of doubt, please contact the embOS support for assistance. The default values depend on the used library mode and are given in the following table for library mode `OS_LIBMODE_DP`.

embOS features

Compile time switch	Description	Permitted values	Default
<code>OS_DEBUG</code>	Enables runtime debug checks	0: Disabled 1: Enabled	1
<code>OS_SUPPORT_STACKCHECK</code>	Support for stack checks	0: Disabled 1: Enabled 2: Stack check with configurable stack check limit	1
<code>OS_SUPPORT_PROFILE</code>	Support for profiling information	0: Disabled 1: Enabled	1
<code>OS_SUPPORT_TICKSTEP</code>	Support for embOSView tick step	0: Disabled 1: Enabled	0
<code>OS_SUPPORT_TRACE</code>	Support for embOSView trace	0: Disabled 1: Enabled	0
<code>OS_SUPPORT_TRACE_API</code>	Support for API trace tools like SystemView	0: Disabled 1: Enabled	1
<code>OS_SUPPORT_TRACE_API_END</code>	Generates additional Trace API-End events	0: Disabled 1: Enabled	1
<code>OS_SUPPORT_RR</code>	Support for Round-Robin scheduling	0: Disabled 1: Enabled	1
<code>OS_SUPPORT_TRACKNAME</code>	Support for task and OS object names	0: Disabled 1: Enabled	1
<code>OS_SUPPORT_SAVE_RESTORE_HOOK</code>	Support for task context extensions	0: Disabled 1: Enabled	1
<code>OS_SUPPORT_STAT</code>	Support for task statistic information	0: Disabled 1: Enabled	1
<code>OS_INIT_EXPLICITLY</code>	Explicitly initialization of internal embOS variables	0: Disabled 1: Enabled	0
<code>OS_SUPPORT_TIMER</code>	Support for embOS software timers	0: Disabled 1: Enabled	1

Compile time switch	Description	Permitted values	Default
OS_SUPPORT_TICKLESS	Support for embOS tickless mode	0: Disabled 1: Enabled	1
OS_SUPPORT_PERIPHERAL_POWER_CTRL	Support for peripheral power control	0: Disabled 1: Enabled	1

embOS settings

Compile time switch	Description	Permitted values	Default
OS_STACKCHECK_LIMIT	Percentage of stack usage that will be detected as a stack overflow error	1-100	100 OS_LIBMODE_SAFE: 70
OS_POWER_NUM_COUNTERS	Number of peripherals which can be used	> 0	5
OS_SPINLOCK_MAX_CORES	Number of cores that should access a spinlock	> 0	4
OS_COM_OUT_BUFFER_SIZE	embOSView communication buffer size	200 - 65535	200
OS_TASKEVENT	Size of task event	OS_U8 OS_U16 OS_U32	8/16-bit CPUs: OS_U8 32-bit CPUs: OS_U32
OS_TASK_PRIORITY	Size of task priority	OS_U8 OS_U16 OS_U32	8/16-bit CPUs: OS_U8 32-bit CPUs: OS_U32

29.4 Source code project

All embOS start projects use the embOS libraries instead of the embOS source code. Even the embOS source shipment does not include a project which uses embOS sources.

It can be useful to have the embOS sources instead of the embOS library in a project, e.g. for easier debugging. To do so you just have to exclude or delete the embOS library from your project and add the embOS sources as described below.

The embOS sources consists of the files in the folder `GenOSSrc`, `CPU` and `CPU\OSSrcCPU`. These files can be found in the embOS source shipment.

Folder	Description
GenOSSrc	embOS generic sources
CPU	RTOS assembler file
CPU\OSSrcCPU	CPU and compiler-specific files

Please add all C and assembler files from these folders to your project and add include paths to these folders to your project settings. For some embOS ports it might be necessary to add additional defines to your preprocessor settings. If necessary you will find more information about it in the CPU and compiler-specific embOS manual.

Note

You must not change compiler settings for single embOS source files only. All embOS source files must be built with the same compiler settings.

29.4.1 Compiler options

While the embOS libraries are built with specific compiler options it is possible to build a source code project with modified or additional compiler options. Some compiler options could require changes in the embOS source code. In case of doubt please contact the embOS support.

Chapter 30

Shipment

30.1 Introduction

embOS can be commercially licensed as object code package or source code package. The source code package extends the object code package by including the embOS source code in addition. The object code package is also available under [SEGGER's Friendly License \(SFL\)](#). This means embOS evaluation and non-commercial use is technically unrestricted.

The following table lists the included features with each package:

Features	Object code package	Source code package
embOS object code	•	•
embOS source code		•
embOSView - Profiling PC tool	•	•
embOS manual	•	•
CPU/Compiler specific manual	•	•
Release notes	•	•
embOS IDE plug-ins	•	•
SystemView instrumentation	•	•
Board support packages	•	•
Feature & maintenance updates	•	•
Technical support	•	•

30.2 Object code package

-  embOSView
-  Start
-  Release_embOS.html
-  Release_embOS_CPU_Compiler.html
-  SYSVIEW_embOS.txt
-  UM01001_embOS.pdf
-  UM01061_embOS_CPU_Compiler.pdf

Directory	File	Description
embOSView	embOSView.exe and embOSView related files	PC utility for runtime analysis (Windows x64 version)
Start\BoardSupport		Board support packages in vendor specific subdirectories
Start\Inc	RTOS.h, BSP.h, OS_Config.h	Include files for embOS
Start\Lib		embOS libraries
	Release_embOS.html	embOS release history
	Release_embOS_CPU_Compiler.html	embOS CPU and compiler-specific release history
	SYSVIEW_embOS.txt	SytemView ID description file
	UM010xx_embOS_CPU_Compiler.pdf	embOS CPU and compiler-specific manual
	UM01001_embOS.pdf	embOS manual

30.3 Source code package

The source code package is identical to the object code package, but in addition also contains the embOS source files and a set of batch files that can be used to rebuild the embOS libraries.

-  CPU
-  embOSView
-  GenOSSrc
-  Start
-  ASM.bat
-  ASM_CPU.bat
-  CC.bat
-  CC_CPU.bat
-  CCD.bat
-  Clean.bat
-  M.bat
-  M1.bat
-  MakeH.bat
-  Prep.bat
-  Release_embOS.html
-  Release_embOS_CPU_Compiler.html
-  SYSVIEW_embOS.txt
-  UM01001_embOS.pdf
-  UM01061_embOS_CPU_Compiler.pdf

Directory	File	Description
embOSView	embOSView.exe and embOSView related files	PC utility for runtime analysis (Windows x64 version)
CPU	OSCHIP.h, OS_Priv.h, RTOS.asm	CPU- and compiler-specific files
CPU\OSSrcCPU		Additional CPU- and compiler-specific source files
GenOSSrc		Generic source files
Start\BoardSupport		Board support packages in vendor specific subdirectories
Start\Inc	RTOS.h, BSP.h, OS_Config.h	Include files for embOS
Start\Lib		embOS libraries
Start\Src		Only used with embOS-Safe, contains additional source files
	Release_embOS.html	embOS release history
	Release_embOS_CPU_Compiler.html	embOS CPU and compiler-specific release history
	SYSVIEW_embOS.txt	SystemView ID description file
	UM010xx_embOS_CPU_Compiler.pdf	embOS CPU and compiler-specific manual
	UM01001_embOS.pdf	embOS manual
	*.bat	Batch files to rebuild the embOS libraries

Chapter 31

Update

31.1 Introduction

This chapter describes how to update an existing project with a newer embOS version. embOS ports are available for different CPUs and compiler. Each embOS port has its own version number.

SEGGER updates embOS ports to a newer software version for different reasons. This is done to fix problems or to include the newest embOS features.

Customers which have a valid support and update agreement will be automatically informed about a new software version via email and may subsequently download the updated software from myaccount.segger.com. The version information and release history is also available at www.segger.com.

31.2 How to update an existing project

If an existing project should be updated to a later embOS version, only files have to be replaced.

Note

Do not use embOS files from different embOS versions in your project!

You should have received the embOS update as a zip file. Unzip this file to the location of your choice and replace all embOS files in your project with the newer files from the embOS update shipment. To do so it is good practice to remove all embOS files from the project and to add then the embOS files from the most recent embOS shipment. embOS files in the most recent embOS shipment could be removed, renamed or new files could be added.

For an easier update procedure, we recommend to not modify the files shipped with embOS. In case these need to be updated, you will have to merge your modifications into the most recent shipment version of that file, or else your modifications will be lost.

In general, the following files have to be updated:

File	Location	Description
embOS libraries	Start\Lib	embOS object code libraries
RTOS.h	Start\Inc	embOS header file
OS_Config.h	Start\Inc	embOS configuration header file
BSP.h	Start\Inc	Board support header file
RTOSInit.c	Start\BoardSupport\...\Setup	BSP related routines
OS_Error.c	Start\BoardSupport\...\Setup	embOS error routine
Additional files	Start\BoardSupport\...\Setup	BSP specific files

If the embOS sources are used in a project the following files needs to be updated as well:

Location	Description
GenOSSrc	embOS generic sources
CPU CPU\OSSrcCPU	embOS CPU and compiler specific sources

31.2.1 My project does not work anymore. What did I do wrong?

One common mistake is to only update the embOS library but not `RTOS.h`. You should always ensure the embOS library and `RTOS.h` belong to the same embOS port version. Also, please ensure further embOS files like `OS_Error.c` and `RTOSInit.c` have been updated to the same version. If you are still experiencing problems, please do not hesitate to contact the embOS support (see *Contacting support* on page 617).

31.3 embOS migration guide

embOS API and RTOS object type names may change with later embOS versions. For example with embOS V5.00 a new embOS API naming scheme was introduced and most embOS API names and some object type names have changed.

Legacy embOS API names can always still be used and there is no need to update the API names in an existing application. embOS always stays 100% API compatible. However, for new projects the latest API names should be used.

If you want to use the latest API names in an existing application you can easily replace all API calls. SEGGER provides a CSV file on request which can be used to automatically replace all API calls.

Please be aware with V5.00 the parameter order of some API functions has changed. This needs to be adapted manually.

`OS_TASK_CREATE()`/`OS_TASK_CREATEEX()`: The order of the parameters `Priority` and `pRoutine` has changed.

`OS_TASKEVENT_Set()`: The order of the parameters `pTask` and `Event` has changed.

`OS_MEMPOOL_Alloc()`/`OS_MEMPOOL_AllocBlocked()`/`OS_MEMPOOL_AllocTimed()`: The parameter `Purpose` does not longer exist.

Misspelled names in older manual versions	Correct name
<code>OS_CREATEEMB()</code>	<code>OS_MAILBOX_Create()</code>
<code>OS_GetUseCnt()</code>	<code>OS_MUTEX_GetValue()</code>
<code>OS_WaitCSema_Timed()</code>	<code>OS_SEMAPHORE_TakeTimed()</code>
<code>OS_WaitEvent_Timed()</code>	<code>OS_TASKEVENT_GetTimed()</code>
<code>OS_GetEventsOccured()</code>	<code>OS_TASKEVENT_Get()</code>
V3.00	V3.01
<code>OS_GetType()</code>	<code>OS_GetLibMode()</code>
V3.60d	V3.60e
<code>OS_HandleTick()</code>	<code>OS_TICK_Handle()</code>
<code>OS_HandleTickEx()</code>	<code>OS_TICK_HandleEx()</code>
<code>OS_AddTickHook()</code>	<code>OS_TICK_AddHook()</code>
<code>OS_RemoveTickHook()</code>	<code>OS_TICK_RemoveHook()</code>
V3.60e	V3.62
<code>OS_TASKID</code>	<code>OS_TASK*</code>
V3.82u	V3.82v
<code>OS_Terminate()</code>	<code>OS_TASK_Terminate()</code>
V4.26	V4.30
<code>OS_AddOnTerminateHook()</code>	<code>OS_TASK_AddTerminateHook()</code>
<code>OS_RemoveOnTerminateHooks()</code>	<code>OS_TASK_RemoveAllTerminateHooks()</code>
<code>OS_ResumeAllSuspendedTasks()</code>	<code>OS_TASK_ResumeAll()</code>
V4.40	V5.00
<code>OS_Config_Stop()</code>	<code>OS_ConfigStop()</code>
<code>OS_InitKern()</code>	<code>OS_Init()</code>
<code>OS_AddExtendTaskContext()</code>	<code>OS_TASK_AddContextExtension()</code>
<code>OS_AddTerminateHook()</code>	<code>OS_TASK_AddTerminateHook()</code>

V4.40	V5.00
OS_CREATETASK()	OS_TASK_CREATE()
OS_CreateTask()	OS_TASK_Create()
OS_CREATETASK_EX()	OS_TASK_CREATEEX()
OS_CreateTaskEx()	OS_TASK_CreateEx()
OS_Delay()	OS_TASK_Delay()
OS_DelayUntil()	OS_TASK_DelayUntil()
OS_Delayus()	OS_TASK_Delay_us()
OS_GetTaskName()	OS_TASK_GetName()
OS_GetNumTasks()	OS_TASK_GetNumTasks()
OS_GetPriority()	OS_TASK_GetPriority()
OS_GetSuspendCnt()	OS_TASK_GetSuspendCnt()
OS_GetTaskID()	OS_TASK_GetID()
OS_GetTimeSliceRem()	OS_TASK_GetTimeSliceRem()
OS_IsTask()	OS_TASK_IsTask()
OS_TaskIndex2Ptr()	OS_TASK_Index2Ptr()
OS_RemoveAllTerminateHooks()	OS_TASK_RemoveAllTerminateHooks()
OS_RemoveTerminateHook()	OS_TASK_RemoveTerminateHook()
OS_Resume()	OS_TASK_Resume()
OS_ResumeAllTasks()	OS_TASK_ResumeAll()
OS_ExtendTaskContext()	OS_TASK_SetContextExtension()
OS_SetDefaultTaskContextExtension()	OS_TASK_SetDefaultContextExtension()
OS_SetDefaultTaskStartHook()	OS_TASK_SetDefaultStartHook()
OS_SetInitialSuspendCnt()	OS_TASK_SetInitialSuspendCnt()
OS_SetTaskName()	OS_TASK_SetName()
OS_SetPriority()	OS_TASK_SetPriority()
OS_SetTimeSlice()	OS_TASK_SetTimeSlice()
OS_Suspend()	OS_TASK_Suspend()
OS_SuspendAllTasks()	OS_TASK_SuspendAll()
OS_TerminateTask()	OS_TASK_Terminate()
OS_WakeTask()	OS_TASK_Wake()
OS_Yield()	OS_TASK_Yield()
OS_CREATETIMER()	OS_TIMER_CREATE()
OS_CreateTimer()	OS_TIMER_Create()
OS_CREATETIMER_EX()	OS_TIMER_CREATEEX()
OS_CreateTimerEx()	OS_TIMER_CreateEx()
OS_DeleteTimer()	OS_TIMER_Delete()
OS_DeleteTimerEx()	OS_TIMER_DeleteEx()
OS_GetpCurrentTimer()	OS_TIMER_GetCurrent()
OS_GetpCurrentTimerEx()	OS_TIMER_GetCurrentEx()
OS_GetTimerPeriod()	OS_TIMER_GetPeriod()
OS_GetTimerPeriodEx()	OS_TIMER_GetPeriodEx()
OS_GetTimerValue()	OS_TIMER_GetRemainingPeriod()
OS_GetTimerValueEx()	OS_TIMER_GetRemainingPeriodEx()

V4.40	V5.00
OS_GetTimerStatus()	OS_TIMER_GetStatus()
OS_GetTimerStatusEx()	OS_TIMER_GetStatusEx()
OS_RetriggerTimer()	OS_TIMER_Restart()
OS_RetriggerTimerEx()	OS_TIMER_RestartEx()
OS_SetTimerPeriod()	OS_TIMER_SetPeriod()
OS_SetTimerPeriodEx()	OS_TIMER_SetPeriodEx()
OS_StartTimer()	OS_TIMER_Start()
OS_StartTimerEx()	OS_TIMER_StartEx()
OS_StopTimer()	OS_TIMER_Stop()
OS_StopTimerEx()	OS_TIMER_StopEx()
OS_TriggerTimer()	OS_TIMER_Trigger()
OS_TriggerTimerEx()	OS_TIMER_TriggerEx()
OS_ClearEvents()	OS_TASKEVENT_Clear()
OS_ClearEventsEx()	OS_TASKEVENT_ClearEx()
OS_GetEventsOccurred()	OS_TASKEVENT_Get()
OS_WaitEvent()	OS_TASKEVENT_GetBlocked()
OS_WaitSingleEvent()	OS_TASKEVENT_GetSingleBlocked()
OS_WaitSingleEventTimed()	OS_TASKEVENT_GetSingleTimed()
OS_WaitEventTimed()	OS_TASKEVENT_GetTimed()
OS_SignalEvent()	OS_TASKEVENT_Set()
OS_EVENT_Wait()	OS_EVENT_GetBlocked()
OS_EVENT_WaitMask()	OS_EVENT_GetMaskBlocked()
OS_EVENT_WaitMaskTimed()	OS_EVENT_GetMaskTimed()
OS_EVENT_WaitTimed()	OS_EVENT_GetTimed()
OS_CreateR sema()	OS_MUTEX_Create()
OS_CREATERSEMA()	OS_MUTEX_Create()
OS_DeleteR sema()	OS_MUTEX_Delete()
OS_GetResourceOwner()	OS_MUTEX_GetOwner()
OS_GetSemaValue()	OS_MUTEX_GetValue()
OS_Request()	OS_MUTEX_Lock()
OS_Use()	OS_MUTEX_LockBlocked()
OS_UseTimed()	OS_MUTEX_LockTimed()
OS_Unuse()	OS_MUTEX_Unlock()
OS_CREATECSEMA()	OS_SEMAPHORE_CREATE()
OS_CreateCSema()	OS_SEMAPHORE_Create()
OS_DeleteCSema()	OS_SEMAPHORE_Delete()
OS_GetCSemaValue()	OS_SEMAPHORE_GetValue()
OS_SignalCSema()	OS_SEMAPHORE_Give()
OS_SignalCSemaMax)	OS_SEMAPHORE_GiveMax()
OS_SetCSemaValue()	OS_SEMAPHORE_SetValue()
OS_CSemaRequest()	OS_SEMAPHORE_Take()
OS_WaitCSema()	OS_SEMAPHORE_TakeBlocked()
OS_WaitCSemaTimed()	OS_SEMAPHORE_TakeTimed()

V4.40	V5.00
OS_ClearMB()	OS_MAILBOX_Clear()
OS_CreateMB()	OS_MAILBOX_Create()
OS_DeleteMB()	OS_MAILBOX_Delete()
OS_GetMailCond()	OS_MAILBOX_Get()
OS_GetMailCond1()	OS_MAILBOX_Get1()
OS_GetMail()	OS_MAILBOX_GetBlocked()
OS_GetMail1()	OS_MAILBOX_GetBlocked1()
OS_GetMessageCnt()	OS_MAILBOX_GetMessageCnt()
OS_GetMailTimed()	OS_MAILBOX_GetTimed()
OS_GetMailTimed1()	OS_MAILBOX_GetTimed1()
OS_Mail_GetPtrCond()	OS_MAILBOX_GetPtr()
OS_Mail_GetPtr()	OS_MAILBOX_GetPtrBlocked()
OS_PeekMail()	OS_MAILBOX_Peek()
OS_Mail_Purge()	OS_MAILBOX_Purge()
OS_PutMailCond()	OS_MAILBOX_Put()
OS_PutMailCond1()	OS_MAILBOX_Put1()
OS_PutMail()	OS_MAILBOX_PutBlocked()
OS_PutMail1()	OS_MAILBOX_PutBlocked1()
OS_PutMailFrontCond()	OS_MAILBOX_PutFront()
OS_PutMailFrontCond1()	OS_MAILBOX_PutFront1()
OS_PutMailFront()	OS_MAILBOX_PutFrontBlocked()
OS_PutMailFront1()	OS_MAILBOX_PutFrontBlocked1()
OS_PutMailTimed()	OS_MAILBOX_PutTimed()
OS_PutMailTimed1()	OS_MAILBOX_PutTimed1()
OS_WaitMail()	OS_MAILBOX_WaitBlocked()
OS_WaitMailTimed()	OS_MAILBOX_WaitTimed()
OS_Q_Clear()	OS_QUEUE_Clear()
OS_Q_Create()	OS_QUEUE_Create()
OS_Q_Delete()	OS_QUEUE_Delete()
OS_Q_GetMessageCnt()	OS_QUEUE_GetMessageCnt()
OS_Q_GetMessageSize()	OS_QUEUE_GetMessageSize()
OS_Q_GetPtrCond()	OS_QUEUE_GetPtr()
OS_Q_GetPtr()	OS_QUEUE_GetPtrBlocked()
OS_Q_GetPtrTimed()	OS_QUEUE_GetPtrTimed()
OS_Q_IsInUse()	OS_QUEUE_IsInUse()
OS_Q_PeekPtr()	OS_QUEUE_PeekPtr()
OS_Q_Purge()	OS_QUEUE_Purge()
OS_Q_Put()	OS_QUEUE_Put()
OS_Q_PutEx()	OS_QUEUE_PutEx()
OS_Q_PutBlocked()	OS_QUEUE_PutBlocked()
OS_Q_PutBlockedEx()	OS_QUEUE_PutBlockedEx()
OS_Q_PutTimed()	OS_QUEUE_PutTimed()
OS_Q_PutTimedEx()	OS_QUEUE_PutTimedEx()

V4.40	V5.00
OS_CallISR()	OS_INT_Call()
OS_CallNestableISR()	OS_INT_CallNestable()
OS_EnterInterrupt()	OS_INT_Enter()
OS_EnterIntStack()	OS_INT_EnterIntStack()
OS_EnterNestableInterrupt()	OS_INT_EnterNestable()
OS_InInterrupt()	OS_INT_InInterrupt()
OS_LeaveInterrupt()	OS_INT_Leave()
OS_LeaveIntStack()	OS_INT_LeaveIntStack()
OS_LeaveNestableInterrupt()	OS_INT_LeaveNestable()
OS_DecRI()	OS_INT_DecRI()
OS_DI()	OS_INT_Disable()
OS_INTERRUPT_MaskGlobal()	OS_INT_DisableAll()
OS_EI()	OS_INT_Enable()
OS_INTERRUPT_UnmaskGlobal()	OS_INT_EnableAll()
OS_RestoreI()	OS_INT_EnableConditional()
OS_IncDI()	OS_INT_IncDI()
OS_INT_PRIO_PRESERVE()	OS_INT_Preserve()
OS_INTERRUPT_PreserveGlobal()	OS_INT_PreserveAll()
OS_INTERRUPT_PreserveAndMaskGlobal()	OS_INT_PreserveAndDisableAll()
OS_INT_PRIO_RESTORE()	OS_INT_Restore()
OS_INTERRUPT_RestoreGlobal()	OS_INT_RestoreAll()
OS_EnterRegion()	OS_TASK_EnterRegion()
OS_LeaveRegion()	OS_TASK_LeaveRegion()
OS_GetTime()	OS_TIME_GetTicks()
OS_GetTime32()	OS_TIME_GetTicks32()
OS_Config_SysTimer()	OS_TIME_ConfigSysTimer()
OS_Timing_GetCycles()	OS_TIME_GetResult()
OS_Timing_Getus()	OS_TIME_GetResult_us()
OS_GetTime_us()	OS_TIME_Get_us()
OS_GetTime_us64()	OS_TIME_Get_us64()
OS_Timing_Start()	OS_TIME_StartMeasurement()
OS_Timing_End()	OS_TIME_StopMeasurement()
OS_AdjustTime()	OS_TICKLESS_AdjustTime()
OS_GetNumIdleTicks()	OS_TICKLESS_GetNumIdleTicks()
OS_StartTicklessMode()	OS_TICKLESS_Start()
OS_StopTicklessMode()	OS_TICKLESS_Stop()
OS_free()	OS_HEAP_free()
OS_malloc()	OS_HEAP_malloc()
OS_realloc()	OS_HEAP_realloc()
OS_MEMF_Request()	OS_MEMPOOL_Alloc()
OS_MEMF_Alloc()	OS_MEMPOOL_AllocBlocked()
OS_MEMF_AllocTimed()	OS_MEMPOOL_AllocTimed()
OS_MEMF_Create()	OS_MEMPOOL_Create()

V4.40	V5.00
OS_MEMF_Delete()	OS_MEMPOOL_Delete()
OS_MEMF_FreeBlock()	OS_MEMPOOL_Free()
OS_MEMF_Release()	OS_MEMPOOL_FreeEx()
OS_MEMF_GetBlockSize()	OS_MEMPOOL_GetBlockSize()
OS_MEMF_GetMaxUsed()	OS_MEMPOOL_GetMaxUsed()
OS_MEMF_GetNumBlocks()	OS_MEMPOOL_GetNumBlocks()
OS_MEMF_GetNumFreeBlocks()	OS_MEMPOOL_GetNumFreeBlocks()
OS_MEMF_IsInPool()	OS_MEMPOOL_IsInPool()
OS_GetObjName()	OS_DEBUG_GetObjName()
OS_SetObjName()	OS_DEBUG_SetObjName()
OS_AddLoadMeasurement()	OS_STAT_AddLoadMeasurement()
OS_STAT_GetTaskExecTime()	OS_STAT_GetExecTime()
OS_GetLoadMeasurement()	OS_STAT_GetLoadMeasurement()
OS_ClearTxActive()	OS_COM_ClearTxActive()
OS_GetNextChar()	OS_COM_GetNextChar()
OS_OnRx()	OS_COM_OnRx()
OS_OnTx()	OS_COM_OnTx()
OS_SendString()	OS_COM_SendString()
OS_SetRxCallback()	OS_COM_SetRxCallback()
OS_TraceEnable()	OS_TRACE_Enable()
OS_TraceEnableAll()	OS_TRACE_EnableAll()
OS_TraceEnableId()	OS_TRACE_EnableId()
OS_TraceEnableFilterId()	OS_TRACE_EnableFilterId()
OS_TraceDisable()	OS_TRACE_Disable()
OS_TraceDisableAll()	OS_TRACE_DisableAll()
OS_TraceDisableId()	OS_TRACE_DisableId()
OS_TraceDisableFilterId()	OS_TRACE_DisableFilterId()
OS_TraceData()	OS_TRACE_Data()
OS_TraceDataPtr()	OS_TRACE_DataPtr()
OS_TracePtr()	OS_TRACE_Ptr()
OS_SetTraceAPI()	OS_TRACE_SetAPI()
OS_TraceU32Ptr()	OS_TRACE_U32Ptr()
OS_TraceVoid()	OS_TRACE_Void()
OS_MPU_AddSanityCheckBuffer()	OS_MPU_SetSanityCheckBuffer()
OS_GetIntStackBase()	OS_STACK_GetIntStackBase()
OS_GetIntStackSize()	OS_STACK_GetIntStackSize()
OS_GetIntStackSpace()	OS_STACK_GetIntStackSpace()
OS_GetIntStackUsed()	OS_STACK_GetIntStackUsed()
OS_GetStackBase()	OS_STACK_GetTaskStackBase()
OS_GetStackSize()	OS_STACK_GetTaskStackSize()
OS_GetStackSpace()	OS_STACK_GetTaskStackSpace()
OS_GetStackUsed()	OS_STACK_GetTaskStackUsed()
OS_GetSysStackBase()	OS_STACK_GetSysStackBase()

V4.40	V5.00
OS_GetSysStackSize()	OS_STACK_GetSysStackSize()
OS_GetSysStackSpace()	OS_STACK_GetSysStackSpace()
OS_GetSysStackUsed()	OS_STACK_GetSysStackUsed()
OS_SetStackCheckLimit()	OS_STACK_SetCheckLimit()
OS_GetStackCheckLimit()	OS_STACK_GetCheckLimit()
OS_GetCPU()	OS_INFO_GetCPU()
OS_GetLibMode()	OS_INFO_GetLibMode()
OS_GetLibName()	OS_INFO_GetLibName()
OS_GetModel()	OS_INFO_GetModel()
OS_GetVersion()	OS_INFO_GetVersion()
OS_RSEMA	OS_MUTEX
OS_CSEMA	OS_SEMAPHORE
OS_Q	OS_QUEUE
OS_Q_SRCLIST	OS_QUEUE_SRCLIST
OS_MEMF	OS_MEMPOOL
OS_TASK_EVENT	OS_TASKEVENT

V5.06	V5.8.0
OS_USEPARA	OS_USE_PARA
OS_ERR_CPU_STATE_ISR_ILLEGAL	OS_ERR_ISR_PRIORITY_INVALID

V5.12.0	V5.14.0
OS_GetpCurrentTask()	OS_TASK_GetID()

V5.16.1	V5.18.0
OS_TASK_Delayus()	OS_TASK_Delay_us()
OS_TIME_GetResultus()	OS_TIME_GetResult_us()
OS_TIME_GetCycles()	OS_TIME_Get_Cycles()
OS_TIME_Getus()	OS_TIME_Get_us()
OS_TIME_Getus64()	OS_TIME_Get_us64()
voidRoutine	OS_ROUTINE_VOID
OS_RX_CALLBACK	OS_ROUTINE_CHAR
OS_WD_RESET_CALLBACK	OS_ROUTINE_WD_PTR
OS_ON_TERMINATE_FUNC	OS_ROUTINE_TASK_PTR
OS_TICK_HOOK_ROUTINE	OS_ROUTINE_VOID
OS_TIMERROUTINE	OS_ROUTINE_VOID
OS_TIMER_EX_ROUTINE	OS_ROUTINE_VOID_PTR
OS_TIMER_HANDLER	OS_ROUTINE_OSGLOBAL_PTR
OS_TIMING	OS_U32

V5.16.1	V5.18.0
OS_MPU_Enable()	OS_MPU_Init(&OS_MPU_DEFAULT_APILIST)
OS_MPU_EnableEx()	OS_MPU_Init()

V5.18.1	V5.20.0
OS_PRIO	OS_TASK_PRIO

Chapter 32

Support

32.1 Contacting support

If you need help or if any problem occurs the following describes how to contact the embOS support.

If you are a registered embOS user there are different ways to contact the embOS support:

1. You can create a support ticket via email to ticket_embos@segger.com.*
2. You can create a support ticket at segger.com/ticket.*
3. You can send an email to support_embos@segger.com.*

Please include the following information in the email or ticket:

- Which embOS do you use? (Core, compiler).
- The embOS version.
- Your embOS license number.
- If you are unsure about the above information you can also use the name of the embOS zip file (which contains the above information).
- A detailed description of the problem.
- Optionally a project with which we can reproduce the problem.

Note

Even without a valid license, feel free to contact our support e.g. in case of questions during your evaluation of embOS or for hobbyist purposes.

Please also take a few moments to help us improve our services by providing a short feedback once your support case has been solved.

32.1.1 Where can I find the license number?

The license number is part of the shipped zip file name. For example `embOS_CortexM_GC-C_SRC_V5.10.2.0_OS-01234_C1010320_200305.zip` where OS-01234 is the license number.

The license number is also part of every *.c- and *.h-file header. For example, if you open RTOS.h you should find the license number as with the example below:

```
-----
Licensing information
Licensor:                SEGGER Microcontroller GmbH
Licensed to:             Customer name
Licensed SEGGER software: embOS
License number:          OS-01234
License model:           SSL
Licensed product:        -
Licensed platform:       Cortex-M, GCC
Licensed number of seats: 1
-----
Support and Update Agreement (SUA)
SUA period:               2020-03-05 - 2021-03-05
Contact to extend SUA:   sales@segger.com
----- END-OF-HEADER -----
File      : RTOS.h
Purpose  : Include file for the OS,
           to be included in every C-module accessing OS-routines
```

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at <https://www.segger.com/legal/privacy-policy/>.

Chapter 33

Performance and Resource Usage

33.1 Introduction

This chapter covers the performance and resource usage of embOS. It explains how to benchmark embOS and contains information about the memory requirements in typical systems which can be used to obtain sufficient estimates for most target systems.

High performance combined with low resource usage has always been a major design consideration. embOS runs on 8/16/32/64-bit CPUs. Depending on which features are being used, even single-chip systems with less than 2048 bytes ROM and 1024 bytes RAM can be supported by embOS.

33.2 Resource Usage

The memory requirements of embOS (RAM and ROM) differs depending on the used features, CPU, compiler, and library model. The following values are typical values for a 32-bit CPU and are taken from embOS Cortex-M ES V5.16.0.0 using embOS library mode `OS_LIBMODE_XR`.

Module	Memory type	Memory requirements
embOS kernel	ROM	~1700 bytes
embOS kernel	RAM	~136 bytes
Task control block	RAM	36 bytes
Software timer	RAM	20 bytes
Task event	RAM	0 bytes
Event object	RAM	12 bytes
Mutex	RAM	16 bytes
Semaphore	RAM	8 bytes
RWLocks	RAM	28 bytes
Mailbox	RAM	24 bytes
Queue	RAM	32 bytes
Watchdog	RAM	12 bytes
Fixed Block Size Memory Pool	RAM	32 bytes

RAM resource measurement of embOS objects:

```
int main(void) {
    unsigned int TCB_size      = sizeof(OS_TASK);
    unsigned int TIMER_size   = sizeof(OS_TIMER);
    unsigned int TASKEVENT_size = 0u;
    unsigned int EVENT_size   = sizeof(OS_EVENT);
    unsigned int MUTEX_size   = sizeof(OS_MUTEX);
    unsigned int SEMAPHORE_size = sizeof(OS_SEMAPHORE);
    unsigned int RWLOCK_size  = sizeof(OS_RWLOCK);
    unsigned int MAILBOX_size  = sizeof(OS_MAILBOX);
    unsigned int QUEUE_size    = sizeof(OS_QUEUE);
    unsigned int WD_size       = sizeof(OS_WD);
    unsigned int MEMPOOL_size  = sizeof(OS_MEMPOOL);
    return 0;
}
```

RAM and ROM resource measurement of embOS kernel:

With the embOS source code and the following defines it is possible to place all embOS kernel and API code and data in specific memory sections.

Define	GCC Example
OS_TEXT_SECTION_ATTRIBUTE	<code>__attribute__((section (".ostext."#name)))</code>
OS_RODATA_SECTION_ATTRIBUTE	<code>__attribute__((section (".osrodata."#name)))</code>
OS_DATA_SECTION_ATTRIBUTE	<code>__attribute__((section (".osdata."#name)))</code>
OS_BSS_SECTION_ATTRIBUTE	<code>__attribute__((section (".osbss."#name)))</code>

The memory map file tells the size of each section and with it the embOS kernel RAM and ROM resource usage. For more details please contact the embOS support.

33.3 Performance

embOS is designed to perform fast context switches. This section describes two different methods to calculate the execution time of a context switch from a task with lower priority to a task with a higher priority.

The first method uses port pins and requires an oscilloscope. The second method uses the embOS time measurement functions. Example programs for both methods are supplied in the `\Application` directory of the embOS BSPs.

SEGGER uses these programs to benchmark embOS performance. You can use these examples to evaluate the benchmark results. Note that the actual performance depends on many factors (CPU, clock speed, tool chain, memory model, optimization, etc.).

Please be aware that the number of cycles are not equal to the number of instructions. Many instructions on ARM need two or three cycles even at zero wait-states, e.g. `LDR` needs 3 cycles.

The following table shows the context switch time for different CPUs. The applications for measurement were compiled using embOS library mode `OS_LIBMODE_XR`.

Target	embOS	CPU Frequency	Time	CPU Cycles
Renesas RZ	embOS ARM ES V5.14.0.0	400 MHz	0.48 us	192
ST STM32F769	embOS Cortex-M ES V5.14.0.0	200 MHz	1.25 us	250

33.3.1 Measurement with Port Pins and Oscilloscope

The example file `OS_MeasureCST_Scope.c` uses the `BSP.c` module to set and clear a port pin. This allows measuring the context switch time with an oscilloscope. The following source code is an excerpt from `OS_MeasureCST_Scope.c`:

```
#include "RTOS.h"
#include "BSP.h"

static OS_STACKPTR int StackHP[128], StackLP[128]; /* Task stacks */
static OS_TASK      TCBHP, TCBLP; /* Task-control-blocks */

/*****
 *
 *      HPTask
 */
static void HPTask(void) {
    while (1) {
        OS_TASK_Suspend(NULL); // Suspend high priority task
        BSP_ClrLED(0);         // Stop measurement
    }
}

/*****
 *
 *      LPTask
 */
static void LPTask(void) {
    while (1) {
        OS_TASK_Delay(100);
        //
        // Display measurement overhead
        //
        BSP_SetLED(0);
        BSP_ClrLED(0);
        //
        // Perform measurement
        //
        BSP_SetLED(0); // Start measurement
        OS_TASK_Resume(&TCBHP); // Resume high priority task to force task switch
    }
}

/*****
 *
 *      main
 */
int main(void) {
    OS_Init(); // Initialize embOS
    OS_IniHW(); // Initialize hardware for embOS
    BSP_Init(); // Initialize LED ports
    OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_Start(); // Start multitasking
    return 0;
}
```

33.3.1.1 Oscilloscope analysis

The context switch time is the time between switching the LED on and off. If the LED is switched on with an active high signal, the context switch time is the time between the rising and the falling edge of the signal. If the LED is switched on with an active low signal, the signal polarity is reversed.

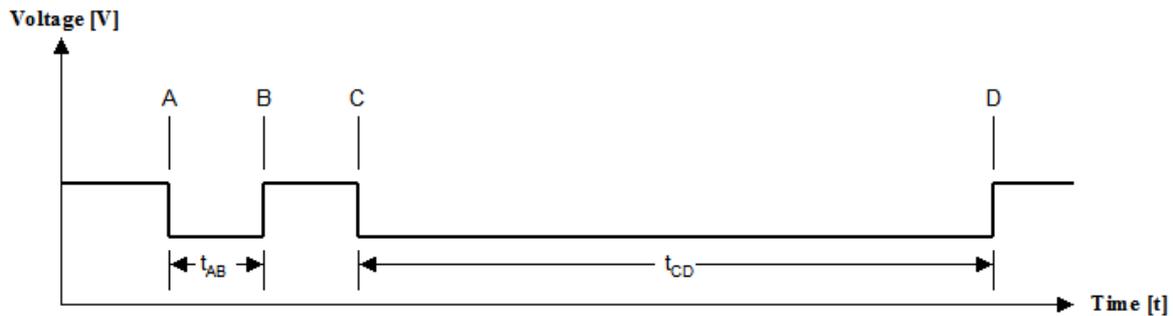
The real context switch time is shorter, because the signal also contains the overhead of switching the LED on and off. The time of this overhead is also displayed on the oscilloscope as a small peak right before the task switch time display and must be subtracted from the displayed context switch time. The picture below shows a simplified oscilloscope signal with an active-low LED signal (low means LED is illuminated). There are switching points to determine:

- A = LED is switched on for overhead measurement
- B = LED is switched off for overhead measurement
- C = LED is switched on right before context switch in low-prio task
- D = LED is switched off right after context switch in high-prio task

The time needed to switch the LED on and off in subroutines is marked as time t_{AB} . The time needed for a complete context switch including the time needed to switch the LED on and off in subroutines is marked as time t_{CD} .

The context switching time t_{CS} is calculated as follows:

$$t_{CS} = t_{CD} - t_{AB}$$



33.3.1.2 Example measurements Renesas RZA1, Thumb2 code in RAM

Configuration

embOS Version: V5.14.0.0

Application: OS_MeasureCST_Scope.c

Hardware: Renesas RZA1

Executed in: internal RAM

CPU Mode: Thumb2

Compiler: SEGGER Embedded Studio V5.50d (SEGGER Compiler)

CPU frequency (f_{CPU}): 400MHz

CPU clock cycle (t_{Cycle}): $1 / f_{CPU} \Rightarrow 1 / 400\text{MHz} = 2.5\text{ns}$

Measuring t_{AB} and t_{CD}



t_{AB} is measured as 540ns.

The number of cycles calculates as follows:

$$\text{Cycles}_{AB} = t_{AB} / t_{Cycle}$$

$$= 540\text{ns} / 2.5\text{ns}$$

$$= 216 \text{ Cycles}$$



t_{CD} is measured as 1020ns.

The number of cycles calculates as follows:

$$\begin{aligned} \text{Cycles}_{CD} &= t_{CD} / t_{\text{Cycle}} \\ &= 1020\text{ns} / 2.5\text{ns} \\ &= 408 \text{ Cycles} \end{aligned}$$

Resulting context switching time and number of cycles

The time which is required for the pure context switch is:

$$\begin{aligned} t_{CS} &= t_{CD} - t_{AB} = 408 \text{ Cycles} - 216 \text{ Cycles} = 192 \text{ Cycles} \\ &=> \mathbf{192 \text{ Cycles (0.48}\mu\text{s @400 MHz)}}. \end{aligned}$$

33.3.2 Measurement with time measurement API

The context switch time may be measured with embOS' time measurement functions. Refer to section *Time Measurement* on page 374 for detailed information about the embOS time measurement API.

The example `OS_MeasureCST_HRTimer_embOSView.c` uses hardware counter to measure the context switch time from a low priority task to a high priority task and displays the results on `embOSView`.

```
#include "RTOS.h"
#include <stdio.h>

static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
static OS_TASK      TCBHP, TCBLP;                 // Task control blocks
static OS_U64       Time;                          // Output buffer
static char         acBuffer[100];

/*****
 *
 *      HPTask()
 */
static void HPTask(void) {
    while (1) {
        OS_TASK_Suspend(NULL); // Suspend high priority task
        Time = OS_TIME_Get_Cycles() - Time; // Stop measurement
    }
}

/*****
 *
 *      LPTask()
 */
static void LPTask(void) {
    OS_U64 MeasureOverhead; // Time for Measure Overhead
    OS_U32 v;               // Real context switching time

    while (1) {
        OS_TASK_Delay(100);
        //
        // Measure overhead for time measurement so we can take this into account by
        // subtracting it. This is done inside the while()-loop to mitigate possible
        // effects of an instruction cache.
        //
        MeasureOverhead = OS_TIME_Get_Cycles();
        MeasureOverhead = OS_TIME_Get_Cycles() - MeasureOverhead;
        //
        // Perform actual measurements
        //
        Time = OS_TIME_Get_Cycles(); // Start measurement
        // Resume high priority task to force task switch
        OS_TASK_Resume(&TCBHP);
        // Calculate real context switching time (w/o overhead measurement)
        Time -= MeasureOverhead;
        // Convert cycles to nanoseconds
        v = (OS_U32)OS_TIME_ConvertCycles2ns(Time);
        sprintf(acBuffer, "Context switch time: %lu.%.3lu microseconds\r",
            (v / 1000uL), (v % 1000uL)); // Create result text
        OS_COM_SendString(acBuffer); // Print out result
    }
}

/*****
 *
 *      main()
 */
int main(void) {
```

```
OS_Init();    // Initialize embOS
OS_Inithw(); // Initialize required hardware
OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
OS_Start();  // Start embOS
return 0;
}
```

The example program calculates and subtracts the measurement overhead. The results will be transmitted to embOSView, so the example runs on every target that supports UART communication to embOSView.

The example program `OS_MeasureCST_HRTimer_Printf.c` is identical to the example program `OS_MeasureCST_HRTimer_embOSView.c` but displays the results with the `printf()` function for those debuggers which support terminal output emulation.

Chapter 34

Supported Development Tools

34.1 Reentrance

Overview

All routines that can be used from different tasks at the same time must be fully reentrant. A routine is in use from the moment it is called until it returns or the task that has called it is terminated. All routines supplied with your RTOS are fully reentrant. If for some reason you need to have non-reentrant routines in your program that can be used from more than one task, it is recommended to use a mutex to avoid this kind of problem.

C routines and reentrance

Normally, the C compiler generates code that is fully reentrant. However, the compiler may have options that force it to generate non-reentrant code. It is recommended not to use these options, although it is possible to do so in certain circumstances.

Assembly routines and reentrance

As long as assembly functions access local variables and parameters only, they are fully reentrant. Everything else needs to be thought about carefully.

34.2 Compiler version

A specific embOS port has been developed with and for a specific C compiler and compiler version for the selected target processor. Please refer to the CPU and compiler specific release notes for details. embOS might work with the specified C compiler version only because other compiler versions might use a different calling conventions (incompatible object file formats) and therefore might be incompatible. However, if you prefer to use a different C compiler version, please contact us and we will run our quality tests again with the requested compiler version and confirm the compatibility.

34.3 Compiler options

embOS libraries were built with specific compiler options. If you use an embOS library instead of embOS sources in your project please ensure to not use any additional incompatible compiler options. This could be a compiler option for padding bytes in structures or the size of specific data types like e.g. an enum. If you have access to the embOS source code you can find all used compiler options in the batch files. You can add there additional compiler options and rebuild the embOS libraries or use the embOS sources instead of the library in your project. However, if you don't have access to the embOS source code but want to use additional compiler options, please contact us and we will share the used compiler options.

34.4 C/C++ standard

embOS was developed in the C programming language. The embOS sources can be built with every existing C standard.

The following table shows which C standards are supported:

C Standard	Description
C90	Supported but // comments are not allowed in C90 and <code>os_global.c</code> uses C99 designator for embOS RL78/RX IAR.
C99	Supported
C11	Supported
C17	Supported

The embOS sources are not designed to be built with a C++ compiler but a C++ application can be used with embOS. To do so the embOS sources must be built with a C compiler and the application with a C++ compiler. You can also simply use an embOS library in your C++ project.

The following table shows which C++ standards are supported:

C++ Standard	Description
C++98	Supported
C++03	Supported
C++11	Supported
C++14	Supported
C++17	Supported
C++20	Supported

Chapter 35

Glossary

Term	Definition
Cooperative multitasking	A scheduling system in which each task is allowed to run until it gives up the CPU; an ISR can make a higher priority task ready, but the interrupted task will be returned to and finished first.
Counting semaphore	A type of semaphore that keeps track of multiple resources. Used when a task must wait for something that can be signaled more than once.
CPU	Central Processing Unit. The "brain" of a microcontroller; the part of a processor that carries out instructions.
Critical region	A section of code which must be executed without interruption.
Event	A message sent to a single, specified task that something has occurred. The task then becomes ready.
Interrupt Handler	Interrupt Service Routine. The routine is called by the processor when an interrupt is acknowledged. ISRs must preserve the entire context of a task (all registers).
ISR	Interrupt Service Routine. The routine is called by the processor when an interrupt is acknowledged. ISRs must preserve the entire context of a task (all registers).
Mailbox	A data buffer managed by an RTOS, used for sending messages to a task or interrupt handler.
Message	An item of data (sent to a mailbox, queue, or other container for data).
Multitasking	The execution of multiple software routines independently of one another. The OS divides the processor's time so that the different routines (tasks) appear to be happening simultaneously.
Mutex	A data structure used for managing resources by ensuring that only one task has access to a resource at a time.
NMI	Non-Maskable Interrupt. An interrupt that cannot be masked (disabled) by software. Example: Watchdog timer interrupt.
Preemptive multitasking	A scheduling system in which the highest priority task that is ready will always be executed. If an ISR makes a higher priority task ready, that task will be executed before the interrupted task is returned to.
Process	Processes are tasks with their own memory layout. Two processes cannot normally access the same memory locations. Different processes typically have different access rights and (in case of MMUs) different translation tables.
Processor	Short for microprocessor. The CPU core of a controller.
Priority	The relative importance of one task to another. Every task in an RTOS has a priority.
Priority inversion	A situation in which a high priority task is delayed while it waits for access to a shared resource which is in use by a lower priority task. A task with medium priority in the ready state may run, instead of the high priority task. embOS avoids this situation by priority inheritance.

Term	Definition
Queue	Like a mailbox, but used for sending larger messages, or messages of individual size, to a task or an interrupt handler.
Ready	Any task that is in "ready state" will be activated when no other task with higher priority is in "ready state".
Resource	Anything in the computer system with limited availability (for example memory, timers, computation time). Essentially, anything used by a task.
RTOS	Real-time Operating System.
Running task	Only one task can execute at any given time. The task that is currently executing is called the running task.
Scheduler	The program section of an RTOS that selects the active task, based on which tasks are ready to run, their relative priorities, and the scheduling system being used.
Semaphore	A data structure used for synchronizing tasks.
Software timer	A data structure which calls a user-specified routine after a specified delay.
Stack	An area of memory with LIFO storage of parameters, automatic variables, return addresses, and other information that needs to be maintained across function calls. In multitasking systems, each task normally has its own stack.
Superloop	A program that runs in an infinite loop and uses no real-time kernel. ISRs are used for real-time parts of the software.
Task	A program running on a processor. A multitasking system allows multiple tasks to execute independently from one another.
Thread	Threads are tasks which share the same memory layout. Two threads can access the same memory locations. If virtual memory is used, the same virtual to physical translation and access rights are used(c.f. Thread, Process)
Tick	The OS timer interrupt. Typically equals 1 millisecond.
Time slice	The time (number of system ticks) for which a task will be executed until a round-robin task change may occur.