Disclaimer
Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

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.
© 1995-2019 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
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: June 19, 2019

<table>
<thead>
<tr>
<th>Software</th>
<th>Revision</th>
<th>Date</th>
<th>By</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.06</td>
<td>2</td>
<td>190619</td>
<td>TS</td>
<td>Minor spelling &amp; wording corrections.</td>
</tr>
<tr>
<td>5.06</td>
<td>1</td>
<td>190401</td>
<td>TS</td>
<td>Minor spelling &amp; wording corrections.</td>
</tr>
<tr>
<td>5.06</td>
<td>0</td>
<td>190219</td>
<td>TS</td>
<td>New API functions added. Minor spelling &amp; wording corrections.</td>
</tr>
<tr>
<td>5.04</td>
<td>0</td>
<td>180913</td>
<td>TS</td>
<td>New API function OS_MUTEX_IsMutex(). Minor spelling &amp; wording corrections.</td>
</tr>
<tr>
<td>5.02</td>
<td>0</td>
<td>180626</td>
<td>TS</td>
<td>New API function OS_STAT_AddLoadMeasurementEx(). Minor spelling &amp; wording corrections.</td>
</tr>
<tr>
<td>5.00</td>
<td>1</td>
<td>180524</td>
<td>TS</td>
<td>OS_TASK_Delay() parameter description corrected. Timeout parameter description added where necessary.</td>
</tr>
<tr>
<td>5.00</td>
<td>0</td>
<td>180508</td>
<td>TS</td>
<td>New API names. Chapter &quot;Debugging&quot; updated. Minor spelling &amp; wording corrections.</td>
</tr>
<tr>
<td>4.38</td>
<td>1</td>
<td>170928</td>
<td>MC</td>
<td>Minor spelling &amp; wording corrections.</td>
</tr>
<tr>
<td>4.38</td>
<td>0</td>
<td>170919</td>
<td>TS</td>
<td>First version generated with emDoc. New function in chapter &quot;Tasks&quot; added: OS_SetDefaultTaskStartHook() New functions in chapter &quot;Debugging&quot; added: OS_GetObjName() OS_GetObjName() Minor corrections/updates.</td>
</tr>
<tr>
<td>4.36</td>
<td>0</td>
<td>170711</td>
<td>TS</td>
<td>New library mode OS_LIBMODE_SAFE added in chapter &quot;Basic Concepts&quot;. New functions in chapter &quot;Stacks&quot; added: OS_GetStackCheckLimit() OS_GetStackCheckLimit() New functions in chapter &quot;MPU&quot; added: OS_MPU_AddSanityCheckBuffer() OS_MPU_SanityCheck() Chapter &quot;Source Code&quot; updated. New functions in chapter &quot;Task Routines&quot; added: OS_Config_Stop() OS_Stop() Minor corrections/updates</td>
</tr>
<tr>
<td>4.34</td>
<td>0</td>
<td>170308</td>
<td>TS</td>
<td>New functions in chapter &quot;Event Objects&quot; added: OS_EVENT_GetMaskMode() OS_EVENT_SetMaskMode()</td>
</tr>
<tr>
<td>Software</td>
<td>Revision</td>
<td>Date</td>
<td>By</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>--------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>4.26</td>
<td>0</td>
<td>160907</td>
<td>RH</td>
<td>Chapter &quot;embOSView&quot;, &quot;Interrupts&quot; and &quot;MPU&quot; updated. Minor corrections/updates.</td>
</tr>
<tr>
<td>4.24</td>
<td>0</td>
<td>160628</td>
<td>MC</td>
<td>Chapter &quot;Multi-core Support&quot; added. Chapter &quot;Debugging&quot; updated.</td>
</tr>
</tbody>
</table>
| 4.22     | 0        | 160525 | MC   | New functions in chapter "Queues" added:  
• 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 "Shipments" updated.  
Chapter "Low Power Support" updated.  
Chapter "Interrupts": Description of  
• OS_INT_PRIO_PRESERVE() and  
• OS_INT_PRIO_RESTORE() added.  
Chapter "Software Timers": Description of  
• 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 functions in chapter "Mailboxes" added:  
• 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   | Minor spelling and wording corrections.  
Chapter "embOSView": added JTAG Chain configuration. |
| 4.10     | 0        | 150430 | TS   | Chapter "embOSView" updated. |
| 4.06b    | 0        | 150324 | MC   | Minor spelling and wording corrections. |
| 4.06a    | 0        | 150318 | MC   | Minor 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"  
• Task priority description updated.  
Chapter "Debugging"  
• 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:  
• OS_Config_SysTimer()  
• OS_GetTime_us()  
• OS_GetTime_us64() |
| 4.00a    | 0        | 140723 | TS   | New functions added in chapter "System Tick":  
• OS_StopTicklesMode()  
New functions added in chapter "Profiling":  
• OS_STAT_Start()  
• OS_STAT_Stop()  
• OS_STAT_GetTaskExecTime() |
<table>
<thead>
<tr>
<th>Software</th>
<th>Revision</th>
<th>Date</th>
<th>By</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.00</td>
<td>0</td>
<td>140606</td>
<td>TS</td>
<td>Tickless support added.</td>
</tr>
<tr>
<td>3.90a</td>
<td>0</td>
<td>140410</td>
<td>AW</td>
<td>Software Update, OS_TerminateTask() modified / corrected.</td>
</tr>
<tr>
<td>3.90</td>
<td>1</td>
<td>140312</td>
<td>SC</td>
<td>Added cross-references to the API-lists.</td>
</tr>
</tbody>
</table>
| 3.90     | 0        | 140303   | AW | New functions to globally enable / disable Interrupts:  
|          |          |          |    | • 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":  
|          |          |          |    | • OS_GetNumIdleTicks()  
|          |          |          |    | • OS_AdjustTime()  
|          |          |          |    | Chapter "System Variables": Description of internal variable OS_GlobalTimeDex 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. |
| 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":  
|          |          |          |    | • 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":  
|          |          |          |    | • 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:  
|          |          |          |    | • 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:  
|          |          |          |    | • 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:  
<p>|          |          |          |    | • Several index entries corrected. |</p>
<table>
<thead>
<tr>
<th>Software</th>
<th>Revision</th>
<th>Date</th>
<th>By</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• OS_EnterRegion() described more in detail.</td>
</tr>
<tr>
<td>3.86d</td>
<td>0</td>
<td>120510</td>
<td>TS</td>
<td>Update to latest software version.</td>
</tr>
<tr>
<td>3.86c</td>
<td>0</td>
<td>120508</td>
<td>TS</td>
<td>Update to latest software version.</td>
</tr>
<tr>
<td>3.86b</td>
<td>0</td>
<td>120502</td>
<td>TS</td>
<td>Chapter “Mailbox”&lt;br&gt;• OS_PeekMail() added.&lt;br&gt;Chapter “Support” added.&lt;br&gt;Chapter “Debugging”:&lt;br&gt;• Application defined error codes added.</td>
</tr>
<tr>
<td>3.86</td>
<td>0</td>
<td>120323</td>
<td>AW</td>
<td>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 priorized tasks.&lt;br&gt;Modified functions:&lt;br&gt;• OS_UseTimed()&lt;br&gt;• OS_WaitCSemaTimed()&lt;br&gt;• OS_GetMailTimed()&lt;br&gt;• OS_WaitMailTimed()&lt;br&gt;• OS_Q_GetPtrTimed()&lt;br&gt;• OS_EVENT_WaitTimed()&lt;br&gt;• OS_MEMF_AllocTimed()&lt;br&gt;New chapter “Extending the Task Context” added.&lt;br&gt;New functions added and described in the manual:&lt;br&gt;• OS_GetTaskName()&lt;br&gt;• OS_GetTimeSliceRem()&lt;br&gt;Handling of queues described more in detail:&lt;br&gt;• OS_Q_GetPtr()&lt;br&gt;• OS_Q_GetPtrCond()&lt;br&gt;• OS_Q_GetPtrTimed()&lt;br&gt;• OS_Q_Purge()&lt;br&gt;Chapter “Priority Inversion / Inheritance” updated.&lt;br&gt;Function names OS_Timing_Start() and OS_Timing_End() corrected in the API table.</td>
</tr>
<tr>
<td>3.84c</td>
<td>1</td>
<td>120130</td>
<td>AW/TS</td>
<td>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.&lt;br&gt;The prototype descriptions of the following API functions are corrected now:&lt;br&gt;• OS_GetTimerValue()&lt;br&gt;• OS_GetTimerStatus()&lt;br&gt;• OS_GetTimerPeriod()&lt;br&gt;• OS_GetSemaValue()&lt;br&gt;• OS_GetResourceOwner()&lt;br&gt;• OS_Q_IsInUse()&lt;br&gt;• OS_Q_GetMessageCnt()&lt;br&gt;• OS_IsTask()&lt;br&gt;• OS_GetEventsOccured()&lt;br&gt;• OS_GetCSemaValue()&lt;br&gt;• OS_TICK_RemoveHook()&lt;br&gt;• OS_MEMF_IsInPool()&lt;br&gt;• OS_MEMF_GetMaxUsed()&lt;br&gt;• OS_MEMF_GetNumBlocks()&lt;br&gt;• OS_MEMF_GetBlockSize()&lt;br&gt;• OS_GetPriority()&lt;br&gt;• OS_EVENT_Get()&lt;br&gt;• OS_Timing_Getus()&lt;br&gt;Chapter “Preface”:&lt;br&gt;• Segger Logo replaced&lt;br&gt;Chapter “Mailbox”:&lt;br&gt;• OS_CREATEMB() changed to OS_CreateMB()&lt;br&gt;Chapter “Queues”:&lt;br&gt;• Typos corrected</td>
</tr>
<tr>
<td>3.84c</td>
<td>0</td>
<td>120104</td>
<td>TS</td>
<td>Chapter “Events”:&lt;br&gt;• Return value of OS_EVENT_WaitTimed() explained in more detail</td>
</tr>
<tr>
<td>3.84b</td>
<td>0</td>
<td>111221</td>
<td>TS</td>
<td>Chapter “Queues”:&lt;br&gt;• OS_Q_PutBlocked() added</td>
</tr>
<tr>
<td>Software</td>
<td>Revision</td>
<td>Date</td>
<td>By</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>-----</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 3.84     | 0        | 110927   | TS  | Chapter "Stacks":  
|          |          |          |     | • 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  | Chapter "Debugging":  
|          |          |          |     | • 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.  
| 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":  
|          |          |          |     | • 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":  
|          |          |          |     | • OS_WaitMailTimed() added  
| 3.82n    | 0        | 101206   | AW  | Chapter "Taskroutines":  
|          |          |          |     | • OS_ResumeAllSuspendedTasks() added  
|          |          |          |     | • OS_GetInitialSuspendCnt() added  
|          |          |          |     | • OS_SuspendAllTasks() added  
|          |          |          |     | Chapter "Time Measurement":  
|          |          |          |     | • Description of OS_GetTime32() corrected  
|          |          |          |     | Chapter "List of Error Codes":  
|          |          |          |     | • New error codes added  
| 3.82k    | 0        | 100927   | TS  | Chapter "Taskroutines":  
|          |          |          |     | • OS_Delayus() added  
|          |          |          |     | • OS_Q_Delete() added  
| 3.82i    | 0        | 100917   | TS  | General updates and corrections  
| 3.89     | 0        | 100621   | AW  | Chapter "Event Objects":  
|          |          |          |     | • Samples added  
|          |          |          |     | Chapter "Configuration of Target System":  
|          |          |          |     | • Detailed description of OS_Idle() added  
| 3.82f    | 1        | 100505   | TS  | Chapter "Profiling" added  
|          |          |          |     | Chapter "System Tick":  
|          |          |          |     | • OS_TickHandleNoHook() added  
| 3.82f    | 0        | 100419   | AW  | Chapter "Tasks":  
|          |          |          |     | • OS_IsRunning() added  
|          |          |          |     | • Description of OS_Start() added  
| 3.82e    | 0        | 100309   | TS  | Chapter "Working with embOS - Recommendations" added.  
|          |          |          |     | Chapter "Basics":  
|          |          |          |     | • Priority inversion image added  
|          |          |          |     | Chapter "Interrupt":  
|          |          |          |     | • 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 function usage (main, task, ISR or timer)  
|          |          |          |     | TOC format corrected  

<table>
<thead>
<tr>
<th>Software</th>
<th>Revision</th>
<th>Date</th>
<th>By</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.80</td>
<td>0</td>
<td>090612</td>
<td>AW</td>
<td>Scheduler optimized for higher task switching speed.</td>
</tr>
</tbody>
</table>
| 3.62c    | 0        | 080903    | SK    | Chapter structure updated.  
Chapter "Interrupts":  
• OS_LeaveNestableInterruptNoSwitch() removed  
• OS_LeaveInterruptNoSwitch() removed  
Chapter "System Tick":  
• OS_TICK_Config() added |
| 3.60     | 2        | 080722    | SK    | Contact address updated. |
| 3.60     | 1        | 080617    | SK    | General updates.  
Chapter "Mailboxes":  
• OS_GetMailCond() / OS_GetMailCond1() corrected  
Chapter "System Tick":  
• OS_TICK_Config() added |
| 3.60     | 0        | 080117    | OO    | General updates.  
Chapter "System Tick" added. |
| 3.52     | 1        | 071026    | AW    | Chapter "Task Routines":  
• OS_SetTaskName() added  
Chapter "Interrupts":  
• OS_ExtendTaskContext() added |
| 3.52     | 0        | 070824    | OO    | Chapter "Task Routines":  
• OS_ExtendTaskContext() added  
Chapter "Interrupts":  
• Updated  
• 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:  
• OS_ERR_ISR_INDEX added  
• OS_ERR_ISR_VECTOR added  
• OS_ERR_RESOURCE.Owner added  
• OS_ERR_SEMA_OVERFLOW added  
Chapter "Task Routines":  
• OS_Yield() added  
Chapter "Counting Semaphores" updated  
• OS_SignalCSema(), additional information adjusted  
Chapter "Performance and Resource Usage" updated:  
• Minor changes in wording. |
| 3.40a    | 1        | 070608    | SK    | Chapter "Counting Semaphores" updated:  
• 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 pcContext corrected.  
Chapter 3: OS_CreateTaskEx() function, type of parameter TimeSlice corrected.  
Chapter 3: OS_CreateTask() function, type of parameter TimeSlice corrected. |
<table>
<thead>
<tr>
<th>Software</th>
<th>Revision</th>
<th>Date</th>
<th>By</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.32</td>
<td>060717</td>
<td>OO</td>
<td>Event objects introduced. Chapter 10 inserted which describes event objects. Previous chapter &quot;Events&quot; renamed to &quot;Task Events&quot;.</td>
</tr>
<tr>
<td></td>
<td>3.30</td>
<td>060519</td>
<td>OO</td>
<td>New software version.</td>
</tr>
<tr>
<td></td>
<td>3.28</td>
<td>060223</td>
<td>OO</td>
<td>All chapters: Added API tables. Some minor changes.</td>
</tr>
<tr>
<td></td>
<td>3.28</td>
<td>051109</td>
<td>AW</td>
<td>Chapter 7: OS_SignalCSemaMax() function added. Chapter 14: Explanation of interrupt latencies and high / low priorities added.</td>
</tr>
<tr>
<td></td>
<td>3.28</td>
<td>050926</td>
<td>AW</td>
<td>Chapter 6: OS_DeleteRSem() function added.</td>
</tr>
<tr>
<td></td>
<td>3.28</td>
<td>050707</td>
<td>AW</td>
<td>Chapter 4: OS_GetSuspendCnt() function added.</td>
</tr>
<tr>
<td></td>
<td>3.28</td>
<td>050425</td>
<td>AW</td>
<td>Version number changed to 3.28 to fit to current embOS version. Chapter 18.1.2: Type return value of OS_GetTime32() corrected.</td>
</tr>
<tr>
<td></td>
<td>3.24</td>
<td>011115</td>
<td>AW</td>
<td>Chapter 6: Some prototype declarations showed in OS_SEMA instead of OS_RSEMA. Corrected.</td>
</tr>
</tbody>
</table>
|          | 3.22     | 040816 | AW    | Chapter 8: New Mailbox functions added  
  - OS_PutMailFront()  
  - OS_PutMailFront1()  
  - OS_PutMailFrontCond()  
  - OS_PutMailFrontCond1()  
  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. |
<p>|          | 3.20     | 040621 | RS/AW | OS_CreateCSem() 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     | 040329 | AW    | OS_PutSem() 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     | 031128 | AW    | OS_CREATEMB() Range for parameter MaxnofMsg corrected. Upper limit is 65535, but was declared 65536 in previous manuals. |
|          | 3.20     | 040831 | AW    | Code samples modified: Task stacks defined as array of int, because most CPUs require alignment of stack on integer aligned addresses. |
|          | 3.12     | 021015 | AW    | Chapter 8: OS_GetMailHeader1() added. Chapter 11 (Heap type memory management) inserted. Chapter 12 (Fixed block size memory pools) inserted. |
|          | 3.10     | 020926 | 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). |</p>
<table>
<thead>
<tr>
<th>Software</th>
<th>Revision</th>
<th>Date</th>
<th>By</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0--13--1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

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:

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

1 Introduction and basic concepts .................................................................................................................. 17
   1.1 What is embOS? ................................................................................................................................. 18
   1.2 embOS ports .................................................................................................................................... 20
   1.3 Tasks ............................................................................................................................................. 21
   1.4 Single-task systems (superloop) ........................................................................................................ 22
   1.5 Multitasking systems .......................................................................................................................... 24
   1.6 Scheduling ........................................................................................................................................ 26
   1.7 Communication between tasks .......................................................................................................... 28
   1.8 How task switching works ............................................................................................................... 29
   1.9 Change of task status ....................................................................................................................... 31
   1.10 How the OS gains control ............................................................................................................... 32
   1.11 Different builds of embOS ............................................................................................................. 33
   1.12 Valid context for embOS API ........................................................................................................... 35
   1.13 Blocking and Non blocking embOS API .......................................................................................... 36
   1.14 API functions ............................................................................................................................... 37

2 Tasks .......................................................................................................................................................... 44
   2.1 Introduction ...................................................................................................................................... 45
   2.2 Cooperative vs. preemptive task switches ....................................................................................... 46
   2.3 Extending the task context ............................................................................................................... 47
   2.4 API functions ................................................................................................................................... 49

3 Software Timers ......................................................................................................................................... 92
   3.1 Introduction ...................................................................................................................................... 93
   3.2 API functions ................................................................................................................................... 95

4 Task Events ............................................................................................................................................... 122
   4.1 Introduction ...................................................................................................................................... 123
   4.2 API functions ................................................................................................................................... 124

5 Event Objects ............................................................................................................................................. 134
   5.1 Introduction ...................................................................................................................................... 135
   5.2 API functions ................................................................................................................................... 138

6 Mutexes ..................................................................................................................................................... 158
   6.1 Introduction ...................................................................................................................................... 159
   6.2 API functions ................................................................................................................................... 161
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 real-time operating systems.

embOS is highly modular. This means that only those functions that are required are linked into an application, keeping the ROM size very small. The minimum memory consumption is little more than 1.7 Kbyte of ROM and about 70 bytes of RAM (plus memory for stacks). 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:

- 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 (limited only by the amount of available memory).
- Unlimited number of semaphores (limited only by the amount of available memory).
- Two types of semaphores: Mutex and counting semaphores.
- Unlimited number of mailboxes (limited only by the amount of available memory).
- Size and number of messages can be freely defined when initializing mailboxes.
- Unlimited number of software timers (limited only by the amount of available memory).
- Up to 32-bit events for every task.
- Time resolution can be freely selected (default is 1 millisecond).
- Easily accessible time variable.
- 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 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 evalboards. 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.

1.2.1 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.

1.2.2 Naming convention

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

1.2.3 Version number convention

SEGGER releases new embOS versions with new features and bug fixes. As soon as a new embOS version is released all embOS ports are updated to this version. Each new embOS version gets a unique version number: V Major.Minor Patch

For example: V4.12a

Major: 4
Minor: 12
Patch: a

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

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. A real-time operating system, 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.

1.3.1 Threads vs. Processes

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.
1.4 Single-task 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 amongst 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()`. If tasks do not cooperate, the system "hangs", which means that other tasks have no chance of being executed by the CPU while the first task is being carried out. This is illustrated in the diagram below. Even if an ISR makes a higher-priority task ready to run, the interrupted task will be resumed and complete 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

Real-time operating systems like embOS operate 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, real-time operating systems such as embOS utilize a timer interrupt that interrupts tasks at periodic intervals 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 Scheduling

There are different algorithms that determine which task to execute, called schedulers. 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.

1.6.1 Round-robin scheduling algorithm

With round-robin scheduling, the scheduler has a list of tasks and, when deactivating the running task, 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:

![Round-Robin Scheduling Diagram]

All tasks share the same priority; the possession of the CPU changes periodically after a predefined execution time. This time is called a time slice and may be defined individually for each task.

1.6.2 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.
1.6.3 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 semaphore 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 semaphore, the high priority task is activated and claims the semaphore.

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 semaphore.
1.7 Communication between tasks

In a multitasking (multithreaded) program, multiple tasks and ISRs work completely separately. Because they all work in the same application, it will sometimes be necessary for them to exchange information with each other.

1.7.1 Periodic polling

The easiest way to communicate between different pieces of code is by using global variables. In certain situations, it can make sense for tasks to communicate via global variables, but most of the time this method has disadvantages.

For example, if you want to synchronize a task to start when the value of a global variable changes, you must continually poll this variable, wasting precious computation time and energy, and the reaction time depends on how often you poll.

1.7.2 Event-driven communication mechanisms

When multiple tasks work with each other, they often have to:

- exchange data,
- synchronize with another task, or
- make sure that a resource is used by no more than one task at a time.

For these purposes embOS offers mailboxes, queues, semaphores and events.

1.7.3 Mailboxes and queues

A mailbox is a data buffer managed by the RTOS and is used for sending a message 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 185 and Queues on page 218.

1.7.4 Semaphores and Mutexes

Semaphores and mutexes are used for task synchronization and to manage resources of any kind. The most common are mutex, although semaphores are also used.

For details and samples, refer to the chapters Mutexes on page 158 and Semaphores on page 172.

1.7.5 Events

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 122 and Event Objects on page 134.
1.8 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 458.

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.
1.8.1 Switching stacks

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

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.
1.9 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.
1.10 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.

```c
void 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
}
```

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**

```
void 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
}
```

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 Different builds of embOS

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 following features are included in the different embOS builds:

**Debug code**

The embOS debug code is mainly implemented as assertions which detect application programming errors like calling an API function from an invalid context.

**Stack check**

The stack check detects stack overflows of task stacks, system stack and interrupt stack. Also the maximum amount of used stack can be calculated.

**Profiling**

embOS supports profiling in profiling builds. Profiling makes precise information available about the execution time of individual tasks. You may always use the profiling libraries, but they require larger task control blocks, additional ROM and additional 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.

**Trace**

embOS 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 task at the same priority periodically run with an according time slice.

**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. With the task context extension support each task control block includes function pointer to save and restore routines which are executed during context switch.
### 1.11.1 List of builds

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).

<table>
<thead>
<tr>
<th>Name</th>
<th>Define</th>
<th>Debug Code</th>
<th>Stack Check</th>
<th>Profiling</th>
<th>Trace</th>
<th>RoundRobin</th>
<th>Object names</th>
<th>Task context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extreme Release</td>
<td>OS_LIBMODE_XR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Smallest fastest build.</td>
</tr>
<tr>
<td>Release</td>
<td>OS_LIBMODE_R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Small, fast build, normally used for release build of application.</td>
</tr>
<tr>
<td>Stack Check</td>
<td>OS_LIBMODE_S</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>Same as release, plus stack checking.</td>
</tr>
<tr>
<td>Stackcheck + Profiling</td>
<td>OS_LIBMODE_SP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>Same as stack check, plus profiling.</td>
</tr>
<tr>
<td>Debug</td>
<td>OS_LIBMODE_D</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Maximum runtime checking.</td>
</tr>
<tr>
<td>Debug + Profiling</td>
<td>OS_LIBMODE_DP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>Maximum runtime checking, plus profiling.</td>
</tr>
<tr>
<td>Debug + Trace + Profiling</td>
<td>OS_LIBMODE_DT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>Maximum runtime checking, plus tracing API calls and profiling.</td>
</tr>
<tr>
<td>Safe Library</td>
<td>OS_LIBMODE_SAFE</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>Additional safety features for pre-certified embOS.</td>
</tr>
</tbody>
</table>

### 1.11.2 OS_Config.h

OS_Config.h is part of every embOS port and located in the Start\Inc folder. Use of OS_Config.h 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 configuration and define nothing in the preprocessor settings in release compile configuration. Subsequently, OS_Config.h will automatically define OS_LIBMODE_DP for debug compile configuration and OS_LIBMODE_R for release compile configuration.

<table>
<thead>
<tr>
<th>Compile Configuration</th>
<th>Preprocessor Define</th>
<th>Define Set by OS_Config.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>DEBUG=1</td>
<td>OS_LIBMODE_DP</td>
</tr>
<tr>
<td>Release</td>
<td></td>
<td>OS_LIBMODE_R</td>
</tr>
</tbody>
</table>
### 1.12 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()`), task, interrupt routines and embOS software timer.

**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.

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

**Example**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>main</th>
<th>Unpriv Task</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OS_TASK_Delay()</code></td>
<td>Suspends the calling task for a specified period of time, or waits actively when called from <code>main()</code></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
1.13 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”.

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.

```c
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.

```c
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.

```c
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
        }
    }
}
```
# 1.14 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Unpriv Task</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS_ConfigStop()</strong></td>
<td>Configures the <strong>OS_Stop()</strong> function.</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OS_DeInit()</strong></td>
<td>De-initializes the embOS kernel.</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OS_Init()</strong></td>
<td>Initializes the embOS kernel.</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OS_IsRunning()</strong></td>
<td>Returns whether <strong>OS_Start()</strong> was called.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OS_Start()</strong></td>
<td>Starts the embOS kernel.</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OS_Stop()</strong></td>
<td>Stops the embOS scheduler and returns from <strong>OS_Start()</strong>.</td>
<td>● ● ● ●</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.14.1 OS_ConfigStop()

Description

Configures the OS_Stop() function.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pContext</td>
<td>Pointer to an object of type OS_MAIN_CONTEXT.</td>
</tr>
<tr>
<td>Addr</td>
<td>Address of the buffer which is used to save the main() stack.</td>
</tr>
<tr>
<td>Size</td>
<td>Size of the buffer.</td>
</tr>
</tbody>
</table>

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. If the buffer is too small, debug builds of embOS will call OS_Error() with the error code OS_ERR_OSSTOP_BUFFER. The structure OS_MAIN_CONTEXT is core and compiler specific; it is specifically defined with each embOS port.

Example

```c
#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.
    while (TheAnswerToEverything == 42) {
        printf("%d", TheAnswerToEverything);
    }
    return 0;
}
```
1.14.2 OS_DeInit()

Description
De-initializes the embOS kernel.

Prototype

```c
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.

Example

```c
#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;
}
```
1.14.3 OS_Init()

Description
Initializes the embOS kernel.

Prototype

```c
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 in main() prior to any other embOS API.

Example

```c
#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;
}
```
1.14.4 OS_IsRunning()

Description
Determines whether the embOS scheduler was started by a call of OS_Start().

Prototype
OS_BOOL OS_IsRunning(void);

Return value
= 0  Scheduler is not started.
≠ 0  Scheduler 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 scheduler 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.

Example

```c
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");
    }
}
```
1.14.5 OS_Start()

Description
Starts the embOS scheduler.

Prototype

```c
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() only.

```c
#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;
}
```
1.14.6 OS_Stop()

**Description**

Stops the embOS scheduler and returns from OS_Start().

**Prototype**

```c
tvoid OS_Stop(void);
```

**Additional information**

This function stops the embOS scheduler and the application returns from OS_Start(). OS_ConfigStop() must be called prior to OS_Stop(). If OS_ConfigStop() was not called, debug builds of embOS will call OS_Error() with the error code OS_ERR_CONFIG_OSSTOP. 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 deinitialize all hardware that was initialized 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.

**Example**

```c
#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 2

Tasks
2.1 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).

2.1.1 Example of a task routine as an endless loop

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

2.1.2 Example of a task routine that terminates itself

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

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 as in the sample start projects.
2.2 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 always allow preemptive task switches.

2.2.1 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:

```c
#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);
    }
}

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;
}
```

2.2.2 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:

```c
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.
2.3 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.

2.3.1 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.

2.3.2 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.

2.3.3 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.

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

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

static void MyTask(void) {
    // Task implementation
}
```
MY_APP_TASK* pThis;
OS_TIME Timeout;
char* pString;
pThis = (MY_APP_TASK*)OS_TASK_GetID();
while (1) {
    Timeout = pThis->Timeout;
pString = pThis->pString;
    printf(pString);
    OS_TASK_Delay(Timeout);
}

/***************************************************************************/
/*
 *       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;
}

/***************************************************************************/
## 2.4 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TASK_AddContextExtension()</td>
<td>Adds an additional task context extension.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_AddTerminateHook()</td>
<td>Adds a hook (callback) function to the list of functions which are called when a task is terminated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_CREATE()</td>
<td>Creates a new task.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_Create()</td>
<td>Creates a new task.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_CREATEEX()</td>
<td>Creates a new task and passes a parameter to the task.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_CreateEx()</td>
<td>Creates a new task and passes a parameter to the task.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_Delay()</td>
<td>Suspends the calling task for a specified period of time, or waits actively when called from main().</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_DelayUntil()</td>
<td>Suspends the calling task until a specified time, or waits actively when called from main().</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_Delayus()</td>
<td>Waits for the given time in microseconds.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_GetName()</td>
<td>Returns a pointer to the name of a task.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_GetNumTasks()</td>
<td>Returns the number of tasks.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_GetPriority()</td>
<td>Returns the task priority of a specified task.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_GetSuspendCnt()</td>
<td>Returns the suspension count and suspension state of the specified task.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_GetID()</td>
<td>Returns a pointer to the task control block structure of the currently running task.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_GetTimeSliceRem()</td>
<td>Returns the remaining time slice value of a task.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_IsTask()</td>
<td>Determines whether a task control block belongs to a valid task.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_Index2Ptr()</td>
<td>Returns the task control block of the task with the specified Index.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_RemoveAllTerminateHooks()</td>
<td>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.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_RemoveTerminateHook()</td>
<td>This function removes a hook function from the OS_ON_TERMINATE_HOOK list which contains the</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
<td>Priv. Task</td>
<td>ISR</td>
<td>SW Timer</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------</td>
<td>-----</td>
<td>----------</td>
</tr>
<tr>
<td>list of functions that are called when a task is terminated.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_Resume()</td>
<td>Decrements the suspend count of the specified task and resumes it if the suspend count reaches zero.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TASK_ResumeAll()</td>
<td>Decrements the suspend count of all tasks that have a nonzero suspend count and resumes these tasks when their respective suspend count reaches zero.</td>
<td>●</td>
<td></td>
<td>● ● ●</td>
</tr>
<tr>
<td>OS_TASK_SetContextExtension()</td>
<td>Makes global variables or processor registers task-specific.</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_SetDefaultContextExtension()</td>
<td>Sets the default task context extension.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_SetDefaultStartHook()</td>
<td>Sets a default hook routine which is executed before a task starts.</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_SetInitialSuspendCnt()</td>
<td>Sets the initial suspend count for newly created tasks to 1 or 0.</td>
<td>● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_SetName()</td>
<td>Allows modification of a task name at runtime.</td>
<td>● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_SetPriority()</td>
<td>Assigns a priority to a specified task.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_SetTimeSlice()</td>
<td>Assigns a specified timeslice period to a specified task.</td>
<td>● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_Suspend()</td>
<td>Suspends the specified task and increments a counter.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_SuspendAll()</td>
<td>Suspends all tasks except the running task.</td>
<td>● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_Terminate()</td>
<td>Ends (terminates) a task.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_Wake()</td>
<td>Ends delay of a specified task immediately.</td>
<td>● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASK_Yield()</td>
<td>Calls the scheduler to force a task switch.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.4.1 OS_TASK_AddContextExtension()

Description

Adds an additional 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(). 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

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

Parameters

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

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. OS_TASK_AddContextExtension() must only be used when OS_TASK_SetContextExtension() has been called before.

Example

```c
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);
   }
}
```
2.4.2  OS_TASK_AddTerminateHook()

Description
Adds a hook (callback) function to the list of functions which are called when a task is terminated.

Prototype

```c
void OS_TASK_AddTerminateHook ( OS_ON_TERMINATE_HOOK * pHook,
                                OS_ON_TERMINATE_FUNC * pfUser );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pHook</td>
<td>Pointer to a variable of type <code>OS_ON_TERMINATE_HOOK</code> which will be inserted into the linked list of functions to be called during <code>OS_TASK_Terminate()</code></td>
</tr>
<tr>
<td>pfUser</td>
<td>Pointer to the function of type <code>OS_TERMINATE_FUNC</code> which shall be called when a task is terminated.</td>
</tr>
</tbody>
</table>

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 function 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 function of type `OS_ON_TERMINATE_FUNC` receives the ID of the terminated task as its parameter. `OS_ON_TERMINATE_FUNC` is defined as:

```c
typedef void OS_ON_TERMINATE_FUNC(OS_CONST_PTR OS_TASK * pTask);
```

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 function when using heap memory for the task control block or task stack.

Example

```c
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(&_TerminateHook, TerminateHookFunc);
    ...
}
2.4.3 OS_TASK_CREATE()

Description
Creates a new task.

Prototype

```c
void OS_TASK_CREATE(       OS_TASK*           pTask ,
const char*              pName ,
OS_PRIO           Priority ,
void*              pRoutine ,
void  OS_STACKPTR*  pStack );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
<tr>
<td>pName</td>
<td>Pointer to the name of the task. Can be NULL (or 0) if not used.</td>
</tr>
<tr>
<td>Priority</td>
<td>Priority of the task. Must be within the following range: 1 ≤ Priority ≤ 2^8-1 = 0xFF for 8/16-bit CPUs 1 ≤ Priority ≤ 2^32-1 = 0xFFFFFFFF for 32-bit CPUs Higher values indicate higher priorities. The type OS_PRIO is defined as 32-bit value for 32-bit CPUs and 8-bit value for 8 or 16-bit CPUs by default.</td>
</tr>
<tr>
<td>pRoutine</td>
<td>Pointer to a function that should run as the task body.</td>
</tr>
<tr>
<td>pStack</td>
<td>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.</td>
</tr>
</tbody>
</table>

Additional information

OS_TASK_CREATE() is a macro which calls OS_TASK_Create(). It creates a task and makes it ready for execution by placing it into the READY state. The newly created task will be activated by the scheduler as soon as there is no other task with higher priority in the READY state. If there is another task with the same priority, the new task will be placed immediately before it. This macro is normally used for creating a task instead of the function call OS_TASK_Create() because it has fewer parameters and is therefore easier to use.

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 tasks easy to understand. The absolute value of Priority is of no importance, only the value in comparison to the priorities of other tasks matters.

OS_TASK_CREATE() determines the size of the stack automatically, using sizeof(). This is possible only if the memory area has been defined at compile time.

Note

The stack that you define 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. The task stack cannot be shared between multiple tasks and must be assigned to one task only. The memory used as task stack cannot be used for other purposes unless the task is terminated.
Example

```c
#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);
    OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
    OS_Start(); // Start embOS
    return 0;
}
```
2.4.4 OS_TASK_Create()

Description
Creates a new task.

Prototype

```c
void OS_TASK_Create(       OS_TASK * pTask,
                            const char* pName,
                            OS_PRIO Priority,
                            void (*pRoutine)(),
                            void OS_STACKPTR *pStack,
                            OS_UINT StackSize,
                            OS_UINT TimeSlice);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
<tr>
<td>pName</td>
<td>Pointer to the name of the task. Can be NULL (or 0) if not used. When using an embOS build without task name support, this parameter does not exist and must be omitted. The embOS OS_LIBMODE_XR libraries do not support task names.</td>
</tr>
<tr>
<td>Priority</td>
<td>Priority of the task. Must be within the following range: 1 ≤ Priority ≤ 28 - 1 = 0xFF for 8/16-bit CPUs 1 ≤ Priority ≤ 232 - 1 = 0xFFFFFFFF for 32-bit CPUs Higher values indicate higher priorities. The type OS_PRIO 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.</td>
</tr>
<tr>
<td>pRoutine</td>
<td>Pointer to a function that should run as the task body.</td>
</tr>
<tr>
<td>pStack</td>
<td>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.</td>
</tr>
<tr>
<td>StackSize</td>
<td>Size of stack in bytes.</td>
</tr>
<tr>
<td>TimeSlice</td>
<td>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 ≤ TimeSlice ≤ 255. When using an embOS build without round-robin support, this parameter does not exist and must be omitted. The embOS OS_LIBMODE_XR libraries do not support round-robin and time slice.</td>
</tr>
</tbody>
</table>

Additional information

This function works the same way as OS_TASK_CREATE(), except that all parameters of the task can be specified. The task can be dynamically created because the stack size is not calculated automatically as it is with the macro. A time slice 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 46)
Note

The stack that you define 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. The task stack cannot be shared between multiple tasks and must be assigned to one task only. The memory used as task stack cannot be used for other purposes unless the task is terminated.

Example

```c
#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, sizeof(StackLP), 2);
    OS_Start();  // Start embOS
    return 0;
}
```
2.4.5 OS_TASK_CREATEEX()

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

Prototype

```c
void OS_TASK_CREATEEX (       OS_TASK *           pTask ,
     const  char *              pName ,
     OS_PRIO           Priority ,
     void *              pRoutine ,
     void *              pStack ,
     void *              pContext );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type <code>OS_TASK</code>.</td>
</tr>
<tr>
<td>pName</td>
<td>Pointer to the name of the task. Can be <code>NULL</code> (or 0) if not used.</td>
</tr>
</tbody>
</table>
| Priority      | Priority of the task. Must be within the following range:
|               | 1 ≤ Priority ≤ 2^8 - 1 = 0xFF for 8/16-bit CPUs
|               | 1 ≤ Priority ≤ 2^32 - 1 = 0xFFFFFFFF for 32-bit CPUs
|               | Higher values indicate higher priorities. The type `OS_PRIO` is defined as
|               | 32-bit value for 32-bit CPUs and 8-bit value for 8 or 16 bit CPUs by
|               | default.                                                                    |
| pRoutine      | Pointer to a function that should 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.                                                                 |
| pContext      | Parameter passed to the created task function.                             |

Additional information

`OS_TASK_CREATEEX()` is a macro calling `OS_TASK_CreateEx()`. It works like `OS_TASK_CREATE()` but allows passing a parameter to the task. Using a void pointer as an additional parameter gives the flexibility to pass any kind of data to the task function.

Note

The stack that you define 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. The task stack cannot be shared between multiple tasks and must be assigned to one task only. The memory used as task stack cannot be used for other purposes unless the task is terminated.
Example
The following example is delivered in the Application folder of embOS.

```c
#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, (void*) 50);
    OS_TASK_CREATEEX(&TCBLP, "LP Task", 50, Task, StackLP, (void*) 200);
    OS_Start();      // Start embOS
    return 0;
}
```
2.4.6 OS_TASK_CreateEx()

Description

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

Prototype

```c
void OS_TASK_CreateEx(   OS_TASK* pTask,
               const char* pName,
               OS_PRIO Priority,
               void ( * pRoutine )( void * pVoid ),
               void OS_STACKPTR * pStack,
               OS_UINT StackSize,
               OS_UINT TimeSlice,
               void* pContext );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
<tr>
<td>pName</td>
<td>Pointer to the name of the task. Can be NULL (or 0) if not used. When using an embOS build without task name support, this parameter does not exist and must be omitted. The embOS OS_LIBMODE_XR libraries do not support task names.</td>
</tr>
<tr>
<td>Priority</td>
<td>Priority of the task. Must be within the following range: 1 ≤ Priority ≤ 2^8 - 1 = 0xFF for 8/16-bit CPUs 1 ≤ Priority ≤ 2^32 - 1 = 0xFFFFFFFF for 32-bit CPUs Higher values indicate higher priorities. The type OS_PRIO 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.</td>
</tr>
<tr>
<td>pRoutine</td>
<td>Pointer to a function that should run as the task body.</td>
</tr>
<tr>
<td>pStack</td>
<td>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.</td>
</tr>
<tr>
<td>StackSize</td>
<td>Size of stack in bytes.</td>
</tr>
<tr>
<td>TimeSlice</td>
<td>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 ≤ TimeSlice ≤ 255. When using an embOS build without round-robin support, this parameter does not exist and must be omitted. The embOS OS_LIBMODE_XR libraries do not support round-robin and time slice.</td>
</tr>
<tr>
<td>pContext</td>
<td>Parameter passed to the created task.</td>
</tr>
</tbody>
</table>

Additional information

This function works the same way as OS_TASK_CREATE(), except that all parameters of the task can be specified. The task can be dynamically created because the stack size is not calculated automatically as it is with the macro. A time slice 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 46)

Note

The stack that you define 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. The task stack cannot be shared between multiple tasks and must be assigned to one task only. The memory used as task stack cannot be used for other purposes unless the task is terminated.

```c
#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, sizeof(StackLP), 2, (void*) 200);
    OS_Start();  // Start embOS
    return 0;
}
```
2.4.7 OS_TASK_Delay()

Description
Suspends the calling task for a specified period of time, or waits actively when called from main().

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Time interval to delay. Must be within the following range: 0 ≤ t ≤ 2^{15} - 1 = 0x7FFF for 8/16-bit CPUs 0 ≤ t ≤ 2^{31} - 1 = 0x7FFFFFFF for 32-bit CPUs Please note that these are signed values.</td>
</tr>
</tbody>
</table>

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 ≤ delay ≤ 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.

Example

```c
void Hello(void) {
    printf("Hello");
    printf("The next output will occur in 5000 system ticks");
    OS_TASK_Delay(5000);
    printf("Delay is over");
}
```
2.4.8 OS_TASK_DelayUntil()

**Description**

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

**Prototype**

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Time to delay until. Must be within the following range: 0 ≤ t ≤ 2^{16} - 1 = 0xFFFF for 8/16-bit CPUs 0 ≤ t ≤ 2^{32} - 1 = 0xFFFFFFFF for 32-bit CPUs Also, the following additional condition must be met: 1 ≤ (t - OS_Global.Time) ≤ 2^{15} - 1 = 0x7FFF for 8/16-bit CPUs 1 ≤ (t - OS_Global.Time) ≤ 2^{31} - 1 = 0xFFFFFFFF for 32-bit CPUs Please note that these are signed values.</td>
</tr>
</tbody>
</table>

**Additional information**

`OS_TASK_DelayUntil()` suspends the calling task until the global time-variable `OS_Global.Time` (see `OS_Global.Time` on page 489) 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 a 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.

**Example**

```c
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++;
        }
    }
}
```

In the example above, using `OS_TASK_Delay()` could lead to accumulating delays and would cause the simple “clock” to be slow. Using `OS_TASK_DelayUntil()` instead avoids accumulating delays.
CHAPTER 2

2.4.9  OS_TASK_Delayus()

Description

Waits for the given time in microseconds.

Prototype

```c
void OS_TASK_Delayus(OS_U16 us);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>us</td>
<td>Number of microseconds to delay. Must be within the following range: (1 \leq us \leq 2^{15} - 1 = 0x7FFF). Please note that these are signed values.</td>
</tr>
</tbody>
</table>

Additional information

This function can be used for short delays. OS_TASK_Delayus() must only be called with interrupts enabled and after OS_Init(), OS_InitHW() and OS_TIME_ConfigSysTimer() have been called. This only works when the embOS system timer is running. An debug build of OS_TASK_Delayus() checks whether interrupts are enabled and calls OS_Error() if they are not.

OS_TASK_Delayus() does not block task switches and does not block interrupts. Therefore, the delay may not be accurate because the function may be interrupted for an undefined time. The delay duration therefore is a minimum delay.

OS_TASK_Delayus() does not suspend the calling task, thus all tasks with lower priority can not interrupt OS_TASK_Delayus() and will not be executed before OS_TASK_Delayus() returns.

Note

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

Example

```c
void Hello(void) {
    printf("Hello");
    printf("The next output will occur in 500 microseconds");
    OS_TASK_Delayus(500);
    printf("Delay is over");
}
```
2.4.10 OS_TASK_GetName()

Description

Returns a pointer to the name of a task.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
</tbody>
</table>

Return value

A pointer to the name of the task. NULL indicates that the task has no name.

When using an embOS build without task name support, OS_TASK_GetName() returns “n/a” in any case. The embOS OS_LIBMODE_XR libraries do not support task names.

Additional information

If pTask is NULL, 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 and does not specify a valid task, a debug build of embOS calls OS_Error(). The release build of embOS cannot check the validity of pTask and may therefore return invalid values if pTask does not specify a valid task.

Example

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

Description
Returns the number of tasks.

Prototype

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

Return value
Number of tasks.

Example

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

Description

Returns the task priority of a specified task.

Prototype

OS_PRIO OS_TASK_GetPriority(OS_CONST_PTR OS_TASK *pTask);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK or NULL for current task.</td>
</tr>
</tbody>
</table>

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 pTask is NULL, the function returns the priority of the currently running task. If pTask does not specify a valid task, the debug build of embOS calls OS_Error(). The release build of embOS cannot check the validity of pTask and may therefore return invalid values if pTask does not specify a valid task.

Note

This function can be called from within an interrupt handler with OS_TASK_GetPriority(NULL) but if the handler interrupts OS_Idle() no task is currently running and no valid task is specified. The debug build of embOS calls OS_Error() in this case. We suggest to call OS_TASK_GetPriority() from an interrupt handler with a pointer to a valid task control block only.

Example

```c
void PrintPriority(const OS_TASK* pTask) {
  OS_PRIO Prio;
  Prio = OS_TASK_GetPriority(pTask);
  printf("Priority of task 0x%x = %u\n", pTask, Prio);
}
```
2.4.13 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**

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
</tbody>
</table>

**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 pTask does not specify a valid task, the debug build of embOS calls OS_Error(). The release build of embOS cannot check the validity of pTask and may therefore return invalid values if pTask does not specify a valid task. When tasks are created and terminated dynamically, OS_TASK_IsTask() may be called prior to calling OS_TASK_GetSuspendCnt() to determine whether a task is valid. The returned value can be used to resume a suspended task by calling OS_TASK_Resume() as often as indicated by the returned value.

**Example**

```c
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--;
    }
}
```
2.4.14 OS_TASK_GetID()

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

Prototype

OS_TASK* OS_TASK_GetID(void);

Return value
A pointer to the task control block. NULL indicates that no task is executing.

Additional information
This function may be used for determining which task is executing. This may be helpful if the reaction of any function depends on the currently running task.

Example

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

Description

Returns the remaining time slice value of a task.

Prototype

OS_U8 OS_TASK_GetTimeSliceRem(OS_CONST_PTR OS_TASK *pTask);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
</tbody>
</table>

Return value

Remaining time slice value of the task.

Additional information

If NULL is passed for pTask, the currently running task is used. However, NULL must not be passed for pTask from main(), a timer callback or from an interrupt handler. A debug build of embOS will call OS_Error() in case pTask does not indicate a valid task. The release build of embOS cannot check the validity of pTask and may therefore return invalid values if pTask does not specify a valid task.

The function is unavailable when using an embOS build without round-robin support. The embOS OS_LIBMODE_XR libraries do not support round-robin. In that case OS_TASK_GetTimeSliceRem() returns zero.

Example

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

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

Description
Determines whether a task control block belongs to a valid task.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
</tbody>
</table>

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.

Example

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

    b = OS_TASK_IsTask(pTask);
    if (b == 0) {
        printf("TCB can be reused for another task.
    } else {
        printf("TCB refers to a valid task.
    }
}
```
2.4.17 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskIndex</td>
<td>Index of a task control block in the task list. This is a zero based index. TaskIndex 0 identifies the first task control block.</td>
</tr>
</tbody>
</table>

Return value

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

Example

```c
void PrintTaskName(int TaskIndex) {
    OS_TASK* pTask;

    pTask = OS_TASK_Index2Ptr(TaskIndex);
    if (pTask != NULL) {
        printf("%s", pTask->Name);
    }
}

void HPTask(void) {
    // Print the task name of the first task in the task list
    PrintTaskName(0);
    while (1) {
        OS_TASK_Delay(100);
    }
}
```
2.4.18 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().

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();
    ...
}
```
2.4.19 OS_TASK_RemoveTerminateHook()

**Description**
This function removes a hook function from the `OS_ON_TERMINATE_HOOK` list which contains the list of functions that are called when a task is terminated.

**Prototype**
```c
void OS_TASK_RemoveTerminateHook(OSCONST_PTR OS_ON_TERMINATE_HOOK *pHook);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pHook</td>
<td>Pointer to a variable of type <code>OS_ON_TERMINATE_HOOK</code>.</td>
</tr>
</tbody>
</table>

**Additional information**
`OS_TASK_RemoveTerminateHook()` removes the specified hook function which was previously added by `OS_TASK_AddTerminateHook()`.

**Example**
```c
OS_ON_TERMINATE_HOOK _TerminateHook;

void TerminateHookFunc(OSCONST_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);
    ...
}
```
2.4.20  OS_TASK_Resume()

Description
Decrement the suspend count of the specified task and resumes it if the suspend count reaches zero.

Prototype
void OS_TASK_Resume(OS_TASK* pTask);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
</tbody>
</table>

Additional information
The specified task’s suspend count is decremented. When the resulting value is zero, the execution of the specified task is resumed. If the task is not blocked by other task blocking mechanisms, the task is placed in the READY state and continues operation according to the rules of the scheduler. In debug builds of embOS, OS_TASK_Resume() checks the suspend count of the specified task. If the suspend count is zero when OS_TASK_Resume() is called, OS_Error() is called with error OS_ERR_RESUME_BEFORE_SUSPEND.

Example
Please refer to the example of OS_TASK_Suspend().
2.4.21 OS_TASK_ResumeAll()

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

Prototype

```c
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.

Example
Please refer to the example of OS_TASK_SetInitialSuspendCnt().
2.4.22 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 coprocessor).
- 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**

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pExtendContext</td>
<td>Pointer to the OS_EXTEND_TASK_CONTEXT structure which contains the addresses of the specific save and restore functions that save and restore the extended task context during task switches.</td>
</tr>
</tbody>
</table>

**Additional information**

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.

OS_TASK_SetContextExtension() is not available in the XR libraries.

**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().

The OS_EXTEND_TASK_CONTEXT structure is defined as follows:

```c
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:

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

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.
Example
The following example is delivered in the Application folder of embOS.

```c
#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;

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

static void OS_STACKPTR* _Save(void OS_STACKPTR* pStack) {
    CONTEXT_EXTENSION* p;
    p = ((CONTEXT_EXTENSION*)pStack) - (1 - OS_STACK_AT_BOTTOM);
    p->GlobalVar = GlobalVar;
    return (void OS_STACKPTR*)p;
}
```
CHAPTER 2  API functions

/*****************************************************************************
*       _Restore()
*  Function description
*    This function restores an extended task context.
*/
static void OS_STACKPTR* _Restore(const void OS_STACKPTR* pStack) {
    const CONTEXT_EXTENSION* p;
    // Create pointer to our structure
    p = (const CONTEXT_EXTENSION*)pStack - (1 - OS_STACK_AT_BOTTOM);
    // Restore all members of the structure
    GlobalVar = p->GlobalVar;
    return (void OS_STACKPTR*)p;
}

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

/*****************************************************************************
*       HPTask()
*  Function description
*    During the execution of this function, the thread-specific
*    global variable GlobalVar always has the same value of 1.
*/
static void HPTask(void) {
    OS_TASK_SetContextExtension(&_SaveRestore);
    GlobalVar = 1;
    while (1) {
        OS_TASK_Delay(10);
    }
}

/*****************************************************************************
*       LPTask()
*  Function description
*    During the execution of this function, the thread-specific
*    global variable GlobalVar always has the same value of 2.
*/
static void LPTask(void) {
    OS_TASK_SetContextExtension(&_SaveRestore);
    GlobalVar = 2;
    while (1) {
        OS_TASK_Delay(50);
    }
}

/*****************************************************************************
*       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;
}
2.4.23 OS_TASK_SetDefaultContextExtension()

**Description**
Sets the default task context extension.

**Prototype**

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pExtendContext</td>
<td>Pointer to the OS_EXTEND_TASK_CONTEXT structure which contains the addresses of the specific save and restore functions that save and restore the extended task context during task switches.</td>
</tr>
</tbody>
</table>

**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.

**Example**

```c
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;
}
```
2.4.24  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

```c
void OS_TASK_SetDefaultStartHook(voidRoutine* pfHook);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfHook</td>
<td>Pointer to the hook routine. If NULL is passed no hook routine gets executed.</td>
</tr>
</tbody>
</table>

Additional information
After calling OS_TASK_SetDefaultStartHook() all newly created tasks will automatically
call this hook routine before the tasks are started for the first time. The same hook function
is used for all tasks.

Example

```c
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;
}
```
2.4.25  **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**

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SuspendCnt</td>
<td>1: Tasks will be created in suspended state.</td>
</tr>
<tr>
<td></td>
<td>0: Tasks will be created normally, unsuspended.</td>
</tr>
</tbody>
</table>

**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.

**Example**

```c
//
// 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.
    }
}
```
2.4.26 OS_TASK_SetName()

Description
Allows modification of a task name at runtime.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
<tr>
<td>s</td>
<td>Pointer to a null-terminated string which is used as task name.</td>
</tr>
</tbody>
</table>

Additional information

If NULL is passed for pTask, the currently running task is modified. However, NULL must not be passed for pTask from main(), from a timer callback or from an interrupt handler. A debug build of embOS will call OS_Error() in case pTask does not indicate a valid task.

When using an embOS build without task name support, OS_TASK_SetName() performs no modifications at all. The embOS OS_LIBMODE_XR libraries do not support task names.

Example

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

Description
Assigns a priority to a specified task.

Prototype

```c
void OS_TASK_SetPriority(OS_TASK* pTask,
OS_PRIO Priority);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pTask</strong></td>
<td>Pointer to a task control block of type OS_TASK or NULL for current task.</td>
</tr>
<tr>
<td><strong>Priority</strong></td>
<td>Priority of the task. Must be within the following range: 1 ≤ Priority ≤ 2^8 - 1 = 0xFF for 8/16-bit CPUs 1 ≤ Priority ≤ 2^32 - 1 = 0xFFFFFFFF for 32-bit CPUs Higher values indicate higher priorities. The type OS_PRIO is defined as 32-bit value for 32-bit CPUs and 8-bit value for 8 or 16-bit CPUs per default.</td>
</tr>
</tbody>
</table>

Additional information

If NULL is passed for **pTask**, the currently running task is modified. However, NULL must not be passed for **pTask** from main(). A debug build of embOS will call OS_Error() in case **pTask** does not indicate a valid task.

Calling this function might lead to an immediate task switch.

Example

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

**Description**

Assigns a specified timeslice period to a specified task.

**Prototype**

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
<tr>
<td>TimeSlice</td>
<td>New time slice period for the task. Must be within the following range:</td>
</tr>
<tr>
<td></td>
<td>0 ≤ TimeSlice ≤ 255.</td>
</tr>
</tbody>
</table>

**Return value**

Previous time slice period of the task.

**Additional information**

If **NULL** is passed for `pTask`, the currently running task is modified. However, **NULL** must not be passed for `pTask` from `main()`, a timer callback or from an interrupt handler. A debug build of embOS will call `OS_Error()` in case `pTask` does not indicate a valid task.

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 46).

The function is unavailable when using an embOS build without round-robin support. The embOS **OS_LIBMODE_XR** libraries do not support round-robin. In that case `OS_TASK_SetTimeSlice()` does nothing and returns zero.

**Example**

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

Description
Suspends the specified task and increments a counter.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
</tbody>
</table>

Additional information

If `pTask` is NULL, the current task suspends. If the function succeeds, execution of the specified task is suspended and the task’s suspend count is incremented. The specified task will be suspended immediately. It can only be restarted by a call of `OS_TASK_Resume()`. Every task has a suspend count with a maximum value of `OS_MAX_SUSPEND_CNT`. If the suspend count is greater than zero, the task is suspended.

In debug builds of embOS, upon calling `OS_TASK_Suspend()` more often than the maximum value without calling `OS_TASK_Resume()` the task’s internal suspend count is not incremented and `OS_Error()` is called with error `OS_ERR_SUSPEND_TOO_OFTEN`. Cannot be called from main(), an interrupt handler or software timer as this function may cause an immediate task switch. The debug build of embOS will call the `OS_Error()` function when `OS_TASK_Suspend()` is not called from a task.

Example

```c
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
}
```
2.4.30  OS_TASK_SuspendAll()

Description
Suspends all tasks except the running task.

Prototype

```c
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()`.

Example
Please refer to the example of `OS_TASK_SetInitialSuspendCnt()`.
2.4.31 OS_TASK_Terminate()

Description
Ends (terminates) a task.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK. A value of NULL terminates the current task.</td>
</tr>
</tbody>
</table>

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.

Example

```c
void Task(void) {
    DoSomething();
    OS_TASK_Terminate(NULL); // Terminate itself
}
```
2.4.32 OS_TASK_Wake()

Description
Ends delay of a specified task immediately.

Prototype
void OS_TASK_Wake(OS_TASK* pTask);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
</tbody>
</table>

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.

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;
}
```
2.4.33 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.

Example

```c
#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 don'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 3

Software Timers
3.1 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 defined with the macro \texttt{OS\_TIMER\_CREATE()}. Timers can be stopped, started and retriggered 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. Generally, timers run in single-shot mode, which means they expire exactly once and call their callback routine exactly once. By calling \texttt{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 \texttt{OS\_TIMER\_GetStatus()}, \texttt{OS\_TIMER\_GetRemainingPeriod()} and \texttt{OS\_TIMER\_GetPeriod()}.

Example

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

static OS\_TIMER TIMER50, TIMER200;

static void Timer50(void) {
    BSP\_ToggleLED(0);
    OS\_TIMER\_Restart(&TIMER50);
}

static void Timer200(void) {
    BSP\_ToggleLED(1);
    OS\_TIMER\_Restart(&TIMER200);
}

int main(void) {
    OS\_Init(); // Initialize embOS
    OS\_InitHW(); // Initialize required hardware
    BSP\_Init(); // Initialize LED ports
    OS\_TIMER\_CREATE(&TIMER50, Timer50, 50);
    OS\_TIMER\_CREATE(&TIMER200, Timer200, 200);
    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 `0x7FFFFFFF00` 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.

**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.
## 3.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>main</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TIMER_CREATE()</td>
<td>Macro that creates and starts a software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_Create()</td>
<td>Creates a software timer without starting it.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_CREATEEX()</td>
<td>Macro that creates and starts an extended software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_CreateEx()</td>
<td>Creates an extended software timer without starting it.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_Delete()</td>
<td>Stops and deletes a software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_DeleteEx()</td>
<td>Stops and deletes an extended software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_GetCurrent()</td>
<td>Returns a pointer to the data structure of the timer that just expired.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_GetCurrentEx()</td>
<td>Returns a pointer to the data structure of the extended software timer that just expired.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_GetPeriod()</td>
<td>Returns the current reload value of a software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_GetPeriodEx()</td>
<td>Returns the current reload value of an extended software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_GetRemainingPeriod()</td>
<td>Returns the remaining timer value of a software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_GetRemainingPeriodEx()</td>
<td>Returns the remaining timer value of an extended software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_GetStatus()</td>
<td>Returns the current timer status of a software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_GetStatusEx()</td>
<td>Returns the current timer status of an extended software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_Restart()</td>
<td>Restarts a software timer with its initial time value.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_RestartEx()</td>
<td>Restarts an extended software timer with its initial time value.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_SetPeriod()</td>
<td>Sets a new timer reload value for a software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_SetPeriodEx()</td>
<td>Sets a new timer reload value for an extended software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_Start()</td>
<td>Starts a software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_StartEx()</td>
<td>Starts an extended software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_Stop()</td>
<td>Stops a software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_StopEx()</td>
<td>Stops an extended software timer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIMER_Trigger()</td>
<td>Ends a software timer at once and calls the timer callback function.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
<td>main</td>
<td>Priv Task</td>
<td>ISR</td>
<td>SW Timer</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
<td>-----------</td>
<td>-----</td>
<td>----------</td>
</tr>
<tr>
<td>OS_TIMER_TriggerEx()</td>
<td>Ends an extended software timer at once and calls the timer callback function.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
## 3.2.1 OS_TIMER_CREATE()

### Description
Macro that creates and starts a software timer.

### Prototype

```
void OS_TIMER_CREATE ( OS_TIMER*         pTimer ,
                       OS_TIMERROUTINE*  Callback ,
                       OS_TIME          Period );
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimer</td>
<td>Pointer to a software timer object of type <code>OS_TIMER</code>.</td>
</tr>
<tr>
<td>Callback</td>
<td>Pointer to the callback routine to be called by the RTOS after expiration of the delay. The callback function must be a void function which does not take any parameters and does not return any value.</td>
</tr>
<tr>
<td>Period</td>
<td>Initial period in embOS system ticks. The data type <code>OS_TIME</code> is defined as an integer, therefore valid values are:</td>
</tr>
</tbody>
</table>
|             | \[
|             | \[1 \leq \text{Period} \leq 2^{15} - 1 = 0x7FFF for 8/16-bit CPUs \]
|             | \[1 \leq \text{Period} \leq 2^{31} - 1 = 0x7FFFFFFF for 32-bit CPUs \]                                                                       |

### Additional information
embOS keeps track of the timers by using a linked list. 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).

This deprecated macro uses the functions `OS_TIMER_Create()` and `OS_TIMER_Start()`. It is supplied for backward compatibility; in newer applications these routines should instead be called directly.

`OS_TIMERROUTINE` is defined in `RTOS.h` as follows:

```
typedef void OS_TIMERROUTINE ( void );
```

Source of the macro (in `RTOS.h`):

```c
#define OS_TIMER_CREATE ( pTimer , c , d )   
  OS_TIMER_Create ( pTimer , c , d );  
  OS_TIMER_Start ( pTimer );
```

### Example

```c
static OS_TIMER TIMER100 ;

static void Timer100(void) {
  BSP_ToggleLED ( 0 );
  OS_TIMER_Restart (& TIMER100 ); // Make timer periodic
}

void InitTask ( void ) {  
  // Create and implicitly start Timer100
  //
  OS_TIMER_CREATE ( &TIMER100 , Timer100 , 100 );
}
```
3.2.2 OS_TIMER_Create()

Description
Creates a software timer without starting it.

Prototype

```c
void OS_TIMER_Create(OS_TIMER *pTimer,
                     OS_TIMERROUTINE *Callback,
                     OS_TIME Period);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimer</td>
<td>Pointer to a software timer object of type OS_TIMER.</td>
</tr>
<tr>
<td>Callback</td>
<td>Pointer to the callback routine to be called by the RTOS after expiration of the delay.</td>
</tr>
</tbody>
</table>
| Period    | Initial period in embOS system ticks. The data type OS_TIME 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

embOS keeps track of the timers by using a linked list. 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_Start() or OS_TIMER_Restart().

OS_TIMERROUTINE is defined in RTOS.h as follows:

```c
typedef void OS_TIMERROUTINE(void);
```

Example

```c
static OS_TIMER TIMER100;

static void Timer100(void) {
    BSP_ToggleLED(0);
    OS_TIMER_Restart(&TIMER100); // Make timer periodic
}

void InitTask(void) {
    // Create Timer100, but start it seperately
    OS_TIMER_Create(&TIMER100, Timer100, 100);
    OS_TIMER_Start(&TIMER100);
}
```
3.2.3 OS_TIMER_CREATEEX()

Description
Macro that creates and starts an extended software timer.

Prototype

```c
void OS_TIMER_CREATEEX(OS_TIMER_EX* pTimerEx, 
                        OS_TIMER_EX_ROUTINE* Callback, 
                        OS_TIME Period, 
                        void* pData);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to an extended software timer object of type OS_TIMER_EX.</td>
</tr>
<tr>
<td>Callback</td>
<td>Pointer to the callback routine to be called by the RTOS after expiration of the delay. The callback function must be of type OS_TIMER_EX_ROUTINE which takes a void pointer as parameter and does not return any value.</td>
</tr>
<tr>
<td>Period</td>
<td>Initial period in embOS system ticks. The data type OS_TIME is defined as an integer, therefore valid values are: 1 \leq Period \leq 2^{15} - 1 = 0x7FFF for 8/16-bit CPUs \n\n 1 \leq Period \leq 2^{31} - 1 = 0x7FFFFFFF for 32-bit CPUs</td>
</tr>
<tr>
<td>pData</td>
<td>A void pointer which is used as parameter for the extended timer callback function.</td>
</tr>
</tbody>
</table>

Additional information
embOS keeps track of the timers by using a linked list. 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). This macro uses the functions OS_TIMER_CreateEx() and OS_TIMER_StartEx().

OS_TIMER_EX_ROUTINE is defined in RTOS.h as follows:

```c
typedef void OS_TIMER_EX_ROUTINE(void* pVoid);
```

Source of the macro (in RTOS.h):

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

Example

```c
static OS_TIMER_EX TIMER100;
static OS_TASK TCB_HP;

static void Timer100(void* pTask) {
    if (pTask != NULL) {
        OS_TASKEVENT_Set(0x01, (OS_TASK*)pTask);
    }
    OS_TIMER_RestartEx(&TIMER100); // Make timer periodic
}

void InitTask(void) {
    // Create and implicitly start Timer100
    OS_TIMER_CREATEEX(&TIMER100);
}```
//
OS_TIMER_CREATEEX(&TIMER100, Timer100, 100, (void*)&TCB_HP);
3.2.4 OS_TIMER_CreateEx()

Description

Creates an extended software timer without starting it.

Prototype

```c
void OS_TIMER_CreateEx(OS_TIMER_EX * pTimerEx, OS_TIMER_EX_ROUTINE * Callback, OS_TIME Period, void * pData);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to an extended software timer object of type OS_TIMER_EX.</td>
</tr>
<tr>
<td>Callback</td>
<td>Pointer to the callback routine of type OS_TIMER_EX_ROUTINE to be called by the RTOS after expiration of the timer.</td>
</tr>
</tbody>
</table>
| Period    | Initial period in embOS system ticks. The data type OS_TIME is defined as an integer, therefore valid values are:  
            1 ≤ Period ≤ 2^{15\text{ - 1}} = 0x7FFF for 8/16-bit CPUs.  
            1 ≤ Period ≤ 2^{31\text{ - 1}} = 0x7FFFFFFF for 32-bit CPUs. |
| pData     | A void pointer which is used as parameter for the extended timer callback function. |

Additional information

embOS keeps track of the timers by using a linked list. 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().

OS_TIMER_EX_ROUTINE is defined in RTOS.h as follows:

```c
typedef void OS_TIMER_EX_ROUTINE(void * pVoid);
```

Example

```c
static OS_TIMER_EX TIMER100;
static OS_TASK TCB_HP;

static void Timer100(void * pTask) {
    if (pTask != NULL) {
        OS_TASKEVENT_Set(0x01, (OS_TASK*)pTask);
    }
    OS_TIMER_RestartEx(&TIMER100);  // Make timer periodic
}

void InitTask(void) {
    // // Create Timer100, but start it seperately
    // OS_TIMER_CreateEx(&TIMER100, Timer100, 100, (void*)&TCB_HP);
    OS_TIMER_Start(&TIMER100);
}
```
3.2.5 OS_TIMER_Delete()

Description
Stops and deletes a software timer.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimer</td>
<td>Pointer to a software timer object of type OS_TIMER.</td>
</tr>
</tbody>
</table>

Additional information
The timer is stopped and therefore removed from the linked list of running timers. In debug builds of embOS, the timer is also marked as invalid.
3.2.6  OS_TIMER_DeleteEx()

   Description
   Stops and deletes an extended software timer.

   Prototype

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

   Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to the OS_TIMER_EX data structure which contains the data of the timer.</td>
</tr>
</tbody>
</table>

   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 as invalid.
3.2.7 OS_TIMER_GetCurrent()

Description
Returns a pointer to the software timer object that just has expired.

Prototype
OS_TIMER* OS_TIMER_GetCurrent (void);

Return value
A pointer to the software timer object of type OS_TIMER.

Additional information
The return value of OS_TIMER_GetCurrent() is valid during execution of a timer callback function; otherwise it is undefined. If only one callback function should be used for multiple timers, this function can be used for examining the timer that expired. The example below shows one usage of OS_TIMER_GetCurrent(). Since version 3.32m of embOS, the extended timer structure and functions may be used to generate and use a software timer with an individual parameter for the callback function. Please be aware that OS_TIMER must be the first member of the structure.

Example

```c
#include "RTOS.h"

typedef struct {
    OS_TIMER Timer;    // OS_TIMER has to be the first element
    void* pUser;       // Any other data type may be used to extend the struct
} TIMER_EX;

static TIMER_EX Timer_User;
static int a;

static void _cb (void) {
    TIMER_EX* p = (TIMER_EX*) OS_TIMER_GetCurrent();
    void* pUser = p->pUser;  // Examine user data
    OS_TIMER_Restart(&p->Timer);  // Make timer periodic
}

static void _CreateTimer (TIMER_EX* timer, OS_TIMERROUTINE* Callback, OS_UINT Period, void* pUser) {
    timer->pUser = pUser;
    OS_TIMER_Create(&timer->Timer, Callback, Period);
}

int main(void) {
    OS_Init();            // Initialize embOS
    OS_InitHW();          // Initialize required hardware
    _CreateTimer(&Timer_User, _cb, 100, &a);
    OS_Start();           // Start embOS
    return 0;
}
```
3.2.8 OS_TIMER_GetCurrentEx()

Description
Returns a pointer to the extended software timer object that just has expired.

Prototype

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

Return value
A pointer to the extended software timer object of type OS_TIMER_EX.

Additional information
The return value of OS_TIMER_GetCurrentEx() is valid during execution of a timer callback function; otherwise it is undefined. If one callback function should be used for multiple extended timers, this function can be used for examining the timer that expired.

Example

```c
OS_TIMER_EX MyTimerEx;

static void _cbTimerEx(void* pData) {
    OS_TIMER_EX* pTimerEx = OS_TIMER_GetCurrentEx();
    OS_TASKEVENT_Set(0x01, (OS_TASK*)pData);
    OS_TIMER_Restart(pTimerEx); // Make timer periodic
}
```
3.2.9 OS_TIMER_GetPeriod()

**Description**

Returns the current reload value of a software timer.

**Prototype**

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pTimer</code></td>
<td>Pointer to a software timer object of type <code>OS_TIMER</code>.</td>
</tr>
</tbody>
</table>

**Return value**

The returned value is the current reload value of a software timer. The data type `OS_TIME` is defined as an integer, therefore valid values are:

- \( 1 \leq x \leq 2^{15} - 1 = 0x7FFF \) for 8/16-bit CPUs.
- \( 1 \leq x \leq 2^{31} - 1 = 0x7FFFFFFF \) for 32-bit CPUs.

**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 retriggered by `OS_TIMER_Restart()`. 
3.2.10  OS_TIMER_GetPeriodEx()

Description
Returns the current reload value of an extended software timer.

Prototype

\[
\text{OS\_TIME OS\_TIMER\_GetPeriodEx(OS\_CONST\_PTR OS\_TIMER\_EX\* pTimerEx)};
\]

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to an extended software timer object of type OS_TIMER_EX.</td>
</tr>
</tbody>
</table>

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

The data type OS\_TIME is defined as an integer, therefore valid values are:

- \(1 \leq x \leq 2^{15} - 1 = 0x7fff\) for 8/16-bit CPUs.
- \(1 \leq x \leq 2^{31} - 1 = 0x7fffffff\) for 32-bit CPUs.

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 retriggered by OS\_TIMER\_RestartEx().
3.2.11  OS_TIMER_GetRemainingPeriod()

Description
Returns the remaining timer value of a software timer.

Prototype
OS_TIME OS_TIMER_GetRemainingPeriod(OS_CONST_PTR OS_TIMER *pTimer);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimer</td>
<td>Pointer to a software timer object of type OS_TIMER.</td>
</tr>
</tbody>
</table>

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

The data type OS_TIME is defined as an integer, therefore valid values are:

- $1 \leq x \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs.
- $1 \leq x \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.
3.2.12 OS_TIMER_GetRemainingPeriodEx()

Description
Returns the remaining timer value of an extended software timer.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to an extended software timer object of type OS_TIMER_EX.</td>
</tr>
</tbody>
</table>

Return value

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

The data type OS_TIME is defined as an integer, therefore valid values are:

- $1 \leq x \leq 2^{15} - 1 = 0xFFFF$ for 8/16-bit CPUs.
- $1 \leq x \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.
3.2.13  OS_TIMER_GetStatus()

Description
Returns the current timer status of a software timer.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimer</td>
<td>Pointer to a software timer object of type OS_TIMER.</td>
</tr>
</tbody>
</table>

Return value

Denotes whether the specified timer is running or not:

- \( = 0 \)  Timer has stopped.
- \( \neq 0 \)  Timer is running.
3.2.14 OS_TIMER_GetStatusEx()

**Description**

Returns the current timer status of an extended software timer.

**Prototype**

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to an extended software timer object of type OS_TIMER_EX.</td>
</tr>
</tbody>
</table>

**Return value**

Denotes whether the specified timer is running or not:

- \( = 0 \)  Timer has stopped.
- \( \neq 0 \)  Timer is running.
3.2.15 OS_TIMER_Restart()

Description
Restarts a software timer with its initial time value.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimer</td>
<td>Pointer to a software timer object of type OS_TIMER.</td>
</tr>
</tbody>
</table>

Additional information

OS_TIMER_Restart() restarts the timer using the initial time value programmed at creation of the timer or with the function OS_TIMER_SetPeriod().

OS_TIMER_Restart() 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.

Example

Please refer to the example for OS_TIMER_CREATE().
3.2.16  OS_TIMER_RestartEx()

Description

Restarts an extended software timer with its initial time value.

Prototype

void OS_TIMER_RestartEx(OS_TIMER_EX* pTimerEx);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to an extended software timer object of type OS_TIMER_EX.</td>
</tr>
</tbody>
</table>

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 of 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.

Example

Please refer to the example for OS_TIMER_CREATEEX().
3.2.17 OS_Timer_SetPeriod()

Description
Sets a new timer reload value for a software timer.

Prototype

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

Parameters

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

Additional information

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

Example

```c
static OS_TIMER TIMERPulse;

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

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

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

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to an extended software timer object of type OS_TIMER_EX.</td>
</tr>
</tbody>
</table>
| Period    | Initial period in embOS system ticks. The data type OS_TIME 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 retriggered 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.

Example

```c
static OS_TIMER_EX TIMERPulse ;
static OS_TASK TCB_HP ;

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

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

Description
Starts a software timer.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimer</td>
<td>Pointer to a software timer object of type OS_TIMER.</td>
</tr>
</tbody>
</table>

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.
3.2.20 OS_TIMER_StartEx()

Description
Starts an extended software timer.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to an extended software timer object of type OS_TIMER_EX.</td>
</tr>
</tbody>
</table>

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.
3.2.21 OS_TIMER_Stop()

Description
Stops a software timer.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimer</td>
<td>Pointer to a software timer object of type OS_TIMER.</td>
</tr>
</tbody>
</table>

Additional information

The actual value of the 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.
3.2.22 OS_TIMER_StopEx()

Description
Stops an extended software timer.

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to an extended software timer object of type OS_TIMER_EX.</td>
</tr>
</tbody>
</table>

Additional information

The actual time 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.
3.2.23 OS_TIMER_Trigger()

Description

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

Prototype

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimer</td>
<td>Pointer to a software timer object of type OS_TIMER.</td>
</tr>
</tbody>
</table>

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.

Example

```c
static OS_TIMER TIMERUartRx;

void TimerUart(void) {
    HandleUartRx();
}

void UartRxIntHandler(void) {
    OS_TIMER_Trigger(&TIMERUartRx); // Character received, stop the software timer
}

void UartSendNextCharacter(void) {
    OS_TIMER_Start(&TIMERUartRx);
    // Send next uart character and wait for Rx character
}

int main(void) {
    OS_TIMER_Create(&TIMERUartRx, TimerUart, 20);
}
```
3.2.24 OS_TIMER_TriggerEx()

Description

Ends an extended software timer at once and calls the timer callback function.

Prototype

```c
void OS_TIMER_TriggerEx (OS_TIMER_EX * pTimerEx);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTimerEx</td>
<td>Pointer to the OS_TIMER_EX data structure which contains the data of the extended software timer.</td>
</tr>
</tbody>
</table>

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.

Example

```c
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 4

Task Events
4.1 Introduction

Task events are another way of communicating between tasks. In contrast to semaphores and mailboxes, task events are messages to a single, specified recipient. 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 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.

Every task has an individual bit mask, which by default is the width of an unsigned integer, usually the word size of the target processor. This means that 32 or 8 different events can be signaled to and distinguished by every task. By calling \texttt{OS_TASKEVENT\_GetBlocked()}, a task waits for one of the events specified as a bitmask. As soon as one of the events occurs, this task must be signaled by calling \texttt{OS_TASKEVENT\_Set()}. The waiting task will then be put in the \texttt{READY} state immediately. It will be activated according to the rules of the scheduler as soon as it becomes the task with the highest priority of all tasks in the \texttt{READY} state.

By changing the definition of \texttt{OS\_TASKEVENT}, which is defined as unsigned long on 32-bit CPUs and unsigned char on 16 or 8-bit CPUs per default, the task events can be expanded to 16 or 32 bits thus allowing more individual events, or reduced to smaller data types on 32-bit CPUs.

Changing the definition of \texttt{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.

Example

```c
#include "RTOS.h"

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

static void HPTask(void) {
    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;
}
```
## 4.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>main</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TASKEVENT_Clear()</td>
<td>Returns the actual state of events and then clears all events of a specified task.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TASKEVENT_ClearEx()</td>
<td>Returns the actual state of events and then clears the specified events for the specified task.</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_TASKEVENT_Get()</td>
<td>Returns a list of events that have occurred for a specified task.</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASKEVENT_GetBlocked()</td>
<td>Waits for one of the events specified in the bitmask and clears the event memory when the function returns.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASKEVENT_GetSingleBlocked()</td>
<td>Waits for one of the specified events and clears only those events that were specified in the event mask.</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASKEVENT_GetSingleTimed()</td>
<td>Waits for one of the specified events for a given time and clears only those events that were specified in the event mask.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASKEVENT_GetTimed()</td>
<td>Waits for the specified events for a given time, and clears the event memory when the function returns.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TASKEVENT_Set()</td>
<td>Signals event(s) to a specified task.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.2.1 OS_TASKEVENT_Clear()

Description
Returns the actual state of events and then clears all events of a specified task.

Prototype

```
OS_TASKEVENT OS_TASKEVENT_Clear(OS_TASK * pTask);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type <code>OS_TASK</code>. The task whose event mask is to be returned, <code>NULL</code> means current task.</td>
</tr>
</tbody>
</table>

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. However, `NULL` must not be passed for `pTask` from `main()`, a timer callback or from an interrupt handler. A debug build of embOS will call `OS_Error()` in case `pTask` does not indicate a valid task.

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);
    }
}
```
4.2.2 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK. The task whose event mask is to be returned, NULL means current task.</td>
</tr>
<tr>
<td>EventMask</td>
<td>The bit mask containing the event bits which shall be cleared.</td>
</tr>
</tbody>
</table>

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. However, NULL must not be passed for pTask from main(), a timer callback or from an interrupt handler. A debug build of embOS will call OS_Error() in case pTask does not indicate a valid task.

Example

```c
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);
    }
}
```
4.2.3 OS_TASKEVENT_Get()

Description

Returns a list of events that have occurred for a specified task.

Prototype

OS_TASKEVENT OS_TASKEVENT_Get (OS_CONST_PTR OS_TASK * pTask);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK. The task whose event mask is to be returned, NULL means current task.</td>
</tr>
</tbody>
</table>

Return value

All events that have been signaled.

Additional information

If NULL is passed for pTask, the currently running task is used. However, NULL must not be passed for pTask from main(), a timer callback or from an interrupt handler. A debug build of embOS will call OS_Error() in case pTask does not indicate a valid task.

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 pTask is NULL, the function clears the events of the currently running task.

```c
void PrintEvents (void)  {
    OS_TASKEVENT MyEvents ;

    MyEvents = OS_TASKEVENT_Get (NULL);
    printf("Events %u\n", MyEvents);
}
```
4.2.4 OS_TASKEVENT_GetBlocked()

Description
Waits for one of the events specified in the bitmask and clears the event memory when
the function returns.

Prototype
OS_TASKEVENT OS_TASKEVENT_GetBlocked(OS_TASKEVENT EventMask);

Parameters
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventMask</td>
<td>The event bit mask containing the event bits, which shall be waited for.</td>
</tr>
</tbody>
</table>

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 event mask. The calling function must handle the
returned value, otherwise events may get lost. Consecutive calls of OS_TASKEVENT_Get-
Blocked() 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.

Example

```c
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().
4.2.5 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventMask</td>
<td>The event bit mask containing the event bits, which shall be waited for and reset.</td>
</tr>
</tbody>
</table>

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 re-
quests. 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 func-
tion 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.

Example

```c
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();
        }
    }
}
```
4.2.6 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventMask</td>
<td>The event bit mask containing the event bits, which shall be waited for and reset.</td>
</tr>
</tbody>
</table>
| Timeout   | Maximum time in embOS system ticks until the events must be signaled. The data type OS_TIME is defined as an integer, therefore valid values are:  
1 ≤ Timeout ≤ 2^{15} - 1 = 0xFFFF for 8/16-bit CPUs. 
1 ≤ Timeout ≤ 2^{31} - 1 = 0x7FFFFFFF for 32-bit CPUs. |

Return value
All requested events that have been signaled and were specified in the EventMask.

Additional information
If none of the specified events 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. If, subsequently, one or more specified event(s) is signaled before the task continues execution, the function will return the resulting event mask. Otherwise it will return zero.

Any bit in the event mask may enable the corresponding event. All unmasked events remain unchanged.

Example

```
void Task(void) {
    OS_TASKEVENT MyEvents;

    while(1) {
        // // Wait for event 0 and 1 to be signaled for 10 system ticks //
        MyEvents = OS_TASKEVENT_GetSingleTimed(3, 10);
        if (MyEvents == 0) {
            _HandleTimeout();
        } else {
            if (MyEvents & 1) {
                _HandleEvent0();
            }
            if (MyEvents & 2) {
                _HandleEvent1();
            }
        }
    }
}
```
4.2.7 OS_TASKEVENT_GetTimed()

Description
Waits for the specified events for a given time, and clears the event memory when the function returns.

Prototype

OS_TASKEVENT OS_TASKEVENT_GetTimed(OS_TASKEVENT EventMask,
OS_TIME Timeout);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventMask</td>
<td>The event bit mask containing the event bits, which shall be waited for.</td>
</tr>
</tbody>
</table>
| Timeout   | Maximum time in embOS system ticks waiting for events to be signaled. The data type OS_TIME is defined as an integer, therefore valid values are:  
1 ≤ Timeout ≤ 2\(^{15} - 1 = 0x7FFF\) for 8/16-bit CPUs.  
1 ≤ Timeout ≤ 2\(^{31} - 1 = 0x7FFFFFFF\) for 32-bit CPUs. |

Return value
All events that have been signaled.

Additional information
If none of the specified events are available, the task is suspended for the given time. The first of the requested 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. If, subsequently, one or more event(s) is signaled before the task continues execution, all signaled events are returned and then cleared. Otherwise the return value will be 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.

Example

```c
void Task(void) {
    OS_TASKEVENT MyEvents;

    while(1) {
        // Wait for event 0 and 1 to be signaled for 10 system ticks
        MyEvents = OS_TASKEVENT_GetTimed(3, 10);
        if ((MyEvents & 3) == 0) {
            _HandleTimeout();
        } else {
            if (MyEvents & 1) {
                _HandleEvent0();
            }
            if (MyEvents & 2) {
                _HandleEvent1();
            }
        }
    }
}
```
4.2.8 OS_TASKEVENT_Set()

Description
Signals event(s) to a specified task.

Prototype

```c
void OS_TASKEVENT_Set(OS_TASK* pTask,
                        OS_TASKEVENT Event);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block of type OS_TASK.</td>
</tr>
<tr>
<td>Event</td>
<td>The event bit mask containing the event bits, which shall be signaled.</td>
</tr>
</tbody>
</table>

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.

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):

```c
#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 5

Event Objects
5.1 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.

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.
5.1.1 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.

```c
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.

```c
#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
```
```c
//
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, "HWTask", 25, HWTask, StackHW);
    OS_EVENT_Create(&HW_Event);
    OS_Start(); // Start multitasking
    return 0;
}
```
## 5.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_EVENT_Create()</td>
<td>Creates an event object and resets the event.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_EVENT_CreateEx()</td>
<td>Creates an extended event object and sets its reset behavior as well as mask bits behavior.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_EVENT_Delete()</td>
<td>Deletes an event object and releases all waiting tasks.</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_EVENT_Get()</td>
<td>Retrieves current state of an event object without modification or suspension.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_EVENT_GetBlocked()</td>
<td>Waits for an event object and suspends task if event is not signaled.</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_EVENT_GetMask()</td>
<td>Returns the bits of an event object that match the given EventMask.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_EVENT_GetMaskBlocked()</td>
<td>Waits for the specified event bits, depending on the current mask mode.</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_EVENT_GetMaskMode()</td>
<td>Retrieves the current mask mode (mask bits behavior) of an event object.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_EVENT_GetMaskTimed()</td>
<td>Waits for the specified event bits with timeout, depending on the current mask mode.</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_EVENT_GetResetMode()</td>
<td>Returns the reset mode (reset behavior) of an event object.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_EVENT_GetTimed()</td>
<td>Waits for an event and suspends the task for a specified time or until the event has been signaled.</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_EVENT_Pulse()</td>
<td>Signals an event object and resumes waiting tasks, then resets the event object to non-signal state.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_EVENT_Reset()</td>
<td>Resets the specified event object to non-signaled state.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_EVENT_Set()</td>
<td>Sets an event object to signaled state, or resumes tasks which are waiting at the event object.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_EVENT_SetMask()</td>
<td>Sets the event mask bits of an event object.</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_EVENT_SetMaskMode()</td>
<td>Sets the mask mode of an event object to OR/AND logic.</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_EVENT_SetResetMode()</td>
<td>Sets the reset behavior of an event object to auto-matic, manual or semiauto.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
CHAPTER 5
API functions

5.2.1 OS_EVENT_Create()

Description
Creates an event object and resets the event. Must be called before the event object can be used.

Prototype

```c
void OS_EVENT_Create(OS_EVENT* pEvent);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
</tbody>
</table>

Additional information

Before the event object can be used, it must be created by a call of OS_EVENT_Create(). On creation, the event is set in non-signaled state, and the list of waiting tasks is empty. Therefore, OS_EVENT_Create() must not be called for an event object which is already created.

A debug build of embOS cannot check whether the specified event object was already created.

The event is created with the default reset behavior which is semiauto. Since version 3.88a of embOS, the reset behavior of the event can be modified by a call of the function OS_EVENT_SetResetMode().

Example

```c
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;
}
```
5.2.2 OS_EVENT_CreateEx()

Description

Creates an extended event object and sets its reset behavior as well as mask bits behavior.

Prototype

```c
void OS_EVENT_CreateEx(OS_EVENT * pEvent,
                        unsigned int Mode);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
<tr>
<td>Mode</td>
<td>Specifies the reset and mask bits behavior of the event object. You can use one of the predefined reset modes: OS_EVENT_RESET_MODE_SEMIAUTO, OS_EVENT_RESET_MODE_MANUAL, OS_EVENT_RESET_MODE_AUTO and one of the mask modes: OS_EVENT_MASK_MODE_OR_LOGIC, OS_EVENT_MASK_MODE_AND_LOGIC which are described under additional information.</td>
</tr>
</tbody>
</table>

Additional information

Before the event object can be used, it must be created by a call of OS_EVENT_Create() or OS_EVENT_CreateEx(). On creation, the event is set in nonsignaled state, and the list of waiting tasks is empty. Therefore, OS_EVENT_CreateEx() must not be called for an event object which is already created. A debug build of embOS cannot check whether the specified event object was already created.

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.
Example

```c
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); // Does not resume HPTask
    OS_EVENT_SetMask(&_Event, 2); // Resume HPTask since both bits are now set
}

int main(void) {
    ...
    OS_EVENT_CreateEx(&_Event, OS_EVENT_RESET_MODE_AUTO |
                      OS_EVENT_MASK_MODE_AND_LOGIC);
    ...
    return 0;
}
```
5.2.3 OS_EVENT_Delete()

Description
Deletes an event object and releases all waiting tasks.

Prototype

```c
void OS_EVENT_Delete(OS_EVENT* pEvent);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
</tbody>
</table>

Additional information

To keep the system fully dynamic, it is essential that event objects can be created dynamically. This also means there must be a way to delete an event object when it is no longer needed. The memory that has been used by the event object’s control structure can then be reused or reallocated.

It is your responsibility to make sure that:
- the program no longer uses the event object to be deleted
- the event object to be deleted actually exists (has been created first)
- no tasks are waiting at the event object when it is deleted.

`pEvent` must address an existing event object, which has been created before by a call of OS_EVENT_Create() or OS_EVENT_CreateEx(). A debug build of embOS will check whether `pEvent` addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error. If any task is waiting at the event object which is deleted, a debug build of embOS calls OS_Error() with error code OS_ERR_EVENT_DELETE.

To avoid any problems, an event object should not be deleted in a normal application.

Example

```c
static OS_EVENT _Event;

void Task(void) {
    ...
    OS_EVENT_Delete(&_Event);
    ...
}
```
5.2.4 OS_EVENT_Get()

**Description**
Retrieves current state of an event object without modification or suspension.

**Prototype**
OS_BOOL OS_EVENT_Get(OS_CONST_PTR OS_EVENT *pEvent);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
</tbody>
</table>

**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. pEvent must address an existing event object, which has been created before by a call of OS_EVENT_Create().
A debug build of embOS will check whether pEvent addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error.

**Example**

```c
static OS_EVENT _Event;

void Task(void) {
    OS_BOOL Status;
    Status = OS_EVENT_Get(&_Event);
    printf("Event Object Status: %d\n", Status);
}
```
5.2.5 OS_EVENT_GetBlocked()

Description
Waits for an event object and suspends task if event is not signaled.

Prototype

```c
void OS_EVENT_GetBlocked(OS_EVENT* pEvent);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
</tbody>
</table>

Additional information

*pEvent* addresses an existing event object, which must be created before the call of OS_EVENT_GetBlocked(). A debug build of embOS will check whether *pEvent* addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error.

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.

Example

```c
static OS_EVENT _Event;

void HPTask(void) {
    OS_EVENT_GetBlocked(&_Event); // Suspends the task
}

void LPTask(void) {
    OS_EVENT_Pulse(&_Event); // Signals the HPTask
}
```
5.2.6 OS_EVENT_GetMask()

Description
Returns the bits of an event object that match the given EventMask.

Prototype

OS_TASKEVENT OS_EVENT_GetMask(OS_EVENT* pEvent,
                                OS_TASKEVENT EventMask);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
<tr>
<td>EventMask</td>
<td>The bit mask containing the event bits which shall be retrieved.</td>
</tr>
</tbody>
</table>

Return value
Matching event object mask bits.

Additional information
The signaled event mask bits are consumed unless OS_EVENT_RESET_MODE_MANUAL is selected. pEvent must address an existing event object, which has been created before by a call of OS_EVENT_Create().
A debug build of embOS will check whether pEvent addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error.

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);
}
```
5.2.7 OS_EVENT_GetMaskBlocked()

Description

Waits for the specified event bits, depending on the current mask mode. The signaled event mask bits are consumed unless OS_EVENT_RESET_MODE_MANUAL is selected.

Prototype

OS_TASKEVENT OS_EVENT_GetMaskBlocked(OS_EVENT * pEvent, OS_TASKEVENT EventMask);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
<tr>
<td>EventMask</td>
<td>The event bit mask containing the event bits, which shall be waited for.</td>
</tr>
</tbody>
</table>

Return value

All requested events that have been signaled and were specified in the EventMask.

Additional information

pEvent addresses an existing event object, which must be created before the call of OS_EVENT_GetMaskBlocked(). A debug build of embOS will check whether pEvent addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error.

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().

Example

```c
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);
    ...
}
```
5.2.8 OS_EVENT_GetMaskMode()

Description
Retrieves the current mask mode (mask bits behavior) of an event object.

Prototype
OS_EVENT_MASK_MODE OS_EVENT_GetMaskMode(OS_CONST_PTR OS_EVENT *pEvent);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
</tbody>
</table>

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

pEvent must address an existing event object, which has been created before by a call of
OS_EVENT_Create() or OS_EVENT_CreateEx().

A debug build of embOS will check whether pEvent addresses a valid event object and will
call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error. 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().

Example

```c
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");
    }
}
```
5.2.9 OS_EVENT_GetMaskTimed()

**Description**

Waits for the specified event bits with timeout, depending on the current mask mode. The task is suspended for the specified time or until the event(s) have been signaled. The signaled event mask bits are consumed unless OS_EVENT_RESET_MODE_MANUAL is selected.

**Prototype**

```c
OS_TASKEVENT OS_EVENT_GetMaskTimed(OS_EVENT * pEvent,
                                  OS_TASKEVENT EventMask,
                                  OS_TIME      Timeout);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
<tr>
<td>EventMask</td>
<td>The event bit mask containing the event bits, which shall be waited for.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Maximum time in embOS system ticks until the event must be signaled. The data type OS_TIME is defined as an integer, therefore valid values are: 1 ≤ Timeout ≤ 215 - 1 = 0x7FFF for 8/16-bit CPUs. 1 ≤ Timeout ≤ 231 - 1 = 0xFFFFFFFF for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

**Return value**

Matching event object mask bits or 0 when a timeout occurred.

**Additional information**

pEvent addresses an existing event object, which must be created before the call of OS_EVENT_GetMaskTimed(). A debug build of embOS will check whether pEvent addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error.

**Example**

```c
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);
    ...
}
```
5.2.10 OS_EVENT_GetResetMode()

Description
Returns the reset mode (reset behavior) of an event object.

Prototype

```c
OS_EVENT_RESET_MODE OS_EVENT_GetResetMode(OS_CONST_PTR OS_EVENT *pEvent);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
</tbody>
</table>

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

pEvent must address an existing event object, which has been created before by a call of OS_EVENT_Create() or OS_EVENT_CreateEx().

A debug build of embOS will check whether pEvent addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error. 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().

Example

```c
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");
    }
}
```
5.2.11 OS_EVENT_GetTimed()

Description

Waits for an event and suspends the task for a specified time or until the event has been signaled. The event is consumed unless OS_EVENT_RESET_MODE_MANUAL is selected.

Prototype

```c
char OS_EVENT_GetTimed(OS_EVENT* pEvent, 
                       OS_TIME   Timeout);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
</tbody>
</table>
| Timeout   | Maximum time in embOS system ticks until the event must be signaled. The data type OS_TIME is defined as an integer, therefore valid values are:

\[
1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF \text{ for 8/16-bit CPUs.} \\
1 \leq \text{Timeout} \leq 2^{31} - 1 = 0x7FFFFFFF \text{ for 32-bit CPUs.}
\]

Return value

- \(= 0\) Success, the event was signaled within the specified time.
- \(\neq 0\) If the event was not signaled within the specified time.

Additional information

pEvent addresses an existing event object, which must be created before the call of OS_EVENT_GetTimed(). A debug build of embOS will check whether pEvent addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error.

Example

```c
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
    }
    ...
}
```
5.2.12 OS_EVENT_Pulse()

Description
Signals an event object and resumes waiting tasks, then resets the event object to non-signaled state.

Prototype

```c
void OS_EVENT_Pulse(OS_EVENT* pEvent);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
</tbody>
</table>

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.

A debug build of embOS will check whether `pEvent` addresses a valid event object and will call `OS_Error()` with the error code `OS_ERR_EVENT_INVALID` in case of an error.

Example

```c
static OS_EVENT _Event;

void HPTask(void) {
    OS_EVENT_GetBlocked(&_Event); // Suspends the task
}

void LPTask(void) {
    OS_EVENT_Pulse(&_Event); // Signales the HPTask
}
```
5.2.13 OS_EVENT_Reset()

Description
Resets the specified event object to non-signaled state.

Prototype

```c
void OS_EVENT_Reset(OS_EVENT* pEvent);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
</tbody>
</table>

Additional information

`pEvent` must address an existing event object, which has been created before by a call of `OS_EVENT_Create()`. A debug build of embOS will check whether `pEvent` addresses a valid event object and will call `OS_Error()` with the error code `OS_ERR_EVENT_INVALID` in case of an error.

Example

```c
static OS_EVENT _Event;

void Task(void) {
    ...
    OS_EVENT_Reset(&_Event);
    ...
}
```
5.2.14 OS_EVENT_Set()

Description
Sets an event object to signaled state, or resumes tasks which are waiting at the event object.

Prototype

```c
void OS_EVENT_Set(OS_EVENT* pEvent);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type <code>OS_EVENT</code>.</td>
</tr>
</tbody>
</table>

Additional information

pEvent must address an existing event object, which must be created before by a call to `OS_EVENT_Create()`. A debug build of embOS will check whether pEvent addresses a valid event object and will call `OS_Error()` with error code `OS_ERR_EVENT_INVALID` in case of an error.

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.

Example

Examples on how to use the `OS_EVENT_Set()` function are shown in *Examples of using event objects* on page .
5.2.15 OS_EVENT_SetMask()

Description
Sets the event mask bits of an event object.

Prototype

```c
void OS_EVENT_SetMask ( OS_EVENT * pEvent ,
                        OS_TASKEVENT EventMask );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
<tr>
<td>EventMask</td>
<td>The event bit mask containing the event bits, which shall be signaled.</td>
</tr>
</tbody>
</table>

Additional information

pEvent must address an existing event object, which must be created before by a call to OS_EVENT_Create(). A debug build of embOS will check whether pEvent addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error.

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.

Example

```c
static OS_EVENT _Event;

void Task ( void ) {
    OS_TASKEVENT EventMask;

    ...
    EventMask = 1 << ((sizeof(OS_TASKEVENT) * 8) - 1); // Set MSB event bit
    OS_EVENT_SetMask (&_Event, EventMask); // Signal MSB event bit
    ...
}
```
5.2.16 OS_EVENT_SetMaskMode()

Description
Sets the mask mode of an event object to OR/AND logic.

Prototype

```c
void OS_EVENT_SetMaskMode(OS_EVENT* pEvent,
                          OS_EVENT_MASK_MODE MaskMode);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
<tr>
<td>MaskMode</td>
<td>Event Mask mode. Modes are defined in enum OS_EVENT_MASK_MODE.</td>
</tr>
<tr>
<td></td>
<td>OS_EVENT_MASK_MODE_OR_LOGIC (0x00u): Mask bits are used with OR logic (default).</td>
</tr>
<tr>
<td></td>
<td>OS_EVENT_MASK_MODE_AND_LOGIC (0x04u): Mask bits are used with AND logic.</td>
</tr>
</tbody>
</table>

Additional information

pEvent must address an existing event object, which has been created before by a call of OS_EVENT_Create() or OS_EVENT_CreateEx().

A debug build of embOS will check whether pEvent addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error.

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.

Example

```c
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);
    ...
}
```
5.2.17 OS_EVENT_SetResetMode()

Description
Sets the reset behavior of an event object to automatic, manual or semiauto.

Prototype

```c
void OS_EVENT_SetResetMode(OS_EVENT * pEvent,
                           OS_EVENT_RESET_MODE ResetMode);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEvent</td>
<td>Pointer to an event object of type OS_EVENT.</td>
</tr>
<tr>
<td>ResetMode</td>
<td>Controls the reset mode of the event object.</td>
</tr>
<tr>
<td>OS_EVENT_RESET_DEFAULT (0x00u): As previous mode.</td>
<td></td>
</tr>
<tr>
<td>OS_EVENT_RESET_MANUAL (0x01u): Event remains set, has to be reset by task.</td>
<td></td>
</tr>
<tr>
<td>OS_EVENT_RESET_AUTO (0x02u): Event is reset automatically.</td>
<td></td>
</tr>
</tbody>
</table>

Additional information

pEvent must address an existing event object, which has been created before by a call of OS_EVENT_Create() or OS_EVENT_CreateEx().

A debug build of embOS will check whether pEvent addresses a valid event object and will call OS_Error() with error code OS_ERR_EVENT_INVALID in case of an error.

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.

Example

```c
static OS_EVENT _Event;

void Task(void) {
    // Set the reset mode for the event object to manual
    OS_EVENT_SetResetMode(&_Event, OS_EVENT_RESET_MANUAL);
}
```
Chapter 6

Mutexes
6.1 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 prom 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:

```
OS_MUTEX_LockBlocked() -> Access resource -> OS_MUTEX_Unlock()
```

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:

```c
#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;
}
```
## 6.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_MUTEX_Create()</td>
<td>Creates a mutex.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MUTEX_Delete()</td>
<td>Deletes a specified mutex.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MUTEX_GetOwner()</td>
<td>Returns the mutex owner if any.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MUTEX_GetValue()</td>
<td>Returns the value of the usage counter of a specified mutex.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MUTEX_IsMutex()</td>
<td>Returns whether a mutex has already been created.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MUTEX_Lock()</td>
<td>Requests a specified mutex and blocks it for other tasks if it is available.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MUTEX_LockBlocked()</td>
<td>Claims a mutex and blocks it for other tasks.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MUTEX_LockTimed()</td>
<td>Tries to claim a mutex and blocks it for other tasks if it is available within a specified time.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MUTEX_Unlock()</td>
<td>Releases a mutex currently in use by a task.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
6.2.1 OS_MUTEX_Create()

Description
Creates a mutex.

Prototype

```c
void OS_MUTEX_Create(OS_MUTEX* pMutex);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMutex</td>
<td>Pointer to a mutex object of type OS_MUTEX.</td>
</tr>
</tbody>
</table>

Additional information

After creation, the mutex is not blocked; the value of the counter is zero.

Example

```c
static OS_MUTEX _Mutex;

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

Description
Deletes a specified mutex. The memory of that mutex may be reused for other purposes or may be used for creating another mutex using the same memory.

Prototype

```c
void OS_MUTEX_Delete(OS_MUTEX* pMutex);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMutex</td>
<td>Pointer to a mutex object of type OS_MUTEX.</td>
</tr>
</tbody>
</table>

Additional information

Before deleting a mutex, make sure that no task is claiming the mutex. A debug build of embOS will call `OS_Error()` with the error code `OS_ERR_MUTEX_DELETE` if a mutex is deleted when it is already in use. In systems with dynamic creation of mutexes, you must delete a mutex before recreating it. Failure to do so may cause mutex handling to work incorrectly.

Example

```c
static OS_MUTEX _Mutex;

int Task(void) {
    ...
    OS_MUTEX_Delete(&_Mutex);
    ...
    return 0;
}
```
6.2.3 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMutex</td>
<td>Pointer to a mutex object of type OS_MUTEX.</td>
</tr>
</tbody>
</table>

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().

Example
Please find an example at OS_MUTEX_GetValue().
6.2.4 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMutex</td>
<td>Pointer to a mutex object of type OS_MUTEX.</td>
</tr>
</tbody>
</table>

Return value
The counter value of the mutex.
A value of zero means the mutex is available.

Example

```c
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);
        }
    }
}
```
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMutex</td>
<td>Pointer to a mutex object of type OS_MUTEX.</td>
</tr>
</tbody>
</table>

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().

Example

```c
static OS_MUTEX _Mutex;

int main(void) {
  ...
  if (OS_MUTEX_IsMutex(&_Mutex) == (OS_BOOL)1) {
    printf("Mutex has already been created");
  } else {
    printf("Mutex has not yet been created");
  }
  ...
  return 0;
}
```
6.2.6 OS_MUTEX_Lock()

**Description**
Requests a specified mutex and blocks it for other tasks if it is available. Continues execution in any case.

**Prototype**

```c
char OS_MUTEX_Lock(OS_MUTEX* pMutex);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMutex</td>
<td>Pointer to a mutex object of type OS_MUTEX.</td>
</tr>
</tbody>
</table>

**Return value**

- = 0 Mutex was not available.
- ≠ 0 Mutex was available, now in use by calling task.

**Additional information**

The following diagram illustrates how OS_MUTEX_Lock() works:

![Diagram of OS_MUTEX_Lock()](image)

**Example**

```c
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
}
```
6.2.7 OS_MUTEX_LockBlocked()

Description
Claims a mutex and blocks it for other tasks.

Prototype

```c
int OS_MUTEX_LockBlocked(OS_MUTEX* pMutex);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMutex</td>
<td>Pointer to a mutex object of type OS_MUTEX.</td>
</tr>
</tbody>
</table>

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.

Example

```c
static OS_MUTEX _Mutex;

void Task(void) {
    ...
    OS_MUTEX_LockBlocked(&_Mutex);
    ...
    OS_MUTEX_Unlock(&_Mutex);
    ...
}
```

The following diagram illustrates how OS_MUTEX_LockBlocked() works:
CHAPTER 6  

API functions

6.2.8   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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMutex</td>
<td>Pointer to a mutex object of type OS_MUTEX.</td>
</tr>
</tbody>
</table>
| Timeout    | Maximum time until the mutex should be available. Timer period in embOS system ticks. The data type OS_TIME is defined as an integer, therefore valid values are:  
            | 1 ≤ Timeout ≤ 2^{15} - 1 = 0x7FFF for 8/16-bit CPUs. |
            | 1 ≤ Timeout ≤ 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.
**Example**

```c
static OS_MUTEX _Mutex;

void Task(void) {
    ...
    if (OS_MUTEX_LockTimed(&_Mutex, 100)) {
        ... // Mutex acquired
    } else {
        ... // Timeout
    }
    ...
}
```
6.2.9 OS_MUTEX_Unlock()

Description

Releases a mutex currently in use by a task.

Prototype

```c
void OS_MUTEX_Unlock(OS_MUTEX * pMutex);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMutex</td>
<td>Pointer to a mutex object of type OS_MUTEX.</td>
</tr>
</tbody>
</table>

Additional information

OS_MUTEX_Unlock() may be used on a mutex only after that mutex has been used by calling OS_MUTEX_LockBlocked() or OS_MUTEX_Lock(). OS_MUTEX_Unlock() decrements the usage counter of the mutex which must never become negative. If this counter becomes negative, a debug build will call the embOS error handler OS_Error() with error code OS_ERR_UNUSE_BEFORE_USE. In a debug build OS_Error() will also be called if OS_MUTEX_Unlock() is called from a task which does not own the mutex. The error code is OS_ERR_RESOURCE_OWNER in this case.

Example

Please find an example at OS_MUTEX_Lock().
Chapter 7

Semaphores
7.1 Introduction

A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a multitasking operating system. While not as widely used as mutexes, events or mailboxes, semaphores can be very useful in specific situations. For example, they are commonly used in "credittracking synchronization" where a task needs to wait for something that can be signaled one or more times.

Example of using semaphores

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:

```c
#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_TICK_HOOK Hook; // Hook to emulate external interrupt

void Task(void) {
    while(1) {
        OS_SEMAPHORE_TakeBlocked(&Sema); // Wait for signaling of received data
        printf("Task is processing data"); // Act on received data
    }
}

void OnTickHookFunction(void) {
    OS_SEMAPHORE_Give(&Sema); // Signal data reception
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_InitHW(); // Initialize required hardware
    // Register tick hook function to emulate an external interrupt
    //
    OS_TICK_AddHook(&Hook, (OS_TICK_HOOK_ROUTINE*)OnTickHookFunction);
    OS_TASK_CREATE(&TCB, "Task", 100, Task, Stack);
    OS_SEMAPHORE_CREATE(&Sema); // Creates semaphore
    OS_Start(); // Start embOS
    return 0;
}
```
## 7.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Unpriv Task</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_SEMAPHORE_CREATE()</td>
<td>Macro that creates a semaphore with an initial count value of zero.</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_SEMAPHORE_Create()</td>
<td>Creates a counting semaphore with a specified initial count value.</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_SEMAPHORE_Delete()</td>
<td>Deletes a counting semaphore.</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_SEMAPHORE_GetValue()</td>
<td>Returns the current counter value of a specified semaphore.</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>OS_SEMAPHORE_Give()</td>
<td>Increments the counter of a semaphore.</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>OS_SEMAPHORE_GiveMax()</td>
<td>Increments the counter of a semaphore up to a specified maximum value.</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>OS_SEMAPHORE_SetValue()</td>
<td>Sets the counter value of a specified semaphore.</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_SEMAPHORE_Take()</td>
<td>Decrements the counter of a semaphore, if it was signaled.</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>OS_SEMAPHORE_TakeBlocked()</td>
<td>Decrements the counter of a semaphore.</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_SEMAPHORE_TakeTimed()</td>
<td>Decrements a semaphore counter if the semaphore is available within a specified time.</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
7.2.1 OS_SEMAPHORE_CREATE()

Description
Macro that creates a semaphore with an initial count value of zero.

Prototype

```c
void OS_SEMAPHORE_CREATE ( OS_SEMAPHORE * pSema );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSema</td>
<td>Pointer to a data structure of type OS_SEMAPHORE.</td>
</tr>
</tbody>
</table>

Additional information
To create a semaphore a data structure of the type OS_SEMAPHORE must be defined in memory and initialized using OS_SEMAPHORE_CREATE(). The value of a semaphore created through this macro is zero. If you need to create a semaphore with an arbitrary initial counting value, use the function OS_SEMAPHORE_Create().

Example
Please refer to the example in Introduction on page 173.
### 7.2.2 OS_SEMAPHORE_Create()

**Description**

Creates a counting semaphore with a specified initial count value.

**Prototype**

```c
void OS_SEMAPHORE_Create(OS_SEMAPHORE *pSema,
                          OS_UINT InitValue);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSema</td>
<td>Pointer to a semaphore object of type OS_SEMAPHORE.</td>
</tr>
<tr>
<td>InitValue</td>
<td>Initial count value of the semaphore:</td>
</tr>
<tr>
<td></td>
<td>(0 \leq \text{InitValue} \leq 2^{16} - 1 = 0xFFFF) for 8/16-bit CPUs.</td>
</tr>
<tr>
<td></td>
<td>(0 \leq \text{InitValue} \leq 2^{32} - 1 = 0xFFFFFFFF) for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

**Additional information**

To create a counting semaphore a data structure of the type OS_SEMAPHORE must be defined in memory and initialized using OS_SEMAPHORE_Create().

**Example**

```c
static OS_SEMA _Sema;

int main(void) {
  ...
  OS_SEMAPHORE_Create(&_Sema, 8);
  ...
  return 0;
}
```
7.2.3 OS_SEMAPHORE_Delete()

**Description**
Deletes a counting semaphore.

**Prototype**

```c
void OS_SEMAPHORE_Delete(OS_SEMAPHORE* pSema);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSema</td>
<td>Pointer to a semaphore object of type OS_SEMAPHORE.</td>
</tr>
</tbody>
</table>

**Additional information**

Before deleting a semaphore, make sure that no task is waiting for it and that no task will signal that semaphore at a later point.
A debug build of embOS will reflect an error if a deleted semaphore is signaled.

**Example**

```c
static OS_SEMA _Sema;

void Task(void) {
    ...
    OS_SEMAPHORE_Delete(&_Sema);
    ...
}
```
7.2.4 OS_SEMAPHORE_GetValue()

Description
Returns the current counter value of a specified semaphore.

Prototype

```c
int OS_SEMAPHORE_GetValue(OS_CONST_PTR OS_SEMAPHORE *pSema);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSema</td>
<td>Pointer to a semaphore object of type OS_SEMAPHORE.</td>
</tr>
</tbody>
</table>

Return value
The current counter value of the semaphore.

Example

```c
static OS_SEMA _Sema;

void PrintSemaValue(void) {
    int Value;
    Value = OS_SEMAPHORE_GetValue(&_Sema);
    printf("Sema Value: %d\n", Value)
}
```
### 7.2.5 OS_SEMAPHORE_SetValue()

**Description**

Sets the counter value of a specified semaphore.

**Prototype**

```c
OS_U8 OS_SEMAPHORE_SetValue(OS_SEMAPHORE* pSema,
                             OS_UINT Value);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSema</td>
<td>Pointer to a semaphore object of type OS_SEMAPHORE.</td>
</tr>
<tr>
<td>Value</td>
<td>Count value of the semaphore:</td>
</tr>
<tr>
<td></td>
<td>0 ≤ Value ≤ 2^{16} - 1 = 0xFFFF for 8/16-bit CPUs.</td>
</tr>
<tr>
<td></td>
<td>0 ≤ Value ≤ 2^{32} - 1 = 0xFFFFFFFF for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

**Return value**

= 0: In any case. The return value can safely be ignored.

**Example**

```c
static OS_SEMA _Sema;

void Task(void) {
    ...
    OS_SEMAPHORE_SetValue(&_Sema, 0);
    ...
}
```
7.2.6  OS_SEMAPHORE_Give()

Description
Increments the counter of a semaphore.

Prototype

```c
void OS_SEMAPHORE_Give(OS_SEMAPHORE* pSema);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSema</td>
<td>Pointer to a semaphore object of type OS_SEMAPHORE.</td>
</tr>
</tbody>
</table>

Additional information

OS_SEMAPHORE_Give() signals an event to a semaphore by incrementing its counter. If one or more tasks are waiting for an event to be signaled to this semaphore, the task with the highest priority becomes the running task. The 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. A debug build of embOS detects a counter overflow and calls OS_Error() with error code OS_ERR_SEMAPHORE_OVERFLOW if an overflow occurs.

Example

Please refer to the example in Introduction on page 173.
7.2.7 OS_SEMAPHORE_GiveMax()

Description
Increments the counter of a semaphore up to a specified maximum value.

Prototype

```c
void OS_SEMAPHORE_GiveMax(OS_SEMAPHORE * pSema,
                           OS_UINT       MaxValue);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSema</td>
<td>Pointer to a semaphore object of type OS_SEMAPHORE.</td>
</tr>
<tr>
<td>MaxValue</td>
<td>Count value of the semaphore:</td>
</tr>
<tr>
<td></td>
<td>1 ≤ MaxValue ≤ 2^{16} - 1 = 0xFFFF for 8/16-bit CPUs.</td>
</tr>
<tr>
<td></td>
<td>1 ≤ MaxValue ≤ 2^{32} - 1 = 0xFFFFFFFF for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

Additional information
As long as current value of the semaphore counter is below the specified maximum value, OS_SEMAPHORE_GiveMax() signals an event to a semaphore by incrementing its counter. If one or more tasks are waiting for an event to be signaled to this semaphore, the tasks are placed into the READY state and the task with the highest priority becomes the running task.

Calling OS_SEMAPHORE_GiveMax() with a MaxValue of 1 makes a counting semaphore behave like a mutex. Consider using a mutex instead.

Example

```c
static OS_SEMA _Sema;

void Task(void) {
    ...
    OS_SEMAPHORE_GiveMax(&_Sema, 8);
    ...
}
```
7.2.8 OS_SEMAPHORE_Take()

**Description**
Decrements the counter of a semaphore, if it was signaled.

**Prototype**

```c
OS_BOOL OS_SEMAPHORE_Take(OS_SEMAPHORE* pSema);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSema</td>
<td>Pointer to a semaphore object of type OS_SEMAPHORE.</td>
</tr>
</tbody>
</table>

**Return value**

- `= 0` Failed, semaphore was not signaled before the call.
- `≠ 0` Success, semaphore was available and counter was decremented once.

**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_Take()` does not wait and does not modify the semaphore counter.

**Example**

```c
static OS_SEMA _Sema;

void Task(void) {
    ...
    if (OS_SEMAPHORE_Take(&_Sema) == 1) {
        printf("Semaphore decremented successfully.\n");
    } else {
        printf("Semaphore not signaled.\n");
    }
    ...
}
```
7.2.9 OS_SEMAPHORE_TakeBlocked()

Description
Decrements the counter of a semaphore.

Prototype
void OS_SEMAPHORE_TakeBlocked(OS_SEMAPHORE* pSema);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSema</td>
<td>Pointer to a semaphore object of type OS_SEMAPHORE.</td>
</tr>
</tbody>
</table>

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.

Example
Please refer to the example in Introduction on page 173.
7.2.10 OS_SEMAPHORE_TakeTimed()

Description

Decrements a semaphore counter if the semaphore is available within a specified time.

Prototype

OS_BOOL OS_SEMAPHORE_TakeTimed(OS_SEMAPHORE* pSema,
OS_TIME Timeout);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSema</td>
<td>Pointer to a semaphore object of type OS_SEMAPHORE.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Maximum time until semaphore should be available. Timeout in embOS system ticks. The data type OS_TIME is defined as an integer, therefore valid values are: $1 \leq Timeout \leq 2^{15} - 1 = 0x7FFF$ for 8/16-bit CPUs. $1 \leq Timeout \leq 2^{31} - 1 = 0x7FFFFFFF$ for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

Return value

= 0 Failed, semaphore not available before timeout.
≠ 0 Success, semaphore was available and counter decremented.

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_TakeTimed() waits until the semaphore is signaled 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 was not signaled within the specified time the program execution continues, but returns a value of zero. 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.

When the calling task is blocked by higher priority tasks for a period longer than the timeout value, it may happen that the counting 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().

Example

```c
static OS_SEMA _Sema;

void Task(void) {
  ...
  if (OS_SEMAPHORE_TakeTimed(&_Sema, 100)) {
    ... // Semaphore acquired
  } else {
    ... // Timeout
  }
  ...
}
```
8.1 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 real-time operating system. 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.

<table>
<thead>
<tr>
<th>Number of messages on 8 or 16-bit CPUs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;= x &lt;= 2^15 - 1 = 0x7FFF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of messages on 32-bit CPUs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;= x &lt;= 2^31 - 1 = 0x7FFFFFFF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message size in bytes on 8 or 16-bit CPUs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;= x &lt;= 2^15 - 1 = 0x7FFF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message size in bytes on 32-bit CPUs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;= x &lt;= 2^15 - 1 = 0x7FFF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maximum buffer size in bytes for one mailbox on 8 or 16-bit CPUs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^16 = 0xFFFF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maximum buffer size in bytes for one mailbox on 32-bit CPUs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^32 = 0xFFFFFFFF</td>
</tr>
</tbody>
</table>

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.
8.1.1 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 \texttt{OS\_mailbox\_PutBlocked()}, \texttt{OS\_mailbox\_Put()}, \texttt{OS\_mailbox\_GetBlocked()}, and \texttt{OS\_mailbox\_Get()} can transfer messages of sizes between 1 and 32,767 bytes each.

Their single-byte equivalents \texttt{OS\_mailbox\_PutBlocked1()}, \texttt{OS\_mailbox\_Put1()}, \texttt{OS\_mailbox\_GetBlocked1()}, and \texttt{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 singlebyte versions if you transfer a lot of single-byte data via mailboxes.

The routines \texttt{OS\_mailbox\_PutBlocked1()}, \texttt{OS\_mailbox\_Put1()}, \texttt{OS\_mailbox\_GetBlocked1()}, and \texttt{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.
#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, "\nHello\0");
        OS_MAILBOX_PutBlocked(&MyMailbox, "\nWorld !\0");
    }
}

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;
}
## 8.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_MAILBOX_Clear()</td>
<td>Clears all messages in the specified mailbox.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_Create()</td>
<td>Creates a new mailbox.</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_Delete()</td>
<td>Deletes a specified mailbox.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_Get()</td>
<td>Retrieves a new message of a predefined size from a mailbox if a message is available.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_Get1()</td>
<td>Retrieves a new message of size 1 from a mailbox if a message is available.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_GetBlocked()</td>
<td>Retrieves a new message of a predefined size from a mailbox.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_GetBlocked1()</td>
<td>Retrieves a new message of size 1 from a mailbox.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_GetMessageCnt()</td>
<td>Returns the number of messages currently available in a specified mailbox.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_GetTimed()</td>
<td>Retrieves a new message of a predefined size from a mailbox if a message is available within a given time.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_GetTimed1()</td>
<td>Retrieves a new message of size 1 from a mailbox if a message is available within a given time.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_GetPtr()</td>
<td>Retrieves a pointer to a new message of a predefined size from a mailbox, if a message is available.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_GetPtrBlocked()</td>
<td>Retrieves a pointer to a new message of a predefined size from a mailbox.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_Peek()</td>
<td>Peeks a mail from a mailbox without removing the mail.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_Purge()</td>
<td>Deletes the last retrieved message in a mailbox.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_Put()</td>
<td>Stores a new message of a predefined size in a mailbox if the mailbox is able to accept one more message.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_Put1()</td>
<td>Stores a new message of size 1 in a mailbox if the mailbox is able to accept one more message.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_PutBlocked()</td>
<td>Stores a new message of a predefined size in a mailbox.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_PutBlocked1()</td>
<td>Stores a new message of size 1 in a mailbox.</td>
<td>● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_PutFront()</td>
<td>Stores a new message of a predefined size into a mailbox in front of all other</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_PutFront1()</td>
<td>Stores a new message of size 1 into a mailbox in front of all other messages if the mailbox is able to accept one more message.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_PutFront-Blocked()</td>
<td>Stores a new message of a predefined size at the beginning of a mailbox in front of all other messages.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_PutFront-Blocked1()</td>
<td>Stores a new message of size 1 at the beginning of a mailbox in front of all other messages.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_PutTimed()</td>
<td>Stores a new message of a predefined size in a mailbox if the mailbox is able to accept one more message within a given time.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_PutTimed1()</td>
<td>Stores a new message of size 1 in a mailbox if the mailbox is able to accept one more message within a given time.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_WaitBlocked()</td>
<td>Waits until a mail is available, but does not retrieve the message from the mailbox.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_MAILBOX_WaitTimed()</td>
<td>Waits until a mail is available or the timeout has expired, but does not retrieve the message from the mailbox.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8.2.1  OS_MAILBOX_Clear()

Description
Cleans all messages in the specified mailbox.

Prototype

```c
void OS_MAILBOX_Clear(OS_MAILBOX* pMB);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
</tbody>
</table>

Additional information

When the mailbox is in use, a debug build of embOS will call `OS_Error()` with error code `OS_ERR_MB_INUSE`.

`OS_MAILBOX_Clear()` may cause a task switch.

Example

```c
static OS_MAILBOX _MBKey;

void ClearKeyBuffer(void) {
    OS_MAILBOX_Clear(&_MBKey);
}
```
8.2.2  OS_MAILBOX_Create()

Description
Creates a new mailbox.

Prototype

```
void OS_MAILBOX_Create ( OS_MAILBOX * pMB ,
OS_U16 sizeofMsg ,
OS_UINT maxnofMsg ,
void * Buffer );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>sizeofMsg</td>
<td>Size of a message in bytes. Valid values are 1 ≤ sizeofMsg ≤ 32,767.</td>
</tr>
<tr>
<td>maxnofMsg</td>
<td>Maximum number of messages. Valid values are 1 ≤ maxnofMsg ≤ 32,767 on 8 or 16-bit CPUs, or 1 ≤ maxnofMsg ≤ 2,147,483,647 on 32-bit CPUs.</td>
</tr>
<tr>
<td>Buffer</td>
<td>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: sizeofMsg * maxnofMsg bytes. For 8/16-bit CPUs the total buffer size for one mailbox is limited to 65,536 Bytes.</td>
</tr>
</tbody>
</table>

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);
```
8.2.3  OS_MAILBOX_Delete()

**Description**

Deletes a specified mailbox.

**Prototype**

```c
void OS_MAILBOX_Delete ( OS_MAILBOX * pMB );
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
</tbody>
</table>

**Additional information**

To keep the system fully dynamic, it is essential that mailboxes can be created dynamically. This also means there must be a way to delete a mailbox when it is no longer needed. The memory that has been used by the mailbox for the control structure and the buffer can then be reused or reallocated.

It is the programmer’s responsibility to:

- make sure that the program no longer uses the mailbox to be deleted
- make sure that the mailbox to be deleted actually exists (i.e. has been created first).

When the mailbox is in use, a debug build of embOS will call `OS_Error()` with error code `OS_ERR_MB_INUSE`.

In a debug build `OS_Error()` will also be called if `OS_MAILBOX_Delete()` is called while tasks are waiting for new data from the mailbox. The error code in this case is `OS_ERR_MAILBOX_DELETE`.

**Example**

```c
static OS_MAILBOX _MBSerIn;

void Cleanup ( void ) {
    OS_MAILBOX_Delete ( & _MBSerIn );
}
```
8.2.4 OS_MAILBOX_Get()

Description
Retrieves a new message of a predefined size from a mailbox if a message is available.

Prototype

```c
char OS_MAILBOX_Get ( OS_MAILBOX * pMB,
                      void * pDest );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pDest</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 `pDest` 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.

Example

```c
#define MESSAGE_SIZE 4

static OS_MAILBOX _MBData;
static char _Buffer[MESSAGE_SIZE];

char GetData(void) {
    return OS_MAILBOX_Get(&_MBData, &_Buffer);
}
```
8.2.5 OS_MAILBOX_Get1()

Description
Retrieves a new message of size 1 from a mailbox if a message is available.

Prototype
char OS_MAILBOX_Get1(OS_MAILBOX* pMB,
                      char* pDest);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pDest</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 pDest 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 187 for differences between OS_MAILBOX_Get() and OS_MAILBOX_Get1().

Example

```c
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;
}
```

8.2.6 OS_MAILBOX_GetBlocked()

Description
Retrieves a new message of a predefined size from a mailbox.

Prototype

```c
void OS_MAILBOX_GetBlocked(OS_MAILBOX *pMB,
                           void* pDest);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pDest</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 ISR.

Example

```c
#define MESSAGE_SIZE 4

static OS_MAILBOX _MBData;
static char _Buffer[MESSAGE_SIZE];

char WaitData(void) {
    return OS_MAILBOX_GetBlocked(&_MBData, &_Buffer);
}
```
8.2.7  OS_MAILBOX_GetBlocked1()

Description
Retrieves a new message of size 1 from a mailbox.

Prototype

```c
void OS_MAILBOX_GetBlocked1(OS_MAILBOX* pMB,
                          char* pDest);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pDest</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 ISR.


Example

```c
static OS_MAILBOX _MBKey;

char WaitKey(void) {
    char c;
    OS_MAILBOX_GetBlocked1(&_MBKey, &c);
    return c;
}
```
8.2.8 OS_MAILBOX_GetMessageCnt()

Description
Returns the number of messages currently available in a specified mailbox.

Prototype
OS_UINT OS_MAILBOX_GetMessageCnt(OS_CONST_PTR OS_MAILBOX *pMB);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
</tbody>
</table>

Return value
The number of messages currently available in the mailbox.

Example

```c
static OS_MAILBOX _MBData;

void PrintAvailableMessages() {
    OS_UINT NumOfMsgs;

    NumOfMsgs = OS_MAILBOX_GetMessageCnt(&_MBData);
    printf("Mailbox contains %d messages.\n", NumOfMsgs);
}
```
8.2.9 OS_MAILBOX_GetTimed()

Description
 Retrieves a new message of a predefined size from a mailbox if a message is available within a given time.

Prototype

```c
char OS_MAILBOX_GetTimed(OS_MAILBOX* pMB,
void* pDest,
OS_TIME Timeout);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pDest</td>
<td>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.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Maximum time until the requested mail must be available. Timer period in embOS system ticks. The data type OS_TIME is defined as an integer, therefore valid values are: 1 ≤ Timeout ≤ 215 - 1 = 0xFFFF for 8/16-bit CPUs. 1 ≤ Timeout ≤ 231 - 1 = 0xFFFFFFFF for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

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 mail is available within the given timeout, or after the timeout value has expired. If the timeout has expired and no mail was available within the timeout the memory area where pDest 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 mail becomes available after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the mail was not available within the requested time. In this case, no mail is retrieved from the mailbox.

Example

```c
#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);
}
```
8.2.10 OS_MAILBOX_GetTimed1()

Description
Retrieves a new message of size 1 from a mailbox if a message is available within a given time.

Prototype

```
char OS_MAILBOX_GetTimed1(OS_MAILBOX* pMB, char* pDest, OS_TIME Timeout);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pDest</td>
<td>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.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Maximum time until the requested mail must be available. Timer period in embOS system ticks. 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.</td>
</tr>
</tbody>
</table>

Return value

- $= 0$ Success; message retrieved.
- $\neq 0$ 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 mail is available within the given timeout, or after the timeout value has expired. If the timeout has expired and no mail was available within the timeout the memory area where pDest 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 mail becomes available after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the mail was not available within the requested time. In this case, no mail is retrieved from the mailbox.

See Single-byte mailbox functions on page 187 for differences between OS_MAILBOX_GetTimed() and OS_MAILBOX_GetTimed1().

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;
}
```
8.2.11   OS_MAILBOX_GetPtr()

Description
Retrieves a pointer to a new message of a predefined size from a mailbox, if a message
is available. Non blocking function.

Prototype

\[
\text{char  OS_MAILBOX\_GetPtr (OS\_MAILBOX*  pMB,
    void**       ppDest);}
\]

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type \text{OS_MAILBOX}.</td>
</tr>
<tr>
<td>ppDest</td>
<td>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.</td>
</tr>
</tbody>
</table>

Return value

\[= 0\]   Success; message retrieved.
\[\neq 0\]   Message could not be retrieved (mailbox is empty); destination remains unchanged.

Additional information
If the mailbox is empty, no message is retrieved and \text{ppDest} remains unchanged, but
the program execution continues. This function never suspends the calling task. It may therefore also be called from an interrupt routine.

The retrieved message is not removed from the mailbox, this must be done by a call of \text{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 \text{OS\_MAILBOX\_Clear()}, \text{OS\_MAILBOX\_Delete()}, \text{OS\_MAILBOX\_GetBlocked*()} and \text{OS\_MAILBOX\_GetPtrBlocked*()} functions are not allowed until \text{OS\_MAILBOX\_Purge()} is called and will call \text{OS\_Error()} in debug builds of embOS.

Example

```c
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);
    }
}
```
8.2.12 OS_MAILBOX_GetPtrBlocked()

Description
Retrieves a pointer to a new message of a predefined size from a mailbox.

Prototype

```c
void OS_MAILBOX_GetPtrBlocked(OS_MAILBOX* pMB,
                              void** ppDest);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>ppDest</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 ISR.

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 and will call OS_Error() in debug builds of embOS.

Example

```c
static OS_MAILBOX _MBKey;

void PrintMessage(void) {
    char* p;

    OS_MAILBOX_GetPtrBlocked(&_MBKey, (void**)&p);
    printf("%d\n", *p);
    OS_MAILBOX_Purge(&_MBKey);
}
```
8.2.13 OS_MAILBOX_Peek()

Description
Peeks a mail from a mailbox without removing the mail. The mail is copied to *pDest if one was available.

Prototype

char OS_MAILBOX_Peek ( OS_CONST_PTR OS_MAILBOX * pMB ,
                       void  * pDest );

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pDest</td>
<td>Pointer to a buffer that should receive the mail.</td>
</tr>
</tbody>
</table>

Return value

= 0   Success, mail was available and is copied to *pDest.
≠ 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 mail was available the memory area where pDest points to remains unchanged.

Example

```c
#define MESSAGE_SIZE  4

static OS_MAILBOX _MBData;
static char _Buffer[MESSAGE_SIZE];

char PeekData(void) {
   return OS_MAILBOX_Peek(&_MBData, &_Buffer);  
}
```
8.2.14 OS_MAILBOX_Purge()

Description
Deletes the last retrieved message in a mailbox.

Prototype

```c
void OS_MAILBOX_Purge(OS_MAILBOX* pMB);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
</tbody>
</table>

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 and will call OS_Error() in debug builds of embOS.

Consecutive calls of OS_MAILBOX_Purge() or calling OS_MAILBOX_Purge() without having retrieved a message from the mailbox will also call OS_Error() in embOS debug builds.

Example

```c
static OS_MAILBOX _MBKey;

void PrintMessage(void) {
    char* p;

    OS_MAILBOX_GetPtrBlocked(&_MBKey, (void**)&p);
    printf("%d\n", *p);
    OS_MAILBOX_Purge(&_MBKey);
}
```
8.2.15 OS_MAILBOX_Put()

Description
Stores a new message of a predefined size in a mailbox if the mailbox is able to accept one more message.

Prototype

char OS_MAILBOX_Put(OS_MAILBOX* pMB,
                      OS_CONST_PTR void *pMail);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pMail</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>

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.

Example

static OS_MAILBOX _MBData;

void AddMessage(struct Data* pSomeData) {
  char Result;

  Result = OS_MAILBOX_Put(&_MBData, pSomeData);
  if (Result == 1) {
    printf("Was not able to add the message to the mailbox.\n");
  }
}
8.2.16 OS_MAILBOX_Put1()

Description
Stores a new message of size 1 in a mailbox if the mailbox is able to accept one more message.

Prototype

```c
char OS_MAILBOX_Put1(OS_MAILBOX* pMB, 
                      OS_CONST_PTR char *pMail);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pMail</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>

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.


Example

```c
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.
8.2.17 OS_MAILBOX_PutBlocked()

Description
Stores a new message of a predefined size in a mailbox.

Prototype

```c
void OS_MAILBOX_PutBlocked(OS_MAILBOX* pMB,
                          OS_CONST_PTR void* pMail);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pMail</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>

Additional information

If the mailbox is full, the calling task is suspended. Because this routine might require a sus-
pension, it must not be called from an interrupt routine. Use OS_MAILBOX_Put() / OS_MAIL-BOX_Put1() instead if you need to store data in a mailbox from within an ISR. When using a debug build of embOS, calling from an interrupt routine will call the error handler OS_Error() with error code OS_ERR_IN_ISR.

Example

```c
static OS_MAILBOX _MBData;

void AddMessage(struct Data* pSomeData) {
  OS_MAILBOX_PutBlocked(&_MBData, pSomeData);
}
```
8.2.18 OS_MAILBOX_PutBlocked1()

Description
Stores a new message of size 1 in a mailbox.

Prototype

```c
void OS_MAILBOX_PutBlocked1 ( OS_MAILBOX * pMB,
                               OS_CONST_PTR char *pMail);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pMail</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>

Additional information

If the mailbox is full, the calling task is suspended. Because this routine might require a sus-
pension, it must not be called from an interrupt routine. Use OS_MAILBOX_Put()/OS_MAIL-
BOX_Put1() instead if you need to store data in a mailbox from within an ISR. When using a
debug build of embOS, calling from an interrupt routine will call the error handler OS_Err-
ror() with error code OS_ERR_IN_ISR.

See Single-byte mailbox functions on page 187 for differences between OS_MAILBOX_Put-
Blocked() and OS_MAILBOX_PutBlockd1().

Example

Single-byte mailbox as keyboard buffer:

```c
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);
}
```
8.2.19 OS_MAILBOX_PutFront()

Description
Stores a new message of a predefined size into a 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

```c
char OS_MAILBOX_PutFront(OS_MAILBOX* pMB,
                          OS_CONST_PTR void *pMail);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pMail</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>

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.

Example

```c
static OS_MAILBOX _MBData;

void AddMessage(struct Data* pSomeData) {
    char Result;
    
    Result = OS_MAILBOX_PutFront(&_MBData, pSomeData);
    if (Result == 1) {
        printf("Was not able to add the message to the mailbox.\n");
    }
}
```
8.2.20 OS_MAILBOX_PutFront1()

Description
Stores a new message of size 1 into a 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* pMB,
   OS_CONST_PTR char *pMail);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pMail</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>

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 187 for differences between OS_MAILBOX_Put-
Front() and OS_MAILBOX_PutFront1().

Example

```c
static OS_MAILBOX _MBData;

void AddMessage(char c) {
    char Result;

    Result = OS_MAILBOX_PutFront1(&_MBData, &c);
    if (Result == 1) {
        printf("Was not able to add the message to the mailbox.\n");
    }
}
```
8.2.21 OS_MAILBOX_PutFrontBlocked()

Description
Stores a new message of a predefined size at the beginning of a mailbox in front of all other messages. This new message will be retrieved first.

Prototype

```c
void  OS_MAILBOX_PutFrontBlocked ( OS_MAILBOX *   pMB ,
                                    OS_CONST_PTR void  * pMail );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pMail</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>

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 ISR.

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.

Example

```c
static OS_MAILBOX _MBData ;

void AddMessage(struct Data* pSomeData) {
    OS_MAILBOX_PutFrontBlocked(&_MBData , pSomeData);
}
```
CHAPTER 8  API functions

8.2.22  OS_MAILBOX_PutFrontBlocked1()

Description
Stores a new message of size 1 at the beginning of a mailbox in front of all other messages. This new message will be retrieved first.

Prototype

```c
void OS_MAILBOX_PutFrontBlocked1(OS_MAILBOX* pMB,
                                  OS_CONST_PTR char *pMail);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pMail</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>

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 ISR.

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 187 for differences between OS_MAILBOX_PutFrontBlocked() and OS_MAILBOX_PutFrontBlocked1().

Example

Single-byte mailbox as keyboard buffer which will follow the LIFO pattern:

```c
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);
}
```
CHAPTER 8
API functions

8.2.23 OS_MAILBOX_PutTimed()

Description
Stores a new message of a predefined size in a 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

```c
OS_BOOL OS_MAILBOX_PutTimed(OS_MAILBOX* pMB,
OS_CONST_PTR void *pMail,
OS_TIME Timeout);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pMail</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>
| Timeout   | Maximum time in embOS system ticks until the given mail must be stored Timer period in embOS system ticks. The data type OS_TIME is defined as an integer, therefore valid values are:  
  \[ 1 \leq \text{Timeout} \leq 2^{15} - 1 = 0x7FFF \text{ for 8/16-bit CPUs.} \]  
  \[ 1 \leq \text{Timeout} \leq 2^{31} - 1 = 0xFFFFFFFF \text{ for 32-bit CPUs.} \] |

Return value

\[ = 0 \quad \text{Success; message stored.} \]
\[ \neq 0 \quad \text{Message could not be stored within the given timeout (mailbox is full). destination remains unchanged.} \]

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 mail 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 mail is stored in the mailbox.

Example

```c
static OS_MAILBOX _MBData;

void AddMessage(struct Data* pSomeData) {
    OS_MAILBOX_PutTimed(&_MBData, pSomeData, 10); // Wait maximum 10 system ticks
}
```
8.2.24 OS_MAILBOX_PutTimed1()

Description
Stores a new message of size 1 in a 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* pMB,
                                 OS_CONST_PTR char *pMail,
                                 OS_TIME      Timeout);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
<tr>
<td>pMail</td>
<td>Pointer to the message to store.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Maximum time in embOS system ticks until the given mail must be stored. Timer period in embOS system ticks. The data type OS_TIME is defined as an integer, therefore valid values are:</td>
</tr>
<tr>
<td></td>
<td>1 ≤ Timeout ≤ 215 - 1 = 0x7FFF for 8/16-bit CPUs.</td>
</tr>
<tr>
<td></td>
<td>1 ≤ Timeout ≤ 231 - 1 = 0x7FFFFFFF for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

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 mail 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 mail is stored in the mailbox.

See Single-byte mailbox functions on page 187 for differences between OS_MAILBOX_PutTimed() and OS_MAILBOX_PutTimed1().

Example

```c
static OS_MAILBOX _MBKey;

void SetKey(char c) {
    OS_MAILBOX_PutTimed1(&_MBKey, &c, 10); // Wait maximum 10 system ticks
}
```
### 8.2.25 **OS_MAILBOX_WaitBlocked()**

**Description**

Waits until a mail is available, but does not retrieve the message from the mailbox.

**Prototype**

```c
void OS_MAILBOX_WaitBlocked(OS_MAILBOX* pMB);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
</tbody>
</table>

**Additional information**

If the mailbox is empty, the task is suspended until a mail is available, otherwise the task continues. The task continues execution according to the rules of the scheduler as soon as a mail is available, but the mail is not retrieved from the mailbox.

**Example**

```c
static OS_MAILBOX _MBData;

void Task(void) {
    while (1) {
        OS_MAILBOX_WaitBlocked(&_MBData);
        ...
    }
}
```
### 8.2.26 OS_MAILBOX_WaitTimed()

**Description**

Waits until a mail is available or the timeout has expired, but does not retrieve the message from the mailbox.

**Prototype**

```c
char OS_MAILBOX_WaitTimed(OS_MAILBOX * pMB,
                          OS_TIME     Timeout);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMB</td>
<td>Pointer to a mailbox object of type OS_MAILBOX.</td>
</tr>
</tbody>
</table>
| Timeout   | Maximum time in embOS system ticks until the requested mail must be available. Timer period in embOS system ticks. 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**

- **= 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 mail 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 mail becomes available after the timeout expired, but before the calling task is resumed. Anyhow, the function returns with timeout, because the mail was not available within the requested time.

**Example**

```c
static OS_MAILBOX _MBData;

void Task(void) {
  char Result;

  Result = OS_MAILBOX_WaitTimed(&_MBData, 10);
  if (Result == 0) {
    // Compute message
  } else {
    // Timeout
  }
}
```
Chapter 9

Queues
9.1 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 realtime 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.

Message size in bytes on 8 or 16-bit CPUs:
\[ 1 \leq x \leq 2^{15} - (1 + \text{OS\_Q\_SIZEOF\_HEADER} + \text{MESSAGE\_ALIGNMENT}) \]

Message size in bytes on 32-bit CPUs:
\[ 1 \leq x \leq 2^{31} - (1 + \text{OS\_Q\_SIZEOF\_HEADER} + \text{MESSAGE\_ALIGNMENT}) \]

Maximum buffer size in bytes for one queue on 8 or 16-bit CPUs:
\[ 2^{16} = 0xFFFF \]

Maximum buffer size in bytes for one queue on 32-bit CPUs:
\[ 2^{32} = 0xFFFFFFFF \]

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**

```c
#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\0", 7);
        OS_QUEUE_Put(&MyQueue, "\nWorld !\0", 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;
}
```
### 9.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_QUEUE_Clear()</td>
<td>Clears all messages in the specified queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_Create()</td>
<td>Creates and initializes a message queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_Delete()</td>
<td>Deletes a specific message queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_GetMessageCnt()</td>
<td>Returns the number of messages that are currently stored in a queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_GetMessageSize()</td>
<td>Returns the size of the first message in the queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_GetPtr()</td>
<td>Retrieve the pointer to a message from the message queue if a message is available.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_GetPtrBlocked()</td>
<td>Retrieve the pointer to a message from the message queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_GetPtrTimed()</td>
<td>Retrieve the pointer to a message from the message queue within a specified time if a message is available.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_IsInUse()</td>
<td>Delivers information whether the queue is actually in use.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_PeekPtr()</td>
<td>Retrieve the pointer to a message from the message queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_Purge()</td>
<td>Deletes the last retrieved message in a queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_Put()</td>
<td>Stors a new message of given size in a queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_PutEx()</td>
<td>Stores a new message, of which the distinct parts are distributed in memory as indicated by a OS_QUEUE_SRCLIST structure, in a queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_PutBlocked()</td>
<td>Stores a new message of given size in a queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_PutBlockedEx()</td>
<td>Stores a new message, of which the distinct parts are distributed in memory as indicated by a OS_QUEUE_SRCLIST structure, in a queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_PutTimed()</td>
<td>Stores a new message of given size in a queue if space is available within a given time.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_QUEUE_PutTimedEx()</td>
<td>Stores a new message, of which the distinct parts are distributed in memory as indicated by a OS_QUEUE_SRCLIST structure, in a queue.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
9.2.1 OS_QUEUE_Clear()

Description
Clears all messages in the specified queue.

Prototype

```c
void OS_QUEUE_Clear(OS_QUEUE* pQ);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
</tbody>
</table>

Additional information

When the queue is in use, a debug build of embOS will call OS_Error() with error code OS_ERR_QUEUE_INUSE.

OS_QUEUE_Clear() may cause a task switch.

Example

```c
static OS_QUEUE _Queue;

void ClearQueue() {
    OS_QUEUE_Clear(&_Queue);
}
```
9.2.2 OS_QUEUE_Create()

Description
Creates and initializes a message queue.

Prototype

```c
void OS_QUEUE_Create ( OS_QUEUE * pQ,
                      void * pData,
                      OS_UINT Size);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
<tr>
<td>pData</td>
<td>Pointer to a memory area used as data buffer for the queue.</td>
</tr>
<tr>
<td>Size</td>
<td>Size in bytes of the data buffer.</td>
</tr>
</tbody>
</table>

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.

Example

```c
#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));
}
```
9.2.3  OS_QUEUE_Delete()

Description

Deletes a specific message queue.

Prototype

```c
void OS_QUEUE_Delete ( OS_QUEUE* pQ );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
</tbody>
</table>

Additional information

To keep the system fully dynamic, it is essential that queues can be created dynamically. This also means there must be a way to delete a queue when it is no longer needed. The memory that has been used by the queue for the control structure and the buffer can then be reused or reallocated.

It is the programmer’s responsibility to:

- make sure that the program no longer uses the queue to be deleted
- make sure that the queue to be deleted actually exists (i.e. has been created first).

When the queue is in use, a debug build of embOS will call `OS_Error()` with error code `OS_ERR_QUEUE_INUSE`.

When tasks are waiting, a debug build of embOS will call `OS_Error()` with error code `OS_ERR_QUEUE_DELETE` is called.

Example

```c
static OS_QUEUE _QSerIn;

void Cleanup(void) {
    OS_QUEUE_Delete (&_QSerIn);
}
```
9.2.4 OS_QUEUE_GetMessageCnt()

Description
Returns the number of messages that are currently stored in a queue.

Prototype

```
int OS_QUEUE_GetMessageCnt(OS_CONST_PTR OS_QUEUE *pQ);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
</tbody>
</table>

Return value
The number of messages in the queue.

Example

```
static OS_QUEUE _Queue;

void PrintNumberOfMessages() {
    int Cnt;

    Cnt = OS_QUEUE_GetMessageCnt(&_Queue);
    printf("%d messages available.\n", Cnt);
}
```
9.2.5 OS_QUEUE_GetMessageSize()

Description
Returns the size of the first message in the queue.

Prototype

```c
int OS_QUEUE_GetMessageSize(OS_CONST_PTR OS_QUEUE *pQ);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
</tbody>
</table>

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.

Example

```c
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);
  }
}
```
9.2.6 OS_QUEUE_GetPtr()

Description
Retrieve the pointer to a message from the message queue if a message is available.

Prototype

```c
int OS_QUEUE_GetPtr(OS_QUEUE* pQ,
                     void** ppData);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
<tr>
<td>ppData</td>
<td>Address of the pointer which will be set to the address of the message.</td>
</tr>
</tbody>
</table>

Return value

- `= 0` No message available in queue.
- `> 0` Size of the message that was retrieved from the queue.

Additional information

If the queue is empty, the function returns zero and `ppData` will not be set. This function never suspends the calling task. It may therefore be called from an interrupt routine or timer. If a message could be retrieved it 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 and will call `OS_Error()` in debug builds of embOS.

Example

```c
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();
        }
    }
}
```
9.2.7 OS_QUEUE_GetPtrBlocked()

Description
Retrieve the pointer to a message from the message queue.

Prototype

```c
int OS_QUEUE_GetPtrBlocked(OS_QUEUE* pQ,
                            void** ppData);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
<tr>
<td>ppData</td>
<td>Address of the pointer which will be set to the</td>
</tr>
<tr>
<td></td>
<td>address of the message.</td>
</tr>
</tbody>
</table>

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_GetPtrCond() 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 and will call OS_Error() in debug builds of embOS.

Example

```c
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
    }
}
```
9.2.8 OS_QUEUE_GetPtrTimed()

Description
Retrieve the pointer to a message from the message queue within a specified time if a message is available.

Prototype

```c
int OS_QUEUE_GetPtrTimed ( OS_QUEUE * pQ,
                           void ** ppData,
                           OS_TIME   Timeout );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
<tr>
<td>ppData</td>
<td>Address of the pointer which will be set to the address of the message.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Maximum time until the requested message must be available. Timer period in embOS system ticks. 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.</td>
</tr>
</tbody>
</table>

Return value

- \(= 0\) No message available in queue.
- \(> 0\) Size of the message that was retrieved from the queue.

Sets the pointer `ppData` 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 `ppData` 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 and will call `OS_Error()` in debug builds of embOS.

Example

```c
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 {
    DoSomethingElse();
}
}
9.2.9 OS_QUEUE_IsInUse()

Description
Delivers information whether the queue is actually in use.

Prototype

```c
OS_BOOL OS_QUEUE_IsInUse(OS_CONST_PTR OS_QUEUE *pQ);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
</tbody>
</table>

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 actually 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.

Example

```c
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();
}
```
9.2.10 OS_QUEUE_PeekPtr()

Description
Retrieve the pointer to a message from the message queue. The message must not be purged.

Prototype

```c
int OS_QUEUE_PeekPtr (OS_CONST_PTR OS_QUEUE *pQ,
                      void**        ppData);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type <code>OS_QUEUE</code>.</td>
</tr>
<tr>
<td>ppData</td>
<td>Address of the pointer which will be set to the address of the message.</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>= 0</td>
<td>No message available.</td>
</tr>
<tr>
<td>≠ 0</td>
<td>Size of message in bytes.</td>
</tr>
</tbody>
</table>

Additional information

Sets the pointer `ppData` to the message that should be retrieved. If no message is available `ppData` 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. `OS_TASK_EnterRegion()` does not inhibit cooperative task switches!

Example

```c
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();
  }
}
```
9.2.11 OS_QUEUE_Purge()

Description
Deletes the last retrieved message in a queue.

Prototype

```c
void OS_QUEUE_Purge(OS_QUEUE* pQ);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
</tbody>
</table>

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 will call the embOS error handler OS_Error() in embOS debug builds.

Example

```c
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
    }
}
```
9.2.12 OS_QUEUE_Put()

Description
Stors a new message of given size in a queue.

Prototype

```c
int OS_QUEUE_Put(OS_QUEUE* pQ,
                  OS_CONST_PTR void *pSrc,
                  OS_UINT Size);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
<tr>
<td>pSrc</td>
<td>Pointer to the message to store.</td>
</tr>
<tr>
<td>Size</td>
<td>Size of the message to store. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>(1 \leq \text{Size} \leq 2^{15} - 1 = 0x7FFF) for 8/16-bit CPUs.</td>
</tr>
<tr>
<td></td>
<td>(1 \leq \text{Size} \leq 2^{31} - 1 = 0x7FFFFFFF) for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

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.

Example

```c
static OS_QUEUE _MemoryQ;

int MEMORY_Write(const char* pData, OS_UINT Len) {
    return OS_QUEUE_Put(&_MemoryQ, pData, Len);
}
```
9.2.13  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 a queue.

Prototype

```c
int OS_QUEUE_PutEx(OS_QUEUE* pQ,
                    OS_CONST_PTR OS_QUEUE_SRCLIST *pSrcList,
                    OS_UINT NumSrc);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
<tr>
<td>pSrcList</td>
<td>Pointer to an array of OS_QUEUE_SRCLIST structures which contain pointers to the data to store.</td>
</tr>
<tr>
<td>NumSrc</td>
<td>Number of OS_QUEUE_SRCLIST structures at pSrcList.</td>
</tr>
</tbody>
</table>

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.

Example

```c
OS_CONST_PTR OS_QUEUE_SRCLIST aDataList[] = { {"Hello ", 6},
                                              {"World!", 6} }
OS_QUEUE_PutEx(&_MemoryQ, aDataList, 2);
```

9.2.13.1  The OS_QUEUE_SRCLIST structure

The OS_QUEUE_SRCLIST structure consists of two elements:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSrc</td>
<td>Pointer to a part of the message to store.</td>
</tr>
<tr>
<td>Size</td>
<td>Size of the part of the message. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>(1 \leq \text{Size} \leq 2^{15} - 1 = 0x7FFF) for 8/16-bit CPUs.</td>
</tr>
<tr>
<td></td>
<td>(1 \leq \text{Size} \leq 2^{31} - 1 = 0x7FFFFFFF) for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

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.
9.2.14 OS_QUEUE_PutBlocked()

Description
Stores a new message of given size in a queue.

Prototype

```c
void OS_QUEUE_PutBlocked(OS_QUEUE* pQ,
                          OS_CONST_PTR void *pSrc,
                          OS_UINT Size);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
<tr>
<td>pSrc</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>
| Size      | 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. |

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.

Example

```c
static OS_QUEUE _MemoryQ;

void StoreMessage(const char* pData, OS_UINT Len)
    OS_QUEUE_PutBlocked(&_MemoryQ, pData, Len);
```
9.2.15  **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 a queue. Blocks the calling task when queue is full.

**Prototype**

```c
void OS_QUEUE_PutBlockedEx(OS_QUEUE* pQ,
                        OS_CONST_PTR OS_QUEUE_SRCLIST* pSrcList,
                        OS_UINT          NumSrc);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type <strong>OS_QUEUE</strong>.</td>
</tr>
<tr>
<td>pSrcList</td>
<td>Pointer to an array of <strong>OS_QUEUE_SRCLIST</strong> structures which contain pointers to the data to store.</td>
</tr>
<tr>
<td>NumSrc</td>
<td>Number of <strong>OS_QUEUE_SRCLIST</strong> structures at pSrcList.</td>
</tr>
</tbody>
</table>

**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.

For more information on the **OS_QUEUE_SRCLIST** structure, refer to The **OS_QUEUE_SRCLIST** structure in the chapter The **OS_QUEUE_SRCLIST** structure on page 235.

**Example**

```c
OS_CONST_PTR OS_QUEUE_SRCLIST aDataList[] = { {"Hello ", 6},
                                         {"World!", 6}  
                          
);
OS_QUEUE_PutEx(&_MemoryQ, aDataList, 2);
```
9.2.16 OS_QUEUE_PutTimed()

Description
Stores a new message of given size in a queue if space is available within a given time.

Prototype

```c
char OS_QUEUE_PutTimed(OS_QUEUE* pQ, OS_CONST_PTR void *pSrc, OS_UINT Size, OS_TIME Timeout);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
<tr>
<td>pSrc</td>
<td>Pointer to the message to store.</td>
</tr>
</tbody>
</table>
| Size      | 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. |
| Timeout   | Maximum time until the given message must be stored. 
|           | Timer period in embOS system ticks. 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

- 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.

Example

```c
static OS_QUEUE _MemoryQ;

int MEMORY_WriteTimed(const char* pData, OS_UINT Len, OS_TIME Timeout) {
    return OS_QUEUE_PutTimed(&_MemoryQ, pData, Len, Timeout);
}
```
9.2.17 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 a queue. Suspends the calling task for a given timeout when the queue is full.

Prototype

```c
char OS_QUEUE_PutTimedEx(OS_QUEUE * pQ,
                         OS_CONST_PTR OS_QUEUE_SRCLIST * pSrcList,
                         OS_UINT NumSrc,
                         OS_TIME Timeout);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQ</td>
<td>Pointer to a queue object of type OS_QUEUE.</td>
</tr>
<tr>
<td>pSrcList</td>
<td>Pointer to an array of OS_QUEUE_SRCLIST structures which contain pointers to the data to store.</td>
</tr>
<tr>
<td>NumSrc</td>
<td>Number of OS_QUEUE_SRCLIST structures at pSrcList.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Maximum time until the given message must be stored. Timer period in embOS system ticks. The data type OS_TIME is defined as an integer, therefore valid values are: 1 ≤ Timeout ≤ 2^{15} - 1 = 0x7FFF for 8/16-bit CPUs. 1 ≤ Timeout ≤ 2^{31} - 1 = 0xFFFFFFFF for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

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.

For more information on the OS_QUEUE_SRCLIST structure, refer to The OS_QUEUE_SRCLIST structure in the chapter The OS_QUEUE_SRCLIST structure on page 235.

Example

```c
OS_CONST_PTR OS_QUEUE_SRCLIST aDataList[] = {
    {"Hello ", 6},
    {"World!", 6}
};
OS_QUEUE_PutEx(&MemoryQ, aDataList, 2, 100);
```
Chapter 10

Watchdog
10.1 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") 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.

Example

```c
#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_USEPARA(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;
}
```
## 10.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>ISR</th>
<th>Unpriv Task</th>
<th>Priv Task</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_WD_Add()</td>
<td>Adds a software watchdog timer to the watchdog list.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_WD_Check()</td>
<td>Checks if a watchdog timer expired.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_WD_Config()</td>
<td>Sets the watchdog callback functions.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_WD_Remove()</td>
<td>Removes a watchdog timer from the watchdog list.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_WD_Trigger()</td>
<td>Triggers a watchdog timer.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
10.2.1 OS_WD_Add()

**Description**

Adds a software watchdog timer to the watchdog list.

**Prototype**

```c
void OS_WD_Add ( OS_WD * pWD ,
                  OS_TIME Timeout );
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pWD</td>
<td>Pointer to a watchdog object of type <code>OS_WD</code>.</td>
</tr>
</tbody>
</table>
| Timeout   | Watchdog timer timeout. Must be within the following range:  
            | 0 ≤ t ≤ 2^{15} - 1 = 0x7FFF for 8/16-bit CPUs  
            | 0 ≤ t ≤ 2^{31} - 1 = 0x7FFFFFFF for 32-bit CPUs  
            | Please note that these are signed values. |

**Example**

```c
static OS_WD _myWD ;

void HPTask (void) {
    OS_WD_Add ( & _myWD , 50 );
    while (1) {
        OS_WD_Trigger ( & _myWD );
        OS_TASK_Delay (50 );
    }
}
```
10.2.2 OS_WD_Check()

Description
Checks if a watchdog timer expired. If no watchdog timer expired the hardware watchdog
is triggered. If a watchdog timer expired, the callback function is called.

Prototype

void OS_WD_Check(void);

Additional information
OS_WD_Check() must be called periodically. It is good practice to call it from the system
tick handler.

Example

```c
void SysTick_Handler(void) {
    OS_INT_Enter();
    OS_Tick_Handle();
    OS_WD_Check();
    OS_INT_Leave();
}
```
10.2.3 OS_WD_Config()

Description
Sets the watchdog callback functions.

Prototype

```c
void OS_WD_Config ( voidRoutine * pfTriggerFunc ,
                   OS_WD_RESET_CALLBACK *pfResetFunc );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfTriggerFunc</td>
<td>Function pointer to hardware watchdog trigger callback function.</td>
</tr>
<tr>
<td>pfResetFunc</td>
<td>Function pointer to callback function which is called in case of an expired watchdog timer. pfResetFunc is optional and may be NULL.</td>
</tr>
</tbody>
</table>

Additional information

pfResetFunc may be used to perform additional operations inside a callback function prior to the reset of the microcontroller. For example, a message may be written to a log file. If pfResetFunc is NULL, no callback function gets executed, but the hardware watchdog will still cause a reset of the microcontroller.

Example

```c
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();
}
```

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.
10.2.4 OS_WD_Remove()

Description
Removes a watchdog timer from the watchdog list.

Prototype

```c
void OS_WD_Remove(OS_CONST_PTR OS_WD *pWD);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pWD</td>
<td>Pointer to a watchdog object of type OS_WD.</td>
</tr>
</tbody>
</table>

Example

```c
int main(void) {
    OS_WD_Add(&_myWD);
    OS_WD_Remove(&_myWD);
}
```
10.2.5 OS_WD_Trigger()

Description
Triggers a watchdog timer.

Prototype

```c
void OS_WD_Trigger(OS_WD * pWD);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pWD</td>
<td>Pointer to a watchdog object of type OS_WD.</td>
</tr>
</tbody>
</table>

Additional information
Each software watchdog timer must be triggered 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).

Example

```c
static OS_WD _myWD;

static void HPTask(void) {
    OS_WD_Add(&_myWD, 50);
    while (1) {
        OS_TASK_Delay(50);
        OS_WD_Trigger(&_myWD);
    }
}
```
Chapter 11

Multi-core Support
11.1 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.

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 multicore 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 \texttt{OS\_SPINLOCK} variable (or \texttt{OS\_SPINLOCK\_SW}, respectively) at an identical memory address. Initialization of the spinlock, however, must be performed by one core only. This API is not available in embOS library mode \texttt{OS\_LIBMODE\_SAFE}.

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:

```c
#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:

```c
#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;
}
```
## 11.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_SPINLOCK_Create()</td>
<td>Creates a hardware-specific spinlock.</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_SPINLOCK_Lock()</td>
<td>Acquires a hardware-specific spinlock. Busy waiting until the spinlock becomes available. This function is unavailable for some architectures.</td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_SPINLOCK_Unlock()</td>
<td>Releases a hardware-specific spinlock.</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_SPINLOCK_SW_Create()</td>
<td>Creates a software-implementation spinlock.</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_SPINLOCK_SW_Lock()</td>
<td>Acquires a software-implementation spinlock.</td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_SPINLOCK_SW_Unlock()</td>
<td>Releases a software-implementation spinlock.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
11.2.0.1 OS_SPINLOCK_Create()

Description
Creates a hardware-specific spinlock. This function is unavailable for architectures that do not support an appropriate instruction set.

Prototype

```c
void OS_SPINLOCK_Create(OS_SPINLOCK* pSpinlock);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSpinlock</td>
<td>Pointer to a spinlock object of type OS_SPINLOCK. The variable must reside in shared memory.</td>
</tr>
</tbody>
</table>

Additional information

After creation, the spinlock is not locked.
11.2.0.2 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

```c
void OS_SPINLOCK_Lock (OS_SPINLOCK* pSpinlock);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSpinlock</td>
<td>Pointer to a variable of type OS_SPINLOCK reserved for the management of the spinlock.</td>
</tr>
</tbody>
</table>

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:
11.2.0.3 OS_SPINLOCK_Unlock()

Description
Releases a hardware-specific spinlock. This function is unavailable for architectures that do not support an appropriate instruction set.

Prototype

```c
void OS_SPINLOCK_Unlock ( OS_SPINLOCK* pSpinlock );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSpinlock</td>
<td>Pointer to a spinlock object of type OS_SPINLOCK.</td>
</tr>
</tbody>
</table>
11.2.0.4 OS_SPINLOCK_SW_Create()

Description
Creates a software-implementation spinlock.

Prototype

```c
void OS_SPINLOCK_SW_Create(OS_SPINLOCK_SW* pSpinlock);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSpinlock</td>
<td>Pointer to a spinlock object of type OS_SPINLOCK. The variable must reside in shared memory.</td>
</tr>
</tbody>
</table>

Additional information

After creation, the spinlock is not locked.
11.2.0.5  OS_SPINLOCK_SW_Lock()

Description
Acquires a 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

```c
void OS_SPINLOCK_SW_Lock(OS_SPINLOCK_SW* pSpinlock,
                          OS_UINT     Id);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSpinlock</td>
<td>Pointer to a spinlock object of type OS_SPINLOCK.</td>
</tr>
<tr>
<td>Id</td>
<td>Unique identifier to specify the core accessing the spinlock. Valid values are 0 ≤ Id &lt; OS_SPINLOCK_MAX_CORES. By default, OS_SPINLOCK_MAX_CORES is defined to 4 and may be changed when using source code. An embOS debug build calls OS_Error() in case invalid values are used.</td>
</tr>
</tbody>
</table>

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().


The following diagram illustrates how OS_SPINLOCK_SW_Lock() works:

```
OS_SPINLOCK_SW_Lock()

Spinlock available?

Yes
Mark Spinlock in use

No

return
```
11.2.0.6   OS_SPINLOCK_SW_Unlock()

Description
Releases a software-implementation spinlock.

Prototype

```c
void OS_SPINLOCK_SW_Unlock(OS_SPINLOCK_SW* pSpinlock,
                            OS_UINT Id);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSpinlock</td>
<td>Pointer to a spinlock object of type OS_SPINLOCK.</td>
</tr>
<tr>
<td>Id</td>
<td>Unique identifier to specify the core accessing the spinlock. Valid values are 0 ≤ Id &lt; OS_SPINLOCK_MAX_CORES. By default, OS_SPINLOCK_MAX_CORES is defined to 4 and may be changed when using source code. An embOS debug build calls OS_Error() in case invalid values are used.</td>
</tr>
</tbody>
</table>
Chapter 12

Interrupts
12.1 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.
12.2 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 262.

12.2.1 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.

12.2.2 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.
- Real-time Operating system (RTOS). An RTOS also needs to temporarily disable the interrupts which can call API-functions of the RTOS. Some RTOSes disable all interrupts,
effectively increasing interrupt latency for all interrupts, some (like embOS) disable only low-priority interrupts and do thereby not affect the latency of high priority interrupts.

12.2.3 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 occurance and the actual execution of the interrupt handler:

```c
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:

```c
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 substracting 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:

```c
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.

### 12.2.4 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.

### 12.2.5 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 interrupts. High priority interrupts are named “Zero latency interrupts” and low priority interrupts are named “embOS interrupts”. The embOS port-specific documentations ex-
plain 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 “embOS interrupts” and the four lowest priority interrupts are considered as “Zero latency 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.

### 12.2.5.1 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.

![Diagram showing interrupt priority levels and usage](Image)

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.
12.3 Rules for interrupt handlers

12.3.1 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.

12.3.2 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 behaviour 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.

12.3.3 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.
## 12.3.4 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Unpriv Task</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OS_INT_Call()</code></td>
<td>Entry function for use in an embOS interrupt handler.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_INT_CallNestable()</code></td>
<td>Entry function for use in an embOS interrupt handler.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_INT_Enter()</code></td>
<td>Informs embOS that interrupt code is executing.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_INT_EnterIntStack()</code></td>
<td>Switches to another stack in interrupt routines.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_INT_EnterNestable()</code></td>
<td>Informs embOS that interrupt code is executing and reenables interrupts.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_INT_InInterrupt()</code></td>
<td>Checks if the calling function runs in an interrupt context.</td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td><code>OS_INT_Leave()</code></td>
<td>Informs embOS that the end of the interrupt routine has been reached; executes task switching within ISR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_INT_LeaveIntStack()</code></td>
<td>Switches back to the interrupt stack.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_INT_LeaveNestable()</code></td>
<td>Informs embOS that the end of the interrupt routine has been reached; executes task switching within ISR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
12.3.4.1 OS_INT_Call()

Description
Entry function for use in an embOS interrupt handler. Nestable interrupts are disabled.

Prototype

void OS_INT_Call(void (*pRoutine)());

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pRoutine</td>
<td>Pointer to a routine that should run on interrupt.</td>
</tr>
</tbody>
</table>

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

```c
#pragma interrupt
void SysTick_Handler(void) {
    OS_INT_Call(_IsrTickHandler);
}
```
12.3.4.2 OS_INT_CallNestable()

Description
Entry function for use in an embOS interrupt handler. Nestable interrupts are enabled.

Prototype
```c
void OS_INT_CallNestable(void (*pRoutine)());
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pRoutine</td>
<td>Pointer to a routine that should run on interrupt.</td>
</tr>
</tbody>
</table>

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
```c
#pragma interrupt
void SysTick_Handler(void) {
  OS_INT_CallNestable(_IsrTickHandler);
}
```
12.3.4.3 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(1, &Task); // Any functionality could be here
    OS_INT_Leave();
}
12.3.4.4 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

```c
void OS_INT_EnterIntStack(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

```c
void ISR_Timer(void) {
  OS_INT_Enter();
  OS_INT_EnterIntStack();
  OS_TASKEVENT_Set(1, &Task);
  OS_INT_LeaveIntStack();
  OS_INT_Leave();
}
```
12.3.4.5  OS_INT_EnterNestable()

Description
Re-enables interrupts and increments the embOS internal critical region counter, thus disabling further task switches.

Prototype

```c
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

```c
_interrupt void ISR_Timer(void) {
    OS_INT_EnterNestable();
    OS_TASKEVENT_Set(1, &Task); // Any functionality could be here
    OS_INT_LeaveNestable();
}
```
12.3.4.6 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

```c
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

```c
void foo() {
  if (OS_INT_InInterrupt() == 1) {
    // Do something within the ISR
  } else {
    printf("No interrupt context.\n")
  }
}
```
12.3.4.7 OS_INT_Leave()

Description
Informs embOS that the end of the interrupt routine has been reached; executes task switching within ISR.

Prototype

```c
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

```c
void ISR_Timer(void) {
    OS_INT_Enter();
    OS_TASKEVENT_Set(1, &Task);  // Any functionality could be here
    OS_INT_Leave();
}
```
12.3.4.8  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

```c
void ISR_Timer(void) {
    OS_INT_Enter();
    OS_INT_EnterIntStack();
    OS_TASKEVENT_Set(1, &Task);
    OS_INT_LeaveIntStack();
    OS_INT_Leave();
}
```
12.3.4.9  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

```c
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_En-
terNestable().

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

```c
_interrupt void ISR_Timer(void) {
    OS_INT_EnterNestable();
    OS_TASKEVENT_Set(1,&Task);  // Any functionality could be here
    OS_INT_LeaveNestable();
}
```
12.4 Interrupt control

12.4.1 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. 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 accesses that are needed, interrupts must be temporarily disabled:

**Bad example:**

```c
volatile long lvar;

void IntHandler(void) {
    lvar++;
}

void routine (void) {
    lvar++;
}
```

**Good example:**

```c
volatile long lvar;

void IntHandler(void) {
    lvar++;
}

void routine (void) {
    OS_INT_Disable();
    lvar++;
    OS_INT_Enable();
}
```

The problem with disabling and re-enabling interrupts is that functions that disable/enable the interrupt cannot be nested.

Your C compiler offers two intrinsic functions for enabling and disabling interrupts. These functions can still be used, but it is recommended to use the functions that embOS offers (to be precise, they only look like functions, but are macros in reality). If you do not use these recommended embOS functions, you may run into a problem if routines which require a portion of the code to run with disabled interrupts are nested or call an OS routine.

We recommend disabling interrupts only for short periods of time, if possible. Also, you should not call functions when interrupts are disabled, because this could lead to long interrupt latency times (the longer interrupts are disabled, the higher the interrupt latency). You may also safely use the compiler-provided intrinsics to disable interrupts but you must ensure to not call embOS functions with disabled interrupts.

12.4.2 Global interrupt enable / disable

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 explicitly enable and disable zero latency interrupts.

In an application it may be required to disable and enable all interrupts. These functions have the suffix `All` and allow a "global" handling of the interrupt enable state of the CPU. These functions affect the state of the CPU unconditionally and should be used with care.
12.4.3 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.

12.4.4 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>UpnpI Task</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_INT_DecRI()</td>
<td>Decrements the counter and enables interrupts if the counter reaches 0.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_Disable()</td>
<td>Disables interrupts. Does not change the interrupt disable counter.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_DisableAll()</td>
<td>Disable all interrupts (high and low priority) unconditionally.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_Enable()</td>
<td>Unconditionally enables interrupts.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_EnableAll()</td>
<td>Enable all interrupts (high and low priority) unconditionally.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_EnableConditional()</td>
<td>Restores the state of the interrupt flag, based on the interrupt disable counter.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_IncDI()</td>
<td>Increments the interrupt disable counter (OS_Global.Counter.s.DI) and disables interrupts.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_Preserve()</td>
<td>Preserves the embOS interrupt state.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_PreserveAll()</td>
<td>Preserves the current interrupt enable state.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_PreserveAndDisableAll()</td>
<td>Preserves the current interrupt enable state and then disables all interrupts.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_Restore()</td>
<td>Restores the embOS interrupt state.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_INT_RestoreAll()</td>
<td>Restores the interrupt enable state which was preserved before.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
12.4.4.1 OS_INT_DecRI()

Description
Short for Decrement and Restore Interrupts. Decrements the counter and enables inter-
rupts if the disable counter reaches zero. It is important that they are used as a pair: first
OS_INT_IncDI(), then OS_INT_DecRI(). OS_INT_IncDI() and OS_INT_DecRI() are actu-
ally macros defined in RTOS.h, so they execute very quickly and are very efficient.

Prototype
void OS_INT_DecRI ( void );

Additional information
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 counter
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.

If you need to disable interrupts for a instant only where no routine is called, as in the
example above, you could also use the pair OS_INT_Disable() and OS_INT_EnableCondi-
tional(). These are slightly more efficient because the interrupt disable counter OS_DICnt
is not modified twice, but only checked once. They have the disadvantage that they do not
work with functions because the status of OS_DICnt is not actually changed, and they should
therefore be used with great care. In case of doubt, use OS_INT_IncDI() and OS_INT_De-
cRI(). You can safely call embOS API between OS_INT_IncDI() and OS_INT_DecRI(). The
embOS API will not enable interrupts.

Example

volatile long lvar;

void routine ( void ) {  
  OS_INT_IncDI();
  lvar ++;
  OS_INT_DecRI();
}

12.4.4.2 OS_INT_Disable()

Description

OS_INT_Disable() disables embOS interrupts but does not change the interrupt disable counter OS_Global.Counters.Cnt.DI.

Prototype

void OS_INT_Disable(void);

Example

```c
void Routine(void) {
    OS_INT_Disable(); // Disable embOS interrupts
    DoSomething();
    OS_INT_Enable();  // Re-enable embOS interrupts unconditionally
}
```
12.4.4.3 **OS_INT_DisableAll()**

**Description**
This function disables embOS and zero latency interrupts unconditionally.

**Prototype**

```c
void OS_INT_DisableAll(void);
```

**Additional information**

`OS_INT_DisableAll()` disables all interrupts (including zero latency interrupts) in a fast and efficient way. Note that the system does not track the interrupt state when calling the function. Therefore the function should not be called when the state is unknown. Interrupts can be re-enabled by calling `OS_INT_EnableAll()`. After calling `OS_INT_DisableAll()`, no embOS function except the interrupt enable function `OS_INT_EnableAll()` should be called, because the interrupt state is not saved by the function. An embOS API function may re-enable interrupts. The exact interrupt enable behaviour depends on the CPU.

**Example**

```c
void Routine(void) {
    OS_INT_DisableAll();  // Disable interrupts
    // Execute any code that should be executed with interrupts disabled
    // No embOS function should be called
    //
    ...
    OS_INT_EnableAll();   // Re-enable interrupts unconditionally
}
```
12.4.4.4 OS_INT_Enable()

Description

OS_INT_Enable() enables embOS interrupts but does not check the interrupt disable counter OS_Global.Counters.Cnt.DI. Refrain from using this function directly unless you are sure that the interrupt disable count has the value zero, because it does not take the interrupt disable counter into account. OS_INT_Disable() / OS_INT_Enable() can be used when no embOS API functions are called between which could enable interrupts before the actual call to OS_INT_Enable() and the interrupt disable count is zero.

Prototype

void OS_INT_Enable(void);

Example

```c
void Routine(void) {
    OS_INT_Disable(); // Disable embOS interrupts
    DoSomething();
    OS_INT_Enable();   // Re-enable embOS interrupts unconditionally
}
```
12.4.4.5 OS_INT_EnableAll()

Description
This function enables embOS and zero latency interrupts unconditionally.

Prototype

```c
void OS_INT_EnableAll(void);
```

Additional information
This function re-enables interrupts which were disabled before by a call of `OS_INT_DisableAll()`. The function re-enables embOS and zero latency interrupts unconditionally. `OS_INT_DisableAll()` and `OS_INT_EnableAll()` should be used as a pair. The call cannot be nested, because the state is not saved. This kind of global interrupt disable/enable should only be used when the interrupt enable state is well known and interrupts are enabled.

Between `OS_INT_DisableAll()` and `OS_INT_EnableAll()`, no function should be called when it is not known if the function alters the interrupt enable state.

If the interrupt state is not known, the functions `OS_INT_PreserveAll()` or `OS_INT_PreserveAndDisableAll()` and `OS_INT_RestoreAll()` shall be used as described later on.

Example

```c
void Routine(void) {
    OS_INT_DisableAll(); // Disable interrupts
    //
    // Execute any code that should be executed with interrupts disabled
    // No embOS function should be called
    //
    ...
    OS_INT_EnableAll(); // Re-enable interrupts unconditionally
}
```
12.4.4.6 OS_INT_EnableConditional()

Description
Restores the interrupt status, based on the interrupt disable counter. Interrupts are only enabled if the interrupt disable counter OS_Global.Counters.Cnt.DI is zero.

Prototype
void OS_INT_EnableConditional(void);

Additional information
You cannot safely call embOS API between OS_INT_Disable() and OS_INT_Enable()/OS_INT_EnableConditional(). The embOS API might already enable interrupts because OS_INT_Disable() does not change the interrupt disable counter. In that case please use OS_INT_IncDI() and OS_INT_DecRI() instead.

Example

volatile long lvar;

void Routine (void) {
    OS_INT_Disable();
    lvar++;
    OS_INT_EnableConditional();
}
12.4.4.7  OS_INT_IncDI()

Description
Short for Disable interrupts and Increment. Increment the counter and disables interrupts. It is important that they are used as a pair: first OS_INT_IncDI(), then OS_INT_DecRI(). OS_INT_IncDI() and OS_INT_DecRI() are actually macros defined in RTOS.h, so they execute very quickly and are very efficient.

Prototype

void OS_INT_DecRI (void);

Additional information

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 counter 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.

If you need to disable interrupts for a instant only where no routine is called, as in the example above, you could also use the pair OS_INT_Disable() and OS_INT_EnableConditional(). These are slightly more efficient because the interrupt disable counter OS_DICnt is not modified twice, but only checked once. They have the disadvantage that they do not work with functions because the status of OS_DICnt is not actually changed, and they should therefore be used with great care. In case of doubt, use OS_INT_IncDI() and OS_INT_DecRI(). You can safely call embOS API between OS_INT_IncDI() and OS_INT_DecRI(). The embOS API will not enable interrupts.

Example

```c
volatile long lvar;

void Routine (void) {
    OS_INT_IncDI();
    lvar ++;
    OS_INT_DecRI();
}
```
12.4.4.8 OS_INT_Preserve()

Description
This function can be called to preserve the current embOS interrupt enable state of the CPU.

Prototype

```c
void OS_INT_Preserve(OS_U32* pState);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pState</td>
<td>Pointer to an OS_U32 variable that receives the interrupt state.</td>
</tr>
</tbody>
</table>

Additional information

If the interrupt enable state is not known and interrupts should be disabled by a call of `OS_INT_Disable()`, the current embOS interrupt enable state can be preserved and restored later by a call of `OS_INT_Restore()`.

Example

```c
void Sample(void) {
    OS_U32 IntState;

    OS_INT_Preserve(&IntState); // Remember the interrupt enable state.
    OS_INT_Disable(); // Disable embOS interrupts
    // // Execute any code that should be executed with embOS interrupts disabled // ...
    OS_INT_Restore(&IntState); // Restore the interrupt enable state
}
```
12.4.4.9  OS_INT_PreserveAll()

Description
This function can be called to preserve the current interrupt enable state of the CPU.

Prototype

void  OS_INT_PreserveAll ( OS_U32*  pState );

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pState</td>
<td>Pointer to an OS_U32 variable that receives the interrupt state.</td>
</tr>
</tbody>
</table>

Additional information
If the interrupt enable state is not known and interrupts should be disabled by a call of OS_INT_DisableAll(), the current interrupt enable state can be preserved and restored later by a call of OS_INT_RestoreAll(). Note that the interrupt state is not stored by embOS. After disabling the interrupts using a call of OS_INT_DisableAll(), no embOS API function should be called because embOS functions might re-enable interrupts.

Example

```c
void  Sample ( void )  {
    OS_U32 IntState ;

    // Remember the interrupt enable 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
    //
    ...
    OS_INT_RestoreAll (&IntState );  // Restore the interrupt enable state
}
```
12.4.4.10  OS_INT_PreserveAndDisableAll()

Description
This function preserves the current interrupt enable state of the CPU and then disables embOS and zero latency interrupts.

Prototype

```c
void OS_INT_PreserveAndDisableAll ( OS_U32 * pState );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pState</td>
<td>Pointer to an OS_U32 variable that receives the interrupt state.</td>
</tr>
</tbody>
</table>

Additional information

The function store the current interrupt enable state into the variable pointed to by pState and then disables embOS and zero latency interrupts. The interrupt state can be restored later by a corresponding call of OS_INT_RestoreAll().

The pair of function calls OS_INT_PreserveAndDisableAll() and OS_INT_RestoreAll() can be nested, as long as the interrupt enable state is stored into an individual variable on each call of OS_INT_PreserveAndDisableAll(). This function pair should be used when the interrupt enable state is not known when interrupts shall be enabled.

Example

```c
void Sample ( void ) {
    OS_U32 IntState ;

    // Remember the interrupt enable state and disables interrupts.
    OS_INT_PreserveAndDisableAll ( &IntState );
    //
    // Execute any code that should be executed with interrupts disabled
    // No embOS function should be called
    //
    ...
    OS_INT_RestoreAll ( &IntState ); // Restore the interrupt enable state
}
```
12.4.4.11  OS_INT_Restore()

Description
This function must be called to restore the embOS interrupt enable state of the CPU which was preserved before.

Prototype

```c
void OS_INT_Restore ( OS_U32* pState );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pState</td>
<td>Pointer to an OS_U32 variable that holds the interrupt enable state.</td>
</tr>
</tbody>
</table>

Additional information

Restores the embOS interrupt enable state which was saved before by a call of `OS_INT_Preserve()`. If embOS interrupts were enabled before they were disabled, the function reenables them.

Example

```c
void Sample(void) {  
  OS_U32 IntState;

  OS_INT_Preserve(&IntState);  // Remember the interrupt enable state. 
  OS_INT_Disable();            // Disable embOS interrupts

  // Execute any code that should be executed with embOS interrupts disabled
  // ...
  OS_INT_Restore(&IntState);   // Restore the interrupt enable state
}
```
12.4.4.12 OS_INT_RestoreAll()

Description
This function must be called to restore the interrupt enable state of the CPU which was preserved before.

Prototype

```c
void OS_INT_RestoreAll ( OS_U32 * pState );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pState</td>
<td>Pointer to an OS_U32 variable that holds the interrupt enable state.</td>
</tr>
</tbody>
</table>

Additional information
Restores the interrupt enable state which was saved before by a call of OS_INT_PreserveAll() or OS_INT_PreserveAndDisableAll(). If interrupts were enabled before they were disabled globally, the function reenables them.

Example

```c
void Sample ( void )  {
    OS_U32 IntState ;

    // Remember the interrupt enable 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
    //
    ...
    OS_INT_RestoreAll (&IntState); // Restore the interrupt enable state
}
```
Chapter 13

Critical Regions
13.1 Introduction

Critical regions are program sections which should not be interrupted by another 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.

A critical region can be defined anywhere during execution of a task. They may also be declared 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

```c
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.

```c
void HPTask(void) {
    OS_TASK_EnterRegion();
    OS_TASK_Delay(100);  // OS_TASK_Delay() will cause a cooperative task switch
    OS_TASK_LeaveRegion();
}
```
## 13.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Unpriv Task</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TASK_EnterRegion()</td>
<td>Indicates the beginning of a critical region to embOS.</td>
<td><img src="chart" alt="•" /></td>
<td><img src="chart" alt="•" /></td>
<td><img src="chart" alt="•" /></td>
<td><img src="chart" alt="•" /></td>
</tr>
<tr>
<td>OS_TASK_LeaveRegion()</td>
<td>Indicates embOS the end of a critical region.</td>
<td><img src="chart" alt="•" /></td>
<td><img src="chart" alt="•" /></td>
<td><img src="chart" alt="•" /></td>
<td><img src="chart" alt="•" /></td>
</tr>
</tbody>
</table>
13.2.1 OS_TASK_EnterRegion()

Description
Indicates the beginning of a critical region to embOS.

Prototype

```c
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.
13.2.2 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

```c
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.
13.3 Disabling context transitions

The following table shows which context transitions may occur after calling appropriate embOS API:

<table>
<thead>
<tr>
<th></th>
<th>Cooperative task switch</th>
<th>Preemptive task switch</th>
<th>Software Timer</th>
<th>embOS interrupt</th>
<th>Zero latency interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular execution</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>In critical region</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>With embOS interrupts disabled</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>With all interrupts disabled</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Example

In the following example DoSomething() in the LPTask cannot be interrupt by the HPTask or the software timer SoftwareTimer. But it can be interrupted by the interrupt routines embOS_ISR and Zero_Latency_ISR.

```c
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 interrupt 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.

```c
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 14

Time Measurement
14.1 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:

```c
#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 voidUserCode(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_TIMING t;
    OS_TIME_StartMeasurement(&t);
    UserCode(); // Execute the user code to be benchmarked
    OS_TIME_StopMeasurement(&t);
    return OS_TIME_GetResultus(&t);
}

void Task(void) {
    int tLo;
    OS_U32 tHi;
    char ac[80];
    while (1) {
        tLo = BenchmarkLoRes();
        tHi = BenchmarkHiRes();
        sprintf(ac, "LoRes: %d system ticks\n", tLo);
        OS_COM_SendString(ac);
        sprintf(ac, "HiRes: %d microseconds\n", tHi);
        OS_COM_SendString(ac);
    }
}

int main(void) {
    OS_Init(); // Initialize embOS
    OS_InitHW(); // Initialize hardware for embOS
    OS_TASK_CREATE(&TCB, "HP Task", 100, Task, Stack);
    OS_Start(); // Start multitasking
    return 0;
}
```
The output of the sample is as follows:

<table>
<thead>
<tr>
<th></th>
<th>LoRes: 7 system ticks</th>
<th>HiRes: 6641 microsecond</th>
</tr>
</thead>
</table>

14.2 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.

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.
## 14.2.1 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TIME_GetTicks()</td>
<td>Returns the current system time in system ticks as a native integer value.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TIME_GetTicks32()</td>
<td>Returns the current system time in system ticks as a 32-bit integer value.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
14.2.1.1 OS_TIME_GetTicks()

**Description**

Returns the current system time in ticks as a native integer value.

**Prototype**

```c
int 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.

**Example**

```c
void PrintTask(void) {
    int Time;

    Time = OS_TIME_GetTicks();
    printf("System Time: %d\n", Time);
}
```
### 14.2.1.2 OS_TIME_GetTicks32()

**Description**

Returns the current system time in system ticks as a 32-bit integer value.

**Prototype**

```c
OS_I32 OS_TIME_GetTicks32(void);
```

**Return value**

The system variable `OS_Global.Time` as a 32-bit 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.

**Example**

```c
define PrintTask(void) {
    OS_I32 Time;

    Time = OS_TIME_GetTicks32();
    printf("System Time: %d\n", Time);
}
```
14.3 High-resolution measurement

Instead of measuring the number of completed system ticks at a given time, the hardware timer counter value is 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 MHz / 1 kHz) = 10,000. This means that with each system tick interrupt, the timer restarts at zero and counts up to 10,000.

```c
void MeasureTask ( void ) {
    OS_U64 t0 ,  t ;
    t0 = OS_TIME_GetCycles () ;
    CodeToBeTimed () ;
    t = OS_TIME_GetCycles () - t0 ;
}
```

The first call to OS_TIME_GetCycles() returns the starting value at 5,000 cycles, while the second call to OS_TIME_GetCycles() returns the ending value at 52,000 cycles. The measured execution time of the code in this example would therefore be (52,000 cycles - 5,000 cycles) = 47,000 cycles, which exactly equals 4.7 millisecond.
### 14.3.1 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TIME_ConfigSysTimer()</td>
<td>Configures the system time parameters for according timing functions. This function is usually called once from <code>OS_InitHW()</code> (implemented in <code>RTOSInit.c</code>).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_ConvertCycles2ns()</td>
<td>Converts timer cycles into nanoseconds.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_ConvertCycles2us()</td>
<td>Converts timer cycles into microseconds.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_Convertns2Cycles()</td>
<td>Converts nanoseconds into timer cycles.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_Convertus2Cycles()</td>
<td>Converts microseconds into timer cycles.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_GetCycles()</td>
<td>Returns the current system time in timer cycles as a 64-bit value.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_GetInts()</td>
<td>Returns the number of system tick interrupts.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_GetResult()</td>
<td>Returns the execution time of the code between <code>OS_TIME_StartMeasurement()</code> and <code>OS_TIME_StopMeasurement()</code> in timer cycles.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_GetResultus()</td>
<td>Returns the execution time of the code between <code>OS_TIME_StartMeasurement()</code> and <code>OS_TIME_StopMeasurement()</code> in microseconds.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_Getus()</td>
<td>Returns the current system time in microseconds as a 32-bit value.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_Getus64()</td>
<td>Returns the current system time in microseconds as a 64-bit value.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_StartMeasurement()</td>
<td>Marks the beginning of a code section to be timed.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TIME_StopMeasurement()</td>
<td>Marks the end of a code section to be timed.</td>
<td>•</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 14.3.1.1 OS_TIME_ConfigSysTimer()

**Description**

Configures the system time parameters for according timing functions. This function is usually called once from `OS_InitHW()` (implemented in RTOSInit.c).

**Prototype**

```c
void OS_TIME_ConfigSysTimer(OS_CONST_PTR OS_SYSTIMER_CONFIG *pConfig);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pConfig</td>
<td>Pointer to a data structure of type <code>OS_SYSTIMER_CONFIG</code>.</td>
</tr>
</tbody>
</table>

**The OS_SYSTIMER_CONFIG struct**

`OS_TIME_ConfigSysTimer()` uses the struct `OS_SYSTIMER_CONFIG`:

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerFreq</td>
<td>System tick hardware timer frequency in Hz</td>
</tr>
<tr>
<td>IntFreq</td>
<td>System tick hardware timer interrupt frequency in Hz</td>
</tr>
<tr>
<td>IsUpCounter</td>
<td>OS_TIMER_DOWNCOUNTING (0): for hardware timer which counts down</td>
</tr>
<tr>
<td></td>
<td>OS_TIMER_UPCOUNTING (1): for hardware timer which counts up</td>
</tr>
<tr>
<td>pfGetTimerCycles</td>
<td>Pointer to a function which returns the current hardware timer count value</td>
</tr>
<tr>
<td>pfGetTimerIntPending</td>
<td>Pointer to a function which indicates whether the hardware timer interrupt pending flag is set</td>
</tr>
</tbody>
</table>

**pfGetTimerCycles()**

**Description**

This callback function must be implemented by the user. It returns the current hardware timer count value.

**Prototype**

```c
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**

```c
unsigned int (*pfGetTimerIntPending)(void);
```
Return value

= 0  Hardware timer interrupt pending flag is not set.
≠ 0  The pending flag is set.

Example

```c
#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);
  ...
}
```
14.3.1.2   OS_TIME_ConvertCycles2ns()

**Description**
Converts timer cycles into nanoseconds.

**Prototype**

```c
OS_U64 OS_TIME_ConvertCycles2ns(OS_U32 Cycles);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>Timer cycles</td>
</tr>
</tbody>
</table>

**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().

**Example**

```c
void Task(void) {
    OS_U64 ns;

    ns = OS_TIME_ConvertCycles2ns(2000);
}
```
14.3.1.3 OS_TIME_ConvertCycles2us()

Description
Converting timer cycles into microseconds.

Prototype
OS_U64 OS_TIME_ConvertCycles2us(OS_U32 Cycles);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>Timer cycles</td>
</tr>
</tbody>
</table>

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().

Example

```c
void Task(void) {
    OS_U64 us;

    us = OS_TIME_ConvertCycles2us(2000);
}
```
14.3.1.4  OS_TIME_Convertns2Cycles()

**Description**
Converts nanoseconds into timer cycles.

**Prototype**

```c
OS_U64 OS_TIME_Convertns2Cycles(OS_U32 ns);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns</td>
<td>Nanoseconds</td>
</tr>
</tbody>
</table>

**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().

**Example**

```c
void Task(void) {
    OS_U64 Cycles;
    Cycles = OS_TIME_Convertns2Cycles(100);
}
```
14.3.1.5 OS_TIME_Convertus2Cycles()

Description
Converts microseconds into timer cycles.

Prototype

```c
OS_U64 OS_TIME_Convertus2Cycles(OS_U32 us);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>us</td>
<td>Microseconds</td>
</tr>
</tbody>
</table>

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().

Example

```c
void Task(void) {
    OS_U64 Cycles;

    Cycles = OS_TIME_Convertus2Cycles(100);
}
```
14.3.1.6 OS_TIME_GetCycles()

Description
Returns the current system time in timer cycles as a 64-bit value.

Prototype

\[
\text{OS_U64} \quad \text{OS_TIME_GetCycles}(\text{void});
\]

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.

\text{OS_TIME_GetCycles()} \text{ returns correct values only if } \text{OS_TIME_ConfigSysTimer()} \text{ was called during initialization. All embOS board support packages already call } \text{OS_TIME_ConfigSysTimer(). With this 64-bit value } \text{OS_TIME_GetCycles()} \text{ can return up to } 2^{64} - 1 \text{ cycles } = 0xFFFFFFFFFFFFFFFF \text{ cycles (} \sim 18 \text{ quintillion cycles).}

Example

\begin{verbatim}
void Task(void) {
    OS_U64 Cycles0, Cycles;
    Cycles0 = OS_TIME_GetCycles();
    DoSomething();
    Cycles = OS_TIME_GetCycles() - Cycles0;
}
\end{verbatim}
14.3.1.7  OS_TIME_GetInts()

**Description**

OS_TIME_GetInts() returns the number of system tick interrupts.

**Prototype**

```c
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.

**Example**

```c
void Task(void) {
  OS_U64 NumInts;

  NumInts = OS_TIME_GetInts();
}
```
14.3.1.8 OS_TIME_GetResult()

Description
Returns the execution time of the code between OS_TIME_StartMeasurement() and OS_TIME_StopMeasurement() in timer cycles.

Prototype
OS_U32 OS_TIME_GetResult(OS_TIMING* pCycle);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCycle</td>
<td>Pointer to a data structure of type OS_TIMING.</td>
</tr>
</tbody>
</table>

Return value
The execution time in timer cycles as a 32-bit integer value.

Additional information
Cycle length depends on the timer clock frequency. Although the function OS_TIME_GetResult() may be used for returning the execution time in cycles as above, it is typically more common to use the function OS_TIME_GetResultus(), which returns the value in microseconds. In the above example, the return value would be 4,700 microseconds.

Example

```c
static void Benchmark(void) {
    OS_TIMING 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);
}
```
### 14.3.1.9 OS_TIME_GetResultus()

**Description**
Returns the execution time of the code between OS_TIME_StartMeasurement() and OS_TIME_StopMeasurement() in microseconds.

**Prototype**
```c
OS_U32 OS_TIME_GetResultus(OS_CONST_PTR OS_TIMING *pCycle);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCycle</td>
<td>Pointer to a data structure of type OS_TIMING.</td>
</tr>
</tbody>
</table>

**Return value**
The execution time in microseconds as a 32-bit integer value.

**Example**
```c
static void Benchmark(void) {
    OS_TIMING t;
    OS_U32    us;

    OS_TIME_StartMeasurement(&t);
   UserCode();    // Execute the user code to be benchmarked
    OS_TIME_StopMeasurement(&t);
    us = OS_TIME_GetResultus(&t);
    printf("Time=%u", us);
}
```
14.3.1.10 OS_TIME_Getus()

Description
Returns the current system time in microseconds as a 32-bit value.

Prototype

```c
OS_U32 OS_TIME_Getus(void);
```

Return value
The current system time in microseconds (usec) 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_Getus() 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_Getus() can return up to $2^{32} - 1$ usec = 0xFFFFFFFF usec (~71 minutes).

Example

```c
void PrintTime(void) {
    OS_U32 Time;

    Time = OS_TIME_Getus();
    printf("System Time: %u usec\n", Time);
}
```
14.3.1.11 OS_TIME_Getus64()

Description
Returns the current system time in microseconds as a 64-bit value.

Prototype
OS_U64 OS_TIME_Getus64(void);

Return value
The current system time in microseconds (usec) 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_Getus64() 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_Getus64() can return up to $2^{64} - 1$ usec = $0xFFFFFFFFFFFFFFFF$ usec (~584,942 years).

Example

```c
void MeasureTime(void) {
    OS_U64 t0, t1;
    OS_U32 delta;

    t0 = OS_TIME_Getus64();
    DoSomething();
    t1 = OS_TIME_Getus64();
    delta = (OS_U32)(t1 - t0);
    printf("Delta: %u usec\n", delta);
}
```
14.3.1.12  OS_TIME_StartMeasurement()

Description
Marks the beginning of a code section to be timed.

Prototype

```c
void OS_TIME_StartMeasurement(OS_TIMING* pCycle);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCycle</td>
<td>Pointer to a data structure of type OS_TIMING.</td>
</tr>
</tbody>
</table>

Additional information
This function must be used with OS_TIME_StopMeasurement().

Example

```c
static void Benchmark(void) {
    OS_TIMING t;
    OS_U32    us;

    OS_TIME_StartMeasurement(&t);
   UserCode(); /* Execute the user code to be benchmarked */
    OS_TIME_StopMeasurement(&t);
    us = OS_TIME_GetResultUs(&t);
    printf("Time=%u", us);
}
```
14.3.1.13 OS_TIME_StopMeasurement()

Description
Marks the end of a code section to be timed.

Prototype

```c
void OS_TIME_StopMeasurement(OS_TIMING* pCycle);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCycle</td>
<td>Pointer to a data structure of type OS_TIMING.</td>
</tr>
</tbody>
</table>

Additional information

This function must be used with OS_TIME_StartMeasurement().

Example

```c
static void Benchmark(void) {
    OS_TIMING t;
    OS_U32    us;

    OS_TIME_StartMeasurement(&t);
   UserCode(); // Execute the user code to be benchmarked
    OS_TIME_StopMeasurement(&t);
    us = OS_TIME_GetResultus(&t);
    printf("Time=%u", us);
}
```
Chapter 15
Low Power Support
15.1 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.

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-dependant. 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:

```c
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 480.
15.2 Tickless support

15.2.1 Introduction

The embOS tickless support stops the periodic system tick interrupt during idle periods. Idle periods are periods of time when there are no tasks and no software timer ready for execution and no interrupt request is pending. Stopping the system tick allows the microcontroller to remain in a power save mode until an interrupt occurs.

The embOS tickless support comes with the functions OS_TICKLESS_GetNumIdleTicks(), OS_TICKLESS_AdjustTime(), OS_TICKLESS_Start() and OS_TICKLESS_Stop(). These can be used to add tickless support to any embOS start project.

OS_Idle()

In order to use the tickless support the OS_Idle() function needs to be modified. The default OS_Idle() function is just an endless loop which starts a power save mode:

```c
void  OS_Idle ( void )  {
    while  ( 1 )  {
        _EnterLowPowerMode ();
    }
}
```

The tickless OS_Idle() function depends on the hardware:

```c
void  OS_Idle ( void )  {
    OS_TIME IdleTicks ;
    OS_INT_Disable ();
    IdleTicks = OS_TICKLESS_GetNumIdleTicks ();
    if  ( IdleTicks > 1 )  {
        if  (( OS_U32 ) IdleTicks > TIMER1_MAX_TICKS )  {
            IdleTicks = TIMER1_MAX_TICKS ;
        }
        OS_TICKLESS_Start ( IdleTicks , & _EndTicklessMode );
        _SetHWTimer ( IdleTicks );
    }
    OS_INT_Enable ();
    while  ( 1 )  {
        _EnterLowPowerMode ();
    }
}
```

The following description explains the tickless OS_Idle() function step by step:

```c
void  OS_Idle ( void )  {
    OS_TIME IdleTicks ;
    OS_INT_Disable ();
    IdleTicks = OS_TICKLESS_GetNumIdleTicks ();
    if  ( IdleTicks > 1 )  {
        if  (( OS_U32 ) IdleTicks > TIMER_MAX_TICKS )  {
            IdleTicks = TIMER_MAX_TICKS ;
        }
        The OS_Idle() function evaluates the number of idle system ticks by calling OS_TICKLESS_GetNumIdleTicks(). The tickless mode is only used when there is more than one idle system tick: If there is one (or none) idle system tick only, the scheduler will be executed with the next system tick, hence it makes no sense to enter the tickless mode in that case.
```

```c
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 system ticks can be reduced to any lower value. For example, if `OS_TICKLESS_GetNumIdleTicks()` returns 200 idle system ticks, but the hardware timer’s duration is limited to 100 ticks maximum, 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 system 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.

```c
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 reprograms the hardware timer to generate a system tick interrupt at the time defined by `IdleTicks`.

```c
OS_INT_Enable();
while (1) {
    _EnterLowPowerMode();
}
```

Interrupts are reenabled and the CPU continually enters power save mode. `_EnterLowPowerMode()` is a hardware-dependent function that activates the power save mode.

**Callback Function**

The callback function calculates how long the processor slept in power save mode and corrects the system time accordingly.

```c
static void _EndTicklessMode(void) {
    OS_U32 NumTicks;
    if (OS_Global.TicklessExpired) {
        OS_TICKLESS_AdjustTime(OS_Global.TicklessFactor);
    } else {
        NumTicks = _GetLowPowerTicks();
        OS_TICKLESS_AdjustTime(NumTicks);
    }
    _SetHWTimer(OS_TIMER_RELOAD);
}
```

The following description explains the callback function step by step:

```c
static void _EndTicklessMode(void) {
    OS_U32 NumTicks;
    if (OS_Global.TicklessExpired) {
        OS_TICKLESS_AdjustTime(OS_Global.TicklessFactor);
    } else {
        NumTicks = _GetLowPowerTicks();
        OS_TICKLESS_AdjustTime(NumTicks);
    }
}
```

If the hardware timer expired and the system tick interrupt was executed the flag `OS_Global.TicklessExpired` is set. This can be used to determine if the system slept in power save mode for the entire idle time. If this flag is set we can use the value in `OS_Global.TicklessFactor` to adjust the system time.

```c
} else {
    NumTicks = _GetLowPowerTicks();
    OS_TICKLESS_AdjustTime(NumTicks);
}
```
_GetLowPowerTicks() is a hardware-dependent function which returns the expired idle ticks if the power save mode was interrupted by any other interrupt than the system tick. We use that value to adjust the system time.

```c
_SetHWtimer(OS_TIMER_RELOAD);
```

_SetHWtimer() is a hardware-dependent function which reprograms the hardware timer to its default value for one system tick.
## 15.2.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>Unpriv Task</th>
<th>ISR</th>
<th>SW Timer</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TICKLESS_AdjustTime()</td>
<td>Adjusts the embOS internal time variable by the specified amount of system ticks.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TICKLESS_GetNumIdleTicks()</td>
<td>Retrieves the number of embOS system ticks until the next time-scheduled action will be started.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TICKLESS_Start()</td>
<td>Start the tickless mode.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TICKLESS_Stop()</td>
<td>Prematurely stops the tickless mode.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
15.2.2.1 OS_TICKLESS_AdjustTime()

Description
Adjusts the embOS internal time variable by the specified amount of system ticks.

Prototype

```c
void OS_TICKLESS_AdjustTime(OS_TIME Time);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>The amount of time which should be added to the embOS internal time variable.</td>
</tr>
</tbody>
</table>

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 1:1, i.e. when `OS_TICK_Config()` was called before. A debug build of embOS will call `OS_Error()` in case the ratio is not 1:1.

Example

Please refer to the example described in `OS_Idle()` on page 480.
15.2.2.2 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().

Example

Please refer to the example described in OS_Idle() on page 480.
15.2.2.3  OS_TICKLESS_Start()

Description

Start the tickless mode. It sets the sleep time and the user callback function which is called from the scheduler after wakeup from power save mode.

Prototype

```c
void OS_TICKLESS_Start ( OS_TIME Time,
voidRoutine* Callback);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Time in ticks which will be spent in power save mode.</td>
</tr>
<tr>
<td>Callback</td>
<td>Callback function to stop the tickless mode.</td>
</tr>
</tbody>
</table>

Additional information

It must be called before the CPU enters a power save mode.

The callback function 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 reset the system tick timer to its default tick period.

Example

Please refer to the example described in `OS_Idle()` on page 480.
15.2.2.4 OS_TICKLESS_Stop()

Description
Prematurely stops the tickless mode.

Prototype

```c
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.
15.2.3 Frequently Asked Questions

Q: Can I use embOS without tickless support?
A: Yes, 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: \texttt{OS\_Idle()} must be modified and the callback function must be implemented. \texttt{OS\_Idle()} is part of the \texttt{RTOSInit.c} file. We suggest to implement the callback function in the same file.

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.
15.3 **Peripheral power control**

15.3.1 **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.

```c
#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()`:

```c
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()`:

```c
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 bitmask 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.

```c
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();
};
15.3.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>main</th>
<th>Priv Task</th>
<th>ISR</th>
<th>Unpriv Task</th>
<th>SW Timer</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_POWER_GetMask()</td>
<td>Retrieves the power management counter.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_POWER_UsageDec()</td>
<td>Decrements the power management counter(s).</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_POWER_UsageInc()</td>
<td>Increments the power management counter(s).</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
15.3.2.1 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.
15.3.2.2 OS_POWER_UsageDec()

**Description**
Decrement the power management counter(s).

**Prototype**

```c
void OS_POWER_UsageDec(OS_UINT Index);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Contains a mask with bits set for those counters which should be updated. (Bit 0 =&gt; Counter 0) The debug version checks for underflow, overflow and undefined counter number.</td>
</tr>
</tbody>
</table>

**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.
15.3.2.3  OS_POWER_UsageInc()

Description
Increments the power management counter(s).

Prototype
void OS_POWER_UsageInc(OS_UINT Index);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Contains a mask with bits set for those counters which should be updated. (Bit 0 =&gt; Counter 0) The debug version checks for underflow, overflow and undefined counter number.</td>
</tr>
</tbody>
</table>

Additional information
When a peripheral is in use this function is called to mark the peripheral as in use.
Chapter 16

Heap Type Memory Management
16.1 Introduction

ANSI C offers some basic dynamic memory management functions. These are 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.

Heap type memory management is part of the embOS libraries. It does not use any resources if it is not referenced by the application (that is, if the application does not use any memory management API function).

Note that another aspect of these routines may still be a problem: the memory used for the functions (known as heap) may fragment. This can lead to a situation where the total amount of memory is sufficient, but there is not enough memory available in a single block to satisfy an allocation request.

This API is not available in embOS library mode `OS_LIBMODE_SAFE`.

Example

```c
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);
    }
}
```
### 16.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Unpriv Task</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_HEAP_free()</td>
<td>Frees a block of memory previously allocated.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_HEAP_malloc()</td>
<td>Allocates a block of memory on the heap.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_HEAP_realloc()</td>
<td>Changes the allocation size.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
16.2.1 OS_HEAP_free()

Description

Frees a block of memory previously allocated. This is the thread safe free() variant.

Prototype

```c
void OS_HEAP_free(void* pMemBlock);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMemBlock</td>
<td>Pointer to a memory block previously allocated with OS_HEAP_malloc().</td>
</tr>
</tbody>
</table>

Example

```c
void UseHeapMem(void) {
    char* sText;

    sText = (char*)OS_HEAP_malloc(20);
    strcpy(sText, "Hello World");
    printf(sText);
    OS_HEAP_free(p);
}
```
16.2.2 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Size of the requested memory block in bytes.</td>
</tr>
</tbody>
</table>

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.

Example

```c
void UseHeapMem(void) {
    char* sText;

    sText = (char*)OS_HEAP_malloc(20);
    strcpy(sText, "Hello World");
    printf(sText);
    OS_HEAP_free(p);
}
```
### 16.2.3 OS_HEAP_realloc()

**Description**
Changes the allocation size.
This is the thread safe realloc() variant.

**Prototype**
```c
void *OS_HEAP_realloc(void *pMemBlock, unsigned int NewSize);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMemBlock</td>
<td>Pointer to a memory block previously allocated with OS_HEAP_malloc().</td>
</tr>
<tr>
<td>NewSize</td>
<td>New size for the memory block in bytes.</td>
</tr>
</tbody>
</table>

**Return value**
Upon successful completion, `OS_HEAP_realloc()` returns a pointer to the reallocated memory block. Otherwise, it returns a NULL pointer.

**Example**
```c
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(p);
}
```
Chapter 17

Fixed Block Size Memory Pool
17.1 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

```c
#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;
## 17.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_MEMPOOL_Alloc()</td>
<td>Requests allocation of a memory block.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MEMPOOL_AllocBlocked()</td>
<td>Allocates a memory block from pool.</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_MEMPOOL_AllocTimed()</td>
<td>Allocates a memory block from pool with a timeout.</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_MEMPOOL_Create()</td>
<td>Creates and initializes a fixed block size memory pool.</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_MEMPOOL_Delete()</td>
<td>Deletes a fixed block size memory pool.</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_MEMPOOL_Free()</td>
<td>Releases a memory block that was previously allocated.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MEMPOOL_FreeEx()</td>
<td>Releases a memory block that was previously allocated.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MEMPOOL_GetBlockSize()</td>
<td>Returns the size of a single memory block in the pool.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MEMPOOL_GetMaxUsed()</td>
<td>Returns maximum number of blocks in a pool that have been used simultaneously since creation of the pool.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MEMPOOL_GetNumBlocks()</td>
<td>Returns the total number of memory blocks in the pool.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MEMPOOL_GetNumFreeBlocks()</td>
<td>Returns the number of free memory blocks in the pool.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MEMPOOL_IsInPool()</td>
<td>Information routine to examine whether a memory block reference pointer belongs to the specified memory pool.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
17.2.1 OS_MEMPOOL_Alloc()

Description
Requests allocation of a memory block. Continues execution without blocking.

Prototype

```c
void *OS_MEMPOOL_Alloc(OS_MEMPOOL *pMEMF);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
</tbody>
</table>

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 delivered to OS_MEMPOOL_FreeEx() as parameter to free the memory block. The pointer must not be modified.

Example

```c
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.
    }
}
```
17.2.2  OS_MEMPOOL_AllocBlocked()

Description
Allocates a memory block from pool. Suspends until memory is available.

Prototype

```c
void *OS_MEMPOOL_AllocBlocked(OS_MEMPOOL * pMEMF);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
</tbody>
</table>

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 retrieved pointer must be delivered to OS_MEMPOOL_FreeEx() as a parameter to free the memory block. The pointer must not be modified.

Example

Please refer to the example in the Introduction on page 344.
17.2.3 OS_MEMPOOL_AllocTimed()

**Description**

Allocates a memory block from pool with a timeout. Suspends until memory is available or a timeout occurs.

**Prototype**

```c
void *OS_MEMPOOL_AllocTimed(OS_MEMPOOL * pMEMF,
                             OS_TIME   Timeout);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Time limit before timeout, given in system ticks. The data type OS_TIME is defined as an integer, therefore valid values are: 0 ≤ Timeout ≤ 2^{15} - 1 = 0xFFF for 8/16-bit CPUs. 0 ≤ Timeout ≤ 2^{31} - 1 = 0x7FFFFFFF for 32-bit CPUs.</td>
</tr>
</tbody>
</table>

**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 delivered to OS_MEMPOOL_FreeEx() as parameter 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.

**Example**

```c
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.
    }
}
```
17.2.4 OS_MEMPOOL_Create()

Description
Creates and initializes a fixed block size memory pool.

Prototype

```c
void OS_MEMPOOL_Create(OS_MEMPOOL * pMEMF,
                        void* pPool,
                        OS_UINT NumBlocks,
                        OS_UINT BlockSize);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
<tr>
<td>pPool</td>
<td>Pointer to memory to be used for the memory pool. Required size is: NumBlocks * BlockSize.</td>
</tr>
<tr>
<td>NumBlocks</td>
<td>Number of blocks in the pool.</td>
</tr>
<tr>
<td>BlockSize</td>
<td>Size in bytes of one block.</td>
</tr>
</tbody>
</table>

Additional information

Before using any memory pool, it must be created. A debug build of libraries keeps track of created and deleted memory pools. The release and stack-check builds do not. The maximum number of blocks and the maximum block size is for 16-Bit CPUs 0x7FFF and for 32-Bit CPUs 0x7FFFFFFF.

Example

```c
#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);
}
```
17.2.5  OS_MEMPOOL_Delete()

Description
Deleted a fixed block size memory pool. After deletion, the memory pool and memory blocks inside this pool can no longer be used.

Prototype

```c
void OS_MEMPOOL_Delete(OS_MEMPOOL* pMEMF);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
</tbody>
</table>

Additional information

This routine is provided for completeness. It is not used in the majority of applications since there is no need to dynamically create/delete memory pools. For most applications, it is suggested to have a static memory pool design: memory pools are created at startup (before calling `OS_Start()`) and never get deleted. A debug build of embOS will explicitly mark a memory pool as deleted.
17.2.6  OS_MEMPOOL_Free()

Description
Releases a memory block that was previously allocated. The memory pool does not need to be denoted.

Prototype

```c
void OS_MEMPOOL_Free(void* pMemBlock);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMemBlock</td>
<td>Pointer to the memory block.</td>
</tr>
</tbody>
</table>

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.

Example

```c
static OS_MEMPOOL _MemPool;

void Task(void) {
    void* pData;

    pData = OS_MEMPOOL_Alloc(&_MemPool); // Allocate memory
    ... // Work with allocated memory
    OS_MEMPOOL_Free(pData); // Free allocated memory
}
```
17.2.7  OS_MEMPOOL_FreeEx()

Description
Releases a memory block that was previously allocated.

Prototype

```c
void OS_MEMPOOL_FreeEx ( OS_MEMPOOL* pMEMF,
                        void* pMemBlock);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
<tr>
<td>pMemBlock</td>
<td>Pointer to memory block to free.</td>
</tr>
</tbody>
</table>

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.

Example
Please refer to the example in the *Introduction* on page 344.
### 17.2.8 OS_MEMPOOL_GetBlockSize()

**Description**

Returns the size of a single memory block in the pool.

**Prototype**

```c
int OS_MEMPOOL_GetBlockSize(OS_CONST_PTR OS_MEMPOOL *pMEMF);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
</tbody>
</table>

**Return value**

Size in bytes of a single memory block in the specified memory pool. This is the value of the parameter when the memory pool was created.

**Example**

```c
static OS_MEMPOOL _MemPool;

void PrintBlockSize(void) {
    int Size;

    Size = OS_MEMPOOL_GetBlockSize(&_MemPool);
    printf("Block Size: %d\n", Size);
}
```
17.2.9 OS_MEMPOOL_GetMaxUsed()

Description
Returns maximum number of blocks in a pool that have been used simultaneously since creation of the pool.

Prototype

```c
int OS_MEMPOOL_GetMaxUsed(OS_CONST_PTR OS_MEMPOOL *pMEMF);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
</tbody>
</table>

Return value

Maximum number of blocks in the specified memory pool that were used simultaneously since the pool was created.

Example

```c
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\%");
    }
}
```
17.2.10 OS_MEMPOOL_GetNumBlocks()

Description
Returns the total number of memory blocks in the pool.

Prototype

```c
int OS_MEMPOOL_GetNumBlocks (OS_CONST_PTR OS_MEMPOOL *pMEMF);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
</tbody>
</table>

Return value

Returns the number of blocks in the specified memory pool. This is the value that was given as parameter during creation of the memory pool.

Please refer to the example of `OS_MEMPOOL_GetMaxUsed()` or `OS_MEMPOOL_GetNumFreeBlocks()`.
17.2.11 OS_MEMPOOL_GetNumFreeBlocks()

Description
Returns the number of free memory blocks in the pool.

Prototype

\[ \text{int OS_MEMPOOL\_GetNumFreeBlocks (OS\_CONST\_PTR OS\_MEMPOOL \* pMEMF);} \]

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
</tbody>
</table>

Return value
The number of free blocks currently available in the specified memory pool.

Example

```c
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)\((\text{(float)}\text{UnusedBlocks} / \text{BlockCnt}) * 100));
    } else {
        printf("Used Memory: 0\%\n");
    }
}
```
17.2.12 OS_MEMPOOL_IsInPool()

Description
Information routine to examine whether a memory block reference pointer belongs to the specified memory pool.

Prototype

```
OS_BOOL OS_MEMPOOL_IsInPool(OS_CONST_PTR OS_MEMPOOL *pMEMF,
                              OS_CONST_PTR void *pMemBlock);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMEMF</td>
<td>Pointer to a memory pool object of type OS_MEMPOOL.</td>
</tr>
<tr>
<td>pMemBlock</td>
<td>Pointer to a memory block that should be checked.</td>
</tr>
</tbody>
</table>

Return value

- = 0  Pointer does not belong to the specified memory pool.
- ≠ 0  Pointer belongs to the specified memory pool.

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");
    }
}
```
18.1 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 Using a different timer to generate tick interrupts for embOS on page ).

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.

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 322 for further information.
### 18.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Unpriv Task</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TICK_Config()</td>
<td>Configures the tick to interrupt ratio.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_TICK_Handle()</td>
<td>Default embOS timer tick handler.</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_TICK_HandleEx()</td>
<td>Alternate tick handler that may be used instead of the default tick handler.</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_TICK_HandleNoHook()</td>
<td>Speed-optimized embOS timer tick handler without hook functionality.</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>
18.2.1 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TickFreq</td>
<td>System tick frequency in Hz</td>
</tr>
<tr>
<td>IntFreq</td>
<td>Timer interrupt frequency in Hz</td>
</tr>
</tbody>
</table>

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;

Example

```c
// The system tick occurs every 1 msec (1 kHz) but
// the timer interrupt occurs every 500 usec (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_Config_SysTimer(&SysTimerConfig);
...
    OS_TICK_Config(OS_TICK_FREQ, OS_INT_FREQ);
// Configure hardware timer to OS_INT_FREQ
// _SetHWTimer(OS_INT_FREQ);
}
```
18.2.2  OS_TICK_Handle()

Description
Default embOS timer tick handler. It assumes a 1:1 tick to interrupt ratio, i.e. one interrupt
increments the tick count by one.

Prototype

```c
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 366), these will be called by `OS_TICK_Handle()`.

Example

```c
__interrupt void SysTick_Handler(void) {
    OS_INT_EnterNestable();
    OS_TICK_Handle();
    OS_INT_LeaveNestable();
}
```
18.2.3 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

```c
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 366), 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()`.

Example

```c
__interrupt void SysTick_Handler(void) {
    OS_INT_EnterNestable();
    OS_TICK_HandleEx();
    OS_INT_LeaveNestable();
}
```
18.2.4 OS_TICK_HandleNoHook()

Description
Speed-optimized embOS timer tick handler without hook functionality.

Prototype

```c
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 366).

Example

```c
__interrupt void SysTick_Handler (void) {
    OS_INT_EnterNestable();
    OS_TICK_HandleNoHook();
    OS_INT_LeaveNestable();
}
```
18.3 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

The same functionality can be achieved with a high-priority task or a software timer with one-tick period time.

**Advantage of using a hook function**

Using a hook function is much faster than performing a task switch or activating a software timer because the hook function is directly called from the embOS timer interrupt handler and does not cause a context switch.

18.3.1 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Unpriv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TICK_AddHook()</td>
<td>Adds a tick hook handler.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TICK_RemoveHook()</td>
<td>Removes a tick hook handler.</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
18.3.1.1 OS_TICK_AddHook()

**Description**

Adds a tick hook handler.

**Prototype**

```c
void OS_TICK_AddHook ( OS_TICK_HOOK * pHook,
                        OS_TICK_HOOK_ROUTINE * pfUser );
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pHook</td>
<td>Pointer to a structure of OS_TICK_HOOK.</td>
</tr>
<tr>
<td>pfUser</td>
<td>Pointer to an OS_TICK_HOOK_ROUTINE function.</td>
</tr>
</tbody>
</table>

**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.

**Example**

```c
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);
    ...
}
```
18.3.1.2  OS_TICK_RemoveHook()

**Description**
Removes a tick hook handler.

**Prototype**

```c
void OS_TICK_RemoveHook(OS_CONST_PTR OS_TICK_HOOK *pHook);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pHook</td>
<td>Pointer to a structure of OS_TICK_HOOK.</td>
</tr>
</tbody>
</table>

**Additional information**

The function may be called to dynamically remove a tick hook function installed by a call to OS_TICK_AddHook().

**Example**

```c
static OS_TICK_HOOK _Hook;

void Task(void) {
    ...
    OS_TICK_RemoveHook(&_Hook);
    ...
}
```
Chapter 19

Debugging
19.1 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(), it does not fix the original application error.

When embOS detects a runtime error, it calls the following routine:

```c
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 simply disables further task switches and then, after re-enabling interrupts, loops forever as follows:

**Example**

```c
// Run time error reaction

void OS_Error(OS_STATUS ErrCode) {
    OS_TASK_EnterRegion(); // Avoid further task switches
    OS_Global.Counters.DI = 0u; // Allow interrupts so we can communicate
    OS_INT_Enable();
    OS_Status = ErrCode;
    while (OS_Status) {
        // Endless loop may be left by setting OS_Status to 0
    }
}
```

If you are using embOSView, you can see the value and meaning of OS_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_Status to your watch window.

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 first statement needs to be the disabling of the scheduler via OS_TASK_EnterRegion(); 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 assigned to the global variable `OS_Status`. 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.

### 19.1.1 List of error codes

<table>
<thead>
<tr>
<th>Value</th>
<th>Define</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>OS_OK</code></td>
<td>No error, everything ok.</td>
</tr>
</tbody>
</table>
| 100   | `OS_ERR_ISR_INDEX`                  | Index value out of bounds during interrupt controller initia-
|       |                                     | lization or interrupt installation.                         |
| 101   | `OS_ERR_ISR_VECTOR`                 | Default interrupt handler called, but interrupt vector not i-
|       |                                     | nitialized.                                                 |
| 102   | `OS_ERR_ISR_PRIO`                   | Wrong interrupt priority.                                   |
| 103   | `OS_ERR_WRONG_STACK`                | Wrong stack used before main().                             |
| 104   | `OS_ERR_ISR_NO_HANDLER`             | No interrupt handler was defined for this interrupt.        |
| 105   | `OS_ERR_TLS_INIT`                   | `OS_TLS_Init()` called multiple times from one task.        |
| 106   | `OS_ERR_MB_BUFFER_SIZE`             | For 16-bit CPUs, the maximum buffer size for a mailbox (64KB) exceeded. |
| 116   | `OS_ERR_EXTEND_CONTEXT`             | `OS_ExtendTaskContext()` called multiple times from one task. |
| 118   | `OS_ERR_INTERNAL`                   | `OS_ChangeTask()` called without Region Counter set (or other internal error). |
| 119   | `OS_ERR_IDLE_RETURNS`               | `OS_Idle()` must not return.                                |
| 120   | `OS_ERR_TASK_STACK`                 | Task stack overflow or invalid task stack.                  |
| 121   | `OS_ERR_SEMaphore_OVERFLOW`         | Semaphore value overflow.                                   |
| 122   | `OS_ERR_POWER_OVER`                 | Counter overflows when calling `OS_POWER_UsageInc()`.        |
| 123   | `OS_ERR_POWER_UNDER`                | Counter underflows when calling `OS_POWER_UsageDec()`.       |
| 124   | `OS_ERR_POWER_INDEX`                | Index too high, exceeds `(OS_POWER_NUM_COUNTERS - 1)`.       |
| 125   | `OS_ERR_SYS_STACK`                  | System stack overflow.                                     |
| 126   | `OS_ERR_INT_STACK`                  | Interrupt stack overflow.                                  |
| 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 over-
|       |                                     | written.                                                    |
| 133   | `OS_ERR_INV_MUTEX`                  | Control block for mutex invalid, not initialized or over-
|       |                                     | written.                                                    |
| 135   | `OS_ERR_MAILBOX_NOT1`               | One of the following 1-byte mailbox functions has been used on a multibyte mailbox: `OS_MAILBOX_Get1()`, `OS_MAIL-

<table>
<thead>
<tr>
<th>Value</th>
<th>Define</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>136</td>
<td>OS_ERR_MAILBOX_DELETE</td>
<td>OS_MAILBOX_Delete() was called on a mailbox with waiting tasks.</td>
</tr>
<tr>
<td>137</td>
<td>OS_ERR_SEMAPHORE_DELETE</td>
<td>OS_SEMAPHORE_Delete() was called on a semaphore with waiting tasks.</td>
</tr>
<tr>
<td>138</td>
<td>OS_ERR_MUTEX_DELETE</td>
<td>OS_MUTEX_Delete() was called on a mutex which is claimed by a task.</td>
</tr>
<tr>
<td>140</td>
<td>OS_ERR_MAILBOX_NOT_IN_LIST</td>
<td>The mailbox is not in the list of mailboxes as expected. Possible reasons may be that one mailbox data structure was overwritten.</td>
</tr>
<tr>
<td>142</td>
<td>OS_ERR_TASKLIST_CORRUPT</td>
<td>The OS internal task list is destroyed.</td>
</tr>
<tr>
<td>143</td>
<td>OS_ERR_QUEUE_INUSE</td>
<td>Queue in use.</td>
</tr>
<tr>
<td>144</td>
<td>OS_ERR_QUEUE_NOT_INUSE</td>
<td>Queue not in use.</td>
</tr>
<tr>
<td>145</td>
<td>OS_ERR_QUEUE_INVALID</td>
<td>Queue invalid.</td>
</tr>
<tr>
<td>146</td>
<td>OS_ERR_QUEUE_DELETE</td>
<td>A queue was deleted by a call of OS_QUEUE_Delete() while tasks are waiting at the queue.</td>
</tr>
<tr>
<td>147</td>
<td>OS_ERR_MB_INUSE</td>
<td>Mailbox in use.</td>
</tr>
<tr>
<td>148</td>
<td>OS_ERR_MB_NOT_INUSE</td>
<td>Mailbox not in use.</td>
</tr>
<tr>
<td>149</td>
<td>OS_ERR_MESSAGE_SIZE_ZERO</td>
<td>Attempt to store a message with size of zero.</td>
</tr>
<tr>
<td>150</td>
<td>OS_ERR_UNUSE_BEFORE_USE</td>
<td>OS_MUTEX.Unlock() has been called on a mutex that hasn’t been locked before.</td>
</tr>
<tr>
<td>151</td>
<td>OS_ERR_LEAVEREGION_BEFORE_ENTERREGION</td>
<td>OS_TASK_LeaveRegion() has been called before OS_TASK_EnterRegion().</td>
</tr>
<tr>
<td>152</td>
<td>OS_ERR_LEAVEINT</td>
<td>Error in OS_INT_Leave().</td>
</tr>
</tbody>
</table>
| 153   | OS_ERR_DICNT | The interrupt disable counter (OS_Global.Counters.Cnt.DI) is out of range (0-15). The counter is affected by the following API calls:  
  - OS_INT_IncDI()  
  - OS_INT_DecRI()  
  - OS_INT_Enter()  
  - OS_INT_Leave() |
<p>| 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(). |</p>
<table>
<thead>
<tr>
<th>Value</th>
<th>Define</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>156</td>
<td>OS_ERR_MUTEX_OWNER</td>
<td>OS_MUTEX_Unlock() has been called from a task which does not own the mutex.</td>
</tr>
<tr>
<td>157</td>
<td>OS_ERR_REGIONCNT</td>
<td>The Region counter overflows (&gt;255).</td>
</tr>
<tr>
<td>158</td>
<td>OS_ERR_DELAYUS_INTERRUPT_T_DISABLED</td>
<td>OS_TASK_Delayus() called with interrupts disabled.</td>
</tr>
<tr>
<td>159</td>
<td>OS_ERR_MUTEX_OVERFLOW</td>
<td>Illegal function call in a software timer routine: A routine that must not be called from within a software timer has been called from within a software timer.</td>
</tr>
<tr>
<td>160</td>
<td>OS_ERR_ILLEGAL_IN_ISR</td>
<td>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.</td>
</tr>
<tr>
<td>161</td>
<td>OS_ERR_ILLEGAL_IN_TIMER</td>
<td>Not a legal API outside interrupt.</td>
</tr>
<tr>
<td>162</td>
<td>OS_ERR_ILLEGAL_OUT_ISR</td>
<td>OS_INT_Enter() has been called, but CPU is not in ISR state.</td>
</tr>
<tr>
<td>163</td>
<td>OS_ERR_NOT_IN_ISR</td>
<td>OS_INT_Enter() has not been called, but CPU is in ISR state.</td>
</tr>
<tr>
<td>164</td>
<td>OS_ERR_IN_ISR</td>
<td>OS_INT_Enter() has not been called, but CPU is in ISR state.</td>
</tr>
<tr>
<td>165</td>
<td>OS_ERR_INIT_NOT_CALLED</td>
<td>embOS API called from ISR with high priority.</td>
</tr>
<tr>
<td>166</td>
<td>OS_ERR_CPU_STATE_ISR_ILLEGAL</td>
<td>CPU runs in illegal mode.</td>
</tr>
<tr>
<td>167</td>
<td>OS_ERR_CPU_STATE_UNKNOWN</td>
<td>CPU runs in unknown mode or mode could not be read.</td>
</tr>
<tr>
<td>168</td>
<td>OS_ERR_CPU_STATE.flexible TICKLESS</td>
<td>Task control block has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>169</td>
<td>OS_ERR_CPU_STATE.flexible TICKLESS_WITH_FRACTIONAL_TICK</td>
<td>Semaphore has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>170</td>
<td>OS_ERR_2USE_TASK</td>
<td>Mutex has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>171</td>
<td>OS_ERR_2USE_TIMER</td>
<td>Mutex has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>172</td>
<td>OS_ERR_2USE_MAILBOX</td>
<td>Mutex has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>173</td>
<td>OS_ERR_2USE_SEMAPHORE</td>
<td>Event object has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>174</td>
<td>OS_ERR_2USE_SEMAPHORE</td>
<td>Event object has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>175</td>
<td>OS_ERR_2USE_MUTEX</td>
<td>Event object has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>176</td>
<td>OS_ERR_2USE_MEMF</td>
<td>Event object has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>177</td>
<td>OS_ERR_2USE_QUEUE</td>
<td>Event object has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>178</td>
<td>OS_ERR_2USE_WATCHDOG</td>
<td>Watchdog has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>179</td>
<td>OS_ERR_2USE_WATCHDOG</td>
<td>Watchdog has been initialized by calling a create function twice.</td>
</tr>
<tr>
<td>180</td>
<td>OS_ERR_NESTED_RX_INT</td>
<td>OS_Rx interrupt handler for embOSView is nested. Disable nestable interrupts.</td>
</tr>
<tr>
<td>Value</td>
<td>Define</td>
<td>Explanation</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>185</td>
<td>OS_ERR_SPINLOCK_INV_CORE</td>
<td>Invalid core ID specified for accessing a OS_SPINLOCK_SW struct.</td>
</tr>
<tr>
<td>190</td>
<td>OS_ERR_MEMF_INV</td>
<td>Fixed size memory block control structure not created before use.</td>
</tr>
<tr>
<td>191</td>
<td>OS_ERR_MEMF_INV_PTR</td>
<td>Pointer to memory block does not belong to memory pool on Release.</td>
</tr>
<tr>
<td>192</td>
<td>OS_ERR_MEMF_PTR_FREE</td>
<td>Pointer to memory block is already free when calling OS_MEMPOOL_Release(). Possibly, same pointer was released twice.</td>
</tr>
<tr>
<td>193</td>
<td>OS_ERR_MEMF_RELEASE</td>
<td>OS_MEMPOOL_Release() was called for a memory pool, that had no memory block allocated (all available blocks were already free before).</td>
</tr>
<tr>
<td>194</td>
<td>OS_ERR_MEMF_POOLADDR</td>
<td>OS_MEMPOOL_Create() was called with a memory pool base address which is not located at a word aligned base address.</td>
</tr>
<tr>
<td>195</td>
<td>OS_ERR_MEMF_BLOCKSIZE</td>
<td>OS_MEMPOOL_Create() was called with a data block size which is not a multiple of processors word size.</td>
</tr>
<tr>
<td>200</td>
<td>OS_ERR_SUSPEND_TOO_OFTEN</td>
<td>Nested call of OS_TASK_Suspend() exceeded OS_MAX_SUSPEND_CNT.</td>
</tr>
<tr>
<td>201</td>
<td>OS_ERR_RESUME_BEFORE_SUSPEND</td>
<td>OS_TASK_Resume() called on a task that was not suspended.</td>
</tr>
<tr>
<td>202</td>
<td>OS_ERR_TASK_PRIORITY</td>
<td>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.</td>
</tr>
<tr>
<td>203</td>
<td>OS_ERR_TASK_PRIORITY_INVALID</td>
<td>The value 0 was used as task priority.</td>
</tr>
<tr>
<td>205</td>
<td>OS_ERR_TIMER_PERIOD_INVALID</td>
<td>The value 0 was used as timer period.</td>
</tr>
<tr>
<td>210</td>
<td>OS_ERR_EVENT_INVALID</td>
<td>An OS_EVENT object was used before it was created.</td>
</tr>
<tr>
<td>212</td>
<td>OS_ERR_EVENT_DELETE</td>
<td>An OS_EVENT object was deleted with waiting tasks.</td>
</tr>
<tr>
<td>220</td>
<td>OS_ERR_WAITLIST_RING</td>
<td>This error should not occur. Please contact the support.</td>
</tr>
<tr>
<td>221</td>
<td>OS_ERR_WAITLIST_PREV</td>
<td>This error should not occur. Please contact the support.</td>
</tr>
<tr>
<td>222</td>
<td>OS_ERR_WAITLIST_NEXT</td>
<td>This error should not occur. Please contact the support.</td>
</tr>
<tr>
<td>223</td>
<td>OS_ERR_TICKHOOK_INVALID</td>
<td>Invalid tick hook.</td>
</tr>
<tr>
<td>224</td>
<td>OS_ERR_TICKHOOK_FUNC_INVALID</td>
<td>Invalid tick hook function.</td>
</tr>
<tr>
<td>225</td>
<td>OS_ERR_NOT_IN_REGION</td>
<td>A function was called without declaring the necessary critical region.</td>
</tr>
<tr>
<td>226</td>
<td>OS_ERR_ILLEGAL_IN_MAIN</td>
<td>Not a legal API call from main().</td>
</tr>
<tr>
<td>227</td>
<td>OS_ERR_ILLEGAL_IN_TASK</td>
<td>Not a legal API after OS_Start().</td>
</tr>
<tr>
<td>228</td>
<td>OS_ERR_ILLEGAL_AFTER_OSS_TART</td>
<td>Not a legal API after OS_Start().</td>
</tr>
<tr>
<td>Value</td>
<td>Define</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>230</td>
<td>OS_ERR_NON_ALIGNED_INVALIDE</td>
<td>Cache invalidation needs to be cache line aligned.</td>
</tr>
<tr>
<td>234</td>
<td>OS_ERR_HW_NOT_AVAILABLE</td>
<td>Hardware unit is not implemented or enabled.</td>
</tr>
<tr>
<td>235</td>
<td>OS_ERR_NON_TIMERCY-CLES_FUNC</td>
<td>Callback function for timer counter value has not been set. Required by OS_TIME_Getus().</td>
</tr>
<tr>
<td>236</td>
<td>OS_ERR_NON_TIMERINT-PENDING_FUNC</td>
<td>Callback function for timer interrupt pending flag has not been set. Required by OS_TIME_Getus().</td>
</tr>
<tr>
<td>240</td>
<td>OS_ERR_MPU_NOT_PRESENT</td>
<td>MPU unit not present in the device.</td>
</tr>
<tr>
<td>241</td>
<td>OS_ERR_MPU_INVALID_REGION</td>
<td>Invalid MPU region index number.</td>
</tr>
<tr>
<td>242</td>
<td>OS_ERR_MPU_INVALID_SIZE</td>
<td>Invalid MPU region size.</td>
</tr>
<tr>
<td>243</td>
<td>OS_ERR_MPU_INVALID_PERMISSION</td>
<td>Invalid MPU region permission.</td>
</tr>
<tr>
<td>244</td>
<td>OS_ERR_MPU_INVALID_ALIGNMENT</td>
<td>Invalid MPU region alignment.</td>
</tr>
<tr>
<td>245</td>
<td>OS_ERR_MPU_INVALID_OBJECT</td>
<td>OS object is directly accessible from the task which is not allowed.</td>
</tr>
<tr>
<td>250</td>
<td>OS_ERR_CONFIG_OSSTOP</td>
<td>OS_Stop() is called without using OS_ConfigStop() before.</td>
</tr>
<tr>
<td>251</td>
<td>OS_ERR_OSSTOP_BUFFER</td>
<td>Buffer is too small to hold a copy of the main() stack.</td>
</tr>
<tr>
<td>253</td>
<td>OS_ERR_VERSION_MISMATCH</td>
<td>OS library and RTOS have different version numbers. Please ensure both are from the same embOS shipment.</td>
</tr>
</tbody>
</table>

### 19.1.2 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**

```c
#define OS_ERR_APPL (0x02u)

void UserAppFunc(void) {
    int r;
    r = DoSomething();
    if (r == 0) {
        OS_Error(OS_ERR_APPL)
    }
}
```
19.2 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 OS_LIBMODE_XR) which is set at task creation. It can be helpful to have human readable identifiers for other OS objects, as well.

Example

```c
static OS_MAILBOX Mailbox;
static OS_OBJNAME MailboxName;
static char Buffer[100];

OS_MAILBOX_Create(&Mailbox, 10, 10, &Buffer);
OS_DEBUG_SetObjName(&MailboxName, &Mailbox, "My Mailbox");
```

With the following API you can easily add human readable identifiers to an unlimited amount of OS objects. Human readable object identifiers are not supported in embOS library mode OS_LIBMODE_XR.

19.2.1 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>main</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_DEBUG_GetObjName()</td>
<td>Returns the name of an OS object.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_DEBUG_SetObjName()</td>
<td>Sets an OS object name.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_DEBUG_RemoveObjName()</td>
<td>Removes an OS object name.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
19.2.1.1 OS_DEBUG_SetObjName()

Description
Sets an OS object name.

Prototype

```c
void OS_DEBUG_SetObjName ( OS_OBJNAME * pObjName ,
                           OS_CONST_PTR void * pOSObjID ,
                           OS_CONST_PTR char * sName );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pObjName</td>
<td>Pointer to a OS_OBJNAME control structure.</td>
</tr>
<tr>
<td>pOSObjID</td>
<td>ID of the OS object.</td>
</tr>
<tr>
<td>sName</td>
<td>Name of the OS object.</td>
</tr>
</tbody>
</table>

Additional information

With OS_DEBUG_SetObjName() every OS 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. This function is not available in OS_LIBMODE_XR.

Example

```c
#include "RTOS.h"
#include <stdio.h>

static OS_STACKPTR int StackHP[128];
static OS_TASK TCBHP;
static OS_MAILBOX Mailbox;
static OS_OBJNAME MailboxName;
static char Buffer[100];

static void HPTask(void) {
    const char* s;
    s = OS_DEBUG_GetObjName(&Mailbox);
    printf(s);

    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_MAILBOX_Create(&Mailbox, 10, 10, &Buffer);
    OS_DEBUG_SetObjName(&MailboxName, &Mailbox, "My Mailbox");
    OS_Start(); // Start embOS
    return 0;
}
```
19.2.1.2 OS_DEBUG_GetObjName()

Description
Returns the name of an OS object.

Prototype

```c
char *OS_DEBUG_GetObjName(OS_CONST_PTR void *pOSObjID);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pOSObjID</td>
<td>Pointer to the OS object.</td>
</tr>
</tbody>
</table>

Return value

= NULL Name was not set for this object.
≠ NULL Pointer to the OS object name.

Additional information

OS_DEBUG_GetObjName() returns the object name which was set before with OS_DEBUG_SetObjName(). This function is not available in OS_LIBMODE_XR.

Example

```c
#include "RTOS.h"
#include <stdio.h>

static OS_STACKPTR int StackHP[128];
static OS_TASK TCBHP;
static OS_MAILBOX Mailbox;
static OS_OBJNAME MailboxName;
static char Buffer[100];

static void HPTask(void) {
    const char* s;
    s = OS_DEBUG_GetObjName(&Mailbox);
    printf(s);
    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_MAILBOX_Create(&Mailbox, 10, 10, &Buffer);
    OS_DEBUG_SetObjName(&MailboxName, &Mailbox, "My Mailbox");
    OS_Start(); // Start embOS
    return 0;
}
```
19.2.1.3 OS_DEBUG_RemoveObjName()

Description
Removes an OS object name.

Prototype

```c
void OS_DEBUG_RemoveObjName(OS_CONST_PTR OS_OBJNAME * pObjName);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pObjName</td>
<td>Pointer to a OS_OBJNAME control structure.</td>
</tr>
</tbody>
</table>

Additional information

OS_DEBUG_RemoveObjName() removes the object name which was set before with OS_DEBUG_SetObjName(). This function is not available in OS_LIBMODE_XR.

Example

```c
#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 Buffer[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, &Buffer);
    OS_DEBUG_SetObjName(&MailboxName, &Mailbox, "My Mailbox");
    OS_Start();  // Start embOS
    return 0;
}
```
19.3  embOS API trace

embOS supports API trace in two different ways. embOS API trace can be used with embOSView which is described in the chapter embOSView.

On the other hand embOS API trace can be used with any other tool like SystemView. To do so the embOS API functions call specific routines which for example store trace events in a memory location. Function pointers to these routines are stored in a structure which is passed as a parameter to OS_TRACE_SetAPI(). These routines are called from the embOS API functions only when profiling is enabled.

Example

```c
void SEGGER_SYSVIEW_Conf ( void ) {
    ..
    OS_TRACE_SetAPI(&embOS_TraceAPI_SYSVIEW);  // Configure embOS to use SYSVIEW.
    ..
}
```

Trace structure

```c
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;
```
### 19.3.1 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TRACE_SetAPI()</td>
<td>Sets the pointer to the trace function table.</td>
<td>•</td>
<td>•</td>
<td>• •</td>
</tr>
</tbody>
</table>
19.3.1.1 OS_TRACE_SetAPI()

Description
Sets the pointer to the trace function table. OS_TRACE_SetAPI() can be used for embOS API trace.

Prototype

```c
void OS_TRACE_SetAPI(OS_CONST_PTR OS_TRACE_API *pTraceAPI);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTraceAPI</td>
<td>Pointer to trace function table.</td>
</tr>
</tbody>
</table>

Example

```c
void SEGGER_SYSVIEW_Conf(void) {
    ..
    OS_TRACE_SetAPI(&embOS_TraceAPI_SYSVIEW); // Configure embOS to use SYSVIEW.
    ..
}
```
Chapter 20

Profiling
20.1 Introduction

This chapter explains the profiling functions that can be used by an application. In software engineering, profiling (“program profiling”, “software profiling”) is a form of dynamic program analysis that measures, for example, the time complexity of a program and duration of function calls.

Example

```c
#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_Delayus(500); // Do something.
        OS_TASK_Delay(1); // Give other tasks a chance to run.
    }
}

static void LPTask(void) {
    while (1) {
        OS_TASK_Delayus(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.
## 20.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_STAT_AddLoadMeasurement()</td>
<td>Initializes the periodic CPU load measurement.</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_STAT_AddLoadMeasurementEx()</td>
<td>Initializes the periodic CPU load measurement.</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_STAT_Disable()</td>
<td>Disables the kernel profiling.</td>
<td>●</td>
<td>●</td>
<td>●●</td>
</tr>
<tr>
<td>OS_STAT_Enable()</td>
<td>Enables the kernel profiling (for an indefinite time).</td>
<td>●</td>
<td>●</td>
<td>●●</td>
</tr>
<tr>
<td>OS_STAT_GetExecTime()</td>
<td>Returns the total task execution time.</td>
<td>●</td>
<td>●</td>
<td>●●</td>
</tr>
<tr>
<td>OS_STAT_GetLoad()</td>
<td>Calculates the current task’s CPU load in permille.</td>
<td>●</td>
<td>●</td>
<td>●●</td>
</tr>
<tr>
<td>OS_STAT_GetLoadMeasurement()</td>
<td>Retrieves the result of the CPU load measurement.</td>
<td>●</td>
<td>●</td>
<td>●●</td>
</tr>
<tr>
<td>OS_STAT_GetNumActivations()</td>
<td>Return the number of task activations.</td>
<td>●</td>
<td>●</td>
<td>●●●</td>
</tr>
<tr>
<td>OS_STAT_GetNumPreemptions()</td>
<td>Return the number of task preemptions.</td>
<td>●</td>
<td>●</td>
<td>●●●</td>
</tr>
<tr>
<td>OS_STAT_Sample()</td>
<td>Starts the kernel profiling and calculates the absolute task run time for all tasks since the last call to OS_STAT_Sample().</td>
<td>●</td>
<td>●</td>
<td>●●●</td>
</tr>
</tbody>
</table>
20.2.1 OS_STAT_AddLoadMeasurement()

Description

Initializes the periodic CPU load measurement. May be used to start the calculation of the total CPU load of an application.

Note

OS_STAT_AddLoadMeasurement() starts a CPU load task with a predefined task stack size of 48 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.

Prototype

```c
void OS_STAT_AddLoadMeasurement ( int     Period ,
                                  OS_U8  AutoAdjust ,
                                  OS_I32 DefaultMaxValue );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period</td>
<td>Measurement period in embOS system ticks.</td>
</tr>
<tr>
<td>AutoAdjust</td>
<td>If not zero, the measurement is autoadjusted once initially.</td>
</tr>
<tr>
<td>DefaultMaxValue</td>
<td>May be used to set a default counter value when AutoAdjust is not used.</td>
</tr>
</tbody>
</table>

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:

```c
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.
20.2.1.1 OS_IdleCnt

Description
This global variable 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().

Declaration

```c
volatile OS_I32 OS_IdleCnt;
```

Additional information
The appropriate DefaultMaxValue may, for example, be examined prior to creating any other task, similar to the given sample below:

```c
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. */
}
```
### 20.2.2 OS_STAT_AddLoadMeasurementEx()

#### Description

Initializes the periodic CPU load measurement. 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().

#### Prototype

```c
void OS_STAT_AddLoadMeasurementEx(int Period,
                                  OS_U8 AutoAdjust,
                                  OS_I32 DefaultMaxValue,
                                  void *pStack,
                                  OS_UINT StackSize);
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period</td>
<td>Measurement period in embOS system ticks.</td>
</tr>
<tr>
<td>AutoAdjust</td>
<td>If not zero, the measurement is autoadjusted once initially.</td>
</tr>
<tr>
<td>DefaultMaxValue</td>
<td>May be used to set a default counter value when AutoAdjust is not used. (See additional information)</td>
</tr>
<tr>
<td>pStack</td>
<td>Pointer to the stack.</td>
</tr>
<tr>
<td>StackSize</td>
<td>Size of the stack.</td>
</tr>
</tbody>
</table>

#### Additional information

Please refer to the description of OS_STAT_AddLoadMeasurement() for more details. This function is not available in OS_LIBMODE_SAFE.
20.2.3   OS_STAT_Disable()

Description
Disables the kernel profiling.

Prototype

```c
void OS_STAT_Disable(void);
```

Additional information
The function OS_STAT_Enable() may be used to start profiling.
20.2.4 OS_STAT_Enable()

**Description**

Enables the kernel profiling (for an indefinite time).

**Prototype**

```c
void OS_STAT_Enable(void);
```

**Additional information**

The function `OS_STAT_Disable()` may be used to stop profiling.
20.2.5  **OS_STAT_GetExecTime()**

**Description**

Returns the total task execution time.

**Prototype**

```c
OS_U32 OS_STAT_GetExecTime(OS_CONST_PTR OS_TASK * pTask);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block.</td>
</tr>
</tbody>
</table>

**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 `pTask` is a NULL pointer, the function returns the total task execution time of the currently running task. If `pTask` does not specify a valid task, a debug build of embOS calls `OS_Error()`.

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.

**Example**

```c
OS_U32 ExecTime;

void MyTask(void) {
    OS_STAT_Enable();
    while (1) {
        ExecTime = OS_STAT_GetExecTime(NULL);
        OS_TASK_Delay(100);
    }
}
```
20.2.6 OS_STAT_GetLoad()

Description
Calculates the current task’s CPU load in permille.

Prototype

```c
int OS_STAT_GetLoad(OS_CONST_PTR OS_TASK *pTask);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block.</td>
</tr>
</tbody>
</table>

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.
20.2.7  OS_STAT_GetLoadMeasurement()

Description
Retrieves the result of the CPU load measurement.

Prototype

```c
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()`.

20.2.7.1  OS_CPU_Load

Description
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.

Declaration

```c
volatile OS_INT OS_CPU_Load;
```

Additional information
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_LIBMOD-E_SAFE`. 
20.2.8 OS_STAT_GetNumActivations()

Description
Return the number of task activations.

Prototype
OS_U32 OS_STAT_GetNumActivations(OS_CONST_PTR OS_TASK *pTask);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to task control block.</td>
</tr>
</tbody>
</table>

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.

Example

```c
void Task(void) {
    OS_U32 NumActivations;

    NumActivations = OS_STAT_GetNumActivations();
}
```
20.2.9 OS_STAT_GetNumPreemptions()

Description
Return the number of task preemptions.

Prototype
OS_U32 OS_STAT_GetNumPreemptions(OS_CONST_PTR OS_TASK *pTask);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to task control block.</td>
</tr>
</tbody>
</table>

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.

Example

```c
void Task(void) {
    OS_U32 NumPreemptions;
    NumPreemptions = OS_STAT_GetNumPreemptions();
}
```
20.2.10 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

```c
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.
Chapter 21
embOSView
21.1 Introduction

The embOSView utility is a helpful tool for the analysis of the running target application. It is shipped as `embOSView.exe` with embOS 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, and DCC for ARM7/9 and Cortex-A CPUs. The hardware dependent routines and defines available for communication with embOSView are implemented inside the source file `RTOSInit.c`. Details on how to modify this file are also included in How to change settings on page .

The communication API is not available in embOS library mode `OS_LIBMODESAFE`. 
21.1.1 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 used in your application.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Builds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prio</td>
<td>Current priority of task.</td>
<td>All</td>
</tr>
<tr>
<td>Id</td>
<td>Task ID, which is the address of the task control block.</td>
<td>All</td>
</tr>
<tr>
<td>Name</td>
<td>Name assigned during creation.</td>
<td>All</td>
</tr>
<tr>
<td>Status</td>
<td>Current state of task (ready, executing, delay, reason for suspension).</td>
<td>All</td>
</tr>
<tr>
<td>Data</td>
<td>Depends on status.</td>
<td>All</td>
</tr>
<tr>
<td>Timeout</td>
<td>Time of next activation.</td>
<td>All</td>
</tr>
<tr>
<td>Stack</td>
<td>Used stack size/max. stack size/stack location.</td>
<td>S, SP, D, DP, DT</td>
</tr>
<tr>
<td>CPULoad</td>
<td>Percentage CPU load caused by task.</td>
<td>SP, DP, DT</td>
</tr>
<tr>
<td>Run Count</td>
<td>Number of activations since reset.</td>
<td>SP, DP, DT</td>
</tr>
<tr>
<td>Time slice</td>
<td>Round-robin time slice.</td>
<td>All</td>
</tr>
</tbody>
</table>

The Task list window is helpful in analysis of stack usage and CPU load for every running task.

21.1.2 System variables window

embOSView shows the state of major system variables in the System variables window. The information shown also depends on the library used by your application:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Builds</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_VERSION</td>
<td>Current version of embOS.</td>
<td>All</td>
</tr>
<tr>
<td>CPU</td>
<td>Target CPU and compiler.</td>
<td>All</td>
</tr>
<tr>
<td>LibMode</td>
<td>Library mode used for target application.</td>
<td>All</td>
</tr>
<tr>
<td>OS_Time</td>
<td>Current system time in system ticks.</td>
<td>All</td>
</tr>
<tr>
<td>OS_NumTasks</td>
<td>Current number of defined tasks.</td>
<td>All</td>
</tr>
<tr>
<td>OS_Status</td>
<td>Current error code (or O.K.).</td>
<td>All</td>
</tr>
<tr>
<td>OS_pActiveTask</td>
<td>Active task that should be running.</td>
<td>SP, D, DP, DT</td>
</tr>
<tr>
<td>OS_pCurrentTask</td>
<td>Actual currently running task.</td>
<td>SP, D, DP, DT</td>
</tr>
<tr>
<td>SysStack</td>
<td>Used size/max. size/location of system stack.</td>
<td>SP, DP, DT</td>
</tr>
<tr>
<td>IntStack</td>
<td>Used size/max. size/location of interrupt stack.</td>
<td>SP, DP, DT</td>
</tr>
<tr>
<td>TraceBuffer</td>
<td>Current count/maximum size and current state of trace buffer.</td>
<td>All trace builds</td>
</tr>
</tbody>
</table>
21.2 Setup embOSView for communication

When the communication to embOSView is enabled in the target application, embOSView can be used to analyze the running application. The communication channel of embOSView has to be setup according to the communication channel which was selected in the project.

21.2.1 Select a UART for communication

Start embOSView and chose menu Setup:

In the Communication tab, choose UART in the Type selection listbox.
In the Host interface box, select the baud rate for communication and the COM port of the PC which should be connected to the target board. The default baud rate of all projects is 38,400. The COM port list box lists all currently available COM ports of the PC.
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.
21.2.2 Select J-Link for communication

embOS supports communication channel to embOSView which uses J-Link to communicate with the running application. embOSView version 3.82g 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:

![Setup menu for J-Link communication](image)

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 listbox.

In the Host interface box, select the USB or TCP/IP channel 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, which may be a JTAG, SWD or FINE connection.

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 a 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 the 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 (which is always 0). Upon selecting the position, the according IR len is determined automatically, which should succeed for most target devices. IR len can 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 5.3 “JTAG interface”).
21.2.3 Select Ethernet for communication

embOS supports communication channel to embOSView which uses Ethernet to communicate with the running application. embOS/IP, or a different TCP/IP 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 listbox.
In the Host interface box, select the IP address of your target and the port number 50021.
21.2.4 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 the same in the debugger settings and in the 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 set the same target interface 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.

21.2.5 Restrictions for using J-Link with embOSView

With the current version of embOSView, J-Link communication with Cortex-M (memory access) can only be used when the Cortex-M vector table of the target application is located at address 0x0.
21.3 Setup target for communication

The communication to embOSView can be enabled by setting the compile time switch OS_VIEW_IFSELECT to an interface define which may be defined in 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 files the OS_VIEW_IFSELECT switch is set to a specific interface if not defined as project option.

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 when using the Debug configuration, and is disabled when using the Release configuration.

<table>
<thead>
<tr>
<th>OS_VIEW_IFSELECT</th>
<th>Communication interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_VIEW_DISABLED</td>
<td>Disabled</td>
</tr>
<tr>
<td>OS_VIEW_IF_UART</td>
<td>UART</td>
</tr>
<tr>
<td>OS_VIEW_IF_JLINK</td>
<td>J-Link</td>
</tr>
<tr>
<td>OS_VIEW_IF_ETHERNET</td>
<td>Ethernet</td>
</tr>
</tbody>
</table>

21.3.0.1 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.

21.3.0.2 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.

21.3.0.3 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 embOS/IP 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, but are available on request. The RTOSInit.c needs to be updated with the section below:

```c
#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
 */
```
Using a different TCP/IP stack than embOS/IP requires modifications to `UDP_Process.c`.

**Note**

For embOS V5.06 and later `OS_TIME_ConfigSysTimer()` must be called before using `embOSView`. 

```c
void OS_COM_Init(void) {
    UDP_Process_Init();
}
#endif
```
### 21.3.1 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>main</th>
<th>Unpriv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_COM_ClearTxActive()</td>
<td>Resets the embOS communication variables.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_COM_GetNextChar()</td>
<td>This routine is used to retrieve the next character to be sent.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_COM_OnRx()</td>
<td>OS_COM_OnRx() informs embOS about a received byte from embOSView.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS_COM_OnTx()</td>
<td>OS_COM_OnTx() returns whether there are more bytes to send.</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 21.3.1.1 OS_COM_ClearTxActive()

**Description**

Resets the embOS communication variables.

**Prototype**

```c
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**

```c
void ResetCom(void) {
    OS_COM_ClearTxActive();
}
```
21.3.1.2  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

```c
void OS_ISR_Tx(void) {
    if (OS_COM_GetNextChar() >= 0u) {
        SendByte(c);
    }
}
```
21.3.1.3  OS_COM_OnRx()

**Description**

OS_COM_OnRx() informs embOS about a received byte from embOSView. This routine is normally called from the rx interrupt service handler when a character was received.

**Prototype**

```c
void OS_COM_OnRx(OS_U8 Data);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Received byte.</td>
</tr>
</tbody>
</table>

**Example**

```c
void OS_ISR_Rx(void) {
    OS_U8 c;
    c = UART_RX_REGISTER;
    OS_COM_OnRx(c);
}
```
21.3.1.4  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

```c
void OS_ISR_Tx(void) {
  if (OS_COM_OnTx() != 0u) {
    UART_TX_INT_ENABLE_REGISTER = 0;
  }
}
```
21.4  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.

21.4.1  API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Tmain</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_COM_SendString()</td>
<td>Sends a string to the embOSView terminal window.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>OS_COM_SetRxCallback()</td>
<td>Sets a callback hook to a routine for receiving one character from embOSView.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
</tbody>
</table>
21.4.1.1 OS_COM_SendString()

**Description**
Sends a string to the embOSView terminal window.

**Prototype**

```c
void OS_COM_SendString(const char* s);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Pointer to a null-terminated string that should be sent to the terminal window.</td>
</tr>
</tbody>
</table>

**Additional information**

This function utilizes the target-specific function `OS_COM_Send1()`.
21.4.1.2 OS_COM_SetRxCallback()

Description
Sets a callback hook to a routine for receiving one character from embOSView.

Prototype

OS_RX_CALLBACK *OS_COM_SetRxCallback(OS_RX_CALLBACK* cb);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cb</td>
<td>Pointer to the application routine that should be called when one character is received over the serial interface.</td>
</tr>
</tbody>
</table>

Return value
This is the pointer to the callback function that was hooked before the call.

Additional information
The user function is called from embOS. The received character is passed as parameter. See the example below.

The callback function is defined as:

```c
typedef void OS_RX_CALLBACK (OS_U8 Data);
```

Example

```c
static void _OnRx(OS_U8 Data) { // Callback to be called from Rx-interrupt
    DisplayChar(Data);
}

void main(void) {
    ...
    OS_COM_SetRxCallback(&_OnRx);
    ...
}
```
21.5 embOS 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:

```
<table>
<thead>
<tr>
<th>Trace</th>
<th>Time</th>
<th>TaskId</th>
<th>TaskName</th>
<th>APIName</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>OS_TASK_AddTerminateHook(void)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>OS_TASK_Delay(1)</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>Task deactivation(void)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0x2000F08</td>
<td>LP Task</td>
<td>Task activation(void)</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0x2000F08</td>
<td>LP Task</td>
<td>OS_TASK_Delay(10)</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0x2000F08</td>
<td>LP Task</td>
<td>Task deactivation(void)</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>Task activation(void)</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>OS_TASK_DelayUntil(10)</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>Task deactivation(void)</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>Task activation(void)</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>OS_TASK_DelayUs(10)</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>OS_TASK_GetNumTasks(void)</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>OS_TASK_GetPrio(void)</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>0x2000EAC</td>
<td>HPTask</td>
<td>OS_TASK_GetSuspendCnt(void)</td>
</tr>
</tbody>
</table>
```

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(10)`. Once the trace buffer is full, trace is automatically stopped. The Trace list and buffer can be cleared from embOSView.

21.5.1 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 0** 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 1** 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.
- **Filters 2 to 4** allow trace of API calls from named tasks.
To enable or disable a filter, simply check or uncheck the corresponding checkboxes labeled Filter 4 Enable to Filter 0 Enable. For any of these five filters, individual API functions can be enabled or disabled by checking or unchecking the corresponding checkboxes in the list. To speed up the process, there are two buttons available:

- **Select all** - enables trace of all API functions for the currently enabled (checked) filters.
- **Deselect all** - disables trace of all API functions for the currently enabled (checked) filters.

**Filter 2, Filter 3, and Filter 4** allow tracing of task-specific API calls. A task name can therefore be specified for each of these filters. In the example above, **Filter 4** is configured to trace calls of `OS_TASK_Delay()` from the task called `MainTask`. After the settings are saved (via the Apply or OK button), the new settings are sent to the target application.
## 21.5.2 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.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>main</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TRACE_Enable()</td>
<td>Enables tracing of filtered API calls.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_EnableAll()</td>
<td>Sets up Filter 0 (any task), enables tracing of all API calls and then enables the trace function.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_EnableId()</td>
<td>Sets the specified ID value in Filter 0 (any task), thus enabling trace of the specified function, but does not start trace.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_EnableFilterId()</td>
<td>Sets the specified ID value in the specified trace filter, thus enabling trace of the specified function, but does not start trace.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_Disable()</td>
<td>Disables tracing of filtered API and user function calls.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_DisableAll()</td>
<td>Sets up Filter 0 (any task), disables tracing of all API calls and also disables trace.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_DisableId()</td>
<td>Resets the specified ID value in Filter 0 (any task), thus disabling trace of the specified function, but does not stop trace.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_DisableFilterId()</td>
<td>Resets the specified ID value in the specified trace filter, thus disabling trace of the specified function, but does not stop trace.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
21.5.2.1 OS_TRACE_Enable()

Description
Enables tracing of filtered API calls.

Prototype

```c
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.
21.5.2.2  OS_TRACE_EnableAll()

**Description**
Sets up Filter 0 (any task), enables tracing of all API calls and then enables the trace function.

**Prototype**

```c
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.
21.5.2.3 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**

```c
void OS_TRACE_EnableId(OS_U8 id);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.
21.5.2.4 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**

```c
void OS_TRACE_EnableFilterId(OS_U8 FilterIndex,
                              OS_U8 id);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilterIndex</td>
<td>Index of the filter that should be affected: 0 ≤ FilterIndex ≤ 4 0 affects Filter 0 (any task) and so on.</td>
</tr>
<tr>
<td>id</td>
<td>ID value of API call that should be enabled for trace: 0 ≤ id ≤ 255 Values from 0 to 99 and 128 to 255 are reserved for embOS.</td>
</tr>
</tbody>
</table>

**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.
21.5.2.5 OS_TRACE_Disable()

**Description**
Disables tracing of filtered API and user function calls.

**Prototype**

```c
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.
21.5.2.6 OS_TRACE_DisableAll()

Description
Sets up Filter 0 (any task), disables tracing of all API calls and also disables trace.

Prototype

```c
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.
21.5.2.7 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**

```c
void OS_TRACE_DisableId(OS_U8 id);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.
21.5.2.8 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**

```c
void OS_TRACE_DisableFilterId(OS_U8 FilterIndex,
                              OS_U8 id);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| FilterIndex | Index of the filter that should be affected: $0 \leq \text{FilterIndex} \leq 4$
| id | 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.
## 21.5.3 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.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>main</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TRACE_Data()</td>
<td>Writes an entry with ID and an integer as parameter into the trace buffer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_DataPtr()</td>
<td>Writes an entry with ID, an integer, and a pointer as parameter into the trace buffer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_Ptr()</td>
<td>Writes an entry with ID and a pointer as parameter into the trace buffer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_PtrU32()</td>
<td>Writes an entry with ID, a pointer, and a 32-bit unsigned integer as parameter into the trace buffer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_U32Ptr()</td>
<td>Writes an entry with ID, a 32-bit unsigned integer, and a pointer as parameter into the trace buffer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_TRACE_Void()</td>
<td>Writes an entry identified only by its ID into the trace buffer.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
21.5.3.1  OS_TRACE_Data()

Description
Writes an entry with ID and an integer as parameter into the trace buffer.

Prototype

```c
void OS_TRACE_Data(OS_U8 id, int v);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>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.</td>
</tr>
<tr>
<td>v</td>
<td>Any integer value that should be recorded as parameter.</td>
</tr>
</tbody>
</table>

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.
21.5.3.2 OS_TRACE_DataPtr()

**Description**
Writes an entry with ID, an integer, and a pointer as parameter into the trace buffer.

**Prototype**

```c
void OS_TRACE_DataPtr( OS_U8 id, int v, volatile OS_CONST_PTR void * p);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>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.</td>
</tr>
<tr>
<td>v</td>
<td>Any integer value that should be recorded as parameter.</td>
</tr>
<tr>
<td>p</td>
<td>Any void pointer that should be recorded as parameter.</td>
</tr>
</tbody>
</table>

**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.
### 21.5.3.3 OS_TRACE_Ptr()

**Description**

Writes an entry with ID and a pointer as parameter into the trace buffer.

**Prototype**

```c
void OS_TRACE_Ptr( OS_U8 id,
                    volatile OS_CONST_PTR void *p);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>ID value of API call that should be enabled for trace: (0 \leq \text{id} \leq 255) \n</td>
</tr>
<tr>
<td>p</td>
<td>Any void pointer that should be recorded as parameter.</td>
</tr>
</tbody>
</table>

**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.
21.5.3.4 OS_TRACE_PtrU32()

Description
Writes an entry with ID, a pointer, and a 32-bit unsigned integer as parameter into the
trace buffer.

Prototype

```c
void OS_TRACE_PtrU32 ( OS_U8 id, volatile OS_CONST_PTR void * p0, OS_U32 p1);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>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.</td>
</tr>
<tr>
<td>p0</td>
<td>Any void pointer that should be recorded as parameter.</td>
</tr>
<tr>
<td>p1</td>
<td>Any unsigned 32-bit value that should be recorded as parameter.</td>
</tr>
</tbody>
</table>

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.
21.5.3.5 OS_TRACE_U32Ptr()

Description

Writes an entry with ID, a 32-bit unsigned integer, and a pointer as parameter into the trace buffer.

Prototype

```c
void OS_TRACE_U32Ptr(           OS_U8        id,
                          OS_U32       p0,
                          volatile OSCONST_PTR void * p1);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| id        | 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. |
| p0        | Any unsigned 32-bit value that should be recorded as parameter. |
| p1        | 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.
21.5.3.6  OS_TRACE_Void()

**Description**

Writes an entry identified only by its ID into the trace buffer.

**Prototype**

```c
void OS_TRACE_Void(OS_U8 id);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| id        | 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**

This functionality is available in trace builds only, and the API call is not removed by the preprocessor.
21.5.4 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

```c
#include "RTOS.h"

#define APP_TRACE_ID_SETSTATE 100 // Application specific trace id

char MainState;

void SetState(char* pState, char Value) {
    #if (OS_TRACE != 0)
        OS_TRACE_DataPtr(APP_TRACE_ID_SETSTATE, Value, pState);
    #endif
    *pState = Value;
}

int main(void) {
    OS_Init();
    OS_InitHW();
    #if (OS_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().
21.5.5 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**

```ini
# 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 paramters. 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 22

MPU - Memory Protection
CHAPTER 22

22.1 Introduction

This chapter describes embOS-MPU. embOS-MPU is a separate product which adds memory protection to embOS.

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.

When a task violates its MPU permissions or causes an exception by other means, it is terminated automatically regardless of its privilege state.

embOS-MPU uses the hardware MPU and additional checks to avoid that a task affects the remaining system. Even if a bug in one task occurs all other tasks and the OS continue execution. The task which caused the issue is terminated automatically and the application is informed via an optional callback function.

Since a hardware MPU is required embOS MPU support is unavailable for some embOS ports. The MPU support is included in separate embOS ports and is not part of the general embOS port.

22.1.1 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.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Privileged</td>
<td>Full access to memory, peripheral and CPU features</td>
</tr>
<tr>
<td>Unprivileged</td>
<td>Only restricted access to memory, no direct access to peripherals, no access to some CPU features</td>
</tr>
</tbody>
</table>

22.1.2 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 can’t be verified for full reliability. As it runs in unprivileged state, it can’t affect the remaining system. Usually, this code is organized in several C files. This can e.g. simplify a certification.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st part</td>
<td>Task and MPU initialization</td>
</tr>
<tr>
<td></td>
<td>Device drivers</td>
</tr>
<tr>
<td>2nd part</td>
<td>Application code from e.g. third party vendor</td>
</tr>
</tbody>
</table>
22.2 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:

<table>
<thead>
<tr>
<th>Permission</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_MPU_NOACCESS</td>
<td>No access to a memory region</td>
</tr>
<tr>
<td>OS_MPU_READONLY</td>
<td>Read only access to a memory region</td>
</tr>
<tr>
<td>OS_MPU_READWRITE</td>
<td>Read and write access to a memory region</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Permission</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_MPU_EXECUTION_ALLOWED</td>
<td>Code execution is allowed</td>
</tr>
<tr>
<td>OS_MPU_EXECUTION_DISALLOWED</td>
<td>Code execution is not allowed</td>
</tr>
</tbody>
</table>

22.2.1 Default memory access permissions

A newly created unprivileged task has per default only access to the following memory regions:

<table>
<thead>
<tr>
<th>Region</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM</td>
<td>OS_MPU_READONLY, OS_MPU_EXECUTION_ALLOWED</td>
</tr>
<tr>
<td>RAM</td>
<td>OS_MPU_READONLY, OS_MPU_EXECUTION_ALLOWED</td>
</tr>
<tr>
<td>Task stack</td>
<td>OS_MPU_READWRITE, OS_MPU_EXECUTION_ALLOWED</td>
</tr>
</tbody>
</table>

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.

22.2.2 Interrupts

Interrupts are always privileged and can access the whole memory.

22.2.3 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.

22.2.4 Access to OS objects

An unprivileged task has no direct write access to embOS objects. It also has per default no access via embOS API functions. Access to OS objects can be added with OS_MPU_SetAllowedObjects(). The object list must be located in ROM memory. The OS object must be created in the privileged part of the task.
22.3 ROM placement of embOS

embOS must be placed in one memory section. embOS-MPU needs this information to e.g. check that supervisor calls are made from embOS API functions only. 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 compiler and linker.

Linker file:

```c
__os_load_start__ = ALIGN(__text_end__, 4);
.os ALIGN(__text_end__, 4) : AT(ALIGN(__text_end__, 4))
{
  __os_start__ = .;
  *(.os .os.*)
}
__os_end__ = __os_start__ + SIZEOF(.os);
__os_size__ = SIZEOF(.os);
__os_load_end__ = __os_end__;
```

C Code:

```c
void OS_InitHW() {
  OS_MPU_ConfigMem(0x08000000u, 0x00100000u, // ROM base address and size
                   0x20000000u, 0x00020000u, // RAM base address and size
                   __os_start__, __os_size__); // OS base address and size
}
```
22.4 Allowed embOS API in unprivileged tasks

Not all embOS API functions are allowed to be called from an unprivileged task. If an API function is allowed to be called from an unprivileged task a dot is placed in the column “Unpriv Task” in the according API table.

Example

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Unpriv Task</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TASK_Delay()</td>
<td>Suspends the calling task for a specified period of time, or waits actively when called from main().</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

It is allowed to call OS_TASK_Delay() from main(), privileged tasks and unprivileged tasks.
22.5 Device driver

22.5.1 Concept

An unprivileged task has no access to any peripheral. Thus a device driver is necessary to use peripherals like UART, SPI or port pins.

A device driver consists of two parts, an unprivileged part and a privileged part. embOS ensures there is only one explicit and safe way to switch from the unprivileged part to the privileged part. The application must call driver functions only in the unprivileged part. The actual peripheral access is performed in the privileged part only.

OS_MPU_CallDeviceDriver() is used to call the device driver. The first parameter is the index of the device driver function. Optional parameters can be passed to the device driver.

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() is the unprivileged part of the driver. It can be called by the unprivileged application.

```c
typedef struct BSP_LED_PARAM_STRUCT {
    BSP_LED_DRIVER_API Action;
    OS_U32             Index;
} BSP_LED_PARAM;

void BSP_ToggleLED(int Index) {
    BSP_LED_PARAM p;
    p.Action = BSP_LED_TOGGLE;
    p.Index  = Index;
    OS_MPU_CallDeviceDriver(0u, &p);
}
```

The device driver itself runs in privileged state and accesses the LED port pin.

```c
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;
    }
}
```
All device driver addresses are stored in one const list which is passed to embOS-MPU with OS_MPU_SetDeviceDriverList().

```c
static const OS_MPU_DEVICE_DRIVER_FUNC _DeviceDriverList[] =
{  BSP_LED_DeviceDriver,
   NULL };  // Last item must be NULL

void BSP_Init(void) {
   OS_MPU_SetDeviceDriverList(_DeviceDriverList);
}
```
### 22.6 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>main</th>
<th>Priv Task</th>
<th>ISR</th>
<th>ISR Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_MPU_AddRegion()</td>
<td>Adds an additional memory region to which the task has access.</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_CallDeviceDriver()</td>
<td>Calls a device driver.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_ConfigMem()</td>
<td>Configures basic memory information.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU.Enable()</td>
<td>Initializes the MPU hardware with the default MPU API list and enables it.</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_EnableEx()</td>
<td>Initializes the MPU hardware with the specified MPU API list and enables it.</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_ExtendTaskContext()</td>
<td>Extends the task context for the MPU registers.</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_GetThreadState()</td>
<td>Returns the current privileged task state.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_SetAllowedObjects()</td>
<td>Sets a task specific list of objects to which the task has access via embOS API functions.</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_SetDeviceDriverList()</td>
<td>Sets the device driver list.</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_SetErrorCallback()</td>
<td>Sets the MPU error callback function.</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_SwitchToUnprivState()</td>
<td>Switches a task to unprivileged state.</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_SwitchToUnprivStateEx()</td>
<td>Switches a task to unprivileged state and calls a task function which runs on a separate task stack.</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_SetSanityCheckBuffer()</td>
<td>Sets the pointer in the task control block to a buffer which holds a copy of the MPU register for sanity check.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS_MPU_SanityCheck()</td>
<td>Performs an MPU sanity check which checks if the MPU register still have the correct value.</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>
22.6.1 OS_MPU_AddRegion()

**Description**

Adds an additional memory region to which the task has access.

**Prototype**

```c
void OS_MPU_AddRegion(OS_TASK* pTask,
    OS_U32   BaseAddr,
    OS_U32   Size,
    OS_U32   Permissions,
    OS_U32   Attributes);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block.</td>
</tr>
<tr>
<td>BaseAddr</td>
<td>Region base address.</td>
</tr>
<tr>
<td>Size</td>
<td>Region size.</td>
</tr>
<tr>
<td>Permissions</td>
<td>Access permissions.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Additional core specific memory attributes.</td>
</tr>
</tbody>
</table>

**Additional information**

This function can be used if a task needs access to additional RAM regions. This RAM region can be e.g. a LCD frame buffer or a queue data buffer. It is CPU specific if the region has to be aligned. Please refer to the according CPU/compiler specific embOS manual for more details.

A memory region can have the following access permissions:

<table>
<thead>
<tr>
<th>Permission</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_MPU_NOACCESS</td>
<td>No access to memory region</td>
</tr>
<tr>
<td>OS_MPU_READONLY</td>
<td>Read only access to memory region</td>
</tr>
<tr>
<td>OS_MPU_READWRITE</td>
<td>Read and write access to memory region</td>
</tr>
<tr>
<td>OS_MPU_EXECUTION_ALLOWED</td>
<td>Code execution is allowed</td>
</tr>
<tr>
<td>OS_MPU_EXECUTION_DISALLOWED</td>
<td>Code execution is not allowed</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Region</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM</td>
<td>Read and execution access for complete ROM</td>
</tr>
<tr>
<td>RAM</td>
<td>Read only and execution access for complete RAM</td>
</tr>
<tr>
<td>Task stack</td>
<td>Read and write and execution access to the task stack</td>
</tr>
</tbody>
</table>

**Note**

OS_MPU_AddRegion() does take affect only when it is called before OS_MPU_SwitchToUnprivState().
Example

```c
static void HPTask(void) {
    OS_MPU_AddRegion(&TCBHP, (OS_U32)MyQBuffer, 512, OS_MPU_READWRITE, 0u);
}
```
22.6.2 OS_MPU_CallDeviceDriver()

Description
Calls a device driver.

Prototype

```c
void OS_MPU_CallDeviceDriver ( OS_U32 Index,
      void* Param );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Index of device driver function.</td>
</tr>
<tr>
<td>Param</td>
<td>Parameter to device driver.</td>
</tr>
</tbody>
</table>

Additional information

Unprivileged tasks have no direct access to any peripherals. A device driver is instead necessary. `OS_MPU_CallDeviceDriver()` is used to let embOS call the device driver which then runs in privileged state. Optional parameter can be passed to the driver function. The device driver is called e.g. for Cortex-M via SVC call.

Example

```c
typedef struct BSP_LED_PARAM_STRUCT {
    BSP_LED_DRIVER_API Action;
    OS_U32             Index;
} BSP_LED_PARAM;

static const OS_MPU_DEVICE_DRIVER_FUNC _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);
}
```
22.6.3 OS_MPU_ConfigMem()

Description

Configures basic memory 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.

Prototype

```c
void OS_MPU_ConfigMem ( OS_U32 ROM_BaseAddr ,
                         OS_U32 ROM_Size ,
                         OS_U32 RAM_BaseAddr ,
                         OS_U32 RAM_Size ,
                         OS_U32 OS_BaseAddr ,
                         OS_U32 OS_Size );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM_BaseAddr</td>
<td>ROM base addr.</td>
</tr>
<tr>
<td>ROM_Size</td>
<td>ROM size.</td>
</tr>
<tr>
<td>RAM_BaseAddr</td>
<td>RAM base addr.</td>
</tr>
<tr>
<td>RAM_Size</td>
<td>RAM size.</td>
</tr>
<tr>
<td>OS_BaseAddr</td>
<td>embOS ROM region base address.</td>
</tr>
<tr>
<td>OS_Size</td>
<td>embOS ROM region size.</td>
</tr>
</tbody>
</table>

Additional information

This function must be called before any unprivileged task is created.

Example

```c
void main(void) {
    OS_MPU_ConfigMem(0x08000000u ,
                      0x00100000u ,
                      0x20000000u ,
                      0x00020000u ,
                      __os_start__,
                      __os_size__);  
}
```
22.6.4 OS_MPU_Enable()

Description
Initializes the MPU hardware with the default MPU API list and enables it.

Prototype

```c
void OS_MPU_Enable(void);
```

Additional information
This function must be called before any embOS-MPU related function is used or any task is created.
22.6.5 OS_MPU_EnableEx()

**Description**
Initializes the MPU hardware with the specified MPU API list and enables it.

**Prototype**

```c
void OS_MPU_EnableEx(OS_CONST_PTR OS_MPU_API_LIST *pAPIList);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAPIList</td>
<td>Pointer to core specific MPU API list.</td>
</tr>
</tbody>
</table>

**Additional information**

This function must be called before any embOS-MPU related function is used or any task is created.

**Example**

```c
void main(void) {
    OS_MPU_EnableEx(&OS_ARMv7M_MPU_API);
}
```
22.6.6 OS_MPU_ExtendTaskContext()

Description
Extends the task context for the MPU registers.

Prototype

```c
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 in OS_LIBMODE_XR.

OS_SetDefaultContextExtension() can be used to automatically add MPU register to the task context of every newly created task.

Example

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

static void HPTask ( void ) {
    OS_MPU_ExtendTaskContext ();
    OS_MPU_SwitchToUnprivState ();
    while ( 1 ) {
        OS_TASK_Delay ( 200 );
    }
}
```

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.
22.6.7 OS_MPU_GetThreadState()

Description
Returns the current privileged task 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.
22.6.8 OS_MPU_SetAllowedObjects()

**Description**
Sets a task specific list of objects to which the task has access via embOS API functions.

**Prototype**

```c
void OS_MPU_SetAllowedObjects ( OS_TASK * pTask ,
                                OS_CONST_PTR OS_MPU_OBJ * pObjList );
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to a task control block.</td>
</tr>
<tr>
<td>pObjList</td>
<td>Pointer to a list of allowed objects.</td>
</tr>
</tbody>
</table>

**Additional information**

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 a control structure is not possible. But if an object is in the list the task can access the object via embOS API functions. This 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}.

The following object types exist:

- OS_MPU_OBJTYPE_TASK
- OS_MPU_OBJTYPE_MUTEX
- OS_MPU_OBJTYPE_SEMA
- OS_MPU_OBJTYPE_EVENT
- OS_MPU_OBJTYPE_QUEUE
- OS_MPU_OBJTYPE_MAILBOX
- OS_MPU_OBJTYPE_SWTIMER
- OS_MPU_OBJTYPE_MEMPOOL
- OS_MPU_OBJTYPE_WATCHDOG

**Example**

```c
static const OS_MPU_OBJ _ObjList[] = {{(OS_U32)&TCBHP, OS_MPU_OBJTYPE_TASK},
                                    {{(OS_U32)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();
}
```
22.6.9 OS_MPU_SetDeviceDriverList()

Description
Sets the device driver list.

Prototype

```c
void OS_MPU_SetDeviceDriverList ( OS_CONST_PTR OS_MPU_DEVICE_DRIVER_FUNC *pList );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pList</td>
<td>Pointer to device driver function address list.</td>
</tr>
</tbody>
</table>

Additional information

All device driver function addresses are stored in one list. The last item must be `NULL`. A device driver is called with the according index to this list.

Example

```c
static const OS_MPU_DEVICE_DRIVER_FUNC _DeviceDriverList[] =
{  BSP_LED_DeviceDriver,
    NULL };  // Last item must be NULL

void BSP_Init (void) {
    OS_MPU_SetDeviceDriverList (_DeviceDriverList);
}
```
22.6.10 OS_MPU_SetErrorCallback()

Description
Sets the MPU error callback function. This function is called when a task is suspended due to an MPU fault.

Prototype
void OS_MPU_SetErrorCallback(OS_MPU_ERROR_CALLBACK pFunc);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFunc</td>
<td>Pointer to callback function.</td>
</tr>
</tbody>
</table>

Additional information
embOS terminates any task that violates its MPU permissions or causes an exception by other means. embOS calls the user callback function 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.

The callback function is called with the following parameter:

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TASK*</td>
<td>Pointer to task control block of the unprivileged task which caused the MPU error.</td>
</tr>
<tr>
<td>OS_MPU_ERRORCODE</td>
<td>Error code which describes the cause for the MPU error.</td>
</tr>
</tbody>
</table>

Example

```c
static void _ErrorCallback(OS_TASK* pTask, OS_MPU_ERRORCODE ErrorCode) {
    printf("%s has been stopped due to error %d\n", 
           pTask->Name, ErrorCode);
}

int main(void) {
    OS_MPU_SetErrorCallback&_ErrorCallback);
}
```

embOS-MPU error codes

<table>
<thead>
<tr>
<th>Define</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_MPU_ERROR_INVALID_REGION</td>
<td>The OS object address is within an allowed task region. This is not allowed. This can for example happen when the object was placed on the task stack.</td>
</tr>
<tr>
<td>OS_MPU_ERROR_INVALID_OBJECT</td>
<td>The unprivileged task is not allowed to access this OS object.</td>
</tr>
<tr>
<td>OS_MPU_ERROR_INVALID_API</td>
<td>The unprivileged task tried to call an embOS API function which is not valid for an unprivileged task. For example unprivileged tasks must not call OS_TASK_EnterRegion().</td>
</tr>
<tr>
<td>OS_MPU_ERROR_HARDFAULT</td>
<td>Indicates that the task caused a hardfault.</td>
</tr>
<tr>
<td>Define</td>
<td>Explanation</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OS_MPU_ERROR_MEMFAULT</td>
<td>An illegal memory access was performed. A unprivileged task tried to write memory without having the access permission.</td>
</tr>
<tr>
<td>OS_MPU_ERROR_BUSFAULT</td>
<td>Indicates that the task caused a bus fault.</td>
</tr>
<tr>
<td>OS_MPU_ERROR_USAGEFAULT</td>
<td>Indicates that the task caused an usage fault.</td>
</tr>
<tr>
<td>OS_MPU_ERROR_SVC</td>
<td>The supervisor call was not made within an embOS API function. This is not allowed.</td>
</tr>
</tbody>
</table>
22.6.11 OS_MPU_SwitchToUnprivState()

Description
Switches a task to unprivileged state.

Prototype

```c
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()`.

Example

```c
static void _Unsecure ( void ) {
    while ( 1 ) {
        OS_TASK_Delay ( 10 );
    }
}

static void HPTask ( void ) {
    //
    // Initialization, e. g. add memory regions
    //
    OS_MPU_ExtendTaskContext ( );
    OS_MPU_SwitchToUnprivState ( );
    _Unsecure ( );
}
```

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.
22.6.12  OS_MPU_SwitchToUnprivStateEx()

Description
Switches a task to unprivileged state and calls a task function which runs on a separate task stack. This is an extended handling which is used with ARMv8M only.

Prototype

```c
void OS_MPU_SwitchToUnprivStateEx(voidRoutine * pRoutine,
                                   void *        OS_STACKPTR *pStack,
                                   OS_UINT      StackSize);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pRoutine</td>
<td>Pointer to a function that should run in unprivileged state.</td>
</tr>
<tr>
<td>pStack</td>
<td>Pointer to the task stack which should be used in unprivileged state.</td>
</tr>
<tr>
<td>StackSize</td>
<td>Size of the task stack.</td>
</tr>
</tbody>
</table>

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_SwitchToUnprivStateEx()`. You must use `OS_MPU_SwitchToUnprivStateEx()` with ARMv8M only.

Example

```c
static unsigned char _Stack[512];

static void _Unsecure(void) { // Runs on the stack _Stack
    while (1) {
        OS_TASK_Delay(10);
    }
}

static void HPTask(void) {
    // Initialization, e.g. add memory regions
    //
    OS_MPU_ExtendTaskContext();
    OS_MPU_SwitchToUnprivStateEx(_Unsecure, _Stack, 512);
}
```

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.
22.6.13 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

```c
void OS_MPU_SetSanityCheckBuffer(OS_TASK* pTask,
        void* p);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>Pointer to the task control block.</td>
</tr>
<tr>
<td>p</td>
<td>Pointer to the MPU register buffer.</td>
</tr>
</tbody>
</table>

Additional information

`OS_MPU_SetSanityCheckBuffer()` is only available in `OS_LIBMODE_SAFE` which is used in the certified embOS-MPU. 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 used prior to calling `OS_MPU_SwitchToUnprivState()` 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`.

Example

```c
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
            }
        }  
    }  
}
```
22.6.14  **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 used 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. If the buffer is not set, `OS_MPU_SanityCheck()` will return 0.

**Example**

```c
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
                }
            }
    }
}
```
23.1 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. A "normal" single-task program needs exactly one stack. In a multitasking system, every task must have its own stack.

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. Therefore, the debug and stack-check builds of embOS monitor the stack size (and, if available, also interrupt stack size) and call OS_Error() if they detect stack overflows.

To detect a stack overflow, the stack is filled with control characters upon its creation, thereby allowing for a check on these characters every time a task is deactivated. However, embOS does not guarantee to reliably detect all stack overflows. A stack that has been defined larger than necessary, on the other hand, does no harm; even though it is a waste of memory.

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 Specifics manual of embOS documentation.

Task stack

Each embOS task has a separate stack. The location and size of this stack is defined when creating the task. The minimum size of a task stack depends on the CPU and the compiler. For details, see the CPU & Compiler Specifics manual of embOS documentation.

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 Specifics manual of embOS documentation.

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. embOS checks at runtime how many bytes at the end of the stack still include the pattern byte. With it the amount of used and unused stack can be calculated.
**Stack-check**

embOS includes stack-check routines. embOS fills the task stacks and also the system stack and the interrupt stack with a pattern byte. embOS periodically checks whether the last pattern byte at the end of the stack was overwritten and calls `OS_Error()` when it was.
### 23.2 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OS_STACK_GetIntStackBase()</code></td>
<td>Returns the base address of the interrupt stack.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetIntStackSize()</code></td>
<td>Returns the size of the interrupt stack.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetIntStackSpace()</code></td>
<td>Returns the amount of interrupt stack which was never used (Free interrupt stack space).</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetIntStackUsed()</code></td>
<td>Returns the amount of interrupt stack which is actually used.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetTaskStackBase()</code></td>
<td>Returns a pointer to the base of a task stack.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetTaskStackSize()</code></td>
<td>Returns the total size of a task stack.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetTaskStackSpace()</code></td>
<td>Returns the amount of task stack which was never used by the task (Free stack space).</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetTaskStackUsed()</code></td>
<td>Returns the amount of task stack which is actually used by the task.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetSysStackBase()</code></td>
<td>Returns the base address of the system stack.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetSysStackSize()</code></td>
<td>Returns the size of the system stack.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetSysStackSpace()</code></td>
<td>Returns the amount of system stack which was never used (Free system stack space).</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetSysStackUsed()</code></td>
<td>Returns the amount of system stack which is actually used.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_GetCheckLimit()</code></td>
<td>Returns the stack check limit in percent.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>OS_STACK_SetCheckLimit()</code></td>
<td>Sets the stack check limit to a percentaged value of the stack size.</td>
<td>● ● ● ● ●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
23.2.1   OS_STACK_GetIntStackBase()

Description
Returns a pointer to the base of the interrupt stack.

Prototype

```c
void* OS_STACK_GetIntStackBase(void);
```

Return value
The pointer to the base address of the interrupt stack.

Additional information
This function is only available when an interrupt stack exists.

Example

```c
void CheckIntStackBase(void) {
    printf("Addr Interrupt Stack %p", OS_STACK_GetIntStackBase());
}
```
23.2.2 OS_STACK_GetIntStackSize()

Description
Returns the size of the interrupt stack.

Prototype

```c
unsigned int OS_STACK_GetIntStackSize(void);
```

Return value

The size of the interrupt stack in bytes.

Additional information

This function is only available when an interrupt stack exists.

Example

```c
void CheckIntStackSize(void) {
    printf("Size Interrupt Stack \%u", OS_STACK_GetIntStackSize());
}
```
23.2.3 OS_STACK_GetIntStackSpace()

**Description**
Returns the amount of interrupt stack which was never used (Free interrupt stack space).

**Prototype**

```c
unsigned int OS_STACK_GetIntStackSpace(void);
```

**Return value**
Amount of interrupt stack which was never used in bytes.

**Additional information**
This function is only available in the debug and stack-check builds and when an interrupt stack exists.

**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.

**Example**

```c
void CheckIntStackSpace(void) {
    printf("Unused Interrupt Stack \%u", OS_STACK_GetIntStackSpace());
}
```
23.2.4 OS_STACK_GetIntStackUsed()

Description
Returns the amount of interrupt stack which is actually used.

Prototype
unsigned int OS_STACK_GetIntStackUsed(void);

Return value
Amount of interrupt stack which is actually used in bytes.

Additional information
This function is only available in the debug and stack-check builds and when an interrupt stack exists.

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.

Example

```c
void CheckIntStackUsed(void) {
    printf("Used Interrupt Stack %u", OS_STACK_GetIntStackUsed());
}
```
23.2.5 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

```c
void OS_STACKPTR *OS_STACK_GetTaskStackBase(OS_CONST_PTR OS_TASK * pTask);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>The task whose stack base should be returned. NULL denotes the current task.</td>
</tr>
</tbody>
</table>

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. However, NULL must not be passed for pTask from main(), a timer callback or from an interrupt handler. A debug build of embOS will call OS_Error() in case pTask does not indicate a valid task.

This function is only available in the debug and stack-check builds of embOS, because only these builds initialize the stack space used for the tasks.

Example

```c
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);
}
```
23.2.6 OS_STACK_GetTaskStackSize()

Description
Returns the total size of a task stack.

Prototype

\[
\text{unsigned int } \text{OS_STACK_GetTaskStackSize(\text{OS_CONST_PTR OS_TASK * pTask});}
\]

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>The task whose stack size should be checked. NULL means current task.</td>
</tr>
</tbody>
</table>

Return value
Total size of the task stack in bytes.

Additional information
If NULL is passed for pTask, the currently running task is used. However, NULL must not be passed for pTask from main(), a timer callback or from an interrupt handler. A debug build of embOS will call OS_Error() in case pTask does not indicate a valid task.

This function is only available in the debug and stack-check builds of embOS, because only these builds initialize the stack space used for the tasks.

Example

```c
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);
}
```
23.2.7 OS_STACK_GetTaskStackSpace()

Description
Returns the amount of task stack which was never used by the task (Free stack space). If no specific task is addressed, the current task is checked.

Prototype

```c
unsigned int OS_STACK_GetTaskStackSpace(OS_CONST_PTR OS_TASK * pTask);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>The task whose stack space should be checked. NULL denotes the current task.</td>
</tr>
</tbody>
</table>

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. However, NULL must not be passed for `pTask` from main(), a timer callback or from an interrupt handler. A debug build of embOS will call `OS_Error()` in case `pTask` does not indicate a valid task.

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_GetTaskStackSpace()`, 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. This function is only available in the debug and stack-check builds of embOS.

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.

Example

```c
void CheckStackSpace(void) {
    printf("Unused Stack[0] %u", OS_STACK_GetTaskStackSpace(&TCB[0]));
    OS_TASK_Delay(1000);
    printf("Unused Stack[1] %u", OS_STACK_GetTaskStackSpace(&TCB[1]));
    OS_TASK_Delay(1000);
}
```
23.2.8 OS_STACK_GetTaskStackUsed()

**Description**

Returns the amount of task stack which is actually used by the task. If no specific task is addressed, the current task is checked.

**Prototype**

```c
unsigned int OS_STACK_GetTaskStackUsed(OS_CONST_PTR OS_TASK *pTask);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTask</td>
<td>The task whose stack usage should be checked. <strong>NULL</strong> denotes the current task.</td>
</tr>
</tbody>
</table>

**Return value**

Amount of task stack which is actually used by the task in bytes.

**Additional information**

If **NULL** is passed for `pTask`, the currently running task is used. However, **NULL** must not be passed for `pTask` from `main()`, a timer callback or from an interrupt handler. A debug build of embOS will call `OS_Error()` in case `pTask` does not indicate a valid task.

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. This function is only available in the debug and stack-check builds of embOS.

**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.

**Example**

```c
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);
}
```
23.2.9 OS_STACK_GetSysStackBase()

**Description**
Returns a pointer to the base of the system stack.

**Prototype**

```c
void* OS_STACK_GetSysStackBase(void);
```

**Return value**
The pointer to the base address of the system stack.

**Example**

```c
void CheckSysStackBase(void) {
    printf("Addr System Stack \p", OS_STACK_GetSysStackBase());
}
```
23.2.10  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.

Example

 void  CheckSysStackSize (void)  {
   printf("Size System Stack \u", OS_STACK_GetSysStackSize());
 }
23.2.11 OS_STACK_GetSysStackSpace()

Description
Returns the amount of system stack which was never used (Free system stack space).

Prototype

```c
unsigned int OS_STACK_GetSysStackSpace(void);
```

Return value
Amount of unused system stack, in bytes.

Additional information
This function is only available in the debug and stack-check builds of embOS.

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.

Example

```c
void CheckSysStackSpace(void) {
    printf("Unused System Stack \%u", OS_STACK_GetSysStackSpace());
}
```
23.2.12 OS_STACK_GetSysStackUsed()

Description
Returns the amount of system stack which is actually used.

Prototype

```
unsigned int OS_STACK_GetSysStackUsed(void);
```

Return value
Amount of used system stack, in bytes.

Additional information
This function is only available in the debug and stack-check builds of embOS.

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.

Example

```
void CheckSysStackUsed(void) {
    printf("Used System Stack \u", OS_STACK_GetSysStackUsed());
}
```
23.2.13  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_CHECKSTACK` 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.

Example

```
void Task(void) {
    OS_U8 Limit;

    Limit = OS_STACK_GetCheckLimit();
    printf("Limit: %u\n", Limit);
}
```
23.2.14 OS_STACK_SetCheckLimit()

Description
Sets the stack check limit to a percentaged value of the stack size.

Prototype

```c
void OS_STACK_SetCheckLimit(OS_U8 Limit);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limit</td>
<td>Stack check limit in percent. Valid values are 0..100%. Values above 100% are trimmed to 100%.</td>
</tr>
</tbody>
</table>

Additional information

This function is only available when the embOS compile time switch OS_CHECKSTACK 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().

Example

```c
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 24

Board Support Packages
24.1  Introduction

This chapter explains the target system specific parts of embOS, called BSP (board support package).

In general, no modification to the board support package is required to get started with embOS. The board support packages supplied with your embOS shipment will execute out of the box on your target hardware. Small modifications to the configuration might be necessary at a later point, for example to configure a different system tick frequency or in order to enable a UART for the optional communication with embOSView.

All 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 and contains most of the functions that may require modifications to match your target hardware.

Furthermore, the file BSP.c is provided in source code as well and may contain routines to initialize and control LEDs, which may require further modifications to match your target hardware. BSP.c is not vital for embOS but is used in some embOS sample applications.

Some board support packages include additional files for e.g. clock and PLL initialization. Usually these files don’t need to be modified. Further details on these are available with the CPU & Compiler specifics manual of the embOS documentation.

24.2  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 by 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.
24.3 Example

This RTOSInit.c serves as a template and shows the basic structure.

```c
#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) {
}
```
24.4 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.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_OS_GetHWTimerCycles()</td>
<td>Returns the current hardware timer count value.</td>
</tr>
<tr>
<td>_OS_GetHWTimerIntPending()</td>
<td>Returns if the hardware timer interrupt pending flag is set.</td>
</tr>
<tr>
<td>OS_Idle()</td>
<td>The idle loop is executed whenever no task is ready for execution.</td>
</tr>
<tr>
<td>OS_COM_Send1()</td>
<td>Sends one character towards embOSView.</td>
</tr>
</tbody>
</table>

(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 297.
24.4.1 OS_Idle()

Description
The function OS_Idle() is called when no task, timer routine or ISR is ready for execution. Usually, OS_Idle() is programmed as an endless loop without any content. With many embOS start projects, however, it activates a power save mode of the target CPU.

Prototype

```c
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 also used by the kernel. Exceptions and interrupts occurring during OS_Idle() will return to OS_Idle() unless they trigger a task switch. When returning to OS_Idle(), execution is continued from where it was interrupted. However, in case a task switch did occur 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 to be preserved. If this is required, please consider implementing a custom idle task (Creating a custom Idle task on page 481).

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(). Using disabled interrupts during OS_Idle() will therefore affect interrupt latency and task activation time.

Peripheral control and tickless API functions may be called from OS_Idle() to save power consumption. Please have a look in the chapter Low Power Support on page 320 for more details.

You must not call any other embOS API from within OS_Idle().

Example

```c
void OS_Idle(void) {  // Idle loop: No task is ready to execute
  while (1) {
  }
}
```
24.4.1.1 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 is ready for execution.

Example

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

static OS_STACKPTR int StackHP[128], StackLP[128], StackIdle[128];
static OS_TASK TCBHP, TCBLP, TCBIdle;

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(&TCBIdle, "Idle Task", 1, IdleTask, StackIdle);
    OS_Start();  // Start multitasking
    return 0;
}
```
24.4.2 OS_COM_Send1()

**Description**
Sends one character towards embOSView via the configured interface.

**Prototype**
```
void OS_COM_Send1(OS_U8 c);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>The character to send towards embOSView.</td>
</tr>
</tbody>
</table>

**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 404.
24.5 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.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SysTick_Handler()</td>
<td>The embOS system tick timer interrupt handler.</td>
</tr>
<tr>
<td>OS_InitHW()</td>
<td>Initializes the hardware required for embOS to run.</td>
</tr>
</tbody>
</table>
24.5.1 SysTick_Handler()

Description
The embOS system timer tick interrupt handler.

Prototype

```c
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

```c
void SysTick_Handler(void) {
    OS_INT_EnterNestable();
    OS_TICK_Handle();
    OS_INT_LeaveNestable();
}
```
### 24.5.2 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**

```c
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**

```c
void OS_InitHW(void) {  
  //  
  // Initialize hardware timer  
  //  
  
  // Setup timing information  
  //  
  OS_TIME_ConfigSysTimer(&SysTimerConfig);  
}
```
24.6 Settings

The following defines are used in the RTOSInit.c.

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TIMER_FREQ</td>
<td>System tick hardware timer frequency in Hz</td>
</tr>
<tr>
<td>OS_TICK_FREQ</td>
<td>System tick frequency in Hz</td>
</tr>
<tr>
<td>OS_INT_FREQ</td>
<td>System tick interrupt frequency in Hz</td>
</tr>
<tr>
<td>OS_VIEW_IFSELECT</td>
<td>embOSView communication interface</td>
</tr>
<tr>
<td>OS_UART</td>
<td>Used UART</td>
</tr>
<tr>
<td>OS_BAUDRATE</td>
<td>UART baudrate</td>
</tr>
</tbody>
</table>

Optional for embOSView

24.6.1 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 milliseconds or any other value. For most applications, however, 1 milliseconds is an appropriate value.

24.6.2 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.

24.6.3 Using a different UART or baudrate for embOSView

Relevant defines
- OS_UART (Selection of UART to be used with embOSView)
- OS_BAUDRATE (Selection of baudrate 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 have been preconfigured for your target hardware.
Chapter 25

System Variables
25.1 Introduction

The system variables are described here for a deeper understanding of how the OS 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, as only the documented API functions are guaranteed to remain unchanged in future versions of embOS. These variables are accessible, but they should only be altered by functions of embOS. However, some of these variables can be very useful, especially the time variables.

**Note**

Do not alter any system variables!
25.2 Time variables

25.2.1 OS_Global

OS_Global is a structure which includes embOS internal variables. The following variables OS_Global.Time and OS_Global.TimeDex are part of OS_Global. Any other part of OS_Global is not explained here as they are not required to use embOS.

25.2.2 OS_Global.Time

Description

This is the time variable which contains the current system time in embOS system ticks (typically equivalent to millisecond).

Additional information

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 297.

25.2.3 OS_Global.TimeDex

For internal use only. Contains the time at which the next task switch or timer activation is due. If \((\text{int})(\text{OS}_\text{Global}.\text{Time} - \text{OS}_\text{Global}.\text{TimeDex})) \neq 0\), 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.
## 25.3 OS information routines

### 25.3.1 API functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>main</th>
<th>Priv Task</th>
<th>ISR</th>
<th>SW Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OS_INFO_GetCPU()</code></td>
<td>Returns the CPU name.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><code>OS_INFO_GetLibMode()</code></td>
<td>Returns the library mode.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><code>OS_INFO_GetLibName()</code></td>
<td>Returns the library name.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><code>OS_INFO_GetModel()</code></td>
<td>Returns the memory model name.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><code>OS_INFO_GetTimerFreq()</code></td>
<td>Returns the system tick hardware timer frequency.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><code>OS_INFO_GetVersion()</code></td>
<td>Returns the embOS version number.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
25.3.1.1 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.
25.3.1.2 OS_INFO_GetLibMode()

Description
Returns the library mode.

Prototype

```c
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".
25.3.1.3 OS_INFO_GetLibName()

Description
Returns the library name.

Prototype

```c
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".
25.3.1.4 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".
25.3.1.5 OS_INFO_GetTimerFreq()

Description
Returns the system tick hardware timer frequency in hertz.

Prototype

```c
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()` is implemented as a macro instead of an actual function. `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.

Example

```c
void Task ( void )  {
    OS_U64 t , t0 ;
    OS_U32 TimerFreq ;
    OS_U64 Result ;

    TimerFreq = OS_INFO_GetTimerFreq () ;
    t0 = OS_TIME_GetCycles () ;
    DoSomeThing () ;
    t = OS_TIME_GetCycles () - t0 ;
    // Result in nanoseconds
    Result = ( t * 1000000000 ) / TimerFreq ;
}
```
25.3.1.6 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. "41203" for embOS version 4.12c.
Chapter 26

Source Code
26.1 Introduction

embOS is available in two versions:

1. Object version: Object code + hardware initialization source.
2. Full source version: Complete source code.

Because this document describes the object version, the internal data structures are not explained in detail. The object version offers the full functionality of embOS including all supported memory models of the compiler, the debug libraries as described and the source code for idle task and hardware initialization. However, the object version does not allow source-level debugging of the library routines and the kernel.

The full source version gives you complete flexibility: embOS can be recompiled for different data sizes; different compile options give you full control of the generated code, making it possible to optimize the system for versatility or minimum memory requirements. You can debug the entire system and even modify it for new memory models or other CPUs.

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 rebuild with the additional batch files in the root folder. All of them are described in the following section.
### 26.2 Building embOS libraries

The embOS libraries can only be built if you have licensed a source code version 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 distribution:

- **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 `OS_Chip.h` and the generic part `OS_RAW.h`. RTOS.h is output in the subfolder `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.
26.3 Compile time switches

Many features of embOS may be modified using compile-time switches. With each embOS distribution, these switches are preconfigured 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.

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.

<table>
<thead>
<tr>
<th>Compile time switch</th>
<th>Description</th>
<th>Permitted values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_DEBUG</td>
<td>Enables runtime debug checks</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_CHECKSTACK</td>
<td>Performs stack checks</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_STACKCHECK_LIMIT</td>
<td>Percentage of stack usage that will be detected as a stack overflow error</td>
<td>1-100</td>
<td>100</td>
</tr>
<tr>
<td>OS_PROFILE</td>
<td>Profiling support</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_SUPPORT_TICKSTEP</td>
<td>embOSView tick step support</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_TRACE</td>
<td>embOSView trace support</td>
<td>0: Disabled 1: Enabled</td>
<td>0</td>
</tr>
<tr>
<td>OS_TRACE_RECORD_API_END</td>
<td>Generates additional SystemView API-End events</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_RR_SUPPORTED</td>
<td>Round-Robin supported</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_TRACKNAME</td>
<td>Allows task and OS object names</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_SUPPORT_SAVE_RESTORE_HOOK</td>
<td>Support task context extensions</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_SUPPORT_STAT</td>
<td>Generate task statistic information</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_SUPPORT_PTLS</td>
<td>Support for thread local storage</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_INIT_EXPLICITLY</td>
<td>Initialization of internal embOS variables</td>
<td>0: Disabled 1: Enabled</td>
<td>0</td>
</tr>
<tr>
<td>OS_SUPPORT_TIMER</td>
<td>Support for embOS software timers</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>OS_SUPPORT_TICKLESS</td>
<td>Support for embOS tickless mode</td>
<td>0: Disabled 1: Enabled</td>
<td>1</td>
</tr>
<tr>
<td>Compile time switch</td>
<td>Description</td>
<td>Permitted values</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>--------------------------------------------------</td>
<td>-------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>OS_SUPPORT_PERIPHERAL_POWER_CTRL</td>
<td>Enables peripheral power control</td>
<td>0: Disabled</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Enabled</td>
<td></td>
</tr>
<tr>
<td>OS_POWER_NUM_COUNTERS</td>
<td>Number of peripherals which can be used</td>
<td>&gt; 0</td>
<td>5</td>
</tr>
<tr>
<td>OS_SPINLOCK_MAX_CORES</td>
<td>Number of cores that should access a spinlock</td>
<td>&gt; 0</td>
<td>4</td>
</tr>
<tr>
<td>OS_SUPPORT_OS_ALLOC</td>
<td>Support for embOS thread safe heap allocation</td>
<td>0: Disabled</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Enabled</td>
<td></td>
</tr>
</tbody>
</table>
26.4 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.

<table>
<thead>
<tr>
<th>Folder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenOSSrc</td>
<td>embOS generic sources</td>
</tr>
<tr>
<td>CPU</td>
<td>RTOS assembler file</td>
</tr>
<tr>
<td>CPU\OSSrcCPU</td>
<td>CPU and compiler-specific files</td>
</tr>
</tbody>
</table>

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.
Chapter 27

Shipment
27.1 Introduction

embOS is available in three different variants: Free, Library, and Source code. The fully functional free variant can easily be downloaded for each embOS port and has no technical limitation.

The following table lists the included features with each of these variants:

<table>
<thead>
<tr>
<th>Features</th>
<th>Free</th>
<th>Library</th>
<th>Source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>embOS libraries</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>embOS source code</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>embOSView - Profiling PC tool</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>embOS manual</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>CPU/Compiler specific manual</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Release notes</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>embOS IDE plug-ins</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Board support packages</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Feature &amp; maintenance updates</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Technical support</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Free for any non-commercial use</td>
<td>●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 27.2 Library variant

<table>
<thead>
<tr>
<th>Directory</th>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start\BoardSupport</td>
<td></td>
<td>Board support packages in vendor specific subfolders</td>
</tr>
<tr>
<td>Start\Inc</td>
<td>RTOS.h, BSP.h, OS_Config.h</td>
<td>Include files for embOS</td>
</tr>
<tr>
<td>Start\Lib</td>
<td>embOSView.exe</td>
<td>PC utility for runtime analysis</td>
</tr>
<tr>
<td></td>
<td>JLinkARM.dll</td>
<td>J-Link DLL used with embOSView</td>
</tr>
<tr>
<td></td>
<td>Release_embOS.html</td>
<td>embOS release history</td>
</tr>
<tr>
<td></td>
<td>Release_embOS_CPU_Compiler.html</td>
<td>embOS CPU and compiler-specific release history</td>
</tr>
<tr>
<td></td>
<td>SYSVIEW_embOS.txt</td>
<td>SystemView ID description file</td>
</tr>
<tr>
<td></td>
<td>UM010xx_embOS_CPU_Complier.pdf</td>
<td>embOS CPU and compiler-specific manual</td>
</tr>
<tr>
<td></td>
<td>UM01001_embOS.pdf</td>
<td>embOS manual</td>
</tr>
</tbody>
</table>
27.3 Free variant

The Free variant is identical to the library variant, except for additional license information governing the use of this variant.

<table>
<thead>
<tr>
<th>Directory</th>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start\BoardSupport</td>
<td></td>
<td>Board support packages in vendor specific subfolders</td>
</tr>
<tr>
<td>Start\Inc</td>
<td>RTOS.h, BSP.h, OS_Config.h</td>
<td>Include files for embOS</td>
</tr>
<tr>
<td>Start\Lib</td>
<td>embOSView.exe</td>
<td>embOS libraries</td>
</tr>
<tr>
<td></td>
<td>JLinkARM.dll</td>
<td>J-Link DLL used with embOSView</td>
</tr>
<tr>
<td></td>
<td>License.txt</td>
<td>License information</td>
</tr>
<tr>
<td></td>
<td>Release_embOS.html</td>
<td>embOS release history</td>
</tr>
<tr>
<td></td>
<td>Release_embOS_CPUCompiler.html</td>
<td>embOS CPU and compiler-specific release history</td>
</tr>
<tr>
<td></td>
<td>SYSVIEW_embOS.txt</td>
<td>SystemView ID description file</td>
</tr>
<tr>
<td></td>
<td>UM010xx_embOS_CPUCompiler.pdf</td>
<td>embOS CPU and compiler-specific manual</td>
</tr>
<tr>
<td></td>
<td>UM01001_embOS.pdf</td>
<td>embOS manual</td>
</tr>
</tbody>
</table>
27.4 Source code variant

The source code variant is identical to the library variant, but in addition also contains the embOS source files and a set of batch files that can be used to rebuild the embOS libraries.

<table>
<thead>
<tr>
<th>Directory</th>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>OSCHIP.h, OS_Priv.h, RTOS.asm</td>
<td>CPU- and compiler-specific files</td>
</tr>
<tr>
<td>CPU\OSSrcCPU</td>
<td></td>
<td>Additional CPU- and compiler-specific source files</td>
</tr>
<tr>
<td>GenOSSrc</td>
<td></td>
<td>Generic source files</td>
</tr>
<tr>
<td>Start\BoardSupport</td>
<td></td>
<td>Board support packages in vendor-specific subfolders</td>
</tr>
<tr>
<td>Start\Inc</td>
<td>RTOS.h, BSP.h, OS_Config.h</td>
<td>Include files for embOS libraries</td>
</tr>
<tr>
<td>Start\Lib</td>
<td></td>
<td>embOS libraries</td>
</tr>
<tr>
<td>Start\Src</td>
<td></td>
<td>Only used with embOS-Safe, contains additional source files</td>
</tr>
<tr>
<td></td>
<td>embOSView.exe</td>
<td>PC utility for runtime analysis</td>
</tr>
<tr>
<td></td>
<td>JLinkARM.dll</td>
<td>J-Link DLL used with embOSView</td>
</tr>
<tr>
<td></td>
<td>Release_embOS.html</td>
<td>embOS release history</td>
</tr>
<tr>
<td></td>
<td>Release_embOS_CPU.Compiler.html</td>
<td>embOS CPU and compiler-specific release history</td>
</tr>
<tr>
<td></td>
<td>SYSVIEW_embOS.txt</td>
<td>SystemView ID description file</td>
</tr>
<tr>
<td></td>
<td>UM010xx_embOS_CPU.Compiler.pdf</td>
<td>embOS CPU and compiler-specific manual</td>
</tr>
<tr>
<td></td>
<td>UM01001_embOS.pdf</td>
<td>embOS manual</td>
</tr>
<tr>
<td></td>
<td>*.bat</td>
<td>Batch files to rebuild the embOS libraries</td>
</tr>
</tbody>
</table>
Chapter 28

Update
28.1 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.
28.2 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.

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:

<table>
<thead>
<tr>
<th>File</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>embOS libraries</td>
<td>Start\Lib</td>
<td>embOS object code libraries</td>
</tr>
<tr>
<td>RTOS.h</td>
<td>Start\Inc</td>
<td>embOS header file</td>
</tr>
<tr>
<td>OS_Config.h</td>
<td>Start\Inc</td>
<td>embOS config header file</td>
</tr>
<tr>
<td>BSP.h</td>
<td>Start\Inc</td>
<td>Board support header file</td>
</tr>
<tr>
<td>RTOSInit.c</td>
<td>Start\BoardSupport_\Setup</td>
<td>Hardware related routines</td>
</tr>
<tr>
<td>OS_Error.c</td>
<td>Start\BoardSupport_\Setup</td>
<td>embOS error routines</td>
</tr>
<tr>
<td>Additional files</td>
<td>Start\BoardSupport_\Setup</td>
<td>CPU and compiler-specific files</td>
</tr>
</tbody>
</table>

28.2.1 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 518).
28.3 embOS API Migration guide

Most embOS API names and some object type names have changed between embOS V4 and V5. The legacy embOS API names can still be used and there is no need to update the user application. embOS is still 100% compatible. However, for new projects the V5 API should be used.

If you want to replace the V4 with the V5 API 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 some API the parameter order has changed. This needs to be adapted manually.

<table>
<thead>
<tr>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_Config_Stop()</td>
<td>OS_ConfigStop()</td>
</tr>
<tr>
<td>OS_InitKern()</td>
<td>OS_Init()</td>
</tr>
<tr>
<td>OS_AddExtendTaskContext()</td>
<td>OS_TASK_AddContextExtension()</td>
</tr>
<tr>
<td>OS_AddTerminateHook()</td>
<td>OS_TASK_AddTerminateHook()</td>
</tr>
<tr>
<td>OS_CREATETASK()</td>
<td>OS_TASK_CREATE()</td>
</tr>
<tr>
<td>OS_CreateTask()</td>
<td>OS_TASK_Create()</td>
</tr>
<tr>
<td>OS_CREATETASK_EX()</td>
<td>OS_TASK_CREATEEX()</td>
</tr>
<tr>
<td>OS_CreateTaskEx()</td>
<td>OS_TASK_CreateEx()</td>
</tr>
<tr>
<td>OS_Delay()</td>
<td>OS_TASK_Delay()</td>
</tr>
<tr>
<td>OS_DelayUntil()</td>
<td>OS_TASK_DelayUntil()</td>
</tr>
<tr>
<td>OS_Delayus()</td>
<td>OS_TASK_Delayus()</td>
</tr>
<tr>
<td>OS_GetTaskName()</td>
<td>OS_TASKGetName()</td>
</tr>
<tr>
<td>OS_GetNumTasks()</td>
<td>OS_TASK_GetNumTasks()</td>
</tr>
<tr>
<td>OS_GetPriority()</td>
<td>OS_TASK_GetPriority()</td>
</tr>
<tr>
<td>OS_GetSuspendCnt()</td>
<td>OS_TASK_GetSuspendCnt()</td>
</tr>
<tr>
<td>OS_GetTaskID()</td>
<td>OS_TASK_GetID()</td>
</tr>
<tr>
<td>OS_GetTimeSliceRem()</td>
<td>OS_TASK_GetTimeSliceRem()</td>
</tr>
<tr>
<td>OS_IsTask()</td>
<td>OS_TASK_IsTask()</td>
</tr>
<tr>
<td>OS_TaskIndex2Ptr()</td>
<td>OS_TASK_Index2Ptr()</td>
</tr>
<tr>
<td>OS_RemoveAllTerminateHooks()</td>
<td>OS_TASK_RemoveAllTerminateHooks()</td>
</tr>
<tr>
<td>OS_RemoveTerminateHook()</td>
<td>OS_TASK_RemoveTerminateHook()</td>
</tr>
<tr>
<td>OS_Resume()</td>
<td>OS_TASK_Resume()</td>
</tr>
<tr>
<td>OS_ResumeAllTasks()</td>
<td>OS_TASK_ResumeAll()</td>
</tr>
<tr>
<td>OS_ExtendTaskContext()</td>
<td>OS_TASK_SetContextExtension()</td>
</tr>
<tr>
<td>OS_SetDefaultTaskContextExtension()</td>
<td>OS_TASK_SetDefaultContextExtension()</td>
</tr>
<tr>
<td>OS_SetDefaultTaskStartHook()</td>
<td>OS_TASK_SetDefaultStartHook()</td>
</tr>
<tr>
<td>OS_SetInitialSuspendCnt()</td>
<td>OS_TASK_SetInitialSuspendCnt()</td>
</tr>
<tr>
<td>OS_SetTaskName()</td>
<td>OS_TASKSetName()</td>
</tr>
<tr>
<td>OS_SetPriority()</td>
<td>OS_TASK_SetPriority()</td>
</tr>
<tr>
<td>OS_SetTimeSlice()</td>
<td>OS_TASK_SetTimeSlice()</td>
</tr>
<tr>
<td>V4</td>
<td>V5</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>OS_Suspend()</td>
<td>OS_TASK_Suspend()</td>
</tr>
<tr>
<td>OS_SuspendAllTasks()</td>
<td>OS_TASK_SuspendAll()</td>
</tr>
<tr>
<td>OS_TerminateTask()</td>
<td>OS_TASK_Terminate()</td>
</tr>
<tr>
<td>OS_WakeTask()</td>
<td>OS_TASK_Wake()</td>
</tr>
<tr>
<td>OS_Yield()</td>
<td>OS_TASK_Yield()</td>
</tr>
<tr>
<td>OS_CREATETIMER()</td>
<td>OS_TIMER_CREATE()</td>
</tr>
<tr>
<td>OS_CreateTimer()</td>
<td>OS_TIMER_Create()</td>
</tr>
<tr>
<td>OS_CREATETIMER_EX()</td>
<td>OS_TIMER_CREATEEX()</td>
</tr>
<tr>
<td>OS_CreateTimerEx()</td>
<td>OS_TIMER_CreateEx()</td>
</tr>
<tr>
<td>OS_DeleteTimer()</td>
<td>OS_TIMER_Delete()</td>
</tr>
<tr>
<td>OS_DeleteTimerEx()</td>
<td>OS_TIMER_DeleteEx()</td>
</tr>
<tr>
<td>OS_GetpCurrentTimer()</td>
<td>OS_TIMER_GetCurrent()</td>
</tr>
<tr>
<td>OS_GetpCurrentTimerEx()</td>
<td>OS_TIMER_GetCurrentEx()</td>
</tr>
<tr>
<td>OS_GetTimerPeriod()</td>
<td>OS_TIMER_GetPeriod()</td>
</tr>
<tr>
<td>OS_GetTimerPeriodEx()</td>
<td>OS_TIMER_GetPeriodEx()</td>
</tr>
<tr>
<td>OS_GetTimerValue()</td>
<td>OS_TIMER_GetRemainingPeriod()</td>
</tr>
<tr>
<td>OS_GetTimerValueEx()</td>
<td>OS_TIMER_GetRemainingPeriodEx()</td>
</tr>
<tr>
<td>OS_GetTimerStatus()</td>
<td>OS_TIMER_GetStatus()</td>
</tr>
<tr>
<td>OS_GetTimerStatusEx()</td>
<td>OS_TIMER_GetStatusEx()</td>
</tr>
<tr>
<td>OS_RetriggerTimer()</td>
<td>OS_TIMER_Restart()</td>
</tr>
<tr>
<td>OS_RetriggerTimerEx()</td>
<td>OS_TIMER_RestartEx()</td>
</tr>
<tr>
<td>OS_SetTimerPeriod()</td>
<td>OS_TIMER_SetPeriod()</td>
</tr>
<tr>
<td>OS_SetTimerPeriodEx()</td>
<td>OS_TIMER_SetPeriodEx()</td>
</tr>
<tr>
<td>OS_StartTimer()</td>
<td>OS_TIMER_Start()</td>
</tr>
<tr>
<td>OS_StartTimerEx()</td>
<td>OS_TIMER_StartEx()</td>
</tr>
<tr>
<td>OS_StopTimer()</td>
<td>OS_TIMER_Stop()</td>
</tr>
<tr>
<td>OS_StopTimerEx()</td>
<td>OS_TIMER_StopEx()</td>
</tr>
<tr>
<td>OS_TriggerTimer()</td>
<td>OS_TIMER_Trigger()</td>
</tr>
<tr>
<td>OS_TriggerTimerEx()</td>
<td>OS_TIMER_TriggerEx()</td>
</tr>
<tr>
<td>OS_ClearEvents()</td>
<td>OS_TASKEVENT_Clear()</td>
</tr>
<tr>
<td>OS_ClearEventsEx()</td>
<td>OS_TASKEVENT_ClearEx()</td>
</tr>
<tr>
<td>OS_GetEventsOccurred()</td>
<td>OS_TASKEVENT_Get()</td>
</tr>
<tr>
<td>OS_WaitEvent()</td>
<td>OS_TASKEVENT_GetBlocked()</td>
</tr>
<tr>
<td>OS_WaitSingleEvent()</td>
<td>OS_TASKEVENT_GetSingleBlocked()</td>
</tr>
<tr>
<td>OS_WaitSingleEventTimed()</td>
<td>OS_TASKEVENT_GetSingleTimed()</td>
</tr>
<tr>
<td>OS_WaitEventTimed()</td>
<td>OS_TASKEVENT_GetTimed()</td>
</tr>
<tr>
<td>OS_SignalEvent()</td>
<td>OS_TASKEVENT_Set()</td>
</tr>
<tr>
<td>OS_EVENT_Wait()</td>
<td>OS_EVENT_GetBlocked()</td>
</tr>
<tr>
<td>OS_EVENT_WaitMask()</td>
<td>OS_EVENT_GetMaskBlocked()</td>
</tr>
<tr>
<td>OS_EVENT_WaitMaskTimed()</td>
<td>OS_EVENT_GetMaskTimed()</td>
</tr>
<tr>
<td>OS_EVENT_WaitTimed()</td>
<td>OS_EVENT_GetTimed()</td>
</tr>
<tr>
<td>OS_CreateRSema()</td>
<td>OS_MUTEX_Create()</td>
</tr>
<tr>
<td>OS_CREATESEMA()</td>
<td>OS_MUTEX_Create()</td>
</tr>
<tr>
<td>V4</td>
<td>V5</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>OS_DeleteRSema()</td>
<td>OS_MUTEX_Delete()</td>
</tr>
<tr>
<td>OS_GetResourceOwner()</td>
<td>OS_MUTEX_GetOwner()</td>
</tr>
<tr>
<td>OS_GetSemaValue()</td>
<td>OS_MUTEX_GetValue()</td>
</tr>
<tr>
<td>OS_Request()</td>
<td>OS_MUTEX_Lock()</td>
</tr>
<tr>
<td>OS_Unuse()</td>
<td>OS_MUTEX_LockBlocked()</td>
</tr>
<tr>
<td>OS_CreateCSEMA()</td>
<td>OS_SEMAPHORE_CREATE()</td>
</tr>
<tr>
<td>OS_CreateCSem()</td>
<td>OS_SEMAPHORE_Create()</td>
</tr>
<tr>
<td>OS_DeleteCSem()</td>
<td>OS_SEMAPHORE_Delete()</td>
</tr>
<tr>
<td>OS_GetCSemValue()</td>
<td>OS_SEMAPHORE_GetValue()</td>
</tr>
<tr>
<td>OS_SignalCSem()</td>
<td>OS_SEMAPHORE_Give()</td>
</tr>
<tr>
<td>OS_SignalCSemMax()</td>
<td>OS_SEMAPHORE_GiveMax()</td>
</tr>
<tr>
<td>OS_SetCSemValue()</td>
<td>OS_SEMAPHORE_SetValue()</td>
</tr>
<tr>
<td>OS_CSemRequest()</td>
<td>OS_SEMAPHORE_Take()</td>
</tr>
<tr>
<td>OS_WaitCSem()</td>
<td>OS_SEMAPHORE_TakeBlocked()</td>
</tr>
<tr>
<td>OS_WaitCSemTimed()</td>
<td>OS_SEMAPHORE_TakeTimed()</td>
</tr>
<tr>
<td>OS_ClearMB()</td>
<td>OS_MAILBOX_Clear()</td>
</tr>
<tr>
<td>OS_CreateMB()</td>
<td>OS_MAILBOX_Create()</td>
</tr>
<tr>
<td>OS_DeleteMB()</td>
<td>OS_MAILBOX_Delete()</td>
</tr>
<tr>
<td>OS_GetMailCond()</td>
<td>OS_MAILBOX_Get()</td>
</tr>
<tr>
<td>OS_GetMailCond1()</td>
<td>OS_MAILBOX_Get1()</td>
</tr>
<tr>
<td>OS_GetMail()</td>
<td>OS_MAILBOX_GetBlocked()</td>
</tr>
<tr>
<td>OS_GetMail1()</td>
<td>OS_MAILBOX_GetBlocked1()</td>
</tr>
<tr>
<td>OS_GetMessageCnt()</td>
<td>OS_MAILBOX_GetMessageCnt()</td>
</tr>
<tr>
<td>OS_GetMailTimed()</td>
<td>OS_MAILBOX_GetTimed()</td>
</tr>
<tr>
<td>OS_GetMailTimed1()</td>
<td>OS_MAILBOX_GetTimed1()</td>
</tr>
<tr>
<td>OS_PutMailCond()</td>
<td>OS_MAILBOX_Put()</td>
</tr>
<tr>
<td>OS_PutMailCond1()</td>
<td>OS_MAILBOX_Put1()</td>
</tr>
<tr>
<td>OS_PutMail()</td>
<td>OS_MAILBOX_PutBlocked()</td>
</tr>
<tr>
<td>OS_PutMail1()</td>
<td>OS_MAILBOX_PutBlocked1()</td>
</tr>
<tr>
<td>OS_PutMailFrontCond()</td>
<td>OS_MAILBOX_PutFront()</td>
</tr>
<tr>
<td>OS_PutMailFrontCond1()</td>
<td>OS_MAILBOX_PutFront1()</td>
</tr>
<tr>
<td>OS_PutMailFront()</td>
<td>OS_MAILBOX_PutFrontBlocked()</td>
</tr>
<tr>
<td>OS_PutMailFront1()</td>
<td>OS_MAILBOX_PutFrontBlocked1()</td>
</tr>
<tr>
<td>OS_PutMailTimed()</td>
<td>OS_MAILBOX_PutTimed()</td>
</tr>
<tr>
<td>OS_PutMailTimed1()</td>
<td>OS_MAILBOX_PutTimed1()</td>
</tr>
<tr>
<td>OS_WaitMail()</td>
<td>OS_MAILBOX_WaitBlocked()</td>
</tr>
<tr>
<td>OS_WaitMailTimed()</td>
<td>OS_MAILBOX_WaitTimed()</td>
</tr>
<tr>
<td>V4</td>
<td>V5</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>OS_Q_Clear()</td>
<td>OS_QUEUE_Clear()</td>
</tr>
<tr>
<td>OS_Q_Create()</td>
<td>OS_QUEUE_Create()</td>
</tr>
<tr>
<td>OS_Q_Delete()</td>
<td>OS_QUEUE_Delete()</td>
</tr>
<tr>
<td>OS_Q_GetMessageCnt()</td>
<td>OS_QUEUE_GetMessageCnt()</td>
</tr>
<tr>
<td>OS_Q_GetMessageSize()</td>
<td>OS_QUEUE_GetMessageSize()</td>
</tr>
<tr>
<td>OS_Q_GetPtrCond()</td>
<td>OS_QUEUE_GetPtr()</td>
</tr>
<tr>
<td>OS_Q_GetPtr()</td>
<td>OS_QUEUE_GetPtrBlocked()</td>
</tr>
<tr>
<td>OS_Q_GetPtrTimed()</td>
<td>OS_QUEUE_GetPtrTimed()</td>
</tr>
<tr>
<td>OS_Q_IsInUse()</td>
<td>OS_QUEUE_IsInUse()</td>
</tr>
<tr>
<td>OS_Q_PeekPtr()</td>
<td>OS_QUEUE_PeekPtr()</td>
</tr>
<tr>
<td>OS_Q_Purge()</td>
<td>OS_QUEUE_Purge()</td>
</tr>
<tr>
<td>OS_Q_Put()</td>
<td>OS_QUEUE_Put()</td>
</tr>
<tr>
<td>OS_Q_PutEx()</td>
<td>OS_QUEUE_PutEx()</td>
</tr>
<tr>
<td>OS_Q_PutBlocked()</td>
<td>OS_QUEUE_PutBlocked()</td>
</tr>
<tr>
<td>OS_Q_PutBlockedEx()</td>
<td>OS_QUEUE_PutBlockedEx()</td>
</tr>
<tr>
<td>OS_Q_PutTimed()</td>
<td>OS_QUEUE_PutTimed()</td>
</tr>
<tr>
<td>OS_Q_PutTimedEx()</td>
<td>OS_QUEUE_PutTimedEx()</td>
</tr>
<tr>
<td>OS_CallISR()</td>
<td>OS_INT_Call()</td>
</tr>
<tr>
<td>OS_CallNestableISR()</td>
<td>OS_INT_CallNestable()</td>
</tr>
<tr>
<td>OS_EnterInterrupt()</td>
<td>OS_INT_Enter()</td>
</tr>
<tr>
<td>OS_EnterIntStack()</td>
<td>OS_INT_EnterIntStack()</td>
</tr>
<tr>
<td>OS_EnterNestableInterrupt()</td>
<td>OS_INT_EnterNestable()</td>
</tr>
<tr>
<td>OS_InInterrupt()</td>
<td>OS_INT_InInterrupt()</td>
</tr>
<tr>
<td>OS_LeaveInterrupt()</td>
<td>OS_INT_Leave()</td>
</tr>
<tr>
<td>OS_LeaveIntStack()</td>
<td>OS_INT_LeaveIntStack()</td>
</tr>
<tr>
<td>OS_LeaveNestableInterrupt()</td>
<td>OS_INT_LeaveNestable()</td>
</tr>
<tr>
<td>OS_DecRI()</td>
<td>OS_INT_DecRI()</td>
</tr>
<tr>
<td>OS_DI()</td>
<td>OS_INT_Disable()</td>
</tr>
<tr>
<td>OS_INTERRUPT_MaskGlobal()</td>
<td>OS_INT_DisableAll()</td>
</tr>
<tr>
<td>OS_EI()</td>
<td>OS_INT_Enable()</td>
</tr>
<tr>
<td>OS_INTERRUPT_UnmaskGlobal()</td>
<td>OS_INT_EnableAll()</td>
</tr>
<tr>
<td>OS_RestoreI()</td>
<td>OS_INT_EnableConditional()</td>
</tr>
<tr>
<td>OS_IncDI()</td>
<td>OS_INT_IncDI()</td>
</tr>
<tr>
<td>OS_INT_PRIO_PRESERVE()</td>
<td>OS_INT_Preserve()</td>
</tr>
<tr>
<td>OS_INTERRUPT_PreserveGlobal()</td>
<td>OS_INT_PreserveAll()</td>
</tr>
<tr>
<td>OS_INTERRUPT_PreserveAndMaskGlobal()</td>
<td>OS_INT_PreserveAndDisableAll()</td>
</tr>
<tr>
<td>OS_INT_PRIO_RESTORE()</td>
<td>OS_INT_Restore()</td>
</tr>
<tr>
<td>OS_INTERRUPT_RestoreGlobal()</td>
<td>OS_INT_RestoreAll()</td>
</tr>
<tr>
<td>OS_EnterRegion()</td>
<td>OS_TASK_EnterRegion()</td>
</tr>
<tr>
<td>OS_LeaveRegion()</td>
<td>OS_TASK_LeaveRegion()</td>
</tr>
<tr>
<td>OS_GetTime()</td>
<td>OS_TIME_GetTicks()</td>
</tr>
<tr>
<td>OS_GetTime32()</td>
<td>OS_TIME_GetTicks32()</td>
</tr>
<tr>
<td>OS_Config_SysTimer()</td>
<td>OS_TIME_ConfigSysTimer()</td>
</tr>
<tr>
<td>V4</td>
<td>V5</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>OS_Timing_GetCycles()</td>
<td>OS_TIME_GetResult()</td>
</tr>
<tr>
<td>OS_Timing_Getus()</td>
<td>OS_TIME_GetResultus()</td>
</tr>
<tr>
<td>OS_GetTime_us()</td>
<td>OS_TIME_Getus()</td>
</tr>
<tr>
<td>OS_GetTime_us64()</td>
<td>OS_TIME_Getus64()</td>
</tr>
<tr>
<td>OS_Timing_Start()</td>
<td>OS_TIME_StartMeasurement()</td>
</tr>
<tr>
<td>OS_Timing_End()</td>
<td>OS_TIME_StopMeasurement()</td>
</tr>
<tr>
<td>OS_AdjustTime()</td>
<td>OS_TICKLESS_AdjustTime()</td>
</tr>
<tr>
<td>OS_GetNumIdleTicks()</td>
<td>OS_TICKLESS_GetNumIdleTicks()</td>
</tr>
<tr>
<td>OS_StartTicklessMode()</td>
<td>OS_TICKLESS_Start()</td>
</tr>
<tr>
<td>OS_StopTicklessMode()</td>
<td>OS_TICKLESS_Stop()</td>
</tr>
<tr>
<td>OS_free()</td>
<td>OS_HEAP_free()</td>
</tr>
<tr>
<td>OS_malloc()</td>
<td>OS_HEAP_malloc()</td>
</tr>
<tr>
<td>OS_realloc()</td>
<td>OS_HEAP_realloc()</td>
</tr>
<tr>
<td>OS_MEMF_Request()</td>
<td>OS_MEMPOOL_Alloc()</td>
</tr>
<tr>
<td>OS_MEMF_Alloc()</td>
<td>OS_MEMPOOL_AllocBlocked()</td>
</tr>
<tr>
<td>OS_MEMF_AllocTimed()</td>
<td>OS_MEMPOOL_AllocTimed()</td>
</tr>
<tr>
<td>OS_MEMF_Create()</td>
<td>OS_MEMPOOL_Create()</td>
</tr>
<tr>
<td>OS_MEMF_Delete()</td>
<td>OS_MEMPOOL_Delete()</td>
</tr>
<tr>
<td>OS_MEMF_FreeBlock()</td>
<td>OS_MEMPOOL_Free()</td>
</tr>
<tr>
<td>OS_MEMF_Release()</td>
<td>OS_MEMPOOL_FreeEx()</td>
</tr>
<tr>
<td>OS_MEMF_GetBlockSize()</td>
<td>OS_MEMPOOL_GetBlockSize()</td>
</tr>
<tr>
<td>OS_MEMF_GetMaxUsed()</td>
<td>OS_MEMPOOL_GetMaxUsed()</td>
</tr>
<tr>
<td>OS_MEMF_GetNumBlocks()</td>
<td>OS_MEMPOOL_GetNumBlocks()</td>
</tr>
<tr>
<td>OS_MEMF_GetNumFreeBlocks()</td>
<td>OS_MEMPOOL_GetNumFreeBlocks()</td>
</tr>
<tr>
<td>OS_MEMF_IsInPool()</td>
<td>OS_MEMPOOL_IsInPool()</td>
</tr>
<tr>
<td>OS_GetObjName()</td>
<td>OS_DEBUG_GetObjName()</td>
</tr>
<tr>
<td>OS_SetObjName()</td>
<td>OS_DEBUG_SetObjName()</td>
</tr>
<tr>
<td>OS_AddLoadMeasurement()</td>
<td>OS_STAT_AddLoadMeasurement()</td>
</tr>
<tr>
<td>OS_STAT_GetTaskExecTime()</td>
<td>OS_STAT_GetExecTime()</td>
</tr>
<tr>
<td>OS_GetLoadMeasurement()</td>
<td>OS_STAT_GetLoadMeasurement()</td>
</tr>
<tr>
<td>OS_ClearTxActive()</td>
<td>OS_COM_ClearTxActive()</td>
</tr>
<tr>
<td>OS_GetNextChar()</td>
<td>OS_COM_GetNextChar()</td>
</tr>
<tr>
<td>OS_OnRx()</td>
<td>OS_COM_OnRx()</td>
</tr>
<tr>
<td>OS_OnTx()</td>
<td>OS_COM_OnTx()</td>
</tr>
<tr>
<td>OS_SendString()</td>
<td>OS_COM_SendString()</td>
</tr>
<tr>
<td>OS_SetRxCallback()</td>
<td>OS_COM_SetRxCallback()</td>
</tr>
<tr>
<td>OS_TraceEnable()</td>
<td>OS_TRACE_Enable()</td>
</tr>
<tr>
<td>OS_TraceEnableAll()</td>
<td>OS_TRACE_EnableAll()</td>
</tr>
<tr>
<td>OS_TraceEnableId()</td>
<td>OS_TRACE_EnableId()</td>
</tr>
<tr>
<td>OS_TraceEnableFilterId()</td>
<td>OS_TRACE_EnableFilterId()</td>
</tr>
<tr>
<td>OS_TraceDisable()</td>
<td>OS_TRACE_Disable()</td>
</tr>
<tr>
<td>OS_TraceDisableAll()</td>
<td>OS_TRACE_DisableAll()</td>
</tr>
<tr>
<td>OS_TraceDisableId()</td>
<td>OS_TRACE_DisableId()</td>
</tr>
</tbody>
</table>
### V4 and V5 API Migration

<table>
<thead>
<tr>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TraceDisableFilterId()</td>
<td>OS_TRACE_DisableFilterId()</td>
</tr>
<tr>
<td>OS_TraceData()</td>
<td>OS_TRACE_Data()</td>
</tr>
<tr>
<td>OS_TraceDataPtr()</td>
<td>OS_TRACE_DataPtr()</td>
</tr>
<tr>
<td>OS_TracePtr()</td>
<td>OS_TRACE_Ptr()</td>
</tr>
<tr>
<td>OS_SetTraceAPI()</td>
<td>OS_TRACE_SetAPI()</td>
</tr>
<tr>
<td>OS_TraceU32Ptr()</td>
<td>OS_TRACE_U32Ptr()</td>
</tr>
<tr>
<td>OS_TraceVoid()</td>
<td>OS_TRACE_Void()</td>
</tr>
<tr>
<td>OS_MPU_AddSanityCheckBuffer()</td>
<td>OS_MPU_SetSanityCheckBuffer()</td>
</tr>
<tr>
<td>OS_GetIntStackBase()</td>
<td>OS_STACK_GetIntStackBase()</td>
</tr>
<tr>
<td>OS_GetIntStackSize()</td>
<td>OS_STACK_GetIntStackSize()</td>
</tr>
<tr>
<td>OS_GetIntStackSpace()</td>
<td>OS_STACK_GetIntStackSpace()</td>
</tr>
<tr>
<td>OS_GetIntStackUsed()</td>
<td>OS_STACK_GetIntStackUsed()</td>
</tr>
<tr>
<td>OS_GetStackBase()</td>
<td>OS_STACK_GetTaskStackBase()</td>
</tr>
<tr>
<td>OS_GetStackSize()</td>
<td>OS_STACK_GetTaskStackSize()</td>
</tr>
<tr>
<td>OS_GetStackSpace()</td>
<td>OS_STACK_GetTaskStackSpace()</td>
</tr>
<tr>
<td>OS_GetStackUsed()</td>
<td>OS_STACK_GetTaskStackUsed()</td>
</tr>
<tr>
<td>OS_GetSysStackBase()</td>
<td>OS_STACK_GetSysStackBase()</td>
</tr>
<tr>
<td>OS_GetSysStackSize()</td>
<td>OS_STACK_GetSysStackSize()</td>
</tr>
<tr>
<td>OS_GetSysStackSpace()</td>
<td>OS_STACK_GetSysStackSpace()</td>
</tr>
<tr>
<td>OS_GetSysStackUsed()</td>
<td>OS_STACK_GetSysStackUsed()</td>
</tr>
<tr>
<td>OS_SetStackCheckLimit()</td>
<td>OS_STACK_SetCheckLimit()</td>
</tr>
<tr>
<td>OS_GetCPU()</td>
<td>OS_INFO_GetCPU()</td>
</tr>
<tr>
<td>OS_GetLibMode()</td>
<td>OS_INFO_GetLibMode()</td>
</tr>
<tr>
<td>OS_GetLibName()</td>
<td>OS_INFO_GetLibName()</td>
</tr>
<tr>
<td>OS_GetModel()</td>
<td>OS_INFO_GetModel()</td>
</tr>
<tr>
<td>OS_GetVersion()</td>
<td>OS_INFO_GetVersion()</td>
</tr>
</tbody>
</table>

### Changed object types:

<table>
<thead>
<tr>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_RSEMA</td>
<td>OS_MUTEX</td>
</tr>
<tr>
<td>OS_CSEMA</td>
<td>OS_SEMAPHORE</td>
</tr>
<tr>
<td>OS_Q</td>
<td>OS_QUEUE</td>
</tr>
<tr>
<td>OS_Q_SRCLIST</td>
<td>OS_QUEUE_SRCLIST</td>
</tr>
<tr>
<td>OS_MEMF</td>
<td>OS_MEMPOOL</td>
</tr>
<tr>
<td>OS_TASK_EVENT</td>
<td>OS_TASKEVENT</td>
</tr>
</tbody>
</table>
Chapter 29
Support
29.1 Contacting support

This chapter should help if any problem occurs and 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 https://segger.com/ticket.
3. You can send an email to support_embos@segger.com*.

The ticket system helps to simplify interactions with the embOS developers and gives the opportunity to share support cases with other developers within the same company.

Please include the following information in the email or ticket:
• Which embOS do you use? (CPU, compiler).
• The embOS version.
• Your embOS registration 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.

*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 30

Performance and Resource Usage
30.1 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-bit CPUs. Depending on which features are being used, even single-chip systems with less than 2 Kbytes ROM and 1 Kbyte RAM can be supported by embOS. The actual performance and resource usage depends on many factors (CPU, compiler, memory model, optimization, configuration, etc.).
30.2 Memory requirements

The memory requirements of embOS (RAM and ROM) differ depending on the used features of the library. The following table shows the memory requirements for the different modules. These values are typical values for a 32-bit CPU and depend on CPU, compiler, and library model used.

<table>
<thead>
<tr>
<th>Module</th>
<th>Memory type</th>
<th>Memory requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>embOS kernel</td>
<td>ROM</td>
<td>1700 bytes</td>
</tr>
<tr>
<td></td>
<td>RAM</td>
<td>71 bytes</td>
</tr>
<tr>
<td>Mailbox</td>
<td>RAM</td>
<td>24 bytes</td>
</tr>
<tr>
<td>Semaphore</td>
<td>RAM</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Mutex</td>
<td>RAM</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Software timer</td>
<td>RAM</td>
<td>20 bytes</td>
</tr>
<tr>
<td>Task event</td>
<td>RAM</td>
<td>0 bytes</td>
</tr>
</tbody>
</table>
30.3 Performance

The following section shows how to benchmark embOS with the supplied example programs.

30.4 Benchmarking

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 high-resolution measurement functions. Example programs for both methods are supplied in the Application directory of your embOS shipment.

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, toolchain, memory model, optimization, configuration, 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 gives an overview about the variations of the context switch time depending on the memory type and the CPU mode:

<table>
<thead>
<tr>
<th>Target</th>
<th>Memory</th>
<th>Time / Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST STM32F756 @ 200 MHz</td>
<td>RAM</td>
<td>1.5us / 260</td>
</tr>
<tr>
<td>Renesas RZ @ 400 MHz</td>
<td>RAM</td>
<td>720ns / 287</td>
</tr>
</tbody>
</table>

All named example performance values in the following section are determined with the following system configuration:

All sources are compiled with IAR Embedded Workbench version 6.40.5, OS_LIBMODE_XR and high optimization level. embOS version 4.14 has been used; values may differ for different builds.
30.4.1 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`:

```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); // Synchronize to tick to avoid jitter
        // 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_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_Start(); // Start multitasking
    return 0;
}
```
30.4.1.1 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:

\[
t_{CS} = t_{CD} - t_{AB}
\]
30.4.1.2 Example measurements Renesas RZ, Thumb2 code in RAM

Task switching time has been measured with the parameters listed below:

- embOS Version V4.14
- Application program: OS_MeasureCST_Scope.c
- Hardware: Renesas RZ processor with 399MHz
- Program is executing in RAM
- Thumb2 mode is used
- Compiler used: SEGGER Embedded Studio V2.10B (GCC)
- CPU frequency (f_{CPU}): 399.0MHz
- CPU clock cycle (t_{Cycle}): \( t_{Cycle} = \frac{1}{f_{CPU}} = \frac{1}{399.0MHz} = 2.506\text{ns} \)

### Measuring \( t_{AB} \) and \( t_{CD} \)

\( t_{AB} \) is measured as 480ns.

The number of cycles calculates as follows:

\[
\text{Cycles}_{AB} = \frac{t_{AB}}{t_{Cycle}} = \frac{480\text{ns}}{2.506\text{ns}} = 191.54 \text{ Cycles} \\
\Rightarrow 192 \text{ Cycles}
\]

\( t_{CD} \) is measured as 12000ns.

The number of cycles calculates as follows:

\[
\text{Cycles}_{CD} = \frac{t_{CD}}{t_{Cycle}} = \frac{1200\text{ns}}{2.506\text{ns}} = 478.85 \text{ Cycles} \\
\Rightarrow 479 \text{ Cycles}
\]

### Resulting context switching time and number of cycles

The time which is required for the pure context switch is:

\[
t_{CS} = t_{CD} - t_{AB} = 479 \text{ Cycles} - 192 \text{ Cycles} = 287 \text{ Cycles} \\
\Rightarrow 287 \text{ Cycles (0.72us @399 MHz)}.\]
30.4.1.3 Measurement with high-resolution timer

The context switch time may be measured with the high-resolution timer. Refer to section *High-resolution measurement* on page 304 for detailed information about the embOS high-resolution measurement.

The example *OS_MeasureCST_HRTimer_embOSView.c* uses a high resolution timer to measure the context switch time from a low priority task to a high priority task and displays the results on embOSView.

```c
#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_U32 Time; // Timer values

/*********************************************************************
*       HPTask
*/
static void HPTask(void) {
    while (1) {
        OS_TASK_Suspend(NULL); // Suspend high priority task
        OS_TIME_StopMeasurement(& _Time); // Stop measurement
    }
}

/*********************************************************************
*       LPTask
*/
static void LPTask(void) {
    char acBuffer[100]; // Output buffer
    OS_U32 MeasureOverhead; // Time for Measure Overhead
    OS_U32 v;

    // Measure Overhead for time measurement so we can take
    // this into account by subtracting it
    OS_TIME_StartMeasurement(& MeasureOverhead);
    OS_TIME_StopMeasurement(& MeasureOverhead);
    // Perform measurements in endless loop
    while (1) {
        OS_TASK_Delay(100); // Sync. to tick to avoid jitter
        OS_TIME_StartMeasurement(& _Time); // Start measurement
        OS_TASK_Resume(& TCBHP);
        // Resume high priority task to force task switch
        v = OS_TIME_GetResult(& _Time);
        v -= OS_TIME_GetResult(& MeasureOverhead);
        v = OS_TIME_ConvertCycles2ns(v); // Convert cycles to nano-seconds
        sprintf(acBuffer, "Context switch time: %lu.%lu usec\r",
                v / 1000uL, v % 1000uL);
        OS_COM_SendString(acBuffer);
    }
}
```

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 31

Supported Development Tools
31.1 Overview

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 may use 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.

Reentrance
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 real-time operating system 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.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cooperative multitasking</td>
<td>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.</td>
</tr>
<tr>
<td>Counting semaphore</td>
<td>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.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit. The “brain” of a microcontroller; the part of a processor that carries out instructions.</td>
</tr>
<tr>
<td>Critical region</td>
<td>A section of code which must be executed without interruption.</td>
</tr>
<tr>
<td>Event</td>
<td>A message sent to a single, specified task that something has occurred. The task then becomes ready.</td>
</tr>
<tr>
<td>Interrupt Handler</td>
<td>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).</td>
</tr>
<tr>
<td>ISR</td>
<td>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).</td>
</tr>
<tr>
<td>Mailbox</td>
<td>A data buffer managed by an RTOS, used for sending messages to a task or interrupt handler.</td>
</tr>
<tr>
<td>Message</td>
<td>An item of data (sent to a mailbox, queue, or other container for data).</td>
</tr>
<tr>
<td>Multitasking</td>
<td>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.</td>
</tr>
<tr>
<td>Mutex</td>
<td>A data structure used for managing resources by ensuring that only one task has access to a resource at a time.</td>
</tr>
<tr>
<td>NMI</td>
<td>Non-Maskable Interrupt. An interrupt that cannot be masked (disabled) by software. Example: Watchdog timer interrupt.</td>
</tr>
<tr>
<td>Preemptive multitasking</td>
<td>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.</td>
</tr>
<tr>
<td>Process</td>
<td>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.</td>
</tr>
<tr>
<td>Processor</td>
<td>Short for microprocessor. The CPU core of a controller.</td>
</tr>
<tr>
<td>Priority</td>
<td>The relative importance of one task to another. Every task in an RTOS has a priority.</td>
</tr>
<tr>
<td>Priority inversion</td>
<td>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.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Queue</td>
<td>Like a mailbox, but used for sending larger messages, or messages of individual size, to a task or an interrupt handler.</td>
</tr>
<tr>
<td>Ready</td>
<td>Any task that is in &quot;ready state&quot; will be activated when no other task with higher priority is in &quot;ready state&quot;.</td>
</tr>
<tr>
<td>Resource</td>
<td>Anything in the computer system with limited availability (for example memory, timers, computation time). Essentially, anything used by a task.</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-time Operating System.</td>
</tr>
<tr>
<td>Running task</td>
<td>Only one task can execute at any given time. The task that is currently executing is called the running task.</td>
</tr>
<tr>
<td>Scheduler</td>
<td>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.</td>
</tr>
<tr>
<td>Semaphore</td>
<td>A data structure used for synchronizing tasks.</td>
</tr>
<tr>
<td>Software timer</td>
<td>A data structure which calls a user-specified routine after a specified delay.</td>
</tr>
<tr>
<td>Stack</td>
<td>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.</td>
</tr>
<tr>
<td>Superloop</td>
<td>A program that runs in an infinite loop and uses no real-time kernel. ISRs are used for real-time parts of the software.</td>
</tr>
<tr>
<td>Task</td>
<td>A program running on a processor. A multitasking system allows multiple tasks to execute independently from one another.</td>
</tr>
<tr>
<td>Thread</td>
<td>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)</td>
</tr>
<tr>
<td>Tick</td>
<td>The OS timer interrupt. Typically equals 1 millisecond.</td>
</tr>
<tr>
<td>Time slice</td>
<td>The time (number of system ticks) for which a task will be executed until a round-robin task change may occur.</td>
</tr>
</tbody>
</table>