pC/OS Reference

V1.95c

Haftungsausschluß
Der Autor übernimmt keinerlei Haftung für durch diesen Code entstandene oder entstehende Schäden an Hard- und Software. Er versichert lediglich, daß er den Code vielfältigen Test's auf unterschiedlicher Hardware unterzogen hat, um seinerseits keine Fehler bestehen zu wissen. Sollten dennoch Fehler auftauchen oder Vorschläge zur Verbesserung des Codes an den Autor weitergegeben werden, so ist dieser bestrebt, Fehler schnellstmöglich auszumerzen oder Vorschläge einzuarbeiten.

liability exclusion
The author takes over no liability for through this code originated or emerging damages to hardware and software. He assures merely that he subjected the code of diverse tests on different hardware, about for his part no mistakes to know exists. Mistakes nevertheless should appear or suggestions are passed on at the author to the improvement of the code, so this is striving, mistakes fastest to wipe out or to incorporate suggestions.

To this for beginners at the fastest to understand real time operating system belongs µC/OS from Jean J. Labrosse (see http://www.micrium.com).
In the versions 1.xx it can administer up to 63 applikationtasks with ever different priorities and belongs to the real time operating systems with the lowest storage demand.
pC/OS was based on the original version µC/OS 1.00 from the Embedded Systems Programming Magazine(1992) developed further.

Since is exchanged by direct transfer of pointers in the original version data between the tasks, and consequently no guarantee for the free usability of the sending-buffers after transfer to another task exists and, much important, the recipient a pointer in the data field of another tasks gets (pointer-error / longitudinal mistake / manipulations among others) I altered the mechanisms for Message-Box and Queue accordingly in a way that the data about a kernel-internal Buffer now are handed over to the recipient. This means that the kernel copies to transferring data into an individual buffer and this copies also again itself with transfer to the recipient in the buffer prepared through the recipient. This admittedly entails a higher storage demand for Queue, secures the processes for it most extensive, however (for real-mode) of each other from.
Kernel constants were transferred in the CODE-Area for security reasons furthermore.
Furthermore I have add pipes, eventgroups, timerservice and dynamic memory management.

In order to declare these alterations unequivocally, I have altered the name, ajar at the always bigger nascent original "µC/OS", on pC/OS like "pico-C..".

Special to: Priority Inversion, the problem and the solutions (actual only in german - sorry !)

known bug:
If a task with lower priority waits for a recource, and a task with higher priority this recource places, so the asleep Task is put into the ready-state. Since the task with higher priority further-runs, this cannot finish reading again the same recource since the lower task 'prematurely' comes with implementation with the return-code OS_TIMEOUT back otherwise.

Since the terms of the various IPC's with data transport are not unique and different implementations can be found in different RTOS systems under the same terms, here is an overview of all the IPC's implemented in the pC/OS.

Semaphor	"Semaphor"		
	- binary/blocking		serialize access to shared objects(s)
	- counting		counting of an event
Mutual-Excusion	"Mutex"		serialze access to shared object(s) w/o the risk of priority inversion but internaly more complex
Event-Group	"EventGroup"		a group of events, w/o counting but with AND/OR options for pending
Mailbox	"MailBox"		a single pointer/value transfer - one pointer/value is temp stored (a one pointer/value buffer)
Byte-Queue	"Queue"	Thread A or ISR bytes size Thread B or ISR	byte-by-byte transfer (FIFO / LIFO) - the bytes are temp stored in the queue (a serial transfer-buffer for bytes)
Package-Queue	"Pipe"	Thread A or ISR max size packages deep Thread B or ISR	package-by-package transfer (FIFO / LIFO) - the hole packages are temp stored in the pipe (a serial transfer-buffer for packages / C-structs)
Smart-Pointer-Queue	"SmartMessageQueue"	Thread A or ISR pointer deep Thread B or ISR	pointer-by-pointer transfer (FIFO / LIFO) - the pointers are temp stored and the memory-heap ownership behind that pointers is transfered too (a serial transfer-buffer for pointers inspired by C++ smart-pointers)

Please note that some functions are declared under the same name as the original but with modified parameters or pointers.

User-Functions:

Task-Control:	Description
OS_Init	Initialization of the kernel
OS_Start	Begin the kernel services
OS_TaskCreate	Generating of a task
OS_ChangePrio	Alteration of the priority of the active task
OS_TaskChangePrio	Alteration of the priority of a active/ready task
OS_TaskDelete	Deletion of a active/ready task
OS_TaskIdDelete	Deletion of a active/ready task by unique ID
OS_TaskGetStatus	returns the current status of the task
OS_TaskIdGetStatus	returns the current status of a task by unique ID
OS_TaskGetID	returns the unique ID of a task
OS_TaskGetPrio	returns the priority of a task
OS_TaskIdDestroy	Deletion of a task by unique ID, even if he waits for an IPC or a mutex has occupied & release all memory allocations
OS_TaskSuspend	Suspending of a task
OS_TaskIdSuspend	Suspending of a task by unique ID
OS_TaskResume	Reactivation of a suspended task
OS_TaskIdResume	Reactivation of a suspended task by unique ID
OS_TimeDly	Put current task for certain time sleeps
OS_TimeDlyResume	Reactivation of an asleep task before course of the put in time
OS_TimeDlyIdResume	Reactivation of an asleep task by unique ID before course of the put in time
OS_Lock	It suppresses the Sheduler (no taskswitch)
OS_Unlock	Reactivation of the Sheduler (taskswitch at event or time)
OS_GetRev	Returns pointer on kernel revision

Dynamic-Memory:	Description
OS_MemoryInit	Generates of the memory pool
OS_MemAlloc	Allocation of memory
OS_MemFree	Release of allocated memory
OS_MemFreeSize	It returns the amount of free memory
OS_MemVerifyPtr	Check if the pointer points inside the memory pool

SmartMessageQueue:	Description
OS_SmqInit	Initialsation of a SmartMessageQueue
OS_SmqInfo	Information about a SmartMessageQueue catches up with
OS_SmqClear	Delete all messages in a SmartMessageQueue
OS_SmqPost	Send a message into a SmartMessageQueue
OS_SmqFrontPost	Send a message to the beginning of a SmartMessageQueue
OS_SmqPostAbbort	Abborts waiting of a sending task (highest waiting prio) onto a SmartMessageQueue
OS_SmqPend	Wait for a message from a SmartMessageQueue
OS_SmqPendAbbort	Abborts waiting of a receiving task (highest waiting prio) onto a SmartMessageQueue

Mailboxes:	Description
OS_MboxInit	Initialisation of a Mailbox
OS_MboxPost	Send data to task with higher priority recipients of this Mailbox
OS_MboxPostAbbort	Abborts waiting of a sending task (highest waiting prio) onto a mailbox
OS_MboxPend	Wait for data from a Mailbox
OS_MboxPendAbbort	Abborts waiting of a receiving Tasks (highest waiting prio) on a Mailbox

Queues:	Description
OS_QueueInit	Initialsation of a Queue
OS_QueueInfo	Information about a Queue catches up with
OS_QueuePost	Send data into a Queue
OS_QueueFrontPost	Send data to the beginning of a Queue
OS_QueuePostAbbort	Abborts waiting of a sending task (highest waiting prio) onto a queue
OS_QueuePend	Wait for data from a Queue
OS_QueuePendAbbort	Abborts waiting of a receiving Tasks (highest waiting prio) on a Queue
OS_QueueClear	Delete all data in a Queue

Pipes:	Description
OS_PipeInit	Initialisation of a Pipe
OS_PipeInfo	Information about a Pipe catches up with
OS_PipePost	Send data into a Pipe
OS_PipeFrontPost	Send data to the beginning of a Pipe
OS_PipePostAbbort	Abborts waiting of a sending task (highest waiting prio) onto a pipe
OS_PipePend	Wait for data from a Pipe
OS_PipePendAbbort	Abborts waiting of a receiving Tasks (highest waiting prio) on a Pipe
OS_PipeClear	Delete all data in a Pipe

inter-core AMP-Pipes:	Description
OS_IccInit	Initialisation of the AMP-Pipe pair
OS_IccInfo	Information about the AMP-Pipe pair catches up with
OS_IccPost	Send data to the other core
OS_IccPend	wait for data from the other core
OS_IccClear	Delete all data of the AMP-Pipe towards the other core

Semaphores:	Description
OS_SemInit	Initalisation of a Semaphore
OS_SemAccept	wait for event and returns number
OS_SemPost	Decontrol of a busy Semaphores / places event
OS_SemPend	Cover one Semaphore / waits on event
OS_SemPendAbbort	Abborts waiting of a Tasks (highest waiting prio) on a Semaphore
OS_SemClear	Clear the Semaphore-Counter

Mutexes:	Description
OS_MutexCreate	Generating of a Mutex
OS_MutexPost	Decontrol of a Mutex
OS_MutexPend	Cover the Mutex
OS_MutexPendAbbort	Abborts waiting of a Tasks (highest waiting prio) on a Mutex

Event-Groups:	Description
OS_EvgInit	Initialisation of a Eventgroup
OS_EvgPost	Place one/many events of an Evengroup
OS_EvgPend	Wait for arriving an or several events of an Eventgroup
OS_EvgPendAbbort	Abborts waiting of a task (highest waiting prio) onto a Eventgroup

Timer-Service:	Description
OS_TimerCreate	Generating of a Timer
OS_TimerDelete	Delete of a generated Timer
OS_TimerStart	(Re-)Start of a generated Timer
OS_TimerStop	Stop of a generated Timer
OS_TimerGetState	returns the status of a generated Timer
OS_TimerGetRemain	returns the remaining time of a running Timer

System-Ticks:	Description
OS_TimeSet	Set ticker to value
OS_TimeGet	returns current ticker-value

Interrupts:	Description
OS_IntEnter	Registration of a called ISR
OS_IntExit	End of a called ISR

History:	Description

OS_HistoryPost	Write entry in History
OS_HistoryRead	return first History-entry and delete this in the table

Error-Codes:

Name	Decimal_Value	Description
OS_SUCCESS / OS_NO_ERR	0	no errors
OS_PARAM_ERR	1	parameter wrong / error
OS_TIMEOUT	10	timeout condition occurs during waiting for a resource
OS_PRIO_EXIST	11	under this priority, a other Task or Mutex is registered
OS_TASK_NOT_EXIST	12	under this priority, no Task is registered
OS_TASK_SUSP_PRIO	13	under this suspend priority, no Task is registered
OS_TASK_NOT_SUSP	14	the task is not suspended
OS_TASK_NOT_RDY	15	the task is not ready
OS_SUSPEND_IDLE	16	the Idle-Task cannot be suspended
OS_PRIO_INVALID	17	the value of priority is bigger OS_MIN_PRIO
OS_TIME_NOT_DLY	18	the task doesn't sleep
OS_SEM_ERR	30	internal error in Semaphore-handling
OS_SEM_NODATA	31	Semaphore occupied / no event (with OS_NO_SUSP)
OS_SEM_OVF	32	Error in the Semaphore-handling (Counter too big)
OS_MUX_ERR	40	Error in Mutex-handling
OS_MUX_NOACC	41	Mutex occupied (with OS_NO_SUSP)
OS_MUX_USED	42	to change Task have a Mutex occupied
OS_MBOX_FULL	50	Mailbox fully (with OS_NO_SUSP)
OS_MBOX_NODATA	51	no message in Mailbox (with OS_NO_SUSP)
OS_Q_FULL	60	Queue fully (with OS_NO_SUSP)
OS_Q_NODATA	61	no byte in Queue (with OS_NO_SUSP)
OS_Q_CLEAR	62	Queue was cleared during waiting
OS_SMQ_ERR	70	Error in SmartMessageQueue handling
OS_SMQ_FULL	71	SmartMessageQueue fully (with OS_NO_SUSP)
OS_SMQ_NODATA	72	no package in SmartMessageQueue (with OS_NO_SUSP)
OS_SMQ_CLEAR	73	SmartMessageQueue was cleared during waiting
OS_P_FULL	80	Pipe fully (with OS_NO_SUSP)
OS_P_NODATA	81	no package in Pipe (with OS_NO_SUSP)
OS_P_CLEAR	82	Pipe was cleared during waiting
OS_P_LEN_ERR	83	Package too long
OS_EVG_ERR	90	Error in Event-Group handling
OS_EVG_NOE	91	Event(s) appeared not (with OS_NO_SUSP)
OS_TMR_NO_TIME	110	no time given on TimerCreate
OS_TMR_NOT_EXIST	111	timer was not created / registered
OS_TMR_EXIST	112	timer still created / registered
OS_MEM_ERR	120	parameter error / internal error
OS_MEM_OVF	121	memeory overflow
OS_HIS_END	130	no (more) entry existing
OS_ICC_ERR	140	parameter error / internal error
OS_ICC_NODATA	141	no package in AMP-Pipe (with OS_NO_SUSP)
OS_ICC_LEN_ERR	142	Package/Message too long
OS_ICC_FULL	143	AMP-Pipe fully (with OS_NO_SUSP)

Configuration of the kernel

The pC/OS kernel can be configured in addition to the to-use hardware port some numbers of ways to configure Services/IPCs as well as to reduce the memory requirements - code-size for the compilers "unused code" may not clearly identify and RAM - available. These are in the file "OS_cfg.h" together.

components configuration	description
OS_SYSTEM_TICKS_PER_SEC	system ticks per secound
OS_TIMER_TICKS_PER_SEC	timer ticks per secound (see Timer-Service / TIMERS), can be tick faster than the kernel(system)-ticks
OS_TASK_EXT_EN	include code for extended TASKS services
OS_TASK_DESTROY_EN	include code for destroy pending/waiting TASKS, needs OS_TASK_EXT_EN too
OS_SEM_EN	include code for SEMAPHORES
OS_SEM_EXT_EN	include code for extended SEMAPHORES services
OS_MUX_EN	include code for MUTEXES
OS_SMQ_EN	include code for SMART-MESSAGE-QUEUE
OS_MBOX_EN	include code for MAILBOXES
OS_Q_EN	include code for QUEUES
OS_P_EN	include code for PIPES
OS_EVG_EN	include code for EVENTGROUPS
OS_TMR_EN	include code for TIMERS
OS_MEM_EN	include code for MEMORY-MANAGER
OS_HIS_EN	include code for HISTORY
OS_ICC_EN	include code for inter-core AMP-PIPES
OS_STK_CHECK_EN	check end-of-stack of old task during context switch
OS_STK_CHECK_FILL	fill stack with 0xEF pattern to get the deep of use

user configuration	description
OS_MAX_TASKS	max created tasks in hole system --> max 64 !
OS_MIN_PRIO	lowest possible prio --> max 64 !
OS_IDLE_STK_SIZE	idle stack size in OS_STK_TYPE with fix (OS_MIN_PRIO - 1) as prio for idle task
OS_TMR_PRIO	timer task prio, if OS_TMR_EN is not 0
OS_TMR_STK_SIZE	timer stack size in OS_STK_TYPE, if OS_TMR_EN is not 0
OS_MAX_HISTORY	history entries, if OS_HIS_EN is not 0
OS_STK_RESERVE	space between real end-of-stack and check-point in OS_STK_TYPE, if OS_STK_CHECK_EN is not 0

to OS_MAX_TASKS and OS_MIN_PRIO:
If a system is needed with 5 tasks, 2 of which tasks are using a shared mutex, you need OS_MAX_TASKS = 7 (including a Mutex and Idle-Task) and OS_MIN_PRIO = 8 whereas the Idle-Task then gets the prio 7 and all other Tasks and the Mutex gets higher priorities (0..6). If the timer-service should used too, it must be this timer task additionally involved.

Managed / Unmanaged Interrupt Service Routinen (ISR)

The pC/OS kernel must be notified when an ISR is running and needs to consider when leaving this if a process change is "preemptive" required. There are, depending on the hardware and used implementation, two ways:

managed ISR	the IRQ entry / exit is central and informs the kernel (see eg. ARM7TDMI ports)
unmanaged ISR	each ISR is independent and is jumped out directly from the vector table -> the kernel must be informed. (see eg. Cortex-Mx ports)

In managed-ISRs the central ISR entry / exit code informs the kernel, so that the ISR itself must not observed for the kernel. In unmanged ISR however, at first `OS_IntEnter()` is to call and at end / at last `OS_IntExit()` is to call !

managed ISR

```
    PUBLIC  OSirqISR
    CODE32
OSirqISR
    ; save registers
    ; register interrupt on kernel
    ; read Interrupt vector for ...
    ; ... this event (eg MyManaged_ISR)

    ; call handler (eg MyManaged_ISR) -->

    ; call OS_IntExit()
    ; restore registers
    END

void MyManaged_ISR(void)
{
    .
    .             // my ISR code
    .
}
```

unmanaged ISR

```
void MyUnManaged_ISR(void)
{
    OS_IntEnter();
    .
    .             // my ISR code
    .
    OS_IntExit();
}
```

Task-Control

OS_Init

```
void OS_Init(void)
```

Initialize the Kernel and installs the Idle-Task. This function must be called before all other kernelservices at the system initialization once.

Parameters

none

Return Value

none

Example

```
void main(void)
{
     .
     .
     OS_Init();
     .
     .
     OS_Start();
}
```

OS_Start

```
void OS_Start(void)
```

Starts the kernel. This function activates the sheduler and never returns back to caller.

Parameters

none

Return Value

none

Example

```
void main(void)
{
    .
    .
    OS_Init();
    .
    .
    OS_Start();
}
```

OS_TaskCreate

```
U08 OS_TaskCreate(void (OS_FAR *task)(void *dptr), void  *data, void  *pstk, U16 stksize, U08  prio)
```

Installs a new Task. This function initializes the Task-Control-Block and writes down the new Task with his Stack and the call parameters. This can from main() take place out in term during the initialization as well as another Task.

Parameters

*dptr	pointer to task-code
*data	pointer to parameter of this task
*pstk	pointer to stack of this task
stksize	stack size in OS_STK_TYPE
prio	priority of this task

Return Value

OS_NO_ERR	Task successfully positioned
OS_PRIO_EXIST	under this priority, already a Task exists
OS_PRIO_INVALID	this priority is reserved for the Idle-Task or the value of priority is bigger than OS_MIN_PRIO

Example

```
OS_STK_TYPE  Task1Stack[STK_SIZE];
U08          Task1Data;

void OS_FAR Task1(void *data);                // forward declaration
.
.

void main(void)
{
    U08  state;

    .
    .
    OS_Init();
    .
    state = OS_TaskCreate(Task1, (void *)&Task1Data, Task1Stack, STK_SIZE, 18);
    .
    OS_Start();
}

.
.

void OS_FAR Task1(void *data)
{
    .
    .
    while(1)
    {
        .
        .
        .
    }
}
```

OS_ChangePrio

```
U08 OS_ChangePrio(U08 newp)
```

Change the priority of the current Tasks. This function can be used in order to change the priority of the tasks on reason of an event for example.

Parameters

newp	new priority of this task

Return Value

OS_NO_ERR	Priority successfully changed
OS_PRIO_EXIST	under this priority, already a task exists
OS_PRIO_INVALID	this priority is reserved for the Idle-Task or the value of priority is bigger than OS_MIN_PRIO
OS_MUX_USED	to change Task have a Mutex occupied

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_ChangePrio(38);
        .
    }
}
```

OS_TaskChangePrio

```
U08 OS_TaskChangePrio(U08 oldp, U08 newp)
```

Change the priority of a Tasks. This function can be used in order to change the priority of a running/ready tasks on reason of an event for example.
The priority of Tasks, waiting on a resource (Semaphore/Queue/Pipe/..) can not changed, because this state is visible for the kernel but not the exact resource itself.

Parameters

oldp	actual/old priority of the task
newp	new priority of this task

Return Value

OS_NO_ERR	Priority successfully changed
OS_PRIO_EXIST	under this priority, already a task exists
OS_PRIO_INVALID	this priority is reserved for the Idle-Task or the value of priority is bigger than OS_MIN_PRIO
OS_TASK_NOT_RDY	the Task is not in RUNNING/READY-state and so the priority can not changed
OS_MUX_USED	to change Task have a Mutex occupied

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_TaskChangePrio(38, 25);
        .
    }
}
```

OS_TaskDelete

```
U08 OS_TaskDelete(U08 prio)
```

Delete the given task. This function can be used, about for example on reason of an event the task too ending/clearing. This task is distant from the Task-Control-Table afterwards. Allocated resources of the tasks are not released automatically on that occasion.
In order to later be able to execute this task again, it must be positioned again by means of OS_TaskCreate regularly.
Tasks, waiting on a resource (Semaphore/Queue/Pipe/..) can not be deleted, because this state is visible for the kernel but not the exact resource itself.

Parameters

prio	priority of the task to delete

Return Value

OS_NO_ERR	Task deleted
OS_TASK_NOT_EXIST	under this priority, no task exists
OS_TASK_NOT_RDY	the Task is not in RUNNING/READY-state and so the task can not be deleted
OS_MUX_USED	to delete Task have a Mutex occupied

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;
    .
    .
    while(1)
    {
        .
        state = OS_TaskDelete(OS_PRIO_SELF);
        .
    }
}
```

OS_TaskIdDelete

```
U08 OS_TaskIdDelete(U08 id)
```

Delete the given task by unique ID. This function can be used, about for example on reason of an event the task too ending/clearing. This task is distant from the Task-Control-Table afterwards. Allocated resources of the tasks are not released automatically on that occasion.
In order to later be able to execute this task again, it must be positioned again by means of OS_TaskCreate regularly.
Tasks, waiting on a resource (Semaphore/Queue/Pipe/..) can not be deleted, because this state is visible for the kernel but not the exact resource itself.

Parameters

id	unique ID of the task to delete

Return Value

OS_NO_ERR	Task deleted
OS_TASK_NOT_EXIST	>under this ID, no task exists
OS_TASK_NOT_RDY	the Task is not in RUNNING/READY-state and so the task can not be deleted
OS_MUX_USED	to delete Task have a Mutex occupied

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;
    .
    .
    while(1)
    {
        .
        state = OS_TaskIdDelete(3);
        .
    }
}
```

OS_TaskGetStatus

```
U08 OS_TaskGetStatus(U08 prio)
```

Returns the current status of the given task.

Parameters

prio	priority of the task

Return Value

status	see "TASK STATUS", Bitmask

Example

```
void OS_FAR Task1(void *data)
{
    U08  status;
    .
    .
    while(1)
    {
        .
        status = OS_TaskGetStatus(6);
        .
    }
}
```

OS_TaskIdGetStatus

```
U08 OS_TaskIdGetStatus(U08 id)
```

Returns the current status of the given task by unique ID.

Parameters

id	unique ID of the task

Return Value

status	see "TASK STATUS", Bitmask

Example

```
void OS_FAR Task1(void *data)
{
    U08  status;
    .
    .
    while(1)
    {
        .
        status = OS_TaskIdGetStatus(2);
        .
    }
}
```

OS_TaskGetID

```
U08 OS_TaskGetID(U08 prio)
```

Returns the unique-ID to the specified tasks. These can later be used to e.g. a task - even if his priority have changed or has just a mutex in use - a violent end (`OS_TaskDestroy()`).

Parameters

prio	current priority of the task

Return Value

id	the unique-ID of this task

Example

```
U08  idT1;

void OS_FAR Task1(void *data)
{
    .
    idT1 = OS_TaskGetID(OS_PRIO_SELF);
    .
    while(1)
    {
        .
        .
    }
}
```

OS_TaskGetPrio

```
U08 OS_TaskGetPrio(U08 id)
```

Returns the priority to the specified tasks by unique ID.

Parameters

id	unique ID of the task

Return Value

prio	the priority of this task

Example

```
U08  prioT1;

void OS_FAR Task1(void *data)
{
    .
    prioT1 = OS_TaskGetPrio(2);
    .
    while(1)
    {
        .
        .
    }
}
```

OS_TaskIdDestroy

```
U08 OS_TaskIdDestroy(U08 id)
```

Removes the specified task completely independently of his status. This feature can be used to e.g. on basis of an event the task is to cancel. This task is then from the Task-Control-Table, from possibly registered IPCs (Semaphore/MBox/Queue/Pipe/..) and the Memory Manager completely removed.
If the tasks at this stage, a mutex has occupied, it will be released and a user-callback function is called to make any necessary reinitialization of the affected hardware, etc. (see OSMutexReInitResource () in "pC_OS_userCB.c"). But this can be done only for one mutex (the last). If the task have two mutexes occupied, it is only the last released!
In order to later be able to execute this task again, it must be positioned again by means of OS_TaskCreate regularly.
ATTENTION:
During this task will be removed from the Task-Control-Table and from the IPCs all interrupts are blocked because a suitable event for this task could result in a access/update conflict. During the subsequent clean up the memory-manager the interrupts are allowed again but sheduling is suppressed to prevent a simultaneous re-booting this task using OS_TaskCreate() (priority of this task).

Parameters

id	unique-ID of the task to destroy

Return Value

OS_NO_ERR	task completely removed
OS_TASK_NOT_EXIST	the specified task does not exist
OS_TASK_NOT_RDY	the task could not be completely removed from the IPCs or a mutex
OS_MEM_ERR	during deallocating the memory allocations of this task an error occurs

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;
    .
    .
    while(1)
    {
        .
        state = OS_TaskIdDestroy(2);
        .
    }
}
```

OS_TaskSuspend

```
U08 OS_TaskSuspend(U08 prio)
```

Suspends a task from the implementation through the kernel. This function can be used in order to deactivate a task for a time on reason of an event for example.

Parameters

prio	priority of task to suspending

Return Value

OS_NO_ERR	Task successfully suspended
OS_SUSPEND_IDLE	the Idle-Task cannot be suspended
OS_PRIO_INVALID	the value of priority is bigger than OS_MIN_PRIO
OS_TASK_SUSP_PRIO	under this priority, no Task is registered
OS_MUX_USED	to suspend Task have a Mutex occupied

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_TaskSuspend(24);
        .
    }
}
```

OS_TaskIdSuspend

```
U08 OS_TaskIdSuspend(U08 id)
```

suspends a task given by unique ID from the implementation through the kernel. This function can be used in order to deactivate a task for a time on reason of an event for example.

Parameters

id	unique ID of task to suspending

Return Value

OS_NO_ERR	Task successfully suspended
OS_SUSPEND_IDLE	the Idle-Task cannot be suspended
OS_TASK_SUSP_PRIO	under this ID, no Task is registered
OS_MUX_USED	to suspend Task have a Mutex occupied

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_TaskIdSuspend(4);
        .
    }
}
```

OS_TaskResume

```
U08 OS_TaskResume(U08 prio)
```

Reactivate a suspended Task. This function can be used in order to activate a suspended task again on reason of an event for example.

Parameters

prio	priority of suspended task

Return Value

OS_NO_ERR	Task successfully reactivated
OS_TASK_NOT_SUSP	the Task is not suspended
OS_PRIO_INVALID	the value of priority is bigger than OS_MIN_PRIO
OS_TASK_NOT_EXIST	under this priority, no Task is registered
OS_MUX_USED	to resume Task have a Mutex occupied

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_TaskResume(24);
        .
    }
}
```

OS_TaskIdResume

```
U08 OS_TaskIdResume(U08 id)
```

Reactivate a suspended Task by given unique ID. This function can be used in order to activate a suspended task again on reason of an event for example.

Parameters

id	unique ID of suspended task

Return Value

OS_NO_ERR	Task successfully reactivated
OS_TASK_NOT_SUSP	the Task is not suspended
OS_TASK_NOT_EXIST	under this ID, no Task is registered
OS_MUX_USED	to resume Task have a Mutex occupied

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_TaskIdResume(4);
        .
    }
}
```

OS_TimeDly

```
void OS_TimeDly(U16 ticks)
```

If puts the current task for kernel-ticks sleeps. This function can be used in order to let pass a defined time on reason of an event for example. ATTENTION! With the parameter OS_SUSPEND (0), the Task for always is deactivated and can never be activated again.

Parameters

ticks	kernel-ticks as sleeping-time (1...65535)

Return Value

none

Example

```
void OS_FAR Task1(void *data)
{
    .
    .
    while(1)
    {
        .
        OS_TimeDly(200);
        .
    }
}
```

OS_TimeDlyResume

```
U08 OS_TimeDlyResume(U08 prio)
```

Breaks off the wait of a tasks prematurely. This function can be used in order to prematurely activate an asleep task again on reason of an event for example.

Parameters

prio	priority of sleeping task

Return Value

OS_NO_ERR	Task successfully wakened
OS_TIME_NOT_DLY	the Task doesn't sleep
OS_PRIO_INVALID	the value of priority is bigger than OS_MIN_PRIO
OS_TASK_NOT_EXIST	under this priority, no Task is registered

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_TimeDlyResume(24);
        .
    }
}
```

OS_TimeDlyIdResume

```
U08 OS_TimeDlyIdResume(U08 id)
```

Breaks off the wait of a tasks prematurely by unique ID. This function can be used in order to prematurely activate an asleep task again on reason of an event for example.

Parameters

id	unique ID of sleeping task

Return Value

OS_NO_ERR	Task successfully wakened
OS_TIME_NOT_DLY	the Task doesn't sleep
OS_TASK_NOT_EXIST	under this ID, no Task is registered

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_TimeDlyIdResume(4);
        .
    }
}
```

OS_Lock

`void OS_Lock(void)`

If turns off the sheduler. This function can be used, about for example atomic (not under-breakable) to be able to execute processes, without task-switches. Interrupts still are served.

Parameters

none

Return Value

none

Example

```
void OS_FAR Task1(void *data)
{
    .
    .
    while(1)
    {
        .
        OS_Lock();
        .
        .         // not under-breakable part
        .
        OS_Unlock();
        .
    }
}
```

OS_Unlock

```
void OS_Unlock(void)
```

If switches on the sheduler again. This function is used, about for example atomic (not under-breakable) to complete processes and to make a taskswitch again possible.

Parameters

none

Return Value

none

Example

```
void OS_FAR Task1(void *data)
{
    .
    .
    while(1)
    {
        .
        OS_Lock();
        .
        .        // not under-breakable part
        .
        OS_Unlock();
        .
    }
}
```

OS_GetRev

```
U08 OS_FAR *OS_GetRev(void)
```

Returns a pointer on the kernelrevision (NULL-terminated ASCII-array).

Parameters

none

Return Value

*pointer	pointer to the address of array

Example

```
void OS_FAR Task1(void *data)
{
    U08 OS_FAR  *Revision;

    .
    .
    while(1)
    {
        .
        Revision = OS_GetRev();
        .
    }
}
```

Dynamic-Memory

OS_MemoryInit

```
U08 OS_MemoryInit(OS_MEM OS_HUGE *mp, U32 size)
```

Initialize the dynamic memory management. This function must be called for the dynamic memory management at the system initialization once. The functions of the vigorous memory management can be used also without current kernel. ATTENTION! It is executed no checkup of the storage area.

Parameters

*mp	startaddress of memory-pool
size	size of memory-pool in bytes

Return Value

OS_NO_ERR	Memory pool positioned
OS_MEM_ERR	one of the parameters is ZERO

Example for LARGE memory in NEAR-model

```
void main(void)
{
    U08  state;

    .
    .                                               // Memory: 512k - 64k(near)
    state = OS_MemoryInit((OS_MEM OS_HUGE *)(0x10000000), 458750);
    .
    .
}
```

Example for model-known memory

```
U08  memorypool[MEMSIZE];

void main(void)
{
    U08  state;

    .
    state = OS_MemoryInit((OS_MEM OS_HUGE *)(memorypool), MEMSIZE);
    .
    .
}
```

OS_MemAlloc

```
U08 OS_MemAlloc(U08 OS_HUGE **MemPtr, U32 size)
```

Allocation of a required memory area.

Parameters

*MemPtr	pointer of pointer to get address of memory-area
size	size of needed memory in bytes

Return Value

OS_NO_ERR	memory successfully allocated
OS_MEM_ERR	size is ZERO or bigger as storage area of the processor
OS_MEM_OVF	not sufficiently free memory

Example

```
void OS_FAR Task1(void *data)
{
    U08 OS_HUGE  *Addr_p;
    U08           state;

    .
    .
    while(1)
    {
        .
        state = OS_MemAlloc(&Addr_p, 3800);
        .
    }
}
```

OS_MemFree

```
U08 OS_MemFree(U08 OS_HUGE **MemPtr)
```

Release of an allocated memory area and clearing of the pointer.

Parameters

**MemPtr	pointer of pointer to address of allocated memory

Return Value

OS_NO_ERR	memory successfully released
OS_MEM_ERR	Pointer is ZERO or not a valid allocation found

Example

```
void OS_FAR Task1(void *data)
{
    U08 OS_HUGE  *Addr_p;
    U08           state;


    .
    .
    while(1)
    {
        .
        state = OS_MemAlloc(&Addr_p, 3800);
        .
        .
        state = OS_MemFree(&Addr_p);
        .
    }
}
```

OS_MemFreeSize

```
U32 OS_MemFreeSize(void)
```

It returns the amount of free memory.

Parameters

none

Return Value

size	free size in memory-pool in bytes

Example

```
void OS_FAR Task1(void *data)
{
    U08 OS_HUGE  *Addr_p;
    U32           fsize;
    U08           state;


    .
    .
    while(1)
    {
        .
        state = OS_MemAlloc(&Addr_p, 3800);
        .
        .
        fsize = OS_MemFreeSize();
        .
    }
}
```

OS_MemVerifyPtr

```
U08 OS_MemVerifyPtr(void OS_HUGE *MemPtr)
```

Check if a pointer is pointing into the memory pool. However, it is not checked whether it is a pointer to an allocated block, only the address range of the memory pool is used.

Parameters

*MemPtr	a pointer

Return Value

OS_NO_ERR	Pointer points into the memory-pool
"1"	Pointer does'nt point into the memory-pool

Example

```
void OS_FAR Task1(void *data)
{
    U08 OS_HUGE  *Addr_p;
    U08           state;


    .
    .
    while(1)
    {
        .
        state = OS_MemAlloc(&Addr_p, 3800);
        .
        .
        state = OS_MemVerifyPtr(Addr_p);
        if (state == OS_NO_ERR)
        {
            .
            state = OS_MemFree(Addr_p);
        }
        .
    }
}
```

SmartMessageQueue

OS_SmqInit

```
U08 OS_SmqInit(OS_SMQ *psmq, void OS_HUGE *buffer, U16 deep)
```

Initialize a SmartMessageQueue. A SmartMessageQueue is used to transfer data by means of a pointer to another process according to the FIFO principle. If the pointer points to an allocated memory block, the ownership of this block is also transferred. If the pointer does not point to an allocated memory block, the risk remains with the user because a non-specific pointer is passed directly. Within this SmartMessageQueue, <deep> pointers can be passed to other processes through the kernel buffer <* buffer>. The buffer can be generated by direct declaration (OS_STK_TYPE SMQ_d [deep]) or by dynamic allocation. For dynamic allocation in segment-based storage management systems, the type declaration OS_HUGE is required to address an area across segment boundaries.

Parameters

*psmq	pointer to SmartMessageQueue
*buffer	pointer to kernel-buffer
deep	size of kernel-buffer in "void *"

Return Value

OS_NO_ERR	SmartMessageQueue initialized

Example

```
OS_SMQ      MQueue;
OS_STK_TYPE MQueue_d[16];

void main(void)
{
    U08  state;

    .
    .
    OS_Init();
    .
    state = OS_SmqInit(&MQueue, MQueue_d, 16);
    .
}
```

OS_SmqInfo

```
U08 OS_SmqInfo(OS_SMQ *psmq, U16 *deep, U16 *used, U08 *prio)
```

Query the status of a SmartMessageQueue. This query can be used to determine various parameters of their initialization and their fill level. Furthermore, the priority of the waiting task can be determined.

Parameters

*psmq	pointer to SmartMessageQueue
*deep	pointer to variable will get the max pointer in pipe
*used	pointer to variable will get the used-pointer at this time
*prio	pointer to variable will get the priority of waiting Task (if zero - no Task is waiting)

Return Value

OS_NO_ERR	no error (for extensions)
OS_SMQ_ERR	no *psmq pointer given

Example

```
OS_SMQ      MQueue;

void OS_FAR Task1(void *data)
{
    U08  state;
    U16  deep;
    U16  used;
    U08  prio;

    .
    .
    while(1)
    {
        .
        state = OS_SmqInfo(&MQueue, &deep, &used, &prio);
        .
    }
}
```

OS_SmqClear

```
U08 OS_SmqClear(OS_SMQ *psmq)
```

Clears the contents of a SmartMessageQueue and reactivates a pending send process. This feature can be used for error handling to restart the data transfer. Included pointers to an allocated memory block are released, otherwise the pointers are deleted and lost.

Parameters

*psmq	pointer to SmartMessageQueue

Return Value

OS_NO_ERR	SmartMessageQueue content deleted

Example

```
OS_SMQ      MQueue;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        .
        state = OS_SmqClear(&MQueue);
        .
    }
}
```

OS_SmqPost

```
U08 OS_SmqPost(OS_SMQ * psmq, void OS_HUGE **msg, U16 timeout)
```

Sends a pointer to a SmartMessageQueue. If the pointer points to an allocated memory block, the kernel takes ownership of this block and clears the sender's pointer. With OS_NO_SUSP is returned immediately even if the SmartMessageQueue was full and with OS_SUSPEND is waited until the pointer can be entered (if necessary, endless).

Parameters

*psmq	pointer to SmartMessageQueue
**msg	pointer to pointer
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	pointer sent into SmartMessageQueue
OS_SMQ_FULL	SmartMessageQueue full (on OS_NO_SUSP)
OS_TIMEOUT	SmartMessageQueue full (after waiting)

Example

```
typedef struct user_s {
    U08 state;
    U16 len;
    U16 curr_offs;
    U08 data[];
} User_st;

OS_SMQ      MQueue;

void OS_FAR Task1(void *data)
{
    User_st OS_HUGE  *user_p;
    U08               state;

    .
    .
    while(1)
    {
        .
        state = OS_MemAlloc(&user_p, sizeof(User_st) + 380);        // struct + user-data
        .
        .
        state = OS_SmqPost(&MQueue, &user_p, 500);
        if (state == OS_NO_ERR)
        {
            .
            if (user_p == NULL)
                // ownership of memory-pool pointer was taken by kernel
            .
        }
    }
}
```

OS_SmqFrontPost

```
U08 OS_SmqFrontPost(OS_SMQ * psmq, void OS_HUGE **msg, U16 timeout)
```

Sends a pointer to the beginning of a SmartMessageQueue. Thus, this pointer is first read out by the receiver (push forward). If the pointer points to an allocated memory block, the kernel takes ownership of this block and clears the sender's pointer. With OS_NO_SUSP is returned immediately even if the SmartMessageQueue was full and with OS_SUSPEND is waited until the pointer can be entered (if necessary, endless).

Parameters

*psmq	pointer to SmartMessageQueue
**msg	pointer to pointer
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	pointer sent into SmartMessageQueue
OS_SMQ_FULL	SmartMessageQueue full (on OS_NO_SUSP)
OS_TIMEOUT	SmartMessageQueue full (after waiting)

Example

```
typedef struct user_s {
    U08 state;
    U16 len;
    U16 curr_offs;
    U08 data[];
} User_st;

OS_SMQ      MQueue;

void OS_FAR Task1(void *data)
{
    User_st OS_HUGE  *user_p;
    U08               state;

    .
    .
    while(1)
    {
        .
        state = OS_MemAlloc(&user_p, sizeof(User_st) + 380);        // struct + user-data bytes
        .
        .
        state = OS_SmqFrontPost(&MQueue, &user_p, 500);
        if (state == OS_NO_ERR)
        {
            .
            if (user_p == NULL)
                // ownership of memory-pool pointer was taken by kernel
            .
        }
    }
}
```

OS_SmqPostAbbort

```
U08 OS_SmqPostAbbort(OS_SMQ * psmq)
```

Cancels the waiting of a sending task at the SmartMessageQueue.
Only the waiting task with the highest priority is quasi "prematurely" sent to TimeOut.

Parameters

*psmq	pointer to SmartMessageQueue

Return Value

OS_NO_ERR	Waiting for a task aborted
OS_TASK_NOT_EXIST	no task is waiting at the SmartMessageQueue

Example

```
OS_SMQ      MQueue;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_SmqPostAbbort(&MQueue);
        .
    }
}
```

OS_SmqPend

```
U08 OS_SmqPend(OS_SMQ * psmq, void OS_HUGE **msg, U16 timeout)
```

Wait for a pointer from a SmartMessageQueue. If the pointer points to an allocated memory block, the ownership of this block is transferred to the receiver. With OS_NO_SUSP is returned immediately even if no pointer was present and with OS_SUSPEND waits until a pointer is present (if necessary, endless).

Parameters

*psmq	pointer to SmartMessageQueue
**msg	pointer to pointer
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	pointer get from SmartMessageQueue
OS_SMQ_NODATA	no pointer in SmartMessageQueue (on OS_NO_SUSP)
OS_TIMEOUT	no pointer in SmartMessageQueue (after waiting)

Example

```
typedef struct user_s {
    U08 state;
    U16 len;
    U16 curr_offs;
    U08 data[];
} User_st;

OS_SMQ      MQueue;

void OS_FAR Task1(void *data)
{
    User_st OS_HUGE  *user_p;
    U08               state;

    .
    .
    while(1)
    {
        .
        state = OS_SmqPend(&MQueue, &user_p, OS_SUSPEND);
        .
        .
        if ((user_p != NULL) && (!OS_MemVerifyPtr(user_p)))
            state = OS_MemFree(user_p);
        .
    }
}
```

OS_SmqPendAbbort

```
U08 OS_SmqPendAbbort(OS_SMQ * psmq)
```

Cancels the waiting for a receiving task at the SmartMessageQueue.
Only the waiting task with the highest priority is quasi "prematurely" sent to TimeOut.

Parameters

*psmq	pointer to SmartMessageQueue

Return Value

OS_NO_ERR	Waiting for a task aborted
OS_TASK_NOT_EXIST	no task is waiting at the SmartMessageQueue

Example

```
OS_SMQ      MQueue;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_SmqPendAbbort(&MQueue);
        .
    }
}
```

Mailboxes

OS_MboxInit

```
U08 OS_MboxInit(OS_MBOX *pmbox)
```

Initialization of a Mailbox. Through a mailbox any kind of data can pass by a pointer. On this, the recipient receives a pointer in the data field of the sender!

Parameters

*pmbox	pointer to Mailbox

Return Value

OS_NO_ERR	Mailbox initialized

Example

```
OS_MBOX  MailBox1;

void main(void)
{
    U08  state;

    .
    .
    OS_Init();
    .
    state = OS_MboxInit(&MailBox1);
    .
}
```

OS_MboxPend

```
U08 OS_MboxPend(OS_MBOX *pmbox, void OS_FAR *msg, U16 timeout)
```

Waits for a message from a mailbox. With OS_NO_SUSP, it immediately is come back even if no news was available and becomes with OS_SUSPEND as long as waited until a message is available, if necessary unending.

Parameters

*pmbox	pointer to Mailbox
*msg	pointer to receiving parameter (U32)
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Message from Mailbox gotten
OS_MBOX_NODATA	no message in Mailbox (with OS_NO_SUSP)
OS_TIMEOUT	no message in Mailbox (after waits)

Example

```
OS_MBOX  MailBox1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U32  Message;

    .
    .
    while(1)
    {
        .
        state = OS_MboxPend(&MailBox1, &Message, 200);
        .
    }
}
```

OS_MboxPendAbbort

```
U08 OS_MboxPendAbbort(OS_MBOX *pmbox)
```

Abborts waiting of a receiving Tasks (highest waiting prio) on a Mailbox. It is only the waiting task with the highest priority quasi "premature" to TimeOut forwarded.

Parameters

*pmbox	pointer to Mailbox

Return Value

OS_NO_ERR	pending of a task abborted
OS_TASK_NOT_EXIST	no pending task on this Maibox

Example

```
OS_MBOX  MailBox1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_MboxPendAbbort(&MailBox1);
        .
    }
}
```

OS_MboxPost

```
U08 OS_MboxPost(OS_MBOX *pmbox, void OS_FAR *msg, U16 timeout)
```

Sends a message into a Mailbox. With OS_NO_SUSP, it immediately is come back even if the Mailbox was full and becomes with OS_SUSPEND as long as waited until the message can be written down, if necessary unending.

Parameters

*pmbox	pointer to Mailbox
*msg	pointer to message (U32)
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Message in Mailbox sent
OS_MBOX_FULL	Mailbox fully (with OS_NO_SUSP)
OS_TIMEOUT	Mailbox fully (after waits)

Example

```
OS_MBOX  MailBox1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U32  Message;

    .
    .
    while(1)
    {
        .
        Message =0x3076;
        state = OS_MboxPost(&MailBox1, &Message, OS_SUSPEND);
        .
    }
}
```

OS_MboxPostAbbort

```
U08 OS_MboxPostAbbort(OS_MBOX *pmbox)
```

Abborts waiting of a sending Tasks (highest waiting prio) on a Mailbox. It is only the waiting task with the highest priority quasi "premature" to TimeOut forwarded.

Parameters

*pmbox	pointer to Mailbox

Return Value

OS_NO_ERR	posting of a task abborted
OS_TASK_NOT_EXIST	no posting task on this Maibox

Example

```
OS_MBOX  MailBox1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_MboxPostAbbort(&MailBox1);
        .
    }
}
```

Queues

OS_QueueInit

```
U08 OS_QueueInit(OS_Q *pq, void OS_HUGE *buffer, U16 size)
```

Initialize a Queue. A Queue serves the byte-expels transfer of data at another process of the FIFO-prinzip. Within this Queue can <size> byte through the kernel buffer <*buffer> at other processes is handed over. The buffer can be generated through direct declaration (UBYTE buffer[size]) or through dynamic allocation. For the dynamic allocation in systems with segment-based memory management, the type declaration is OS_HUGE necessary in order to be able to go down well away with an area over segment borders.

Parameters

*pq	pointer to Queue
*buffer	pointer to kernel-buffer
size	size of kernel-buffer in bytes

Return Value

OS_NO_ERR	Queue initialized

Example

```
OS_Q  Queue1;

U08   Q_Data1[256];

void main(void)
{
    U08  state;

    .
    .
    OS_Init();
    .
    state = OS_QueueInit(&Queue1, &Q_Data1[0], 256);
    .
}
```

OS_QueueInfo

```
U08 OS_QueueInfo(OS_Q *pq, U16 *size, U16 *used, U08 *prio)
```

Retrieval of the status of a Queue. Through this retrieval, miscellaneous parameters their initialization as well as their filling stand can be determined. Furthermore, the priority of the waiting tasks can be determined.

Parameters

*pq	pointer to Queue
*size	pointer to variable will get the size
*used	pointer to variable will get the used-bytes at this time
*prio	pointer to variable will get the priority of waiting Task (if zero - no Task is waiting)

Return Value

OS_NO_ERR	no mistake (for expansions)

Example

```
OS_Q  Queue1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U16  size;
    U16  used;
    U08  prio;

    .
    .
    while(1)
    {
        .
        state = OS_QueueInfo(&Queue1, &size, &used, &prio);
        .
    }
}
```

OS_QueuePend

```
U08 OS_QueuePend(OS_Q *pq, U08 OS_FAR *msg, U16 timeout)
```

Wait on one byte from a Queue. With OS_NO_SUSP, it immediately is come back even if no bytes were available and become with OS_SUSPEND as long as waited until one byte is available, if necessary unending.

Parameters

*pq	pointer to Queue
*msg	pointer to receiving Byte
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Byte from Queue gotten
OS_Q_NODATA	no byte in Queue (with OS_NO_SUSP)
OS_TIMEOUT	no byte in Queue (after waits)

Example

```
OS_Q  Queue1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U08  Receive;

    .
    .
    while(1)
    {
        .
        state = OS_QueuePend(&Queue1, &Receive, 100);
        .
    }
}
```

OS_QueuePendAbbort

```
U08 OS_QueuePendAbbort(OS_Q *pq)
```

Abborts waiting of a receiving Tasks (highest waiting prio) on a Queue. It is only the waiting task with the highest priority quasi "premature" to TimeOut forwarded.

Parameters

*pq	pointer to Queue

Return Value

OS_NO_ERR	pending of a task abborted
OS_TASK_NOT_EXIST	no pending task on this Queue

Example

```
OS_Q  Queue1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_QueuePendAbbort(&Queue1);
        .
    }
}
```

OS_QueuePost

```
U08 OS_QueuePost(OS_Q *pq, U08 msg, U16 timeout)
```

Sends one byte into a Queue. With OS_NO_SUSP, it immediately is come back even if the Queue was full and becomes with OS_SUSPEND as long as waited until the byte can be written down, if necessary unending.

Parameters

*pq	pointer to Queue
msg	byte to send
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Byte in Queue sent
OS_Q_FULL	Queue fully (with OS_NO_SUSP)
OS_TIMEOUT	Queue fully (after waits)

Example

```
OS_Q  Queue1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U08  Message;

    .
    .
    while(1)
    {
        .
        Message = 0x6A;
        state = OS_QueuePost(&Queue1, Message, 5000);
        .
    }
}
```

OS_QueueFrontPost

```
U08 OS_QueueFrontPost(OS_Q *pq, U08 msg, U16 timeout)
```

Sends one byte at the beginning of a Queue. Consequently, this byte first is finished reading again by the recipient (cuts in line). With OS_NO_SUSP, it immediately is come back even if the Queue was full and becomes with OS_SUSPEND as long as waited until the byte can be written down, if necessary unending.

Parameters

*pq	pointer to Queue
msg	byte to send
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Byte in Queue sent
OS_Q_FULL	Queue fully (with OS_NO_SUSP)
OS_TIMEOUT	Queue fully (after waits)

Example

```
OS_Q  Queue1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U08  Message;

    .
    .
    while(1)
    {
        .
        Message = 0x2D;
        state = OS_QueueFrontPost(&Queue1, Message, OS_NO_SUSP);
        .
    }
}
```

OS_QueuePostAbbort

```
U08 OS_QueuePostAbbort(OS_Q *pq)
```

Abborts waiting of a sending Tasks (highest waiting prio) on a Queue.
It is only the waiting task with the highest priority quasi "premature" to TimeOut forwarded.

Parameters

*pq	pointer to Queue

Return Value

OS_NO_ERR	posting of a task abborted
OS_TASK_NOT_EXIST	no posting task on this Queue

Example

```
OS_Q  Queue1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_QueuePostAbbort(&Queue1);
        .
    }
}
```

OS_QueueClear

```
U08 OS_QueueClear(OS_Q *pq)
```

Deletes the content of a Queue and reactivates a waiting posting-process. This function can be used to reactivate the datatransfer after a hang-on. The deleted data get lost on that occasion.

Parameters

*pq	pointer to Queue

Return Value

OS_NO_ERR	content of Queue deleted

Example

```
OS_Q  Queue1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        .
        state = OS_QueueClear(&Queue1);
        .
    }
}
```

Pipes

OS_PipeInit

```
U08 OS_PipeInit(OS_P *pp, void OS_HUGE *buffer, U16 size, U08 deep)
```

Initialize a pipe. A pipe is used to transfer data in packets to another process in FIFO-style.
The kernel offers two different modi to choose at the time of creation.

- **array-modi**:
  The pipe works with a permanently assigned array of fixed size. Within this pipe maximum <deep> packages with a maximum of <size> bytes of a packet through the kernel buffer <*buffer> be transferred to another process. A packet can be between 1 byte and <size> bytes large but always occupies one row of the array of <size> bytes.
  The buffer can be generated by direct declaration (`U08 Buffer[(size+2)*deep]`) or by dynamic allocation. For dynamic allocation in segment-based storage management systems, the type declaration OS_HUGE is required to address an area across segment boundaries. The additional length of 2 bytes per packet is required for the storage of packet length.

- **dynamic-modi**:
  The pipe uses the kernel memory manager and automatically allocates the required memory for a packet (plus 6 bytes of management) and release it again after reading of the package. Thus, only the memory required for a packet is always allocated (plus 6 bytes of pipe concatenation + n bytes of memory manager). As a result, this pipe is never full, only the free memory of the memory manager can be used up. Depending on the location and fragmentation of the memory, it can not be determined where the package is stored exactly. Furthermore, the accumulated packages within the pipe are directly concatenated and are therefore prone to erroneous memory accesses by user-tasks.
  In addition, the processing of 'allocate + copy' and 'copy + free' are interruptible mostly by interrupts and partly by sheduling. The integrity is secure at all times, but occasionally these processing may take a long time to complete. Furthermore, a sending task during '`OS_PipePost(..)`' should NEVER be destroyed by an ISR or other task using '`OS_TaskDestroy(..)`'.

Parameters

*pp	pointer to Pipe
*buffer	pointer to kernel-buffer (array mode)
size	max size of data-bytes per paket (array mode)
deep	max pakets in pipe (array mode)

Return Value

OS_NO_ERR	Pipe initialized

Example

```
OS_P  PipeA;
OS_P  PipeD;

U08   P_DataA[(MAXPAKET+2)*8];

void main(void)
{
    U08  state;

    .
    .
    OS_Init();
    .
    state = OS_PipeInit(&PipeA, P_DataA, MAXPAKET+2, 8);    // use array-modi of 'size*deep'
    .
    state = OS_PipeInit(&PipeD, NULL, 0, 0);                // use dynamic-modi (Memory-Manager)
    .
}
```

OS_PipeInfo

```
U08 OS_PipeInfo(OS_P *pp, U16 *size, U08 *deep, U08 *used, U08 *prio)
```

Retrieval of the status of a Pipe. Through this retrieval, miscellaneous parameters of their initialization as well as their filling stand can be determined. Furthermore, the priority of the waiting Tasks can be determined.

Parameters

*pp	pointer to Pipe
*size	pointer to variable will get the size per paket
*deep	pointer to variable will get the max pakets in pipe
*used	pointer to variable will get the used-pakets at this time
*prio	pointer to variable will get the priority of waiting Task (if zero - no Task is waiting)

Return Value

OS_NO_ERR	no mistake (for expansions)

Example

```
OS_P  Pipe1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U16  size;
    U08  deep;
    U08  used;
    U08  prio;

    .
    .
    while(1)
    {
        .
        state = OS_PipeInfo(&Pipe1, &size, &deep, &used, &prio);
        .
    }
}
```

OS_PipePend

```
U08 OS_PipePend(OS_P *pp, U08 OS_HUGE *msg, U16 *lng, U16 timeout)
```

Waits on a data package from a Pipe. The receiver-buffer must be included sufficiently big in order to be able to pick up the package. Since the receiver-buffer can also be dynamically allocated, the type declaration is OS_HUGE necessary on the other hand in order to be able to go down well away with an area over segment borders. With OS_NO_SUSP, it immediately is come back even if no package was available and becomes with OS_SUSPEND as long as waited until one package is available, if necessary unending.

Parameters

*pp	pointer to Pipe
*msg	pointer to receiving array
*lng	pointer to variable will get the lenght of paket
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Package from Pipe gotten
OS_P_NODATA	no package in Pipe (with OS_NO_SUSP)
OS_TIMEOUT	no package in Pipe (after waits)

Example

```
OS_P  Pipe1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U16  rLenght;
    U08  Receive[1024];

    .
    .
    while(1)
    {
        .
        state = OS_PipePend(&Pipe1, Receive, &rLenght, OS_SUSPEND);
        .
    }
}
```

OS_PipePendAbbort

```
U08 OS_PipePendAbbort(OS_P *pp)
```

Abborts waiting of a receiving Tasks (highest waiting prio) on a Pipe. It is only the waiting task with the highest priority quasi "premature" to TimeOut forwarded.

Parameters

*pp	pointer to Pipe

Return Value

OS_NO_ERR	pending of a task abborted
OS_TASK_NOT_EXIST	no pending task on this Pipe

Example

```
OS_P  Pipe1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_PipePendAbbort(&Pipe1);
        .
    }
}
```

OS_PipePost

```
U08 OS_PipePost(OS_P *pp, U08 OS_HUGE *msg, U16 lenght, U16 timeout)
```

Sends one package into a Pipe. So the transmitter-buffer also dynamically allocated can be, the type declaration is OS_HUGE necessary on the other hand in order to be able to go down well away with an area over segment borders. With OS_NO_SUSP, it immediately is come back even if the Pipe was full and becomes with OS_SUSPEND as long as waited until the package can be written down, if necessary unending.

Parameters

*pp	pointer to Pipe
*msg	pointer to data-paket
lenght	lenght of data-paket
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Package in Pipe sent
OS_P_FULL	Pipe fully (with OS_NO_SUSP) - on array-modi
OS_P_LEN_ERR	Package too long - on array-modi
OS_TIMEOUT	Pipe fully (after waits) - on array-modi
see `OS_MemAlloc()`	memory error - on dynamic-modi

Example

```
OS_P  Pipe1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U08  Message[]={"Hello World!"};

    .
    .
    while(1)
    {
        .
        .
        state = OS_PipePost(&Pipe1, Message, strlen(Message), 500);
        .
    }
}
```

OS_PipeFrontPost

```
U08 OS_PipeFrontPost(OS_P *pp, U08 OS_HUGE *msg, U16 lenght, U16 timeout)
```

Sends one package at the beginning of a Pipe. Consequently, this package first is finished reading again by the recipient (cuts in line). Since the transmitter-buffer can also be dynamically allocated, the type declaration is OS_HUGE necessary on the other hand in order to be able to go down well away with an area over segment borders. With OS_NO_SUSP, it immediately is come back even if the Pipe was full and becomes with OS_SUSPEND as long as waited until the package can be written down, if necessary unending.

Parameters

*pp	pointer to Pipe
*msg	pointer to data-paket
lenght	lenght of data-paket
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Package in Pipe sent
OS_P_FULL	Pipe fully (with OS_NO_SUSP) - on array-modi
OS_P_LEN_ERR	Package too long - on array-modi
OS_TIMEOUT	Pipe fully (after waits) - on array-modi
see `OS_MemAlloc()`	memory error - on dynamic-modi

Example

```
OS_P  Pipe1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U08  Message[]={"Hallo Welt !"};

    .
    .
    while(1)
    {
        .
        .
        state = OS_PipeFrontPost(&Pipe1, Message, strlen(Message), OS_NO_SUSP);
        .
    }
}
```

OS_PipePostAbbort

```
U08 OS_PipePostAbbort(OS_P *pp)
```

Abborts waiting of a sending Tasks (highest waiting prio) on a Pipe. It is only the waiting task with the highest priority quasi "premature" to TimeOut forwarded.

Parameters

*pp	pointer to Pipe

Return Value

OS_NO_ERR	posting of a task abborted
OS_TASK_NOT_EXIST	no posting task on this Pipe

Example

```
OS_P  Pipe1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_PipePostAbbort(&Pipe1);
        .
    }
}
```

OS_PipeClear

```
U08 OS_PipeClear(OS_P *pp)
```

Deletes the content of a Pipe and reactivates a waiting transmitter-process. This function can be used to reactivate the datatransfer after a hang-on. The deleted packages get lost on that occasion.

Parameters

*pp	pointer to Pipe

Return Value

OS_NO_ERR	Pipe-content deleted

Example

```
OS_P  Pipe1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        .
        state = OS_PipeClear(&Pipe1);
        .
    }
}
```

inter-core AMP-Pipe

OS_IccInit

```
U08 OS_IccInit(void)
```

Initialize the AMP-pipe pair. The AMP-pipe pair is used for the packet-wise transfer of data to another processor core according to the FIFO principle.

The maximum size of a packet, the maximum number of packets and the memory location is specified in the hardware-dependent adaptation 'OS_ICC_xxx.c' in order to provide both processor cores with exactly identical values.
A packet can be between 1 byte and <size> bytes in size, but always occupies one line of the array of <size> bytes.
The additional length of 2 bytes per packet is required to store the real packet length.

Parameters

none

Return Value

OS_NO_ERR	AMP-pipe pair initialized

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    state = OS_IccInit();
    .
}
```

OS_IccInfo

```
U08 OS_IccInfo(U16 *size, U08 *deep, U08 *usedRx, U08 *usedTx)
```

Query the status of the AMP-pipe pair. This query can be used to determine various initialization parameters and their fill levels.

Parameters

*size	pointer to variable will get the size per paket
*deep	pointer to variable will get the max pakets in pipe
*usedRx	pointer to variable will get the receiving AMP-Pipe used-pakets at this time
*usedTx	pointer to variable will get the tranmitting AMP-Pipe used-pakets at this time

Return Value

OS_NO_ERR	no error (for extensions)

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;
    U16  size;
    U08  deep;
    U08  usedRx, usedTx;

    .
    .
    while(1)
    {
        .
        state = OS_IccInfo(&size, &deep, &usedRx, &usedTx);
        .
    }
}
```

OS_IccPost

```
U08 OS_IccPost(U08 OS_HUGE *msg, U16 lenght, U16 timeout)
```

Sends a packet in the AMP-pipe to the other core. Since the sender buffer can also be dynamically allocated, the type declaration OS_HUGE is required in order to be able to address an area across segment boundaries. OS_NO_SUSP returns immediately, even if the AMP-pipe was full, and OS_SUSPEND waits until the package can be entered (if necessary, endlessly).

Parameters

*msg	pointer to data-paket
lenght	lenght of data-paket
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Packet sent in AMP-pipe
OS_ICC_FULL	AMP-pipe full (at OS_NO_SUSP)
OS_ICC_LEN_ERR	Packet too long
OS_TIMEOUT	AMP-pipe full (after waiting)

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;
    U08  Message[]={"Hi other core!"};

    .
    .
    while(1)
    {
        .
        .
        state = OS_IccPost(Message, strlen(Message), 500);
        .
    }
}
```

OS_IccPend

```
U08 OS_IccPend(U08 OS_HUGE *msg, U16 *lng, U16 timeout)
```

Wait for a packet from the AMP-pipe from the other core. The recipient buffer must be large enough to accommodate the package. Since the recipient buffer can also be dynamically allocated, the type declaration OS_HUGE is required in order to be able to address an area across segment boundaries.   OS_NO_SUSP returns immediately, even if there was no package, and OS_SUSPEND waits until a package is available (if necessary, endlessly).

Parameters

*msg	pointer to receiving array
*lng	pointer to variable will get the lenght of paket
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Receive package from AMP-pipe
OS_ICC_NODATA	no package in AMP-pipe (with OS_NO_SUSP)
OS_TIMEOUT	no package in AMP-pipe (after waiting)

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;
    U16  rLenght;
    U08  Receive[512];

    .
    .
    while(1)
    {
        .
        state = OS_IccPend(Receive, &rLenght, OS_SUSPEND);
        .
    }
}
```

OS_IccClear

```
U08 OS_IccClear(void)
```

Deletes the content of the respective sending AMP pipe and reactivates a waiting sending process. This function can be used for error handling to restart the data transfer. The deleted packages are lost.

Parameters

none

Return Value

OS_NO_ERR	AMP pipe content deleted

Example

```
void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        .
        state = OS_IccClear();
        .
    }
}
```

Semaphores

OS_SemInit

```
U08 OS_SemInit(OS_SEM *psem, U16 cnt)
```

Initialize a Semaphore. One semaphores serves the process-syncronisation. Two variations of the utilization belong semaphores to it to one.
- binary: for example the syncronisation of accesses on common recources/variables
- counting: Attendants on entering of a signal, to the control of the sequence of processes.

With a binary semaphore, a state-machine can become protected before simultaneous accesses of different processes (Read/Write), for example. So, inconsistent conditions or data are avoided. Can appear however in some cases *Priority Inversion*, occupied i.e. this a low Task the recource, a higher Task therefore must wait and then for example through an INT a middle Task (and its heirs) whom lower Task interrupts for an uncertain time. In such a case, it is initialized the semaphores with 1 as cnt, the access asked by means of OS_SemPend and released again by means of OS_SemPost.

With a counting semaphore, arriving is signalled by events. So, a process can wait for a signal to pause about the sequence of processes. By means of OS_SemAccept, also the number of the meanwhile entered events can be determined on that occasion. In such a case, it is initialized the semaphores with 0 as cnt, waited on the event by means of OS_SemPend or OS_SemAccept and reported the event by means of OS_SemPost.

Parameters

*psem	pointer to Semaphore

Return Value

OS_NO_ERR	Semaphore initialized

Example

```
OS_SEM  State1;

void main(void)
{
    U08  state;

    .
    .
    OS_Init();
    .
    state = OS_SemInit(&State1, 1);
    .
}
```

OS_SemPend

```
U08 OS_SemPend(OS_SEM *psem, U16 timeout)
```

Reserve one on the protected recource semaphore and consequently the access as well as wait for an event. With OS_NO_SUSP, it immediately is come back even if was not freely the semaphores as well as no event was available and becomes with OS_SUSPEND as long as waited until the semaphores the event could be reserved as well as could happen, if necessary unending.

Parameters

*psem	pointer to Semaphore
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Semaphore reserve / event was available
OS_SEM_NODATA	Semaphore occupy / no event (with OS_NO_SUSP)
OS_TIMEOUT	Semaphore occupy / no event (after waits)

Example

```
OS_SEM  State1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_SemPend(&State1, OS_SUSPEND);
        .
        .
    }
}
```

OS_SemPendAbbort

```
U08 OS_SemPendAbbort(OS_SEM *psem)
```

Abborts waiting of a Tasks (highest waiting prio) on a Semaphore. It is only the waiting task with the highest priority quasi "premature" to TimeOut forwarded.

Parameters

*psem	pointer to Semaphore

Return Value

OS_NO_ERR	pending of a task abborted
OS_TASK_NOT_EXIST	no pending task on this Semaphore

Example

```
OS_SEM  State1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_SemPendAbbort(&State1);
        .
    }
}
```

OS_SemAccept

```
U08 OS_SemAccept(OS_SEM *psem, U16 *cnt, U16 timeout)
```

It is used as Event-Counter with utilization of the semaphores. The Semaphore-counter is not influenced on that occasion. If the counter bigger than 0 the current counter will return. With OS_NO_SUSP, it immediately is come back even if the semaphoren-counter 0 is and becomes with OS_SUSPEND as long as waited until the event once appeared, if necessary unending.

Parameters

*psem	pointer to Semaphore
*cnt	pointer to variable will get the counter-value
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Event min. 1 times happened
OS_SEM_NODATA	Counter immediately 0 (with OS_NO_SUSP)
OS_TIMEOUT	Counter immediately 0 (after waits)

Example

```
OS_SEM  Event1;

void OS_FAR Task1(void *data)
{
    U08  state;
    U16  EventCnt;

    .
    .
    while(1)
    {
        .
        state = OS_SemAccept(&Event1, &EventCnt, 500);
        .
    }
}
```

OS_SemPost

```
U08 OS_SemPost(OS_SEM *psem)
```

Gives a retiring semaphore and consequently the access to the protected recource again freely as well as signals an event.

Parameters

*psem	pointer to Semaphore

Return Value

OS_NO_ERR	Semaphores released / event reported
OS_SEM_OVF	Error in the Semaphore-handling (Counter too big)

Example

```
OS_SEM  State1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        .
        state = OS_SemPost(&State1);
        .
    }
}
```

OS_SemClear

```
U08 OS_SemClear(OS_SEM *psem)
```

Deletes the Counter of the semaphore. This function can be used to reset a counting-semaphore or for error-handling to restart the semaphore-handling. With application of binary semaphore to the control of accesses on a protected recource must be released the semaphore in the connection with application to the mistake-handling by OS_SemPost so much times, how simultaneously processes can access the recource.

Parameters

*psem	pointer to Semaphore

Return Value

OS_NO_ERR	count of semaphore reset to 0

Example

```
OS_SEM  State1;

void OS_FAR Task1(void *data)
{
    U08 state;

    .
    .
    while(1)
    {
        .
        state = OS_SemClear(&State1);
        .
    }
}
```

Mutexes

OS_MutexCreate

```
U08 OS_MutexCreate(OS_MUX *pmux, U08 prio)
```

Aims as well as initializing of a Mutex (Mutual-Exclusion). A Mutex serves the syncronisation of accesses to common recources/variables. With a Mutex, state-machines are protected from simultaneous accesses of different processes (Read/Write), for example. The second process must wait, until the first process finished his access (Read/Write). So, inconsistent conditions or data are avoided. In contrast to the utilization of semaphores, the effect of the *Priority Inversion* cannot kick open on that occasion here. The priority of the Mutex must be included higher, as the highest priority of the Tasks accessing it. The Mutex is written down as not current Task, so under the same priority no other Task can run.

Parameters

*pmux	pointer to Mutex

Return Value

OS_NO_ERR	Mutex written down and initialized

Example

```
OS_MUX  Mutex1;

void main(void)
{
    U08  state;

    .
    .
    OS_Init();
    .
    state = OS_MutexCreate(&Mutex1, 10);
    .
}
```

OS_MutexPend

```
U08 OS_MutexPend(OS_MUX *pmux, U16 timeout)
```

Reserve a Mutex and consequently the access on the protected recource. With OS_NO_SUSP, it immediately is come back even if the Mutex was not free and becomes with OS_SUSPEND as long as waited until the Mutex could be reserved, if necessary unending.

Parameters

*pmux	pointer to Mutex
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Mutex reserved
OS_MUX_NOACC	Mutex is occupied (with OS_NO_SUSP)
OS_TIMEOUT	Mutex is occupied (after waits)
OS_MUX_ERR	Error in Mutex-handling

Example

```
OS_MUX  Mutex1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_MutexPend(&Mutex1, OS_SUSPEND);
        .
        .
    }
}
```

OS_MutexPendAbbort

```
U08 OS_MutexPendAbbort(OS_MUX *pmux)
```

Abborts waiting of a Tasks (highest waiting prio) on a Mutex.
It is only the waiting task with the highest priority quasi "premature" to TimeOut forwarded.

Parameters

*pmux	pointer to Mutex

Return Value

OS_NO_ERR	pending of a task abborted
OS_TASK_NOT_EXIST	no pending task on this Mutex

Example

```
OS_MUX  Mutex1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_MutexPendAbbort(&Mutex1);
        .
    }
}
```

OS_MutexPost

```
U08 OS_MutexPost(OS_MUX *pmux)
```

Gives a reserved Mutex free and again the access on the protected recource.

Parameters

*pmux	pointer to Mutex

Return Value

OS_NO_ERR	Mutex released
OS_MUX_ERR	Error in Mutex-handling

Example

```
OS_MUX  Mutex1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        .
        state = OS_MutexPost(&Mutex1);
        .
    }
}
```

Event-Groups

OS_EvgInit

`U08 OS_EvgInit(OS_EVG *pevg)`

Initialize an Eventgroup. An Eventgroup consists individually can be processed of 32 single-events, that summarized in an ULONG, as also grouped. Each Event within the group can report the appearance of an event, however any statement about it doesn't meet, how often the event appeared in the meantime. In order to also be able to count events, you must be used semaphores as Counting-Semaphore for every individual event. (semaphore with 0 initialize, at appearance of the event "OS_SemPost" and when wait "OS_SemAccept")

Parameters

*pevg	pointer to Eventgroup

Return Value

OS_NO_ERR	Event-group initialized

Example

```
OS_EVG  Events1;

void main(void)
{
    U08  state;

    .
    .
    OS_Init();
    .
    state = OS_EvgInit(&Events1);
    .
}
```

OS_EvgPost

```
U08 OS_EvgPost(OS_EVG *pevg, U32 events, U08 mode)
```

Report the appearance an as well as several Events of an Eventgroup. The bit mask is interpreted as OR of the Events on that occasion. I.e. all Events, that are set in the bit mask, are reported. Mode is used the utilization of this function for the erasure of Events.

Parameters

*pevg	pointer to Eventgroup
events	bit-mask of events
mode	mode of usement "OS_EVG_OR / OS_EVG_CLR"

Return Value

OS_NO_ERR	Event(s) reported

Example

```
OS_EVG  Events1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_EvgPost(&Events1,~0x00101000, OS_EVG_CLR);   // clear this events
        state = OS_EvgPost(&Events1, 0x01000100, OS_EVG_OR);    // set this events
        .
    }
}
```

OS_EvgPend

```
U08 OS_EvgPend(OS_EVG *pevg, U32 *events, U08 mode, U16 timeout)
```

Waits on one as well as several Events of an Eventgroup. The bit mask is interpreted modes as connection of the Events on that occasion together with him. I.e., already an Event, that is set in the bit mask, is enough with OS_EVG_OR for the function and with OS_EVG_AND, all Events, that are set in the bit mask, had to arrive. For a special case events of an Eventgroup can wake up multiple tasks at once. For this the Eventgroup differs on EvgPend and EvgPost between "OS_EVG_OR_C / OS_EVG_AND_C" for normal use (consuming event) or just "OS_EVG_OR / OS_EVG_AND" to wake up multiple tasks. But the events are then manually to delete again by a task. After returning the bit mask contains the occurred events that triggered the return. With OS_NO_SUSP, it immediately is come back even if no Event appeared and becomes with OS_SUSPEND as long as waited until the Event(s) appeared, if necessary indefinitely.

Parameters

*pevg	pointer to Eventgroup
*events	pointer to bit-mask of events waiting for, returns the events on return
mode	mode of usement "OS_EVG_OR_C / OS_EVG_AND_C"
timeout	kernel-ticks as waiting-time (1...65534)

Return Value

OS_NO_ERR	Event(s) appeared
OS_EVG_NOE	Event(s) not appeared (with OS_NO_SUSP)
OS_TIMEOUT	Event(s) not appeared (after waits)

Example

```
OS_EVG  Events1;

void OS_FAR Task1(void *data)
{
    U32  event;
    U08  state;

    .
    .
    while(1)
    {
        .
        event = 0x00100100;
        state = OS_EvgPend(&Events1, &event, OS_EVG_OR_C, OS_SUSPEND);
        .
    }
}
```

OS_EvgPendAbbort

```
U08 OS_EvgPendAbbort(OS_EVG *pevg)
```

Abborts waiting of a task (highest waiting prio) onto a Eventgroup. It is only the waiting task with the highest priority quasi "premature" to TimeOut forwarded.

Parameters

*pevg	pointer to Eventgroup

Return Value

OS_NO_ERR	pending of a task abborted
OS_TASK_NOT_EXIST	no pending task on this eventgroup

Example

```
OS_EVG  Events1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_EvgPendAbbort(&Events1);
        .
    }
}
```

Timer-Service

OS_TimerCreate

```
U08 OS_TimerCreate(OS_TMR *ptmr, U32 time, void(*tFct)(void *), void *tArg, U08 mode)
```

Creating / initialize a timer.
A timer can e.g. for subsequent purposes:
• timeouts within protocol layers and applications such as TCP / IP, X25, HTTP, FTP, ...
• prevent the "starvation" of tasks by defining a timeout and corresponding measures such as priority raising or other
• Periodic management of services
• soft-deadline / watchdog of services

As *mode* subsequent details can be made:
• OS_TMR_ENABLE - starts the Timer immediately
• OS_TMR_RONCE - Timer is of type "run-once", this means single timeout and after it is automatically disabled, but remains registered
• OS_TMR_CYCL - Timer is of type "cyclic / periodic" this means the timer will automatically restarted after each timeout
• OS_TMR_CLR - Timer is of type "run-once auto-erase," this means the timer is single timeout and is automatically deleted and must be created new for further use

It is OS_TMR_CLR the scheme goes before OS_TMR_CYCL and this before OS_TMR_RONCE.

The registered callback function should be as short as possible. For information-sharing an argument can be used.

Parameters

*ptmr	pointer to Timer
time	timeout of this timer (in timer-ticks)
tFct	address of timeout-callback function
tArg	argument of timeout-callback function
mode	mode of this timer (run-once / cyclic / auto-clear)

Return Value

OS_NO_ERR	Timer registered and initialized
OS_TMR_NO_TIME	no valid time a parameter given (time == 0)
OS_TMR_EXIST	Timer was still registered and initialized

Example

```
OS_TMR  Timer1;

void TCP_To_CB(void *session)
{
    OS_QueuePost(&TCPIP_To_Q, (U08)session, OS_NO_SUSP);
}

U08 TCP_send(void)
{
    U08  state;
    U08  session;

    .
    .
    state = OS_TimerCreate(&Timer1, 30, TCP_To_CB, &session, OS_TMR_ENABLE | OS_TMR_CLR);
    .
}
```

OS_TimerDelete

```
U08 OS_TimerDelete(OS_TMR *ptmr)
```

Deactivate and delete a Timer.

Parameters

*ptmr	pointer to Timer

Return Value

OS_NO_ERR	Timer deleted
OS_TMR_NOT_EXIST	the Timer was not registered

Example

```
OS_TMR  Timer1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_TimerDelete(&Timer1);
        .
        .
    }
}
```

OS_TimerStart

```
U08 OS_TimerStart(OS_TMR *ptmr, U32 time)
```

Starts a deactivated, restart a run-once or restart a running Timer.

Parameters

*ptmr	pointer to Timer
time	new timeout (if not zero) of this timer (in timer-ticks)

Return Value

OS_NO_ERR	Timer (re-)started
OS_TMR_NOT_EXIST	the Timer is not registered

Example

```
OS_TMR  Timer1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        state = OS_TimerStart(&Timer1, 0);
        .
        .
    }
}
```

OS_TimerStop

```
U08 OS_TimerStop(OS_TMR *ptmr)
```

Stops / deactivat a running Timer.

Parameters

*ptmr	pointer to Timer

Return Value

OS_NO_ERR	Timer stopped
OS_TMR_NOT_EXIST	the Timer is not registered

Example

```
OS_TMR  Timer1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        .
        state = OS_TimerStop(&Timer1);
        .
    }
}
```

OS_TimerGetState

```
U08 OS_TimerGetState(OS_TMR *ptmr)
```

Returns the status of a created timer.
The following information is provided:
• OS_TMR_ENABLE - the Timer is actually running
• OS_TMR_RONCE - Timer is of type "run-once", this means single timeout and after it is automatically disabled, but remains registered
• OS_TMR_CYCL - Timer is of type "cyclic / periodic" this means the timer will automatically restarted after each timeout
• OS_TMR_CLR - Timer is of type "run-once auto-erase," this means the timer is single timeout and is automatically deleted and must be created new for further use

If in the returns status not OS_TMR_ENABLE but OS_TMR_CLR, the timer was never created or the timer was "run-once auto-erase" and the time had expired.

Parameters

*ptmr	pointer to Timer

Return Value

status	status of the timers (see "modi" on OS_TimerCreate)

Example

```
OS_TMR  Timer1;

void OS_FAR Task1(void *data)
{
    U08  state;

    .
    .
    while(1)
    {
        .
        .
        state = OS_TimerGetState(&Timer1);
        if(state & OS_TMR_ENABLE)
        {
            .
            .
        }
        .
    }
}
```

OS_TimerGetRemain

```
U32 OS_TimerGetRemain(OS_TMR *ptmr)
```

Returns the remaining time (in timer-ticks) of a running timer.
Is the returned time equal to 0, the timer was expired or was never activated by OS_TimerCreate.

Parameters

*ptmr	pointer to Timer

Return Value

time	remaining time in timer-ticks

Example

```
OS_TMR  Timer1;

void OS_FAR Task1(void *data)
{
    U32  rtime;

    .
    .
    while(1)
    {
        .
        .
        rtime = OS_TimerGetRemain(&Timer1);
        .
    }
}
```

System-Ticks

OS_TimeSet

```
void OS_TimeSet(U32 ticks)
```

Places the kernel-internal Tick-Counter on handed over value.

Parameters

ticks	new value of tick-counter

Return Value

none

Example

```
void OS_FAR Task1(void *data)
{
    .
    .
    while(1)
    {
        .
        OS_TimeSet(24837);
        .
    }
}
```

OS_TimeGet

```
U32 OS_TimeGet(void)
```

It returns the current value of the kernel-internal Tick-Counter.

Parameters

none

Return Value

ticks	actual value of tick-counter

Example

```
void OS_FAR Task1(void *data)
{
    U32  time;

    .
    .
    while(1)
    {
        .
        time = OS_TimeGet();
        .
    }
}
```

Interrupts

OS_IntEnter

```
void OS_IntEnter(void)
```

Register an Interrupt-Level. No contextswitch are generated by it. This function is necessary for C-Code ISRs.

Parameters

none

Return Value

none

Example

```
void OS_FAR ISR1(void)
{
    OS_IntEnter();
    .
    .
    .
    OS_IntExit();
}
```

OS_IntExit

```
void OS_IntExit(void)
```

Unregister an Interrupt-Level. Contextswitches are generated again by it. This function is necessary for C-Code ISRs.

Parameters

none

Return Value

none

Example

```
void OS_FAR ISR1(void)
{
    OS_IntEnter();
    .
    .
    .
    OS_IntExit();
}
```

History

OS_HistoryPost

```
U08 OS_HistoryPost(U32 param1, U32 param2)
```

Writes down an entry into the History-Table of the kernel. Additional to the two parameters still becomes the priority of the Tasks and the Tick-Counter, as time stamps, written down.

Parameters

param1	first 32-bit parameter for table
param2	secound 32-bit parameter for table

Return Value

OS_NO_ERR	Entry written

Example

```
OS_Q  Queue5;

void OS_FAR Task2(void *data)
{
    U08  state;
    U08  Message;

    .
    .
    while(1)
    {
        .
        Message = 0x2D;
        state = OS_QueueFrontPost(&Queue5, Message, 200);
        if(state != OS_NO_ERR)
            OS_HistoryPost((U32)state, 0x0205);
        .
    }
}
```

OS_HistoryRead

```
U08 OS_HistoryRead(U32 *param1, U32 *param2, U08 *prio, U32 *time)
```

Reads next entry from the History-Table of the kernel and deletes this on that occasion.

Parameters

*param1	pointer to variable will get the first 32-bit parameter
*param2	pointer to variable will get the secound 32-bit parameter
*prio	pointer to variable will get priority of task who has this written
*time	pointer to variable will get time-stamp of this entry

Return Value

OS_NO_ERR	Entry read
OS_HIS_END	no entry existing

Example

```
void OS_FAR Task3(void *data)
{
    U08  state;
    U32  Hpara1;
    U32  Hpara2;
    U08  Tprio;
    U32  stamp;

    .
    .
    while(1)
    {
        .
        state = OS_HistoryRead(&Hpara1, &Hpara2, &Tprio, &stamp);
        .
    }
}
```

Comments

Comments

Comments