embedded-os.de
a little world of RTOS and data-communication protocols

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.

Zu den für Anfänger am schnellsten zu verstehenden Echtzeitbetriebssystemen gehört µC/OS von Jean J. Labrosse (siehe http://www.micrium.com).
Es kann in den Versionen 1.xx bis zu 63 Applikationstasks mit je unterschiedlichen Prioritäten verwalten und gehört zu den Echtzeitbetriebssystemen mit dem geringsten Speicherbedarf.
pC/OS wurde basierend auf der Orginalversion µC/OS 1.00 aus dem Embedded Systems Programming Magazine(1992) weiterentwickelt.

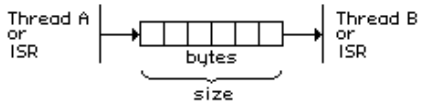
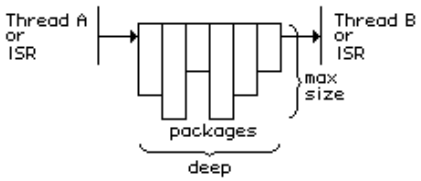
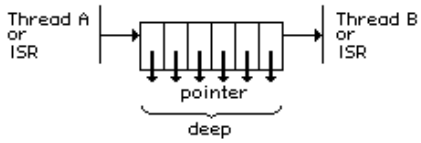
Da in der Originalversion Daten durch direkte Übergabe von Zeigern zwischen den Tasks ausgetauscht werden, und somit keine Garantie für die freie Verwendbarkeit des Absender-buffers nach Übergabe an einen anderen Task besteht und, viel wichtiger, der Empfänger einen Zeiger in das Datenfeld eines anderen Tasks erhält (Pointer-Fehler / Längen-Fehler / Manipulationen u.a.) wurden die Mechanismen für Message-Box und Queue entsprechend so geändert, daß nun die Daten über einen Kernel-internen Buffer an den Empfänger übergeben werden. Das bedeutet, daß der Kernel zu übertragende Daten in einen eigenen Buffer kopiert und bei Übergabe an den Empfänger diese auch wieder selber in den durch den Empfänger bereitgestellten Buffer kopiert. Das bringt zwar einen höheren Speicherbedarf je Queue mit sich, sichert aber dafür die Prozesse weitestgehend (für Real-Mode) voneinander ab.
Aus Sicherheitsgründen wurden außerdem Kernel-Konstanten in den CODE-Area verlegt.
Desweiteren wurde der Kernel um die Diente Pipe, Eventgroup, Mutex, Timerservice und dyn. Memorymanagement erweitert.

Um diese Änderungen eindeutig zu deklarieren wurde der Namen, angelehnt an dem immer größer werdenden Original "µC/OS", auf pC/OS wie "pico-C.." geändert.

Special zu: Priority Inversion, das Problem und die Lösungsansätze

bekannter Bug:
Wenn ein niederpriorisierter Task auf eine Recource wartet, und ein höherpriorisierter Task diese Recource setzt, so wird der schlafende Task in den Ready-state versetzt. Da der höherpriorisierte Task weiterläuft, darf dieser die selbe Recource nicht wieder auslesen, da ansonsten der niederpriorisierte Task bei Ausführung 'vorzeitig' mit dem Return-Code OS_TIMEOUT zurück kommt.

Da die Begriffe der verschiedenen IPC's mit Datentransport nicht eindeutig sind und unter den selben Begriffen unterschiedliche Implementierungen in verschiedenen RTOS-Systemen vorzufinden sind, hier eine Übersicht aller im pC/OS implementierten IPC's.

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"		byte-by-byte transfer (FIFO / LIFO) - the bytes are temp stored in the queue (a serial transfer-buffer for bytes)
Package-Queue	"Pipe"		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"		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)

Bitte beachten Sie, daß einige Funktionen unter dem selben Namen wie im Original aber mit modifizierten Parametern bzw. Zeigern deklariert sind.

User-Functions:

Task-Control:	Description
OS_Init	Initialisierung des Kernels
OS_Start	Beginn der Kernelservices
OS_TaskCreate	Anlegen eines Tasks
OS_ChangePrio	Änderung der Priorität des aktiven Tasks
OS_TaskChangePrio	Änderung der Priorität eines aktiven/ready Tasks
OS_TaskDelete	Löschen eines aktiven/ready Tasks
OS_TaskIdDelete	Löschen eines aktiven/ready Tasks via unique ID
OS_TaskGetStatus	Gibt den aktuellen Status eines Tasks zurück
OS_TaskIdGetStatus	Gibt den aktuellen Status eines Tasks via unique ID zurück
OS_TaskGetID	Gibt die unique ID eines Tasks zurück
OS_TaskGetPrio	Gibt die Priorität eines Tasks zurück
OS_TaskIdDestroy	Löschen eines Tasks via unique ID, auch wenn dieser an einer IPC wartet oder eine Mutex inne hat & freigeben aller Memory Allokationen
OS_TaskSuspend	Suspendiert einen Task
OS_TaskIdSuspend	Suspendiert einen Task via unique ID
OS_TaskResume	Wiederaufwecken eines suspendierten Tasks
OS_TaskIdResume	Wiederaufwecken eines suspendierten Tasks via unique ID
OS_TimeDly	Legt laufenden Task für bestimmte Zeit schlafen
OS_TimeDlyResume	Wiederaufwecken eines schlafenden Tasks vor Ablauf der eingestellten Zeit
OS_TimeDlyIdResume	Wiederaufwecken eines schlafenden Tasks via unique ID vor Ablauf der eingestellten Zeit
OS_Lock	Unterdrücken des Shedulers (keine Taskwechsel)
OS_Unlock	Wiederzuschaltung des Shedulers (Taskwechsel bei Ereignis oder Zeit)
OS_GetRev	Gibt Zeiger auf Kernel-Revision zurück

Dynamic-Memory:	Description
OS_MemoryInit	Erzeugen des Speicherpools
OS_MemAlloc	Allokieren von Speicher
OS_MemFree	Freigeben von allokiertem Speicher
OS_MemFreeSize	Gibt den Freispeicher zurück
OS_MemVerifyPtr	Prüft, ob der Zeiger innerhalb des Speicherpools zeigt

SmartMessageQueue:	Description
OS_SMQueueInit	Anlegen einer SmartMessageQueue
OS_SMQueueInfo	Informationen über eine SmartMessageQueue einholen
OS_SMQueueClear	Löscht alle Messages in einer SmartMessageQueue
OS_SMQueuePost	Sendet eine Message in eine SmartMessageQueue
OS_SMQueueFrontPost	Sendet eine Message an den Anfang einer SmartMessageQueue
OS_SMQueuePostAbbort	Bricht Warten eines sendenden Tasks (höchste wartende Prio) an einer SmartMessageQueue ab
OS_SMQueuePend	Wartet auf eine Message aus einer SmartMessageQueue
OS_SMQueuePendAbbort	Bricht Warten eines empfangenden Tasks (höchste wartende Prio) an einer SmartMessageQueue ab

Mailboxes:	Description
OS_MboxInit	Anlegen einer Mailbox
OS_MboxPost	Sendet Daten an höchstpriorisierten Empfänger dieser Mailbox
OS_MboxPostAbbort	Bricht Warten eines sendenden Tasks (höchste wartende Prio) an einer Mailbox ab
OS_MboxPend	Wartet auf Daten aus einer Mailbox
OS_MboxPendAbbort	Bricht Warten eines empfangenden Tasks (höchste wartende Prio) an einer Mailbox ab

Queues:	Description
OS_QueueInit	Anlegen einer Queue
OS_QueueInfo	Informationen über eine Queue einholen
OS_QueuePost	Sendet Daten in eine Queue
OS_QueueFrontPost	Sendet Daten an den Anfang einer Queue
OS_QueuePostAbbort	Bricht Warten eines sendenden Tasks (höchste wartende Prio) an einer Queue ab
OS_QueuePend	Wartet auf Daten aus einer Queue
OS_QueuePendAbbort	Bricht Warten eines empfangenden Tasks (höchste wartende Prio) an einer Queue ab
OS_QueueClear	Löscht alle Daten in einer Queue

Pipes:	Description
OS_PipeInit	Anlegen einer Pipe
OS_PipeInfo	Informationen über eine Pipe einholen
OS_PipePost	Sendet Daten in eine Pipe
OS_PipeFrontPost	Sendet Daten an den Anfang einer Pipe
OS_PipePostAbbort	Bricht Warten eines sendenden Tasks (höchste wartende Prio) an einer Pipe ab
OS_PipePend	Wartet auf Daten aus einer Pipe
OS_PipePendAbbort	Bricht Warten eines empfangenden Tasks (höchste wartende Prio) an einer Pipe ab
OS_PipeClear	Löscht alle Daten in einer Pipe

inter-core AMP-Pipes:	Description
OS_IccInit	Initialisieren des AMP-Pipe Paares
OS_IccInfo	Informationen über das AMP-Pipe Paar einholen
OS_IccPost	Sendet Daten zum anderen Core
OS_IccPend	Wartet auf Daten vom anderen Core
OS_IccClear	Löscht alle Daten in der AMP-Pipe hin zum anderen Core

Semaphores:	Description
OS_SemInit	Anlegen einer Semaphore
OS_SemAccept	wartet auf Ereignis und gibt Anzahl zurück
OS_SemPost	Freigabe einer belegten Semaphore / setzt Ereignis
OS_SemPend	Belegt eine Semaphore / wartet auf Ereignis
OS_SemPendAbbort	Bricht Warten eines Tasks (höchste wartende Prio) an einer Semaphore ab
OS_SemClear	Löscht Semaphoren-Counter

Mutexes:	Description
OS_MutexCreate	Anlegen einer Mutex
OS_MutexPost	gibt Mutex wieder frei
OS_MutexPend	Besetzt die Mutex
OS_MutexPendAbbort	Bricht Warten eines Tasks (höchste wartende Prio) an einer Mutex ab

Event-Groups:	Description
OS_EvgInit	Anlegen einer Eventgruppe
OS_EvgPost	Setzt ein/mehrere Events einer Evengruppe
OS_EvgPend	Wartet auf das Eintreffen eines oder mehrere Events einer Eventgruppe
OS_EvgPendAbbort	Bricht Warten eines Tasks (höchste wartende Prio) an einer Eventgruppe ab

Timer-Service:	Description
OS_TimerCreate	Anlegen eines Timers
OS_TimerDelete	Löschen eines angelegten Timers
OS_TimerStart	(Re-)Starten eines angelegten Timers
OS_TimerStop	Stoppen eines angelegten Timers
OS_TimerGetState	gibt den Status eines angelegten Timers zurück
OS_TimerGetRemain	gibt die verbleibende Zeit eines laufenden Timers zurück

System-Ticks:	Description
OS_TimeSet	Setzt Ticker auf übergebenen Wert
OS_TimeGet	Gibt aktuellen Ticker-Wert zurück

Interrupts:	Description
OS_IntEnter	Registrierung einer aufgerufenen ISR
OS_IntExit	Ende einer aufgerufenen ISR

History:	Description
OS_HistoryPost	Schreibt Eintrag in History
OS_HistoryRead	Gibt ersten History-Eintrag und löscht diesen in der Tabelle

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)

Konfiguration des Kernels

Der pC/OS Kernel stellt neben der zu verwendenden Hardware-Portierung mehrere Möglichkeiten zur Konfiguration von Services/IPCs sowie zur Reduzierung des Speicherbedarfs - Code-size bei Compilern die "unused code" nicht eindeutig identifizieren können und RAM - zur Verfügung. Diese sind in der Datei "OS_cfg.h" zusammengefaßt.

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

zu OS_MAX_TASKS und OS_MIN_PRIO:
Wenn ein System benötigt wird mit 5 Tasks wovon 2 Tasks eine gemeinsame Mutex benutzen, braucht man also OS_MAX_TASKS = 7 (incl. eine Mutex und Idle-Task) und OS_MIN_PRIO = 8 wobei der Idle-Task dann die Prio 7 erhält und alle anderen Tasks und die Mutex höhere Prioritäten bekommen (0..6). Soll auch der Timer-Service verwendet werden, so ist dieser Timer-Tasks zusätzlich einzubeziehen.

Managed / Unmanaged Interrupt Service Routinen (ISR)

Der pC/OS Kernel muss informiert sein, wenn eine ISR läuft und muss bei Verlassen dieser prüfen, ob ein Prozesswechsel "preemptiv" notwendig ist. Dazu gibt es, abhängig von der Hardware und der verwendeten Implementierung zwei Möglichkeiten:

managed ISR	der IRQ Entry/Exit ist zentral und informiert den Kernel (siehe zB. ARM7TDMI ports)
unmanaged ISR	jede ISR ist eigenständig und wird direkt aus der Vector-Tabelle heraus angesprungen -> der Kernel muss informiert werden. (siehe zB. Cortex-Mx ports)

Bei managed-ISRs übernimmt der zentrale ISR Entry/Exit code die Aufgabe den Kernel zu informieren, sodaß die ISR selbst nichts für den Kernel beachten muss. Bei unmanged ISRs hingegen muss als Erstes `OS_IntEnter()` und am Ende `OS_IntExit()` aufgerufen werden !

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)
```

Initialisiert den Kernel und installiert den Idle-Task. Diese Funktion muß vor allen anderen Kerneldiensten bei der Systeminitialisierung einmal aufgerufen werden.

Parameters

none

Return Value

none

Example

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

OS_Start

```
void OS_Start(void)
```

Startet den Kernel. Diese Funktion aktiviert das Multitasking und kehrt nicht zurück.

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)
```

Legt einen neuen Task an. Diese Funktion initialisiert den Task-Control-Block und trägt den neuen Task mit seinem Stack und den Aufrufparametern ein. Dieses kann aus main() während der Initialisierung bzw. einem anderen Task in Laufzeit heraus erfolgen.

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 erfolgreich angelegt
OS_PRIO_EXIST	unter dieser Priorität existiert bereits ein Task
OS_PRIO_INVALID	diese Priorität ist für den Idle-Task reserviert bzw. der Wert der Priorität ist größer 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)
```

Ändert die Priorität des aktuellen Tasks. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (z.B. Events) die Priotität des Tasks zu verändern.

Parameters

newp	new priority of this task

Return Value

OS_NO_ERR	Priorität erfolgreich geändert
OS_PRIO_EXIST	unter dieser Priorität existiert bereits ein Task
OS_PRIO_INVALID	diese Priorität ist für den Idle-Task reserviert bzw. der Wert der Priorität ist größer OS_MIN_PRIO
OS_MUX_USED	der Task hat eine Mutex in Besitz

Example

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

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

OS_TaskChangePrio

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

Ändert die Priorität eines aktiven/ready Tasks. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (z.B. Events) die Priotität eines aktiven/ready Tasks zu verändern.
Tasks die an einer Resource (Semaphore/Queue/Pipe/..) warten können nicht umpriorisiert werden, da dieser Zustand für den Kernel erkennbar ist aber nicht die exakte Resource selbst.

Parameters

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

Return Value

OS_NO_ERR	Priorität erfolgreich geändert
OS_PRIO_EXIST	unter dieser Priorität existiert bereits ein Task
OS_PRIO_INVALID	diese Priorität ist für den Idle-Task reserviert bzw. der Wert der Priorität ist größer OS_MIN_PRIO
OS_TASK_NOT_RDY	der Task ist nicht im RUNNING/READY-state und kann daher nicht umpriorisiert werden
OS_MUX_USED	der Task hat eine Mutex in Besitz

Example

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

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

OS_TaskDelete

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

Entfernt den angegebenen Task. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (Events) einen Task zu beenden/löschen. Dieser Task ist anschließend aus der Task-Control-Tabelle entfernt. Allokierte Resourcen des Tasks werden dabei NICHT automatisch freigegeben.
Um diesen Task später erneut ausführen zu können, muß er regulär mittels OS_TaskCreate neu angelegt werden.
Tasks die an einer Resource (Semaphore/Queue/Pipe/..) warten können so nicht beendet werden, da dieser Zustand für den Kernel erkennbar ist aber nicht die exakte Resource selbst.

Parameters

prio	priority of the task to delete

Return Value

OS_NO_ERR	Task gelöscht
OS_TASK_NOT_EXIST	unter dieser prio ist kein Task eingetragen
OS_TASK_NOT_RDY	der Task ist nicht im RUNNING/READY-state und kann daher nicht gelöscht werden
OS_MUX_USED	der Task hat eine Mutex in Besitz

Example

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

OS_TaskIdDelete

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

Entfernt den angegebenen Task via unique-ID. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (Events) einen Task zu beenden/löschen. Dieser Task ist anschließend aus der Task-Control-Tabelle entfernt. Allokierte Resourcen des Tasks werden dabei NICHT automatisch freigegeben.
Um diesen Task später erneut ausführen zu können, muß er regulär mittels OS_TaskCreate neu angelegt werden.
Tasks die an einer Resource (Semaphore/Queue/Pipe/..) warten können so nicht beendet werden, da dieser Zustand für den Kernel erkennbar ist aber nicht die exakte Resource selbst.

Parameters

id	unique ID of the task to delete

Return Value

OS_NO_ERR	Task gelöscht
OS_TASK_NOT_EXIST	unter dieser ID ist kein Task eingetragen
OS_TASK_NOT_RDY	der Task ist nicht im RUNNING/READY-state und kann daher nicht gelöscht werden
OS_MUX_USED	der Task hat eine Mutex in Besitz

Example

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

OS_TaskGetStatus

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

Gibt den aktuellen Status des angegebenen Tasks zurück.

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)
```

Gibt den aktuellen Status des via unique-ID angegebenen Tasks zurück.

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)`

Gibt die unique-ID das angegebenen Tasks zurück. Diese kann später verwendet werden, um z.B. einen Task - auch wenn er seine Prio inzwischen geändert hat oder gerade eine Mutex inne hat - gewaltsam zu beenden (see `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)
```

Gibt die aktuelle Priorität des via unique-ID angegebenen Tasks zurück.

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)
```

Entfernt den angegebenen Task vollständig unabhängig von seinem Status. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (Events) einen Task abzubrechen. Dieser Task ist anschließend aus der Task-Control-Tabelle, aus eventuell eingetragenen IPCs (Semaphore/MBox/Queue/Pipe/..) und dem Memory-Manager komplett entfernt.
Sollte der Tasks zu diesem Zeitpunkt eine Mutex besetzt haben, so wird diese freigegeben und eine User-CallBack Funktion aufgerufen, um eine eventuell notwendige Reinitialisierung der betroffenen Hardware o.ä. ausfähren zu können (siehe OSMutexReInitResource() in "pC_OS_userCB.c"). Dies kann aber nur für eine Mutex (die letzte) erfolgen. Sollte der Task zwei Mutexes inne haben, so wird nur die zuletzt besetzte freigegeben !
Um diesen Task später erneut ausführen zu können, muß er regulär mittels OS_TaskCreate neu angelegt werden.
ACHTUNG:
Während dieser Task aus der der Task-Control-Tabelle und aus IPCs entfernt wird sind alle Interrupte unterbunden, da ein für diesen Task passendes Event zu einem Zugriff/Update Konflikt führen könnte. Während dem anschließenden Aufräumen im Memory-Manager sind Interrupte wieder zulässig aber ein Sheduling wird unterdrückt um ein gleichzeitiges Wiederhochfahren dieses Tasks mittels OS_TaskCreate() zu verhindern (prio of this task).

Parameters

id	unique-ID of the task to destroy

Return Value

OS_NO_ERR	Task vollständig entfernt
OS_TASK_NOT_EXIST	der angegebene Task existiert nicht
OS_TASK_NOT_RDY	der Task konnte nicht vollständig aus den IPCs oder einer Mutex entfernt werden
OS_MEM_ERR	beim Freigeben der Memory-Allokationen des Tasks ist ein Fehler aufgetreten

Example

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

OS_TaskSuspend

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

Suspendiert einen Task von der Ausführung durch den Kernel. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (Events) einen Task vorrübergehend zu deaktivieren.

Parameters

prio	priority of task to suspending

Return Value

OS_NO_ERR	Task erfolgreich suspendiert
OS_SUSPEND_IDLE	der Idle-Task darf nicht suspendiert werden
OS_PRIO_INVALID	der Wert der Priorität ist größer OS_MIN_PRIO
OS_TASK_SUSP_PRIO	unter dieser Priorität ist kein Task eingetragen
OS_MUX_USED	der Task hat eine Mutex in Besitz

Example

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

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

OS_TaskIdSuspend

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

Suspendiert einen Task via seiner unique ID von der Ausführung durch den Kernel. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (Events) einen Task vorrübergehend zu deaktivieren.

Parameters

id	unique ID of task to suspending

Return Value

OS_NO_ERR	Task erfolgreich suspendiert
OS_SUSPEND_IDLE	der Idle-Task darf nicht suspendiert werden
OS_TASK_SUSP_PRIO	unter dieser ID ist kein Task eingetragen
OS_MUX_USED	der Task hat eine Mutex in Besitz

Example

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

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

OS_TaskResume

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

Reaktiviert einen suspendierten Task. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (z.B. Events) einen suspendierten Task wieder zu aktivieren.

Parameters

prio	priority of suspended task

Return Value

OS_NO_ERR	Task erfolgreich reaktiviert
OS_TASK_NOT_SUSP	der Task ist nicht suspendiert
OS_PRIO_INVALID	der Wert der Priorität ist größer OS_MIN_PRIO
OS_TASK_NOT_EXIST	unter dieser Priorität ist kein Task eingetragen
OS_MUX_USED	der Task hat eine Mutex in Besitz

Example

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

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

OS_TaskIdResume

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

Reaktiviert einen suspendierten Task basierend auf seiner unique ID. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (z.B. Events) einen suspendierten Task wieder zu aktivieren.

Parameters

id	unique ID of suspended task

Return Value

OS_NO_ERR	Task erfolgreich reaktiviert
OS_TASK_NOT_SUSP	der Task ist nicht suspendiert
OS_TASK_NOT_EXIST	unter dieser ID ist kein Task eingetragen
OS_MUX_USED	der Task hat eine Mutex in Besitz

Example

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

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

OS_TimeDly

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

Legt den aktuellen Task für angegebene Kernel-Ticks schlafen. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (Events) eine definierte Zeit verstreichen zu lassen. ACHTUNG! Mit dem Parameter OS_SUSPEND (0) wird der Task für immer deaktiviert und kann nie wieder aktiviert werden.

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)
```

Bricht die Wartezeit eines Tasks vorzeitig ab. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (z.B. Events) einen schlafenden Task vorzeitig wieder zu aktivieren.

Parameters

prio	priority of sleeping task

Return Value

OS_NO_ERR	Task erfolgreich aufgeweckt
OS_TIME_NOT_DLY	der Task wartet nicht
OS_PRIO_INVALID	der Wert der Priorität ist größer OS_MIN_PRIO
OS_TASK_NOT_EXIST	unter dieser Priorität ist kein Task eingetragen

Example

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

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

OS_TimeDlyIdResume

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

Bricht die Wartezeit eines Tasks basierend auf seiner unique ID vorzeitig ab. Diese Funktion kann verwendet werden, um z.B. auf Grund eines Ereignisses (z.B. Events) einen schlafenden Task vorzeitig wieder zu aktivieren.

Parameters

id	unique ID of sleeping task

Return Value

OS_NO_ERR	Task erfolgreich aufgeweckt
OS_TIME_NOT_DLY	der Task wartet nicht
OS_TASK_NOT_EXIST	unter dieser ID ist kein Task eingetragen

Example

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

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

OS_Lock

```
void OS_Lock(void)
```

Schaltet den Sheduler aus. Diese Funktion kann verwendet werden, um z.B. atomare (nicht unterbrechbare) Vorgänge ausführen zu können, ohne daß ein Taskwechsel zwischendrin stattfinden kann.Die Interrupte werden weiterhin bedient.

Parameters

none

Return Value

none

Example

```
void OS_FAR Task1(void *data)
{
    .
    .
    while(1)
    {
        .
        OS_Lock();
        .
        .        // nicht unterbrechbarer Teil
        .
        OS_Unlock();
        .
    }
}
```

OS_Unlock

```
void OS_Unlock(void)
```

Schaltet den Sheduler wieder ein. Diese Funktion wird verwendet, um z.B. atomare (nicht unterbrechbare) Vorgänge abzuschliessen und einen Taskwechsel wieder zu ermöglichen

Parameters

none

Return Value

none

Example

```
void OS_FAR Task1(void *data)
{
    .
    .
    while(1)
    {
        .
        OS_Lock();
        .
        .        // nicht unterbrechbarer Teil
        .
        OS_Unlock();
        .
    }
}
```

OS_GetRev

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

Gibt einen Zeiger auf die Kernelrevision (NULL-terminiertes ASCII-Array) zurück.

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)
```

Initialisiert das dynamische Memory-Management. Diese Funktion muß für das dynamische Memory-Management bei der Systeminitialisierung einmal aufgerufen werden. Die Funktionen des dynamische Memory-Managements können auch ohne laufenden Kernel genutzt werden. ACHTUNG! Es wird keine Überprüfung des Speicherbereiches durchgeführt.

Parameters

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

Return Value

OS_NO_ERR	Memory-Pool angelegt
OS_MEM_ERR	einer der Parameter ist NULL

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)
```

Allokieren eines benötigten Memorybereiches.

Parameters

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

Return Value

OS_NO_ERR	Speicher erfolgreich allokiert
OS_MEM_ERR	size ist NULL oder größer als Speicherbereich des Prozessors
OS_MEM_OVF	nicht genügend freier Speicher

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)
```

Freigeben eines allokierten Memorybereiches und löschen des Zeigers.

Parameters

**MemPtr	pointer of pointer to address of allocated memory

Return Value

OS_NO_ERR	Speicher erfolgreich freigegeben
OS_MEM_ERR	Pointer ist NULL oder keine gültige Allokation vorgefunden

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)
```

Gibt den Freispeicher zurück.

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)
```

Prüft, ob ein Pointer in den Memory-Pool zeigt. Dabei wird jedoch nicht geprüft, ob es sich um einen Zeiger auf einen allokierten Block handelt, nur der Address-range des Memory-Pools wird herangezogen.

Parameters

*MemPtr	a pointer

Return Value

OS_NO_ERR	Pointer zeigt in den Memory-Pool
"1"	Pointer zeigt nicht in den 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_SMQueueInit

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

Initialisieren einer SmartMessageQueue. Eine SmartMessageQueue dient der Übergabe von Daten mittels eines Pointers an einen anderen Prozess nach dem FIFO-Prinzip. Wenn der Pointer auf einen allokierten Memory-Block zeigt, so wird die Eigentümerschaft dieses Blocks auch übertragen. Zeigt der Pointer nicht auf einen allokierten Memory-Block so bleibt das Risiko beim Anwender, da ein unspezifischer Pointer direkt übergeben wird. Innerhalb dieser SmartMessageQueue können <deep> Pointer durch den Kernel-Puffer <*buffer> an andere Prozesse übergeben werden. Der Puffer kann durch direkte Deklaration (OS_STK_TYPE SMQ_d[deep]) oder durch dynamische Allokation erzeugt werden. Für die dynamische Allokation in Systemen mit segmentbasierter Speicherverwaltung ist die Typdeklaration OS_HUGE erforderlich, um über Segmentgrenzen hinweg ein Area ansprechen zu können.

Parameters

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

Return Value

OS_NO_ERR	SmartMessageQueue initialisiert
OS_SMQ_ERR	kein *psmq Pointer übergeben

Example

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

void main(void)
{
    U08  state;

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

OS_SMQueueInfo

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

Abfrage des Status einer SmartMessageQueue. Durch diese Abfrage lassen sich diverse Parameter ihrer Initialisierung sowie ihr Füllstand ermitteln. Desweiteren kann die Priorität des wartenden Tasks ermittelt werden.

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	kein Fehler (für Erweiterungen)

Example

```
OS_SMQ      MQueue;

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

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

OS_SMQueueClear

```
U08 OS_SMQueueClear(OS_SMQ *psmq)
```

Löscht den Inhalt einer SmartMessageQueue und reaktiviert einen wartenden Sendeprozess. Diese Funktion kann für Fehlerbehandlungen verwendet werden, um den Datentransfer wieder neu zu starten. Enthaltene Pointer auf einen allokierten Memory-Block werden freigegeben, ansonsten werden die Pointer gelöscht und gehen dabei verloren.

Parameters

*psmq	pointer to SmartMessageQueue

Return Value

OS_NO_ERR	SmartMessageQueue-Inhalt gelöscht

Example

```
OS_SMQ      MQueue;

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

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

OS_SMQueuePost

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

Sendet einen Pointer in eine SmartMessageQueue. Wenn der Pointer auf einen allokierten Memory-Block zeigt, so wird die Eigentümerschaft dieses Blocks vom Kernel übernommen und der Pointer des Senders gelöscht. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn die SmartMessageQueue voll war und mit OS_SUSPEND wird solange gewartet bis der Pointer eingetragen werden kann (notfalls endlos).

Parameters

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

Return Value

OS_NO_ERR	Pointer in SmartMessageQueue gesendet
OS_SMQ_FULL	SmartMessageQueue voll (bei OS_NO_SUSP)
OS_TIMEOUT	SmartMessageQueue voll (nach warten)

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_SMQueuePost(&MQueue, &user_p, 500);
        if (state == OS_NO_ERR)
        {
            .
            if (user_p == NULL)
                // ownership of memory-pool pointer was taken by kernel
            .
        }
    }
}
```

OS_SMQueueFrontPost

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

Sendet einen Pointer an den Anfang einer SmartMessageQueue. Somit wird dieser Pointer durch den Empfänger zuerst wieder ausgelesen (vordrängeln). Wenn der Pointer auf einen allokierten Memory-Block zeigt, so wird die Eigentümerschaft dieses Blocks vom Kernel übernommen und der Pointer des Senders gelöscht. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn die SmartMessageQueue voll war und mit OS_SUSPEND wird solange gewartet bis der Pointer eingetragen werden kann (notfalls endlos).

Parameters

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

Return Value

OS_NO_ERR	Pointer in SmartMessageQueue gesendet
OS_SMQ_FULL	SmartMessageQueue voll (bei OS_NO_SUSP)
OS_TIMEOUT	SmartMessageQueue voll (nach warten)

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_SMQueueFrontPost(&MQueue, &user_p, 500);
        if (state == OS_NO_ERR)
        {
            .
            if (user_p == NULL)
                // ownership of memory-pool pointer was taken by kernel
            .
        }
    }
}
```

OS_SMQueuePostAbbort

```
U08 OS_SMQueuePostAbbort(OS_SMQ * psmq)
```

Bricht das Warten eines sendenden Tasks an der SmartMessageQueue ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*psmq	pointer to SmartMessageQueue

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der SmartMessageQueue

Example

```
OS_SMQ      MQueue;

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

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

OS_SMQueuePend

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

Warte auf einen Pointer aus einer SmartMessageQueue. Wenn der Pointer auf einen allokierten Memory-Block zeigt, so wird die Eigentümerschaft dieses Blocks an den Empfänger übertragen. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn kein Pointer vorlag und mit OS_SUSPEND wird solange gewartet bis ein Pointer vorliegt (notfalls endlos).

Parameters

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

Return Value

OS_NO_ERR	Pointer aus SmartMessageQueue erhalten
OS_SMQ_NODATA	kein Pointer in SmartMessageQueue (bei OS_NO_SUSP)
OS_TIMEOUT	kein Pointer in SmartMessageQueue (nach warten)

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_SMQueuePend(&MQueue, &user_p, OS_SUSPEND);
        .
        .
        if ((user_p != NULL) && (!OS_MemVerifyPtr(user_p)))
            state = OS_MemFree(user_p);
        .
    }
}
```

OS_SMQueuePendAbbort

```
U08 OS_SMQueuePendAbbort(OS_SMQ * psmq)
```

Bricht das Warten eines empfangenden Tasks an der SmartMessageQueue ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*psmq	pointer to SmartMessageQueue

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der SmartMessageQueue

Example

```
OS_SMQ      MQueue;

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

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

Mailboxes

OS_MboxInit

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

Initialisieren einer Mailbox. Mittels einer Mailbox können Daten jeglicher Art durch einen Zeiger übergeben werden. Dabei erhält der Empfänger jedoch einen Zeiger in das Datenfeld des Absenders!

Parameters

*pmbox	pointer to Mailbox

Return Value

OS_NO_ERR	Mailbox initialisiert

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)
```

Warten auf eine Nachricht aus einer Mailbox. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn keine Nachrichten vorlagen und mit OS_SUSPEND wird solange gewartet bis eine Nachricht vorliegt (notfalls endlos).

Parameters

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

Return Value

OS_NO_ERR	Nachricht aus Mailbox erhalten
OS_MBOX_NODATA	keine Nachricht in Mailbox (bei OS_NO_SUSP)
OS_TIMEOUT	keine Nachricht in Mailbox (nach warten)

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)
```

Bricht das Warten eines empfangenden Tasks an der Mailbox ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*pmbox	pointer to Mailbox

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der 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)
```

Sendet eine Nachricht in eine Mailbox. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn die Mailbox voll war und mit OS_SUSPEND wird solange gewartet bis die Nachricht eingetragen werden kann (notfalls endlos).

Parameters

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

Return Value

OS_NO_ERR	Nachricht in Mailbox gesendet
OS_MBOX_FULL	Mailbox voll (bei OS_NO_SUSP)
OS_TIMEOUT	Mailbox voll (nach warten)

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)
```

Bricht das Warten eines sendenden Tasks an der Mailbox ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*pmbox	pointer to Mailbox

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der 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)
```

Initialisieren einer Queue. Eine Queue dient der byte-weisen Übergabe von Daten an einen anderen Prozess nach dem FIFO-Prinzip. Innerhalb dieser Queue können <size> Bytes durch den Kernel-Puffer <*buffer> an andere Prozesse übergeben werden. Der Buffer kann durch direkte Deklaration (U08 Buffer[size]) oder durch dynamische Allokation erzeugt werden. Für die dynamische Allokation in Systemen mit segmentbasierter Speicherverwaltung ist die Typdeklaration OS_HUGE erforderlich, um über Segmentgrenzen hinweg ein Area ansprechen zu können.

Parameters

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

Return Value

OS_NO_ERR	Queue initialisiert

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)
```

Abfrage des Status einer Queue. Durch diese Abfrage lassen sich diverse Parameter ihrer Initialisierung sowie ihr Füllstand ermitteln. Desweiteren kann die Priorität des wartenden Tasks ermittelt werden.

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	kein Fehler (für Erweiterungen)

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)
```

Warten auf ein Byte aus einer Queue. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn keine Bytes vorlagen und mit OS_SUSPEND wird solange gewartet bis ein Byte vorliegt (notfalls endlos).

Parameters

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

Return Value

OS_NO_ERR	Byte aus Queue erhalten
OS_Q_NODATA	kein Byte in Queue (bei OS_NO_SUSP)
OS_TIMEOUT	kein Byte in Queue (nach warten)

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)
```

Bricht das Warten eines empfangenden Tasks an der Queue ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*pq	pointer to Queue

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der 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)
```

Sendet ein Byte in eine Queue. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn die Queue voll war und mit OS_SUSPEND wird solange gewartet bis das Byte eingetragen werden kann (notfalls endlos).

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 gesendet
OS_Q_FULL	Queue voll (bei OS_NO_SUSP)
OS_TIMEOUT	Queue voll (nach warten)

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)
```

Sendet ein Byte an den Anfang einer Queue. Somit wird dieses Byte durch den Empfänger zuerst wieder ausgelesen (vordrängeln). Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn die Queue voll war und mit OS_SUSPEND wird solange gewartet bis das Byte eingetragen werden kann (notfalls endlos).

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 gesendet
OS_Q_FULL	Queue voll (bei OS_NO_SUSP)
OS_TIMEOUT	Queue voll (nach warten)

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)
```

Bricht das Warten eines sendenden Tasks an der Queue ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*pq	pointer to Queue

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der 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)
```

Löscht den Inhalt einer Queue und reaktiviert einen wartenden Sendeprozess. Diese Funktion kann für Fehlerbehandlungen verwendet werden, um den Datentransfer wieder neu zu starten. Die gelöschten Daten gehen dabei verloren.

Parameters

*pq	pointer to Queue

Return Value

OS_NO_ERR	Queue-Inhalt gelöscht

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)`

Initialisieren einer Pipe. Eine Pipe dient der Paket-weisen Übergabe von Daten an einen anderen Prozess nach dem FIFO-Prinzip.
Der Kernel bietet hierfür zwei verschiedene Modi an, die zum Zeitpunkt des Anlegens zu wählen sind.

- **array-modi**:
  Die Pipe arbeitet mit einem fest zugewiesenen Array festgelegter Größe. Innerhalb dieser Pipe können maximal <deep> Pakete mit je maximal <size> Bytes eines Paketes durch den Kernel-Puffer <*buffer> an einen anderen Prozess übergeben werden. Eine Paket kann dabei zwischen 1 Byte und <size> Bytes groß sein, belegt aber immer eine Zeile des Arrays von <size> Bytes.
  Der Buffer kann durch direkte Deklaration (`U08 Buffer[(size+2)*deep]`) oder durch dynamische Allokation erzeugt werden. Für die dynamische Allokation in Systemen mit segmentbasierter Speicherverwaltung ist die Typdeklaration OS_HUGE erforderlich, um über Segmentgrenzen hinweg ein Area ansprechen zu können. Die zusätzliche Länge von 2 Bytes pro Paket wird für die Speicherung der Paketlänge benötigt.

- **dynamic-modi**:
  Die Pipe verwendet den Memory-Manager und allokiert selbstständig den benötigen Speicher für ein Paket (plus 6 Bytes Verwaltung) und gibt diesen nach Lesen des Pakets auch wieder frei. Somit wird immer nur der Speicher belegt, der für ein Paket auch benötigt wird (plus 6 Bytes Pipe-Verkettung + n Bytes Memory-Manager). Dadurch ist diese Pipe nie voll, nur der Speicher des Memory-Managers kann aufgebraucht sein. Je nach Lage und Fragmentierung des Speichers kann aber nicht bestimmt werden wo das Paket genau gespeichert wird. Desweiteren werden die angesammelten Pakete innerhalb der Pipe direkt verkettet und sind somit anfällig für fehlerhafte Speicherzugriffe durch User-Tasks.
  Außerdem sind die Vorgänge 'allocate + copy' und 'copy + free' meistens durch Interrupte und teilweise auch durch Sheduling unterbrechbar. Die Intigrität ist zu jeder Zeit gesichert, jedoch können diese Vorgänge dadurch vereinzelt auffällig lange brauchen. Desweiteren sollte ein sendender Task während '`OS_PipePost(..)`' NIE durch eine ISR oder einen anderen Task mittels '`OS_TaskDestroy(..)`' zerstört werden.

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 initialisiert

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)
```

Abfrage des Status einer Pipe. Durch diese Abfrage lassen sich diverse Parameter ihrer Initialisierung sowie ihr Füllstand ermitteln. Desweiteren kann die Priorität des wartenden Tasks ermittelt werden.

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	kein Fehler (für Erweiterungen)

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)
```

Warte auf ein Daten-Paket aus einer Pipe. Dabei muß der Empfänger-buffer ausreichend groß sein, um das Paket aufnehmen zu können. Da der Empfänger-buffer auch dynamisch allokiert sein kann, ist wiederum die Typdeklaration OS_HUGE erforderlich, um über Segmentgrenzen hinweg ein Area ansprechen zu können. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn kein Paket vorlag und mit OS_SUSPEND wird solange gewartet bis ein Paket vorliegt (notfalls endlos).

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	Paket aus Pipe erhalten
OS_P_NODATA	kein Paket in Pipe (bei OS_NO_SUSP)
OS_TIMEOUT	kein Paket in Pipe (nach warten)

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_PipePendAbort

```
U08 OS_PipePendAbort(OS_P *pp)
```

Bricht das Warten eines empfangenden Tasks an der Pipe ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*pp	pointer to Pipe

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der Pipe

Example

```
OS_P  Pipe1;

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

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

OS_PipePost

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

Sendet ein Paket in eine Pipe. Da der Sender-buffer auch dynamisch allokiert sein kann, ist wiederum die Typdeklaration OS_HUGE erforderlich, um über Segmentgrenzen hinweg ein Area ansprechen zu können. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn die Pipe voll war und mit OS_SUSPEND wird solange gewartet bis das Paket eingetragen werden kann (notfalls endlos).

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	Paket in Pipe gesendet
OS_P_FULL	Pipe voll (bei OS_NO_SUSP) - on array-modi
OS_P_LEN_ERR	Paket zu lang - on array-modi
OS_TIMEOUT	Pipe voll (nach warten) - on array-modi
see `OS_MemAlloc()`	Speicher Fehler - on dynamic-modi

Example

```
OS_P  Pipe1;

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

    .
    .
    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)
```

Sendet ein Paket an den Anfang einer Pipe. Somit wird dieses Paket durch den Empfänger zuerst wieder ausgelesen (vordrängeln). Da der Sender-buffer auch dynamisch allokiert sein kann, ist wiederum die Typdeklaration OS_HUGE erforderlich, um über Segmentgrenzen hinweg ein Area ansprechen zu können. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn die Pipe voll war und mit OS_SUSPEND wird solange gewartet bis das Paket eingetragen werden kann (notfalls endlos).

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	Paket in Pipe gesendet
OS_P_FULL	Pipe voll (bei OS_NO_SUSP) - on array-modi
OS_P_LEN_ERR	Paket zu lang - on array-modi
OS_TIMEOUT	Pipe voll (nach warten) - on array-modi
see `OS_MemAlloc()`	Speicher Fehler - 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)
```

Bricht das Warten eines sendenden Tasks an der Pipe ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*pp	pointer to Pipe

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der 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)
```

Löscht den Inhalt einer Pipe und reaktiviert einen wartenden Sendeprozess. Diese Funktion kann für Fehlerbehandlungen verwendet werden, um den Datentransfer wieder neu zu starten. Die gelöschten Pakete gehen dabei verloren.

Parameters

*pp	pointer to Pipe

Return Value

OS_NO_ERR	Pipe-Inhalt gelöscht

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)
```

Initialisieren des AMP-Pipe Paares. Das AMP-Pipe Paar dient der Paket-weisen Übergabe von Daten an einen anderen Prozessor-Kern nach dem FIFO-Prinzip.

Die maximale Größe eines Paketes, die maximale Zahl der Pakete sowie die Speicher-Lokation ist dabei in der HW-abhängige Adaption 'OS_ICC_xxx.c' festgelegt um beide Prozessor-Kerne mit exakt identischen Werten auszustatten.
Eine Paket kann dabei zwischen 1 Byte und <size> Bytes groß sein, belegt aber immer eine Zeile des Arrays von <size> Bytes.
Die zusätzliche Länge von 2 Bytes pro Paket wird für die Speicherung der realen Paketlänge benötigt.

Parameters

none

Return Value

OS_NO_ERR	AMP-Pipe Paar initialisiert

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)
```

Abfrage des Status des AMP-Pipe Paares. Durch diese Abfrage lassen sich diverse Parameter der Initialisierung sowie ihre Füllstände ermitteln.

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	kein Fehler (für Erweiterungen)

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)
```

Sendet ein Paket in die AMP-Pipe zum anderen Core. Da der Sender-buffer auch dynamisch allokiert sein kann, ist wiederum die Typdeklaration OS_HUGE erforderlich, um über Segmentgrenzen hinweg ein Area ansprechen zu können. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn die AMP-Pipe voll war und mit OS_SUSPEND wird solange gewartet bis das Paket eingetragen werden kann (notfalls endlos).

Parameters

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

Return Value

OS_NO_ERR	Paket in AMP-Pipe gesendet
OS_ICC_FULL	AMP-Pipe voll (bei OS_NO_SUSP)
OS_ICC_LEN_ERR	Paket zu lang
OS_TIMEOUT	AMP-Pipe voll (nach warten)

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)
```

Warte auf ein Paket aus der AMP-Pipe vom anderen Core. Dabei muß der Empfänger-buffer ausreichend groß sein, um das Paket aufnehmen zu können. Da der Empfänger-buffer auch dynamisch allokiert sein kann, ist wiederum die Typdeklaration OS_HUGE erforderlich, um über Segmentgrenzen hinweg ein Area ansprechen zu können. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn kein Paket vorlag und mit OS_SUSPEND wird solange gewartet bis ein Paket vorliegt (notfalls endlos).

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	Paket aus AMP-Pipe erhalten
OS_ICC_NODATA	kein Paket in AMP-Pipe (bei OS_NO_SUSP)
OS_TIMEOUT	kein Paket in AMP-Pipe (nach warten)

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)
```

Löscht den Inhalt der jeweils sendenden AMP-Pipe und reaktiviert einen wartenden Sendeprozess. Diese Funktion kann für Fehlerbehandlungen verwendet werden, um den Datentransfer wieder neu zu starten. Die gelöschten Pakete gehen dabei verloren.

Parameters

none

Return Value

OS_NO_ERR	AMP-Pipe Inhalt gelöscht

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)
```

Initialisieren einer Semaphore. Eine Semaphore dient der Bedingungssynchonisation. Dazu gehören zwei Varianten der Nutzung einer Semaphore.
- binär: z.B. der Syncronisation von Zugriffen auf gemeinsame Recourcen/Variablen
- counting: Warten auf das Eintreten einer Bedingung(Signal) zur Steuerung der Reihenfolge von Prozessen.

Mit einer binären Semaphore kann z.B. eine State-Machine vor gleichzeitigen Zugriffen unterschiedlicher Prozesse (Read/Write) geschützt werden. So werden inkonsistente Zustände oder Daten vermieden. Jedoch kann in einigen Fällen *Priority Inversion* auftreten (siehe Sonder-Dokument), d.h. das ein niedriger Task die Recourcen besetzt hat, ein höherer Task deshalb warten muß und dann z.B. durch einen INT ein mittlerer Task(und seine Erben) die niedrigere Task für eine unbestimmte Zeit unterbricht. In einem solchen Fall wird die Semaphore mit 1 als cnt initialisiert, der Zugriff mittels OS_SemPend angefragt und mittels OS_SemPost wieder freigegeben.

Mit einer counting Semaphore wird das Eintreffen von Ereignissen signalisiert. So kann ein Prozess auf ein Signal warten um die Abarbeitungsreihenfolge einzuhalten. Mittels OS_SemAccept kann dabei auch die Zahl der inzwischen eingetretenen Ereignisse ermittelt werden. In einem solchen Fall wird die Semaphore mit 0 als cnt initialisiert, auf das Ereignis mittels OS_SemPend oder OS_SemAccept gewartet und mittels OS_SemPost das Ereignis gemeldet.

Parameters

*psem	pointer to Semaphore

Return Value

OS_NO_ERR	Semaphore initialisiert

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)
```

Reserviert eine Semaphore und somit den Zugriff auf die geschützte Recource bzw. wartet auf ein Ereignis. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn die Semaphore nicht frei war bzw. kein Ereignis vorlag und mit OS_SUSPEND wird solange gewartet bis die Semaphore reserviert werden konnte bzw. das Ereignis eingetreten ist (notfalls endlos).

Parameters

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

Return Value

OS_NO_ERR	Semaphore reserviert / Ereignis lag vor
OS_SEM_NODATA	Semaphore besetzt / kein Ereignis (bei OS_NO_SUSP)
OS_TIMEOUT	Semaphore besetzt / kein Ereignis (nach warten)

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)
```

Bricht das Warten eines Tasks an der Semaphore ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*psem	pointer to Semaphore

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der 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)
```

Wird verwendet bei Nutzung der Semaphore als Event-Counter. Der Semaphoren-Counter wird dabei nicht beeinflußt. Ist der Counter größer 0 so wird der aktuelle Counterwert zurückgegeben. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn der Semaphoren-Counter 0 ist und mit OS_SUSPEND wird solange gewartet bis das Ereignis einmal aufgetreten ist (notfalls endlos).

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	Ereignis mind. 1 mal eingetreten
OS_SEM_NODATA	Counter gleich 0 (bei OS_NO_SUSP)
OS_TIMEOUT	Counter gleich 0 (nach warten)

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)
```

Gibt eine reservierte Semaphore und somit den Zugriff auf die geschützte Recource wieder frei bzw. signalisiert ein Ereignis.

Parameters

*psem	pointer to Semaphore

Return Value

OS_NO_ERR	Semaphore freigegeben / Ereignis gemeldet
OS_SEM_OVF	Fehler im Semaphoren-Handling (Counter zu groß)

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)
```

Löscht den Counter einer Semaphore. Diese Funktion kann zum Rücksetzten einer Counting-Semaphore oder für Fehlerbehandlungen verwendet werden, um das Semaphoren-Handling wieder neu zu starten. Bei Verwendung als binär Semaphore zur Steuerung von Zugriffen auf eine geschützte Recource muß im Anschluß bei Verwendung zum Fehlerhandling durch OS_SemPost die Semaphore sooft freigegeben werden, wie gleichzeitig Prozesse auf die Recource zugreifen dürfen.

Parameters

*psem	pointer to Semaphore

Return Value

OS_NO_ERR	Semaphoren-cnt gelöscht

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)
```

Anlegen bzw. Initialisieren einer Mutex (Mutual-Exclusion). Eine Mutex dient der Syncronisation von Zugriffen auf gemeinsame Recourcen/Variablen. Mit einer Mutex werden z.B. State-Machines vor gleichzeitigen Zugriffen unterschiedlicher Prozesse (Read/Write) geschützt. Der zweite Prozess muß warten, bis der erste Prozess seinen Zugriff (Read/Write) beendet hat. So werden inkonsistente Zustände oder Daten vermieden. Im Gegensatz zur Nutzung von Semaphoren kann dabei hier nicht der Effekt der *Priority Inversion* direkt auftreten (siehe Sonder-Dokument). Die Priorität der Mutex muß dabei höher sein, als die höchste Priorität der darauf zugreifenden Tasks. Die Mutex wird als nicht laufende Task eingetragen, sodaß unter der selben Priorität kein anderer Task laufen kann.

Parameters

*pmux	pointer to Mutex

Return Value

OS_NO_ERR	Mutex eingetragen und initialisiert

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)
```

Reserviert eine Mutex und somit den Zufriff auf die geschützte Recource. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn die Mutex nicht frei war und mit OS_SUSPEND wird solange gewartet bis die Mutex reserviert werden konnte (notfalls endlos).

Parameters

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

Return Value

OS_NO_ERR	Mutex reserviert
OS_MUX_NOACC	Mutex besetzt (bei OS_NO_SUSP)
OS_TIMEOUT	Mutex besetzt (nach warten)
OS_MUX_ERR	Fehler im 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)
```

Bricht das Warten eines Tasks an der Mutex ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*pmux	pointer to Mutex

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der 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)
```

Gibt eine reservierte Mutex und somit den Zufriff auf die geschützte Recource wieder frei.

Parameters

*pmux	pointer to Mutex

Return Value

OS_NO_ERR	Mutex freigegeben
OS_MUX_ERR	Fehler im 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)
```

Initialisieren einer Event-Gruppe. Eine Eventgruppe besteht aus 32 Einzelevents, die in einem U32 zusammengefaßt einzeln als auch gruppiert verarbeitet werden können. Jedes Event innerhalb der Gruppe kann das Auftreten eines Ereignisses melden, jedoch keine Aussage darüber treffen, wie oft das Ereignis in der Zwischenzeit aufgetreten ist. Um Ereignisse auch zählen zu können, muß eine Semaphore als Counting-Semaphore für jedes einzelne Ereignis verwendet werden.(Semaphore mit 0 initialisieren , bei Auftreten des Ereignisses "OS_SemPost" und beim Warten "OS_SemAccept")

Parameters

*pevg	pointer to Eventgroup

Return Value

OS_NO_ERR	Event-Gruppe initialisiert

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)
```

Meldet das Auftreten eines bzw. mehrerer Events einer Event-Gruppe. Die Bit-Maske wird dabei als OR-Verknüpfung der Events interpretiert. D.h. alle Events, die in der Bit-Maske gesetzt sind, werden gemeldet. Mode dient der Nutzung dieser Funktion zum Löschen von 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) gemeldet

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)
```

Warten auf ein bzw. mehrere Events einer Event-Gruppe. Die Bit-Maske wird dabei zusammen mit dem Modi als Verknüpfung der Events interpretiert. D.h. bei OS_EVG_OR reicht bereits ein Event, das in der Bit-Maske gesetzt ist, für die Funktion und bei OS_EVG_AND müssen alle Events, die in der Bit-Maske gesetzt sind, eingetroffen sein. Für einen Spezialfall können Events einer Evengruppe auch mehrere Tasks auf einmal aufwecken. Dafür unterscheidet die Eventgruppe beim EvgPend und EvgPost zwischen "OS_EVG_OR_C / OS_EVG_AND_C" für die normale Verwendung (konsumiere Event) oder eben "OS_EVG_OR / OS_EVG_AND" zum Aufwecken mehrerer Tasks. Die Events sind dann aber wieder manuell durch einen Task zu löschen. Nach der Rückkehr enthält die Bit-Maske die aufgetretenen Events, die die Rückkehr ausgelöst haben. Mit OS_NO_SUSP wird sofort zurückgekehrt auch wenn kein Event aufgetreten ist und mit OS_SUSPEND wird solange gewartet bis das/ein/die Event(s) aufgetreten ist(sind) (notfalls endlos).

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) aufgetreten
OS_EVG_NOE	Event(s) nicht aufgetreten (bei OS_NO_SUSP)
OS_TIMEOUT	Event(s) nicht aufgetreten (nach warten)

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)
```

Bricht Warten eines Tasks (höchste wartende Prio) an einer Eventgruppe ab.
Dabei wird nur der wartende Task mit der höchsten Priorität quasi "vorzeitig" ins TimeOut geschickt.

Parameters

*pevg	pointer to Eventgroup

Return Value

OS_NO_ERR	Warten eines Tasks abgebrochen
OS_TASK_NOT_EXIST	kein Task wartet an der Eventgruppe

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)
```

Anlegen bzw. Initialisieren eines Timers.
Ein Timer kann z.B. zu nachfolgenden Zwecken verwendet werden:
• timeouts innerhalb von Protokollschichten und Applikationen wie TCP/IP, X25, HTTP, FTP, ...
• verhindern des "verhungern" von Tasks durch Definition eines timeouts und entsprechenden Maßnahmen wie Prioritätsanhebung oder Anderes
• Steuerung periodischer Dienste oder Services
• soft-deadline / Watchdog von Diensten oder Services

Als *mode* können nachfolgende Angaben gemacht werden:
• OS_TMR_ENABLE - startet den Timer sofort
• OS_TMR_RONCE - Timer ist von Typ "run-once", d.h. nach einmaligem timeout wird er automatisch deaktiviert, bleibt aber eingetragen
• OS_TMR_CYCL - Timer ist vom Typ "cyclic / periodic", d.h. der Timer wird nach jedem timeout automatisch neu gestartet
• OS_TMR_CLR - Timer ist vom Typ "run-once auto-erase", d.h. wie run-once nur wird dieser Timer automatisch gelöscht und muß für einen weiteren Einsatz neu angelegt werden

Dabei gilt die Regelung OS_TMR_CLR geht vor OS_TMR_CYCL und dies vor OS_TMR_RONCE.

Die eingetragene CallBack-Funktion sollte möglichst kurz sein. Zur Informationsweitergabe kann ein Argument verwendet werden.

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 eingetragen und initialisiert
OS_TMR_NO_TIME	keine gültige Zeit als Parameter (time == 0)
OS_TMR_EXIST	Timer war bereits eingetragen und initialisiert

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)
```

Deaktiviert und löscht einen Timer.

Parameters

*ptmr	pointer to Timer

Return Value

OS_NO_ERR	timer gelöscht
OS_TMR_NOT_EXIST	der Timer war nicht eingetragen

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)
```

Startet einen deaktivierten, restarted einen run-once oder restartet einen laufenden 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	der Timer ist nicht eingetragen

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)
```

Stoppt / deaktiviert einen laufenden Timer.

Parameters

*ptmr	pointer to Timer

Return Value

OS_NO_ERR	Timer gestoppt
OS_TMR_NOT_EXIST	der Timer ist nicht eingetragen

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)
```

Gibt den Status eines angelegten Timers zurück.
Dabei werden folgende Informationen bereitgestellt:
• OS_TMR_ENABLE - der Timer läft aktuell
• OS_TMR_RONCE - Timer ist von Typ "run-once", d.h. nach einmaligem timeout wird er automatisch deaktiviert, bleibt aber eingetragen
• OS_TMR_CYCL - Timer ist vom Typ "cyclic / periodic", d.h. der Timer wird nach jedem timeout automatisch neu gestartet
• OS_TMR_CLR - Timer ist vom Typ "run-once auto-erase", d.h. wie run-once nur wird dieser Timer automatisch gelöscht und muß für einen weiteren Einsatz neu angelegt werden

Wenn der zurückgegebene Status nicht OS_TMR_ENABLE ist und OS_TMR_CLR, so wurde dieser Timer nie angelegt oder war "run-once auto-erase" und die Zeit war abgelaufen.

Parameters

*ptmr	pointer to Timer

Return Value

status	Status des Timers (siehe "modi" bei 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)
```

Gibt die verbleibende Zeit (in Timer-Ticks) eines laufenden Timers zurück.
Ist die Zeit gleich 0, so war der Timer abgelaufen oder wurde nie per OS_TimerCreate angelegt.

Parameters

*ptmr	pointer to Timer

Return Value

time	verbleibende Zeit 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)
```

Setzt den Kernel-internen Tick-Counter auf übergebenen Wert.

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)
```

Gibt den aktuellen Kernel-internen Tick-Counter-Wert zurück.

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)
```

Registriert eine Interrupt-Level. Dadurch werden keine Taskwechsel generiert. Diese Funktion ist für C-Code ISRs notwendig auf Prozessoren, bei denen Interrupte durch andere Interrupte unterbrochen werden können.

Parameters

none

Return Value

none

Example

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

OS_IntExit

```
void OS_IntExit(void)
```

Unregistriert eine Interrupt-Level. Dadurch werden Taskwechsel wieder generiert. Diese Funktion ist für C-Code ISRs notwendig auf Prozessoren, bei denen Interrupte durch andere Interrupte unterbrochen werden können.

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)
```

Trägt einen Eintrag in die History-Tabelle des Kernels ein. Zusätzlich zu den beiden Parametern werden noch die Priorität des Tasks und der Tick-Counter (als Zeitstempel) eingetragen.

Parameters

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

Return Value

OS_NO_ERR	Eintrag geschrieben

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)
```

Liest nächsten Eintrag aus der History-Tabelle des Kernels und löscht diesen dabei.

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	Eintrag gelesen
OS_HIS_END	kein Eintrag vorhanden

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