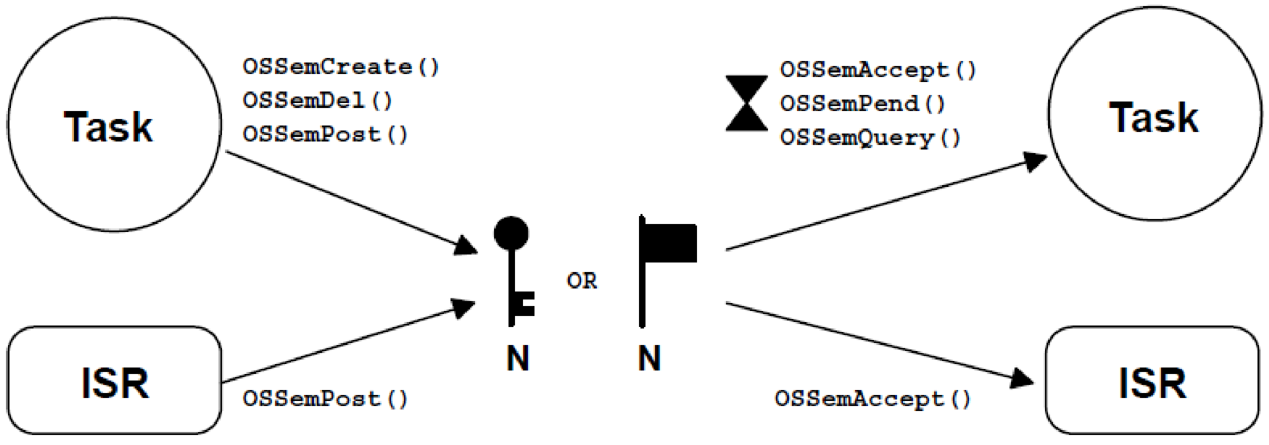
~~~ Semaphore Management In uC/OS-II~~~

In the MicroC/OS-II real-time operating system, semaphores are used to solve synchronization and mutual exclusion problems. A semaphore is essentially a counter that is used to provide access to a shared resource. It can be thought of as tickets to access a resource. If the counter is positive, a task can decrement the counter and continue execution. If it's zero, the task must wait.



The following functions are used in MicroC/OS-II for Semaphore management.

OSSemAccept() - Checks the state of the semaphore without blocking the calling task. If the semaphore count is zero, the function returns immediately with a zero. If the count is greater than zero, the function decrements the count and returns the new count.

OSSemCreate() - Creates a semaphore with an initial count. The count can be zero or a positive number. It returns a pointer to the created semaphore.

OSSemDel() - Deletes a semaphore. It frees up any resources used by the semaphore and unblocks any tasks that are waiting on the semaphore.

OSSemPend() - Is used by a task to wait for a semaphore. The task will be blocked if the semaphore count is zero. If multiple tasks are waiting on the semaphore, they will be unblocked based on their priority.

OSSemPendAbort() - Aborts the wait on the semaphore for the highest priority task that is waiting. If the task was waiting with a timeout, the timeout is cleared. It can be called from an ISR or a task.

OSSemPost() - Signals a semaphore. It increments the semaphore count and unblocks the highest priority task waiting on the semaphore, if any.

OSSemQuery() - Allows you to find out the state of a semaphore. It fills in a data structure with information about the semaphore, including the semaphore count and the list of tasks waiting on the semaphore.

OSSemSet() - Sets the semaphore count to a specified value. It can be used for error recovery purposes. It's not recommended to use this function in normal system operation as it can lead to unpredictable results.

Semaphore data, data structure:

The **OS_SEM_DATA** structure provides a way for the OS to keep track of the state of a semaphore and manage task synchronization.

```
typedef struct os_sem_data {
   INT16U  OSCnt;                        /* Semaphore count                               */
   OS_PRIO OSEventTbl[OS_EVENT_TBL_SIZE];  /* List of tasks waiting for event to occur              */
   OS_PRIO OSEventGrp;                  /* Group corresponding to tasks waiting for event to occur */
} OS_SEM_DATA;
```

OSCnt: This variable represents the semaphore count. It's of type **INT16U** (unsigned 16-bit integer). This count is initially set when the semaphore is created via **OSSemCreate**(). Every time a task performs a **OSSemPend**() operation, this count is decremented. When a **OSSemPost**() operation is performed, the count is incremented. If the count is zero, tasks calling **OSSemPend**() will be blocked until another task or an interrupt service routine (ISR) performs a **OSSemPost**() operation.

OSEventTbl: This is an array of **OS_PRIO** (priority values) which represents the list of tasks waiting for the semaphore. The size of the array is defined by **OS_EVENT_TBL_SIZE**. Each entry corresponds to a task that is waiting for the semaphore to be signaled. If a task performs an **OSSemPend**() operation and the semaphore count is zero, the task's priority is added to this list.

OSEventGrp: This variable is of type **OS_PRIO** and represents a group corresponding to the tasks waiting for the semaphore to be signaled. It's essentially a bitmap where each bit corresponds to a task priority level. This makes it quick to find the highest priority task that is waiting on the semaphore.

OSSemAccept():

```
INT16U      OSSemAccept          (OS_EVENT      *pevent);
```

OSSemAccept is used to check the state of a semaphore without blocking the calling task. It checks if an event occurred or if resource is available. It takes in the argument pevent that is a pointer to event control block and returns a value greater than 0 if the resource is available or if the event did not occur the semaphore is decremented.

```
INT16U  OSSemAccept (OS_EVENT *pevent)
{
    INT16U     cnt;
#if OS_CRITICAL_METHOD == 3u                          /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr = 0u;
#endif



#if OS_ARG_CHK_EN > 0u
    if (pevent == (OS_EVENT *)0) {                    /* Validate 'pevent'                             */
        return (0u);
    }
#endif
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {   /* Validate event block type                     */
        return (0u);
    }
    OS_ENTER_CRITICAL();
    cnt = pevent->OSEventCnt;
    if (cnt > 0u) {                                   /* See if resource is available                  */
        pevent->OSEventCnt--;                         /* Yes, decrement semaphore and notify caller    */
    }
    OS_EXIT_CRITICAL();
    return (cnt);                                     /* Return semaphore count                        */
}
```

The function first checks if the **pevent** pointer is **NULL**. If it is, the function returns **0u**, indicating an error. This is a basic form of argument checking to prevent null pointer dereferencing.

Next, the function checks the type of the event to ensure it's a semaphore. If it's not, the function also returns **0u**.

The function then enters a critical section to prevent race conditions. It checks the count of the semaphore. If the count is greater than **0u**, it means the semaphore is available, and the function decrements the semaphore count. If the count is **0u**, it means the semaphore is not available, and the function does nothing.

Finally, the function exits the critical section and returns the original count of the semaphore. If the count was greater than **0u**, this means the semaphore was available. If the count was **0u**, this means the semaphore was not available.

This function is typically used when a task wants to check a semaphore but doesn't want to block if the semaphore isn't available. It's a non-blocking alternative to **OSSemPend**(), which does block the calling task if the semaphore isn't available.

OSSemCreate():

```
OS_EVENT    *OSSemCreate          (INT16U         cnt);
```

This function is called to create a semaphore. It take an initial value for the semaphore as a parameter. If the value is 0, no resource is available or no event has occurred. The semaphore is initialized by the used to a non zero value to specify how many resources are available.

```
OS_EVENT  *OSSemCreate (INT16U cnt)
{
    OS_EVENT  *pevent;
#if OS_CRITICAL_METHOD == 3u                           /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr = 0u;
#endif


#ifdef OS_SAFETY_CRITICAL_IEC61508
    if (OSSafetyCriticalStartFlag == OS_TRUE) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return ((OS_EVENT *)0);
    }
#endif

    if (OSIntNesting > 0u) {                           /* See if called from ISR ...               */
        return ((OS_EVENT *)0);                        /* ... can't CREATE from an ISR             */
    }
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                          /* Get next free event control block        */
    if (OSEventFreeList != (OS_EVENT *)0) {            /* See if pool of free ECB pool was empty   */
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                     /* Get an event control block               */
        pevent->OSEventType    = OS_EVENT_TYPE_SEM;
        pevent->OSEventCnt     = cnt;                  /* Set semaphore value                      */
        pevent->OSEventPtr     = (void *)0;            /* Unlink from ECB free list                */
#if OS_EVENT_NAME_EN > 0u
        pevent->OSEventName    = (INT8U *)(void *)"?";
#endif
        OS_EventWaitListInit(pevent);                  /* Initialize to 'nobody waiting' on sem.   */

        OS_TRACE_SEM_CREATE(pevent, pevent->OSEventName);
    }
    return (pevent);
}
```

The function first checks if it's being called from an interrupt service routine (ISR) by checking the value of **OSIntNesting**. If it is being called from an ISR, the function returns **NULL**, because a semaphore cannot be created from an ISR.

The function then enters a critical section to prevent race conditions. It retrieves a free ECB from the **OSEventFreeList** pool. If there are no free ECBs, the function will return **NULL**.

```
pevent = OSEventFreeList;
```

The function then initializes the **ECB**, **pevent = OSEventFreeList** , to represent a semaphore. It sets the event type to semaphore, sets the semaphore count to the value passed as an argument, and initializes the event pointer to **NULL**.

```
pevent->OSEventType    = OS_EVENT_TYPE_SEM; // ECB is now a semaphore
pevent->OSEventCnt     = cnt;               // to the initial count of the semaphore
pevent->OSEventPtr     = (void *)0;         // to unlink the ECB from the list of free ECBs
```

The function also initializes the event's wait list, which is a list of tasks waiting on the semaphore. Initially, this list is empty, because no tasks are waiting on a newly created semaphore.

```
OS_EventWaitListInit(pevent);
```

Finally, the function exits the critical section and returns a pointer to the newly created semaphore.

This function is typically called when a task wants to create a semaphore. The task must provide the initial count of the semaphore, which is typically the number of resources controlled by the semaphore or 1 for a binary semaphore.

OSSemDel():

```
OS_EVENT    *OSSemDel   (OS_EVENT       *pevent,  INT8U          opt,   INT8U          *perr);
```

This function deletes a semaphore and readies all tasks pending on the semaphore.

It takes in 3 arguments, '**pevent**' which is a pointer to the event control block associated with the desired semaphore, '**opt**' which are delete options:

OS_DEL_NO_PEND - Deletes a semaphore only if no tasks are pending.
OS_DEK_ALWAYS – Deletes the semaphore even if tasks are waiting. The pending tasks would be readied.

'**perr**' is a pointer to an error code that has one of the following values.

OS_ERR_NONE - The call was successful and the semaphore was deleted.
OS_ERR_DEL_ISR   - If you attempted to delete the semaphore from an ISR.
OS_ERR_ILLEGAL_DEL_RUN_TIME - If you tried to delete the semaphore after safety critical operation started.
OS_ERR_INVALID_OPT - An invalid option was specified.
OS_ERR_TASK_WAITING - One or more tasks were waiting on the semaphore.
OS_ERR_EVENT_TYPE - If you didn't pass a pointer to a semaphore.
OS_ERR_PEVENT_NULL - If **'pevent'** is a NULL pointer.

```c
OS_EVENT  *OSSemDel (OS_EVENT  *pevent,
                     INT8U      opt,
                     INT8U     *perr)
{
    BOOLEAN    tasks_waiting;
    OS_EVENT  *pevent_return;
#if OS_CRITICAL_METHOD == 3u                               /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr = 0u;
#endif


#ifdef OS_SAFETY_CRITICAL
    if (perr == (INT8U *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return ((OS_EVENT *)0);
    }
#endif

#ifdef OS_SAFETY_CRITICAL_IEC61508
    if (OSSafetyCriticalStartFlag == OS_TRUE) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        *perr = OS_ERR_ILLEGAL_DEL_RUN_TIME;
        return ((OS_EVENT *)0);
    }
#endif

#if OS_ARG_CHK_EN > 0u
    if (pevent == (OS_EVENT *)0) {                         /* Validate 'pevent'                        */
        *perr = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
#endif

    OS_TRACE_SEM_DEL_ENTER(pevent, opt);

    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {        /* Validate event block type                */
        *perr = OS_ERR_EVENT_TYPE;
        OS_TRACE_SEM_DEL_EXIT(*perr);
        return (pevent);
    }
    if (OSIntNesting > 0u) {                               /* See if called from ISR ...               */
        *perr = OS_ERR_DEL_ISR;                            /* ... can't DELETE from an ISR             */
        OS_TRACE_SEM_DEL_EXIT(*perr);
        return (pevent);
    }
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0u) {                        /* See if any tasks waiting on semaphore    */
        tasks_waiting = OS_TRUE;                           /* Yes                                      */
    } else {
        tasks_waiting = OS_FALSE;                          /* No                                       */
    }
    switch (opt) {
        case OS_DEL_NO_PEND:                                    /* Delete semaphore only if no task waiting */
             if (tasks_waiting == OS_FALSE) {
#if OS_EVENT_NAME_EN > 0u
                 pevent->OSEventName    = (INT8U *)(void *)"?";
#endif
                 pevent->OSEventType    = OS_EVENT_TYPE_UNUSED;
                 pevent->OSEventPtr     = OSEventFreeList; /* Return Event Control Block to free list  */
                 pevent->OSEventCnt     = 0u;
                 OSEventFreeList        = pevent;          /* Get next free event control block        */
                 OS_EXIT_CRITICAL();
                 *perr                  = OS_ERR_NONE;
                 pevent_return          = (OS_EVENT *)0;   /* Semaphore has been deleted               */
             } else {
                 OS_EXIT_CRITICAL();
                 *perr                  = OS_ERR_TASK_WAITING;
                 pevent_return          = pevent;
             }
             break;

        case OS_DEL_ALWAYS:                                     /* Always delete the semaphore              */
             while (pevent->OSEventGrp != 0u) {            /* Ready ALL tasks waiting for semaphore    */
                 (void)OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM, OS_STAT_PEND_ABORT);
             }
#if OS_EVENT_NAME_EN > 0u
             pevent->OSEventName    = (INT8U *)(void *)"?";
#endif
             pevent->OSEventType    = OS_EVENT_TYPE_UNUSED;
             pevent->OSEventPtr     = OSEventFreeList;     /* Return Event Control Block to free list  */
             pevent->OSEventCnt     = 0u;
             OSEventFreeList        = pevent;              /* Get next free event control block        */
             OS_EXIT_CRITICAL();
             if (tasks_waiting == OS_TRUE) {                 /* Reschedule only if task(s) were waiting  */
                 OS_Sched();                               /* Find highest priority task ready to run  */
             }
             *perr                  = OS_ERR_NONE;
             pevent_return          = (OS_EVENT *)0;       /* Semaphore has been deleted               */
             break;

        default:
             OS_EXIT_CRITICAL();
             *perr                  = OS_ERR_INVALID_OPT;
             pevent_return          = pevent;
             break;
    }

    OS_TRACE_SEM_DEL_EXIT(*perr);
    return (pevent_return); }
```

The safety critical checks are performed first before performing deletion operation on any of the semaphore.

```
if (pevent == (OS_EVENT *)0) {                          /* Validate 'pevent'                            */
        *perr = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
```

Checks to ensure that the pointer to ECS is not **NULL**. If **pevent** is **NULL**, any attempt to access it would lead to undefined behviour, hence this check.

```
if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {         /* Validate event block type                    */
        *perr = OS_ERR_EVENT_TYPE;
        OS_TRACE_SEM_DEL_EXIT(*perr);
        return (pevent);
    }
```

This check ensures that the event type of ECB is a semaphore. This check is needed as this function is designed to only delete semaphores.

```
if (OSIntNesting > 0u) {                                /* See if called from ISR ...                   */
        *perr = OS_ERR_DEL_ISR;                             /* ... can't DELETE from an ISR             */
        OS_TRACE_SEM_DEL_EXIT(*perr);
        return (pevent);
    }
```

This check ensures that the function is not being called from an ISR. This check is required because any OS resources, including semaphores, cannot be deleted from within an ISR.

```
if (pevent->OSEventGrp != 0u) {                         /* See if any tasks waiting on semaphore    */
        tasks_waiting = OS_TRUE;                            /* Yes                                      */
    } else {
        tasks_waiting = OS_FALSE;                           /* No                                       */
    }
```

This check determines whether any tasks are waiting on the semaphore. This information is used to decide whether to delete the semaphore and how to handle tasks that we waiting on it.

Once all the checks are performed, the function continues execution onto switch cases. Based on the given options the program flow is as follows:

OS_DEL_NO_PEND: This case represents the scenario where the semaphore should be deleted only if no tasks are waiting on it. If **tasks_waiting == OS_FALSE**, which means no tasks are waiting on the semaphore, it proceeds with the deletion. The event name (if enabled) is set to **"?",** the event type is marked as unused, the event's pointer is added to the free list, and the semaphore count is set to zero. The semaphore is then added to the head of the free list. The function then exits the critical section, sets the error code to **OS_ERR_NONE** (indicating no error), and sets the return pointer to null (indicating that the semaphore has been deleted). If there are tasks waiting on the semaphore (**tasks_waiting == OS_TRUE**), the function just exits the critical section, sets the error code to **OS_ERR_TASK_WAITING**, and returns the original semaphore.

OS_DEL_ALWAYS: This case represents the scenario where the semaphore should always be deleted, regardless of whether any tasks are waiting on it. If there are tasks waiting on the semaphore, it uses a **while** loop to make all waiting tasks ready. The event name (if enabled) is set to **"?",** the event type is marked as unused, the event's pointer is added to the free list, and the semaphore count is set to zero. The semaphore is then added to the head of the free list. The function then exits the critical section. If there were tasks waiting on the semaphore, it calls **OS_Sched()** to trigger a context switch to the highest priority task that is ready to run. It then sets the error code to **OS_ERR_NONE** (indicating no error), and sets the return pointer to null (indicating that the semaphore has been deleted).

default: This case handles invalid options. If an invalid option is provided, the function just exits the critical section, sets the error code to **OS_ERR_INVALID_OPT**, and returns the original semaphore.

The function finally returns a pointer to the deleted semaphore (or the original semaphore in case of an error). If the semaphore was successfully deleted, it returns a null pointer.

This function is used when a task wants to delete a semaphore. The task must provide a pointer to the semaphore's ECB and choose an appropriate deletion option. If the function returns a null pointer and ***perr == OS_ERR_NONE**, it indicates that the semaphore was successfully deleted. If not, the task must check ***perr** to find out what went wrong.

When deleting a semaphore with **OS_DEL_ALWAYS**, the function also reschedules tasks if any were waiting on the semaphore. This is done using the **OS_Sched()** function, which triggers a context switch to the highest priority task that is ready to run. This is necessary because the state of the tasks may have changed after the semaphore was deleted.

OSSemPend():

```
void        OSSemPend           (OS_EVENT       *pevent, INT32U         timeout,  INT8U          *perr);
```

This function waits for a semaphore. It takes in 3 arguments, '**pevent**' a pointer to an event control block associated with the desired semaphore. '**timeout**' is counted in clock ticks. If non-zero, your task will wait for the resource up to the amount of time specified by timeout argument. If it is specified with 0, the task will wait forever at the specified semaphore or until the resource becomes available. '**perr**' is a pointer to where an error message will be deposited. The possible error message for this function is as follow:
OS_ERR_NONE - The call was successful and your task owns the resource or, the event you are waiting for occurred.
OS_ERR_TIMEOUT - The semaphore was not received within the specified 'timeout'.
OS_ERR_PEND_ABORT - The wait on the semaphore was aborted.
OS_ERR_EVENT_TYPE - If you didn't pass a pointer to a semaphore.
OS_ERR_PEND_ISR - If you called this function from an ISR and the result would lead to a suspension.
OS_ERR_PEVENT_NULL - If 'pevent' is a NULL pointer.
OS_ERR_PEND_LOCKED - If you called this function when the scheduler is locked

```
void  OSSemPend (OS_EVENT  *pevent,
                 INT32U     timeout,
                 INT8U     *perr)
{
#if OS_CRITICAL_METHOD == 3u                          /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr = 0u;
#endif


#ifdef OS_SAFETY_CRITICAL
```

```
    if (perr == (INT8U *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif
#if OS_ARG_CHK_EN > 0u
    if (pevent == (OS_EVENT *)0) {                    /* Validate 'pevent'                             */
        *perr = OS_ERR_PEVENT_NULL;
        return;
    }
#endif

    OS_TRACE_SEM_PEND_ENTER(pevent, timeout);

    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {   /* Validate event block type                     */
        *perr = OS_ERR_EVENT_TYPE;
        OS_TRACE_SEM_PEND_EXIT(*perr);
        return;
    }
    if (OSIntNesting > 0u) {                          /* See if called from ISR ...                    */
        *perr = OS_ERR_PEND_ISR;                      /* ... can't PEND from an ISR                    */
        OS_TRACE_SEM_PEND_EXIT(*perr);
        return;
    }
    if (OSLockNesting > 0u) {                         /* See if called with scheduler locked ...       */
        *perr = OS_ERR_PEND_LOCKED;                   /* ... can't PEND when locked                    */
        OS_TRACE_SEM_PEND_EXIT(*perr);
        return;
    }
    OS_ENTER_CRITICAL();
    if (pevent->OSEventCnt > 0u) {                    /* If sem. is positive, resource available ...   */
        pevent->OSEventCnt--;                         /* ... decrement semaphore only if positive.     */
        OS_EXIT_CRITICAL();
        *perr = OS_ERR_NONE;
        OS_TRACE_SEM_PEND_EXIT(*perr);
        return;
    }
                                                      /* Otherwise, must wait until event occurs       */
    OSTCBCur->OSTCBStat     |= OS_STAT_SEM;           /* Resource not available, pend on semaphore     */
    OSTCBCur->OSTCBStatPend  = OS_STAT_PEND_OK;
    OSTCBCur->OSTCBDly       = timeout;               /* Store pend timeout in TCB                     */
    OS_EventTaskWait(pevent);                         /* Suspend task until event or timeout occurs    */
    OS_EXIT_CRITICAL();
    OS_Sched();                                       /* Find next highest priority task ready         */
    OS_ENTER_CRITICAL();
    switch (OSTCBCur->OSTCBStatPend) {                /* See if we timed-out or aborted                */
        case OS_STAT_PEND_OK:
             *perr = OS_ERR_NONE;
             break;
        case OS_STAT_PEND_ABORT:
             *perr = OS_ERR_PEND_ABORT;                /* Indicate that we aborted                     */
             break;

        case OS_STAT_PEND_TO:
        default:
             OS_EventTaskRemove(OSTCBCur, pevent);
             *perr = OS_ERR_TIMEOUT;                   /* Indicate that we didn't get event within TO   */
             break;
    }
    OSTCBCur->OSTCBStat          =  OS_STAT_RDY;      /* Set   task  status to ready                   */
    OSTCBCur->OSTCBStatPend      =  OS_STAT_PEND_OK;  /* Clear pend  status                            */
    OSTCBCur->OSTCBEventPtr      = (OS_EVENT  *)0;    /* Clear event pointers                          */
#if (OS_EVENT_MULTI_EN > 0u)
    OSTCBCur->OSTCBEventMultiPtr = (OS_EVENT **)0;
#endif
    OS_EXIT_CRITICAL();
    OS_TRACE_SEM_PEND_EXIT(*perr);
}
```

Before waiting on a semaphore, few checks are performed.

Null pointer check:

```
    if (pevent == (OS_EVENT *)0) {                     /* Validate
'pevent'                              */
        *perr = OS_ERR_PEVENT_NULL;
        return;
    }
```

This is a null pointer check. If argument checking (**OS_ARG_CHK_EN**) is enabled, this section checks whether the **pevent** pointer is null. If it is, the error pointer **perr** is assigned **OS_ERR_PEVENT_NULL** and the function returns immediately. This is important because the rest of the function assumes **pevent** is a valid pointer. If it's null, dereferencing it would lead to a segmentation fault or similar error.

Event Type Check:

```
if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {   /* Validate event block type                     */
        *perr = OS_ERR_EVENT_TYPE;
        OS_TRACE_SEM_PEND_EXIT(*perr);
        return;
    }
```

This block validates the event type of **pevent**. It checks if the event type of the passed event control block is a semaphore. If it's not, the function is dealing with the wrong type of event, which is an error. In that case, the error pointer **perr** is assigned **OS_ERR_EVENT_TYPE** and the function returns immediately.

ISR Check:

```
if (OSIntNesting > 0u) {                          /* See if called from ISR ...                    */
        *perr = OS_ERR_PEND_ISR;                      /* ... can't PEND from an ISR                    */
        OS_TRACE_SEM_PEND_EXIT(*perr);
        return;
    }
```

This block checks if the function was called from an interrupt service routine (ISR). The **OSIntNesting** variable keeps track of the nesting level of ISR calls. If **OSIntNesting** is greater than zero, this means that the function was called from an ISR. In this case, the error pointer **perr** is assigned **OS_ERR_PEND_ISR**, as pending a semaphore from an ISR is not allowed, and the function returns immediately.

Scheduler Lock Check:

```
    if (OSLockNesting > 0u) {                         /* See if called with scheduler locked ...       */
        *perr = OS_ERR_PEND_LOCKED;                   /* ... can't PEND when locked                    */
        OS_TRACE_SEM_PEND_EXIT(*perr);
        return;
    }
```

This block checks if the scheduler is currently locked. The **OSLockNesting** variable keeps track of the nesting level of scheduler lock calls. If **OSLockNesting** is greater than zero, this means that the scheduler is currently locked. In this case, the error pointer **perr** is assigned **OS_ERR_PEND_LOCKED**, as pending a semaphore when the scheduler is locked is not allowed, and the function returns immediately.

Semaphore Check:

```
if (pevent->OSEventCnt > 0u) {                    /* If sem. is positive, resource available ...   */
        pevent->OSEventCnt--;                         /* ... decrement semaphore only if positive.     */
        OS_EXIT_CRITICAL();
        *perr = OS_ERR_NONE;
        OS_TRACE_SEM_PEND_EXIT(*perr);
        return;
    }
```

This block enters a critical section to protect the semaphore from being modified by other tasks or ISRs during the check. It checks if the semaphore count (**pevent->OSEventCnt**) is greater than zero. If it is, this means that the semaphore is available. In this case, the semaphore count is decremented (since it is now being used), the function exits the critical section, the error pointer **perr** is assigned **OS_ERR_NONE**, and the function returns immediately. If the semaphore count is not greater than zero, the function continues to the next section where it will put the task to sleep until the semaphore becomes available or a timeout occurs.

```
    OSTCBCur->OSTCBStat     |= OS_STAT_SEM;           /* Resource not available, pend on semaphore     */
    OSTCBCur->OSTCBStatPend  = OS_STAT_PEND_OK;
    OSTCBCur->OSTCBDly       = timeout;               /* Store pend timeout in TCB                     */
    OS_EventTaskWait(pevent);                         /* Suspend task until event or timeout occurs    */
    OS_EXIT_CRITICAL();
    OS_Sched();                                       /* Find next highest priority task ready         */
    OS_ENTER_CRITICAL();
```

The above block of code manages the task’s transition into and out of a waiting state when a semaphore is not readily available. It starts by setting the current task's status to indicate that it is waiting for a semaphore. It also sets the task's pending status to signify that the task is actively waiting for an event, in this case, a semaphore, and that no errors have been encountered. To implement the timeout feature for waiting on a semaphore, the delay field of the current task is set to the specified timeout. If the semaphore is not available within this period, the task will stop waiting and the **OSSemPend** function will return with a timeout error. To manage the task suspension and resumption processes effectively, the current task is suspended to wait for the semaphore or timeout, after which the critical section is exited to allow the scheduler to switch context to another task. The scheduler then selects and switches context to the next highest priority task that is ready to run. Once the scheduler has switched the context, a critical section is re-entered in preparation for the next phase of operations when the task is resumed.

```
switch (OSTCBCur->OSTCBStatPend) {                  /* See if we timed-out or aborted                */
        case OS_STAT_PEND_OK:
             *perr = OS_ERR_NONE;
             break;

        case OS_STAT_PEND_ABORT:
             *perr = OS_ERR_PEND_ABORT;                 /* Indicate that we aborted                      */
             break;

        case OS_STAT_PEND_TO:
        default:
             OS_EventTaskRemove(OSTCBCur, pevent);
             *perr = OS_ERR_TIMEOUT;                    /* Indicate that we didn't get event within TO   */
             break;
    }
```

The above section handles the possible outcomes after the task’s attempt to obtain a semaphore.

A **switch** statement checks the current task's pending status (**OSTCBCur->OSTCBStatPend**) to determine why the task was resumed from its waiting state. If the status is **OS_STAT_PEND_OK**, it means the semaphore was successfully obtained before the timeout and no error occurred, so the error code ***perr** is set to **OS_ERR_NONE**. If the status is **OS_STAT_PEND_ABORT**, the wait operation was aborted, possibly due to the semaphore being deleted while the task was still waiting or the task being explicitly readied by another task or ISR. In this case, ***perr** is set to **OS_ERR_PEND_ABORT**. If the status is **OS_STAT_PEND_TO** or any other value, it means the task did not obtain the semaphore and it timed out. The task is then removed from the event's waiting list and the error code **OS_ERR_TIMEOUT** is set in ***perr**.

Once that’s done, the code resets the state of the current task after it has attempted to acquire a semaphore leaving it in a ready state where it can either begin execution if it’s the highest priority tasks or wait to be scheduled for execution in the following operations.

OSTCBCur->OSTCBStat = OS_STAT_RDY; sets the task's status to **'ready'**. This doesn't necessarily mean the task will immediately begin execution; it simply indicates that the task is ready to run and can be scheduled for execution.

Next, **OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK**; clears the task's **pending** status. This indicates that the task is no longer waiting for a semaphore or any other event to occur.

OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0; clears the event pointer associated with the task. This pointer is used to link the task to the event control block of the semaphore it was waiting for, and clearing it means the task is no longer associated with that semaphore.

The line **OSTCBCur->OSTCBEventMultiPtr = (OS_EVENT **)0**; clears the task's pointer to an array of event control blocks, if the feature for waiting for multiple events is enabled (**OS_EVENT_MULTI_EN > 0u**). This indicates that the task is not waiting for any events.

Finally, **OS_EXIT_CRITICAL()**; marks the end of a critical section, allowing other tasks or interrupt routines to run. This is paired with an **OS_ENTER_CRITICAL()**; call from earlier in the function.

OSSemPendAbort():

```
INT8U       OSSemPendAbort       (OS_EVENT       *pevent, INT8U          opt, INT8U          *perr);
```

This function aborts and readies any task currently waiting on a semaphore. This function should be used to fault-abort the wait on the semaphore rather than to normally signal the semaphore via **OSSemPost().**

It takes in 3 arguments, ‘**pevent**’ a pointer to an ECB associated with the desired semaphore. ‘**opt**’ that would determine the type of abort performed.
OS_PEND_OPT_NONE - ABORT waits for a single task (HPT) waiting on the semaphore.
OS_PEND_OPT_BROADCAST - ABORT waits for ALL tasks that are waiting on the semaphore.

‘**perr**’ that is a pointer to where an error message will be deposited. The following are the error messages for this function:

OS_ERR_NONE - No tasks were waiting on the semaphore.

OS_ERR_PEND_ABORT - At least one task waiting on the semaphore was readied and informed of the aborted wait; check the return value for the number of tasks whose wait on the semaphores aborted.

OS_ERR_EVENT_TYPE - If you didn't pass a pointer to a semaphore.

OS_ERR_PEVENT_NULL - If 'pevent' is a NULL pointer.

```
INT8U  OSSemPendAbort (OS_EVENT  *pevent,
                       INT8U      opt,
                       INT8U     *perr)
{
    INT8U      nbr_tasks;
#if OS_CRITICAL_METHOD == 3u                          /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr = 0u;
#endif


#ifdef OS_SAFETY_CRITICAL
    if (perr == (INT8U *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return (0u);
    }
#endif

#if OS_ARG_CHK_EN > 0u
```

```
    if (pevent == (OS_EVENT *)0) {                         /* Validate 'pevent'                              */
        *perr = OS_ERR_PEVENT_NULL;
        return (0u);
    }
#endif
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {    /* Validate event block type                      */
        *perr = OS_ERR_EVENT_TYPE;
        return (0u);
    }
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0u) {                        /* See if any task waiting on semaphore?          */
        nbr_tasks = 0u;
        switch (opt) {
            case OS_PEND_OPT_BROADCAST:                    /* Do we need to abort ALL waiting tasks?         */
                 while (pevent->OSEventGrp != 0u) {    /* Yes, ready ALL tasks waiting on semaphore      */
                     (void)OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM, OS_STAT_PEND_ABORT);
                     nbr_tasks++;
                 }
                 break;

            case OS_PEND_OPT_NONE:
            default:                                       /* No,  ready HPT       waiting on semaphore      */
                 (void)OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM, OS_STAT_PEND_ABORT);
                 nbr_tasks++;
                 break;
        }
        OS_EXIT_CRITICAL();
        OS_Sched();                                        /* Find HPT ready to run                          */
        *perr = OS_ERR_PEND_ABORT;
        return (nbr_tasks);
    }
    OS_EXIT_CRITICAL();
    *perr = OS_ERR_NONE;
    return (0u);                                           /* No tasks waiting on semaphore                  */
}
```

Initially different checks are performed to ensure the operation functions in the desired manner.

Safety Check:

```
#ifdef OS_SAFETY_CRITICAL
    if (perr == (INT8U *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return (0u);
    }
#endif
```

The system is configured as "safety critical" (indicated by the defined macro **OS_SAFETY_CRITICAL**), there's a validation to ensure the error pointer **perr** isn't **NULL**. If **perr** is **NULL**, it signifies the function lacks a mechanism to report errors, a serious concern in safety-critical systems. As a response, the

OS_SAFETY_CRITICAL_EXCEPTION() function or macro is invoked, likely handling this critical exception by logging, halting the system, or transitioning it to a safe state. Post exception handling, the function immediately exits, returning **0u** to indicate no tasks were aborted due to the semaphore operation.

Conditional Compilation Check:

```
#if OS_ARG_CHK_EN > 0u
    if (pevent == (OS_EVENT *)0) {                      /* Validate 'pevent'                             */
        *perr = OS_ERR_PEVENT_NULL;
        return (0u);
    }
#endif
```

The argument validation is conditionally compiled based on whether the **OS_ARG_CHK_EN** macro is set to a non-zero value. When argument checking is enabled, the function checks if the **pevent** pointer, which points to an event control block, is **NULL**. If **pevent** is **NULL**, it indicates an invalid or uninitialized event control block passed to the function. In this scenario, the error code **OS_ERR_PEVENT_NULL** is assigned to the location pointed by **perr** to signify that a null pointer for the event was provided. Subsequently, the function exits immediately, returning **0u** to indicate no tasks were aborted due to the semaphore operation. This check helps in the early detection of misuse or errors in the system by ensuring only valid event control blocks are processed.

Validation Check:

```
if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {   /* Validate event block type                     */
        *perr = OS_ERR_EVENT_TYPE;
        return (0u);
    }
```

The code validates whether the event associated with the **pevent** pointer is of the correct type, specifically a semaphore. By accessing the **OSEventType** member of the **pevent** structure and comparing it to the predefined constant **OS_EVENT_TYPE_SEM**, the function checks if the event type is a semaphore. If the event type doesn't match, an error code **OS_ERR_EVENT_TYPE** is set, indicating an unexpected or incorrect event type, and the function returns **0u** for an early exit due to this error. This validation ensures that the provided **pevent** is indeed associated with a semaphore, and if not, the function identifies and handles the discrepancy.

Check if any task is waiting for the semaphore and perform the operation. It is about handling tasks waiting on a semaphore, either making all of them ready or just the highest priority one based on the options provided.

```
if (pevent->OSEventGrp != 0u) {                   /* See if any task waiting on semaphore?         */

        nbr_tasks = 0u;
        switch (opt) {
            case OS_PEND_OPT_BROADCAST:                 /* Do we need to abort ALL waiting tasks?        */
                 while (pevent->OSEventGrp != 0u) {   /* Yes, ready ALL tasks waiting on semaphore     */
                     (void)OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM, OS_STAT_PEND_ABORT);
                     nbr_tasks++;
                 }
                 break;

            case OS_PEND_OPT_NONE:
            default:                                    /* No,  ready HPT       waiting on semaphore     */
                 (void)OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM, OS_STAT_PEND_ABORT);
```

```
                nbr_tasks++;
                break;
        }
        OS_EXIT_CRITICAL();
        OS_Sched();                                     /* Find HPT ready to run                            */
        *perr = OS_ERR_PEND_ABORT;
        return (nbr_tasks);
    }
```

The line **if (pevent->OSEventGrp != 0u)** checks if there are tasks waiting on the semaphore. The **OSEventGrp** member of **pevent** indicates the group of tasks waiting on this event. If it's non-zero, it means there are tasks waiting. Initial the task counter **nbr_tasks** to keep track of how many tasks were made ready due to the operation. Once initialized, handle the tasks based on the specified option '**opt**'.

Broadcast option (OS_PEND_OPT_BROADCAST): If the option specified is to broadcast (meaning to make all waiting tasks ready), the code enters a loop with the line **while (pevent->OSEventGrp != 0u)**. For every iteration, the function **OS_EventTaskRdy()** is called to make a waiting task ready. The parameters passed indicate that the task was waiting on a semaphore (**OS_STAT_SEM**) and that it's being made ready due to an abort (**OS_STAT_PEND_ABORT**). After making each task ready, the counter **nbr_tasks** is incremented.

Default option (OS_PEND_OPT_NONE and default): If the option specified is not to broadcast (or if an unrecognized option is provided), only the highest priority task (HPT) waiting on the semaphore is made ready. Again, **OS_EventTaskRdy()** is used for this, and the counter **nbr_tasks** is incremented.

After handling the tasks based on the specified option, the code exits the critical section with **OS_EXIT_CRITICAL()**. Following this, the **OS_Sched()** function is called to reschedule tasks, ensuring the highest priority task that is now ready runs next.

Once done, the function will exit the critical section and initialize per with **OS_ERR_NONE** indicating that there is no error and would exit the function returning 0 indicating that no tasks were made ready because no tasks were waiting on the semaphore.

OSSemPost():

```
INT8U       OSSemPost           (OS_EVENT       *pevent);
```

This function is used to signal a semaphore. It is a synchronization mechanism used for inter-task communication. Upon invocation, it first validates the provided event pointer and type. If valid, it checks for tasks waiting on this semaphore. If tasks are waiting, the function readies the highest-priority task to run, potentially leading to a context switch. If no tasks are waiting, it increments the semaphore's count, ensuring it doesn't exceed a set maximum value (65535). This function is essential in multitasking environments, enabling tasks or interrupt service routines (ISRs) to signal events or resource availability to other tasks, facilitating coordinated operations and efficient resource sharing. For example, an ISR reading data might signal a processing task using **OSSemPost** to indicate that new data is ready for processing.

It takes in 1 argument, 'pevent' which is a pointer to the event control block associated with the desired semaphore.

It returns one of the following macro:

OS_ERR_NONE - The call was successful and the semaphore was signaled.
OS_ERR_SEM_OVF - If the semaphore count exceeded its limit. In other words, you have signaled the semaphore more often than you waited on it with either OSSemAccept() or OSSemPend().
OS_ERR_EVENT_TYPE - If you didn't pass a pointer to a semaphore.
OS_ERR_PEVENT_NULL - If **'pevent'** is a NULL pointer.

```
INT8U  OSSemPost (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3u                          /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr = 0u;
#endif


#if OS_ARG_CHK_EN > 0u
    if (pevent == (OS_EVENT *)0) {                    /* Validate 'pevent'                             */
        return (OS_ERR_PEVENT_NULL);
    }
#endif

    OS_TRACE_SEM_POST_ENTER(pevent);

    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {   /* Validate event block type                     */
        OS_TRACE_SEM_POST_EXIT(OS_ERR_EVENT_TYPE);
        return (OS_ERR_EVENT_TYPE);
    }
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0u) {                   /* See if any task waiting for semaphore         */
                                                      /* Ready HPT waiting on event                    */
        (void)OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM, OS_STAT_PEND_OK);
        OS_EXIT_CRITICAL();
        OS_Sched();                                   /* Find HPT ready to run                         */
        OS_TRACE_SEM_POST_EXIT(OS_ERR_NONE);
        return (OS_ERR_NONE);
    }
    if (pevent->OSEventCnt < 65535u) {                /* Make sure semaphore will not overflow         */
        pevent->OSEventCnt++;                         /* Increment semaphore count to register event   */
        OS_EXIT_CRITICAL();
        OS_TRACE_SEM_POST_EXIT(OS_ERR_NONE);
        return (OS_ERR_NONE);
    }
    OS_EXIT_CRITICAL();                               /* Semaphore value has reached its maximum       */
    OS_TRACE_SEM_POST_EXIT(OS_ERR_SEM_OVF);

    return (OS_ERR_SEM_OVF);
}
```

Initially, the argument '**pevent'** validation is performed and a block type check is performed to ensure that the event type is a semaphore. Once it is passed, the code enters into the critical section to perform the operation. It handles two scenarios when an attempt is made to signal (or post to) a semaphore,

Tasks Waiting for Semaphore: The first **if** condition checks if any tasks are waiting on the semaphore by examining the **OSEventGrp** of the provided event (**pevent**). If **OSEventGrp** is not zero, it indicates that there are tasks waiting. The function **OS_EventTaskRdy** is then called to ready the highest priority task (HPT) that's waiting on this semaphore. This task has been waiting for the semaphore to be signaled, and now that it has been signaled, the task can proceed. After making the task ready, the function exits the critical section (ensuring atomic operations) and then calls **OS_Sched()**. This is a scheduler function that potentially causes a context switch to the highest priority task that's ready to run. The function then logs the successful post operation and returns with a status indicating success (**OS_ERR_NONE**).

Semaphore Count Increment: If no tasks are waiting for the semaphore, the next operation is to increase the semaphore's count, which keeps track of the number of posts that have occurred. Before doing so, the function ensures that this count does not overflow its maximum allowed value (65535 in this case). If the count is below this threshold, it's incremented, indicating that the semaphore has been signaled. The function then exits the critical section, logs the successful post operation, and returns with a status indicating success (**OS_ERR_NONE**).

If none of the above conditions is satisfied, the semaphore has reached its maximum count and a value OS_ERR_SEM_OVF is returned indicating a semaphore overflow error.

OSSemQuery():

```
INT8U       OSSemQuery          (OS_EVENT       *pevent, OS_SEM_DATA     *p_sem_data);
```

The **OSSemQuery** function provides a mechanism to retrieve information about a specified semaphore. By accepting two arguments - a pointer to the semaphore's event control block (**pevent**) and a pointer to a data structure (**p_sem_data**) where the semaphore's information will be stored - the function allows users to inspect the state and configuration of the semaphore. Initial checks ensure the validity of the provided pointers and confirm that the event type corresponds to a semaphore. Once validated, the function enters a critical section to ensure atomic operations. It then copies details about tasks waiting on the semaphore into the provided data structure and retrieves the semaphore's current count. This function is typically used in scenarios where there's a need to monitor, debug, or assess the state of a semaphore without modifying its state. For instance, a system health check routine might use **OSSemQuery** to ensure that certain semaphores are not being over- or under-utilized, helping in early detection of potential deadlocks or resource leaks.

The function takes in 2 arguments, ‘**pevent’** that is a pointer to the event control block. ‘**p_sem_data**’ which is a pointer to a structure that will contain information about the semaphore.

It returns one of the following value:
OS_ERR_NONE   - The call was successful and the message was sent.
OS_ERR_EVENT_TYPE - If you are attempting to obtain data from a non semaphore.
OS_ERR_PEVENT_NULL- If 'pevent'   is a NULL pointer.
OS_ERR_PDATA_NULL - If 'p_sem_data' is a NULL pointer.

Initial checks are performed on ‘**pevent’** and ‘**p_sem_data**’ to make sure that the given value is not null. Another check is performed on the event type to ensure that the current event is a semaphore.

```
INT8U  OSSemQuery (OS_EVENT     *pevent,
                   OS_SEM_DATA  *p_sem_data)
{
    INT8U       i;
    OS_PRIO    *psrc;
    OS_PRIO    *pdest;
#if OS_CRITICAL_METHOD == 3u                              /* Allocate storage for CPU status register */
    OS_CPU_SR   cpu_sr = 0u;
#endif



#if OS_ARG_CHK_EN > 0u
    if (pevent == (OS_EVENT *)0) {                        /* Validate 'pevent'                        */
        return (OS_ERR_PEVENT_NULL);
    }
    if (p_sem_data == (OS_SEM_DATA *)0) {                 /* Validate 'p_sem_data'                    */
        return (OS_ERR_PDATA_NULL);
    }
#endif
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {       /* Validate event block type                */
        return (OS_ERR_EVENT_TYPE);
    }
    OS_ENTER_CRITICAL();
    p_sem_data->OSEventGrp = pevent->OSEventGrp;          /* Copy message mailbox wait list           */
    psrc                   = &pevent->OSEventTbl[0];
    pdest                  = &p_sem_data->OSEventTbl[0];
    for (i = 0u; i < OS_EVENT_TBL_SIZE; i++) {
        *pdest++ = *psrc++;
    }
    p_sem_data->OSCnt = pevent->OSEventCnt;               /* Get semaphore count                      */
    OS_EXIT_CRITICAL();
    return (OS_ERR_NONE);
}
```

The next section is dedicated to copying detailed information about a semaphore's state into the **p_sem_data** structure. Initially, the priority group of tasks waiting on the semaphore is copied with **p_sem_data->OSEventGrp = pevent->OSEventGrp;**. The code then prepares to delve into the finer details by setting pointers **psrc** and **pdest** to the start of the event tables in the original semaphore and the **p_sem_data** structure, respectively. A loop subsequently iterates over the entire event table, transferring the specifics about each waiting task (often sorted by priority) from the original semaphore to the **p_sem_data**. Concluding the section, the current count of the semaphore, indicating its available "signals" or "posts", is copied over with **p_sem_data->OSCnt = pevent->OSEventCnt;**. Altogether, this process furnishes a snapshot of the semaphore's current condition, encapsulating details like the tasks awaiting it and its current count. Once the operation is performed successfully, an **OS_ERR_NONE** is returned by the function indicating that there was no error in querying about the semaphore.

OSSemSet():

```
void        OSSemSet              (OS_EVENT       *pevent, INT16U          cnt, INT8U          *perr);
```

This function sets the semaphore count to the value specified as an argument. It is used when a semaphore is used as a single mechanism and you want to reset the count value.

When invoked, it first checks for a set of preconditions, including the validity of the input semaphore pointer and its type. Entering a critical section ensures atomicity during the operation. The function then checks if the semaphore already has a count. If it does, the count is directly set to the desired value. If the semaphore count is zero, it checks if there are any tasks waiting on this semaphore. If no tasks are waiting, the semaphore count is set to the specified value. If tasks are waiting, an error indicating that tasks are pending on this semaphore is returned. This function is mainly used when there's a requirement to reset or adjust the signaling mechanism in real-time operating systems, ensuring that semaphores reflect the intended state in synchronized operations.

This function takes in 3 arguments, '**pevent**' that is a pointer to the event control block, '**cnt**' that is the new value for the semaphore count. It would be passed 0 to reset the semaphore count and '**perr**' which is a pointer to an error cod returned by the function; it is one of the following error code for this function:

OS_ERR_NONE - The call was successful and the semaphore value was set.
OS_ERR_EVENT_TYPE - If you didn't pass a pointer to a semaphore.
OS_ERR_PEVENT_NULL - If 'pevent' is a NULL pointer.
OS_ERR_TASK_WAITING - If tasks are waiting on the semaphore.

```
void  OSSemSet (OS_EVENT  *pevent,
                INT16U     cnt,
                INT8U     *perr)
{
#if OS_CRITICAL_METHOD == 3u                          /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr = 0u;
#endif



#ifdef OS_SAFETY_CRITICAL
    if (perr == (INT8U *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

#if OS_ARG_CHK_EN > 0u
    if (pevent == (OS_EVENT *)0) {                    /* Validate 'pevent'                             */
        *perr = OS_ERR_PEVENT_NULL;
        return;
    }
#endif
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {   /* Validate event block type                     */
        *perr = OS_ERR_EVENT_TYPE;
        return;
    }
    OS_ENTER_CRITICAL();
    *perr = OS_ERR_NONE;
    if (pevent->OSEventCnt > 0u) {                    /* See if semaphore already has a count          */
        pevent->OSEventCnt = cnt;                     /* Yes, set it to the new value specified.      */
    } else {                                          /* No                                            */
        if (pevent->OSEventGrp == 0u) {               /*      See if task(s) waiting?                  */
            pevent->OSEventCnt = cnt;                 /*      No, OK to set the value                  */
        } else {
            *perr              = OS_ERR_TASK_WAITING;
        }
    }
    OS_EXIT_CRITICAL();
}
#endif
```

Initially, safety checks are performed on '**pevent'** and validation of block type to check if the received event is of semaphore type. The code then enters into the critical section to perform the operation. It determines whether the count of the semaphore can be set to desired value based on its current state and if tasks are waiting on it. If the semaphore already has a count or no tasks are waiting on it when its count is zero, it can be adjusted. If tasks are waiting on the semaphore when its count is zero, it signifies a condition where the semaphore cannot be changed and error is returned.

Checking if Semaphore Already Has a Count: The first line **if (pevent->OSEventCnt > 0u)** checks if the semaphore already has a count (i.e., its count is greater than zero).

If it does (**pevent->OSEventCnt > 0u** is true), then the code directly sets the semaphore's count to the desired value **cnt** with the line **pevent->OSEventCnt = cnt;**.

Handling the Case where Semaphore Count is Zero: If the semaphore doesn't have a count (i.e., its count is zero), the code moves to the **else** section.

Within this section, it checks if there are tasks waiting on this semaphore using the condition **if (pevent->OSEventGrp == 0u)**. The **OSEventGrp** being zero indicates that no tasks are waiting on the semaphore.

If no tasks are waiting (**pevent->OSEventGrp == 0u** is true), the code then sets the semaphore's count to the desired value **cnt** with the line **pevent->OSEventCnt = cnt;**.

However, if there are tasks waiting on the semaphore (**pevent->OSEventGrp == 0u** is false), the code sets the error pointer ***perr** to **OS_ERR_TASK_WAITING** to indicate that there are tasks pending on this semaphore and thus the count cannot be changed.

The function then exits the critical section after finishing the required operation.

This was an article describing Semaphore Management in MicroC/OS-II. Hope you enjoyed learning about it.

References: MicroC/OS-II The Real-Time Kernel, Second Edition, Jean J. Labrosse.

Article Written By: Yashwanth Naidu Tikkisetty

Follow **#oswithyash** to get updates related to Embedded Systems, Space and Technology.