Unilateral Rendez-vous

semaphores

The figure below shows that a task can be synchronized with an ISR (or another task) by using a semaphore. In this case, no data is exchanged, however there is an indication that the ISR or the task (on the left) has occurred. Using a semaphor for this type of synchronization is called a unilateral rendez-vous.

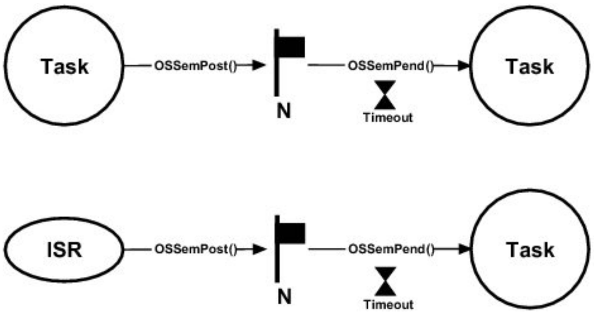

Figure - Unilateral Rendezvous

A unilateral rendez-vous is used when a task initiates an I/O operation and waits (i.e., call `OSSemPend()`) for the semaphore to be signaled (posted). When the I/O operation is complete, an ISR (or another task) signals the semaphore (i.e., calls `OSSemPost()`), and the task is resumed. This process is also shown on the timeline of the figure below and described below. The code for the ISR and task is shown in the listing follows the figure below.

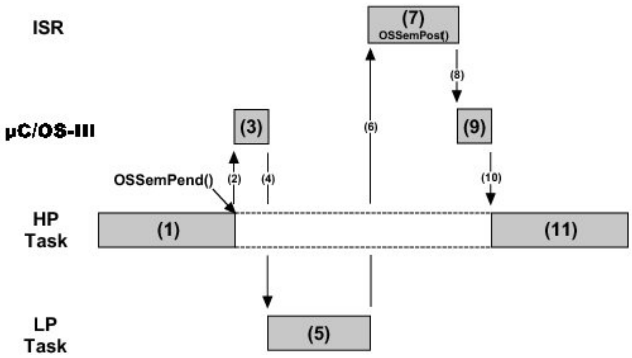

Figure - Unilateral Rendezvous, Timing Diagram

(1) A high priority task is executing. The task needs to synchronize with an ISR (i.e., wait for the ISR to occur) and call `OSSemPend()`.

(2-3) Since the ISR has not occurred, the task will be placed in the waiting list for the semaphore until the event occurs The scheduler in µC/OS-III will then select the next most important task and context switch to that task.

(5) The low-priority task executes.

(6) The event that the original task was waiting for occurs. The lower-priority task is immediately preempted (assuming interrupts are enabled), and the CPU vectors to the interrupt handler for the event.

(7-8) The ISR handles the interrupting device and then calls `OSSemPost()` to signal the semaphore. When the ISR completes, µC/OS-III is called (i.e. `OSIntExit()`).

(9-10) µC/OS-III notices that a higher-priority task is waiting for this event to occur and context switches back to the original task.

(11) The original task resumes execution immediately after the call to `OSSemPend()`.

```
OS_SEM  MySem;
```

```


void MyISR (void)
{
    OS_ERR  err;


    /* Clear the interrupting device */
    OSSemPost(&MySem,                    (7)
              OS_OPT_POST_1,
              &err);
    /* Check “err” */
}


void MyTask (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;
    :
    :
    while (DEF_ON) {
        OSSemPend(&MySem,                (1)
                  10,
                  OS_OPT_PEND_BLOCKING,
                  &ts,
                  &err);
        /* Check “err” */                (11)
        :
        :
    }
}
```

Listing - Pending (or waiting) on a Semaphore

A few interesting things are worth noting about this process. First, the task does not need to know about the details of wha happens behind the scenes. As far as the task is concerned, it called a function (`OSSemPend()`) that will return when the event it is waiting for occurs. Second, µC/OS-III maximizes the use of the CPU by selecting the next most important task, which executes until the ISR occurs. In fact, the ISR may not occur for many milliseconds and, during that time, the CPU w work on other tasks. As far as the task that is waiting for the semaphore is concerned, it does not consume CPU time whil is waiting. Finally, the task waiting for the semaphore will execute immediately after the event occurs (assuming it is the mo important task that needs to run).