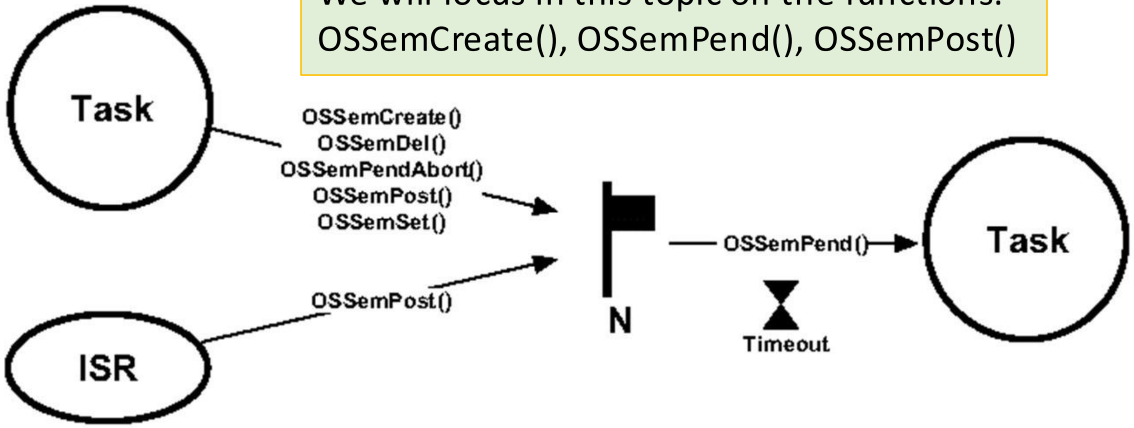
Synchronization

- How can tasks synchronize their activities with interrupt service routines or other tasks?
- When an ISR executes, it can signal a task telling the task that an event of interest has occurred …
- Servicing interrupting devices from task level is preferred since it reduces the amount of time that interrupts are disabled, and the code is easier to debug
- uC/OSIII provides two main mechanisms for synchronization:
  1. Use of counting semaphores
  2. Use of event flags

Synchronization Using Counting Semaphores

When used in this context the semaphores are drawn as a flag!

We will focus in this topic on the functions: OSSemCreate(), OSSemPend(), OSSemPost()



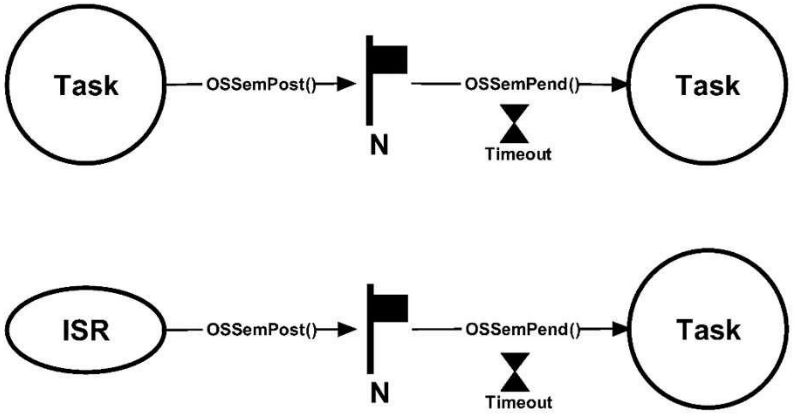
- Semaphores are best used to synchronize an ISR to a task, or synchronize a task with another task
- The N value next to the flag indicates that the semaphore can accumulate events or credits

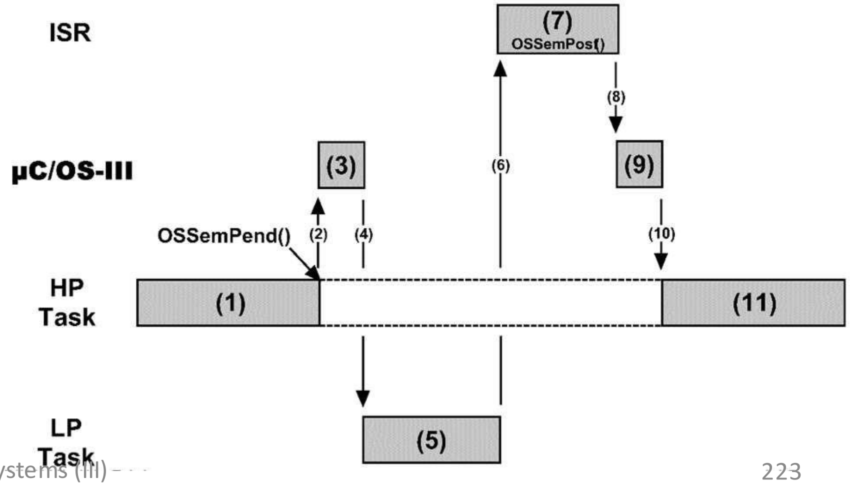
Synchronization Techniques Using Semaphores

- Unilateral rendez-vous using a semaphore: used when no data needs to be exchanged, but an indication that an ISR or task has occurred
- Credit tracking: allows the tasks to execute without losing track of the events
- Multiple tasks waiting on a semaphore: broadcasting is used to synchronize multiple tasks and tell that all task waiting on the semaphore be made ready-to-run not only the HPT waiting
- Usage of task semaphore: uC/OSIII allows built in task semaphores that can be used in similar way for synchronization as the semaphore object

Unilateral Rendez-Vous

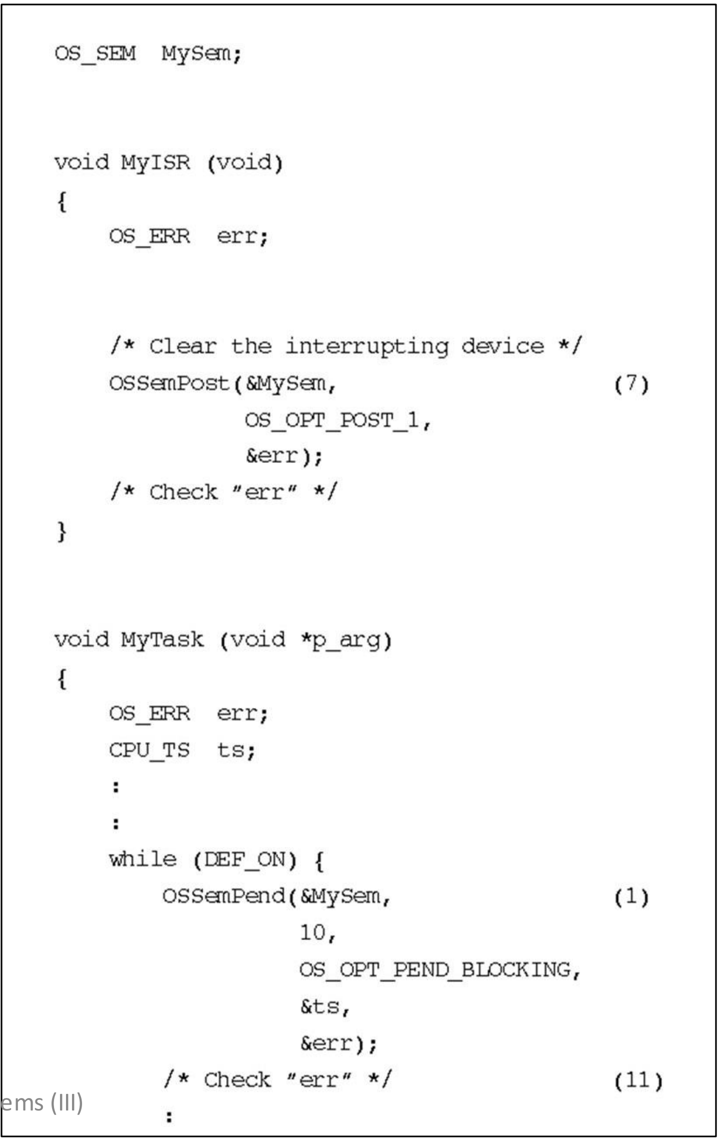
- Unilateral rendez-vous is used when a task initiates an I/O operation and waits for a semaphore to be posted.
- The task that waits synchronizes with an ISR or another task
- The rendez-vous process is shown in the task diagram …





ISR and Task Code for Rendez-Vous

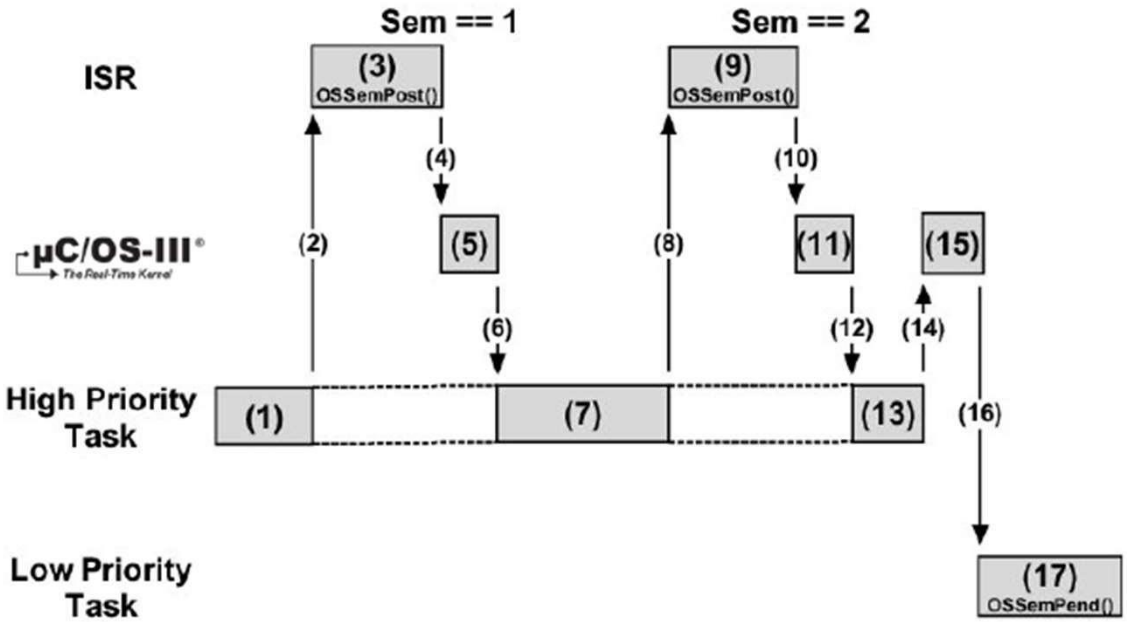
- (1) … in this process the task just needs to call the function OSSemPend(), which will return when the event occurs …
- (7) … the task waiting for the semaphore will execute when the event occurs assuming it is the HPT ready to run …
- (11) …



```
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)
        :
```

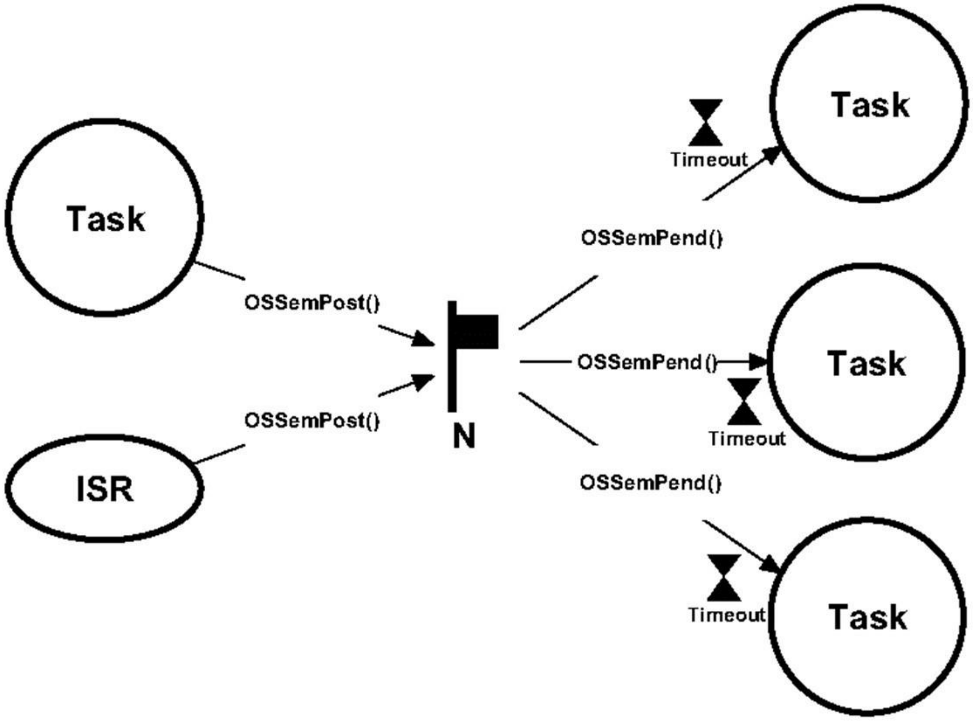
Credit Tracking

- A counting semaphore remembers how many times was signaled
- When the task waiting becomes the HPT ready-to-run task, it will execute without blocking until all accumulated counts expire.
- (1), …
- (2), (3) …
- (4), (5), (6) …
- (7), …
- (8), (9) …
- …
- (16), (17) …



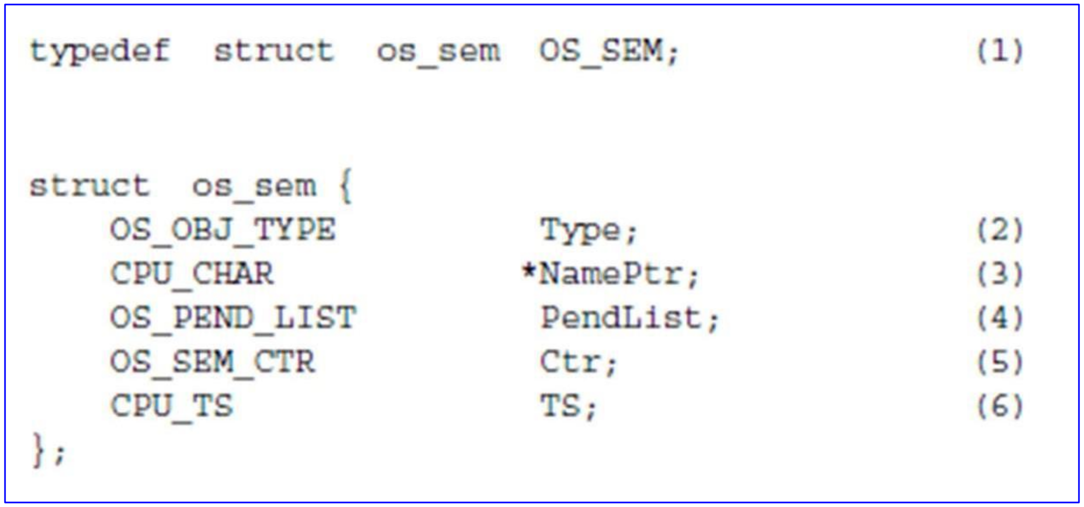
Multiple Tasks Waiting on a Semaphore

- In general, the HPT ready to run can execute after a post operation
- Or, we can broadcast to all task using OS_OPT_POST_ALL as an option
- Broadcasting is a technique used to synchronize multiple tasks.
- If some tasks waiting on the same semaphore do not want to synchronize, we can resolve this problem by combining semaphores and event flags.



Semaphore Internals (for synchronization)

- A semaphore is a kernel object as defined by the OS_SEM data type, which is derived from the structure os_sem (see os.h)
- Notice the fields utilized in building a semaphore object
- (1) …
- …



```
typedef  struct  os_sem  OS_SEM;                    (1)


struct  os_sem {
    OS_OBJ_TYPE          Type;                      (2)
    CPU_CHAR            *NamePtr;                   (3)
    OS_PEND_LIST         PendList;                  (4)
    OS_SEM_CTR           Ctr;                       (5)
    CPU_TS               TS;                        (6)
};
```

Semaphores must be created OSSemCreate(…) before they can be used by an application

A task waits for a signal from an ISR or another task by calling OSSemPend(…)

To signal a task (either from an ISR or a task), simply call OSSemPost(…); you can post to only 1 task or more by specifying the appropriate parameter

Task Semaphores

- In uC/OS-III each task has its own semaphore built in and cannot be disabled at compile time as can other services; its default value is 0
- A task can pend on its semaphore, and other tasks or ISR can post if your code knows which task to signal when the event occurs

Task semaphore services in µC/OS-III start with the OSTaskSem???() prefix

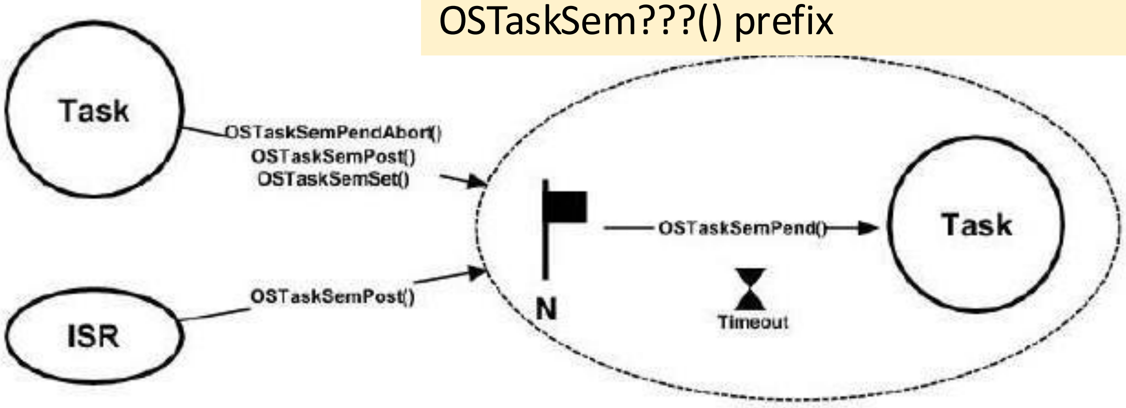
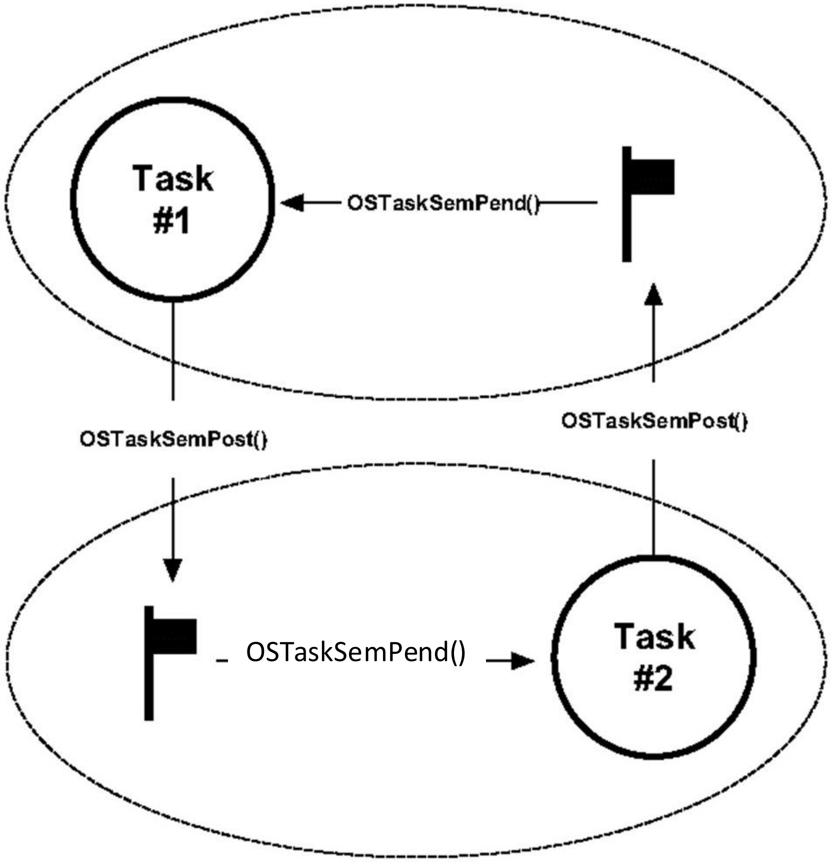


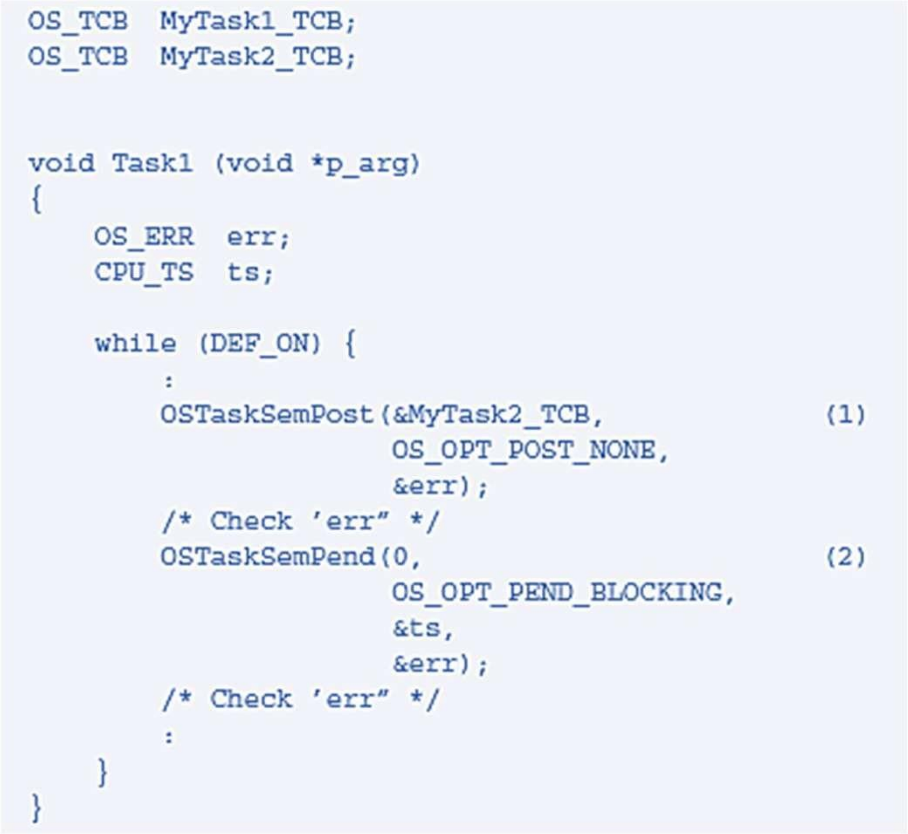
Figure - Semaphore built-into a Task

Bilateral Rendez-Vous – Task Synchronization

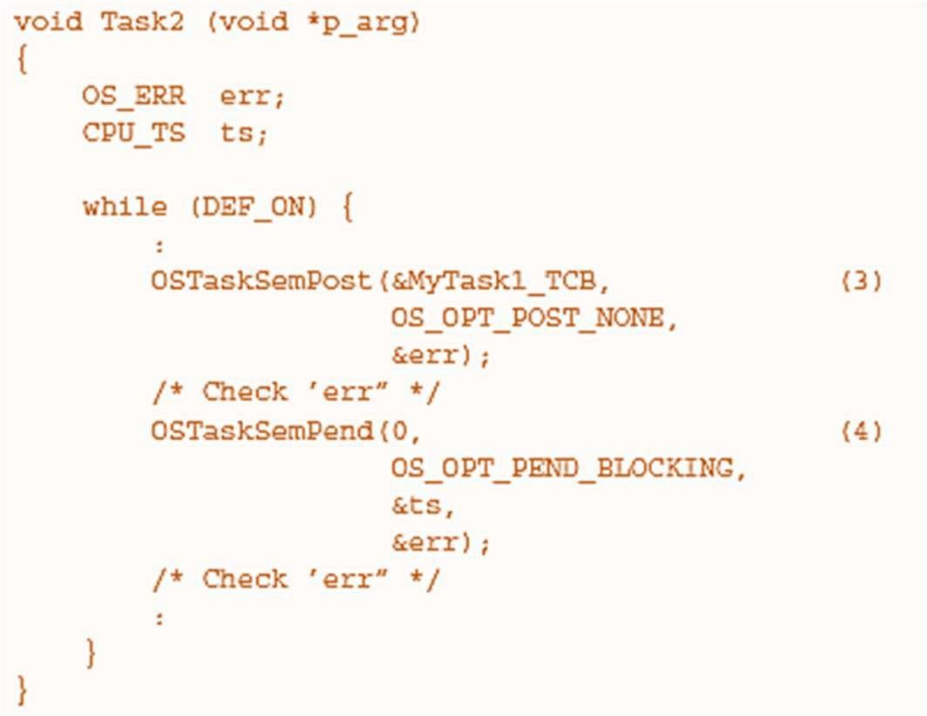
- The task semaphore initial value is zero
- An ISR or a task can signal a task by calling OSTaskSemPost()
- Two tasks can synchronize their activities by using two task semaphores in a bilateral way



Bilateral Rendez-Vous Code



```
OS_TCB  MyTask1_TCB;
OS_TCB  MyTask2_TCB;


void Task1 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;

    while (DEF_ON) {
        :
        OSTaskSemPost(&MyTask2_TCB,                (1)
                      OS_OPT_POST_NONE,
                      &err);
        /* Check 'err" */
        OSTaskSemPend(0,                           (2)
                      OS_OPT_PEND_BLOCKING,
                      &ts,
                      &err);
        /* Check 'err" */
        :
    }
}
```

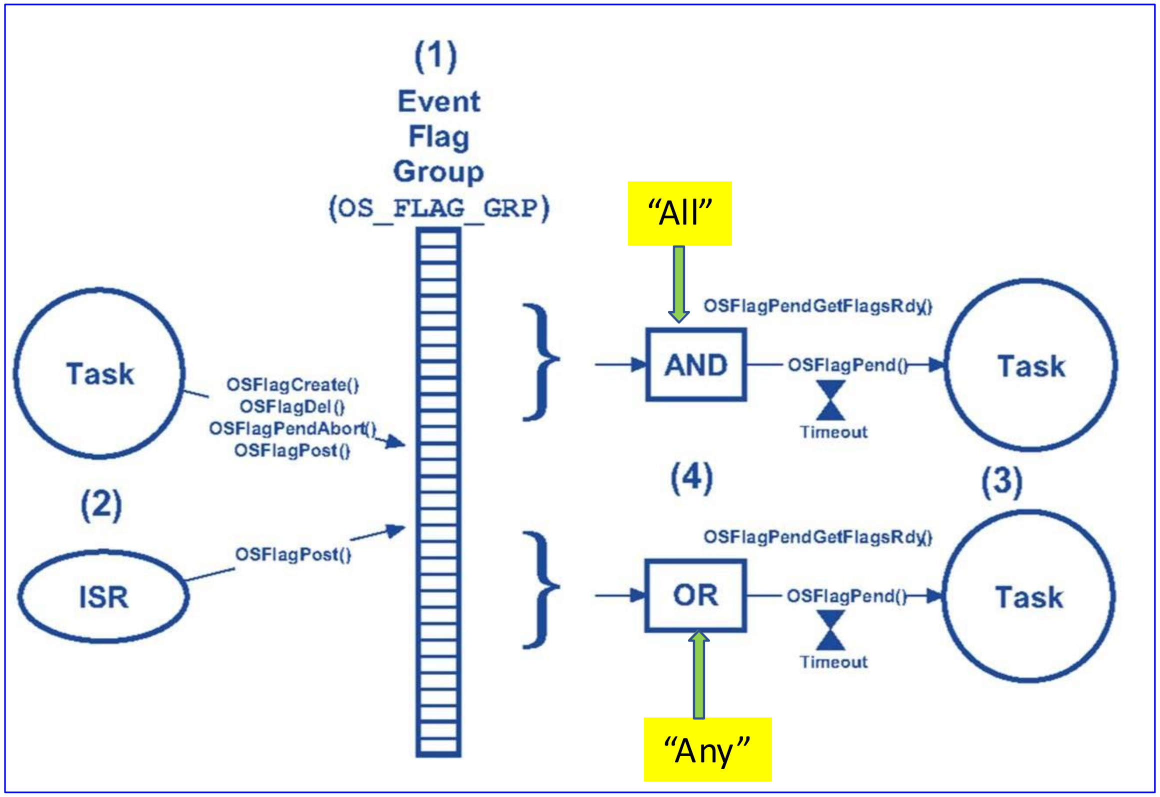


```
void Task2 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;

    while (DEF_ON) {
        :
        OSTaskSemPost(&MyTask1_TCB,                (3)
                      OS_OPT_POST_NONE,
                      &err);
        /* Check 'err" */
        OSTaskSemPend(0,                           (4)
                      OS_OPT_PEND_BLOCKING,
                      &ts,
                      &err);
        /* Check 'err" */
        :
    }
}
```

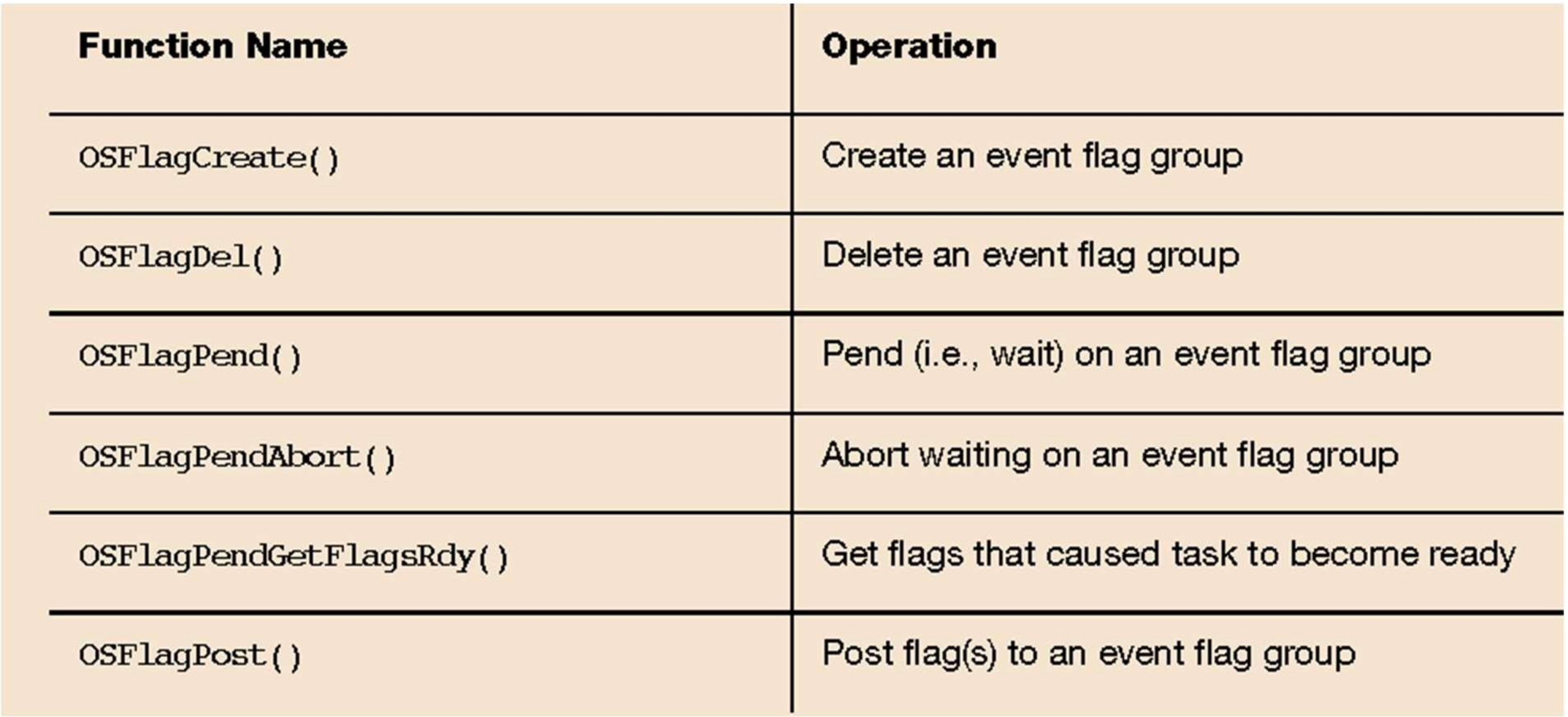
Event Flags

Enabled by setting the configuration constant OS_CFG_FLAG_EN to DEF_ENABLED in os_cfg.h.

- Event flags are used when a task needs to synchronize with the occurrence of multiple events
- (1) … event flag group is a kernel object of type OS_FLAG_GRP and consists of a series of bits
- (2) … functions call types …
- (3) … tasks waiting …
- (4) … wait type …



Event Group Flag Functions



Function Name	Operation
OSFlagCreate()	Create an event flag group
OSFlagDel()	Delete an event flag group
OSFlagPend()	Pend (i.e., wait) on an event flag group
OSFlagPendAbort()	Abort waiting on an event flag group
OSFlagPendGetFlagsRdy()	Get flags that caused task to become ready
OSFlagPost()	Post flag(s) to an event flag group

Using Event Flags

Example of event flag setup:

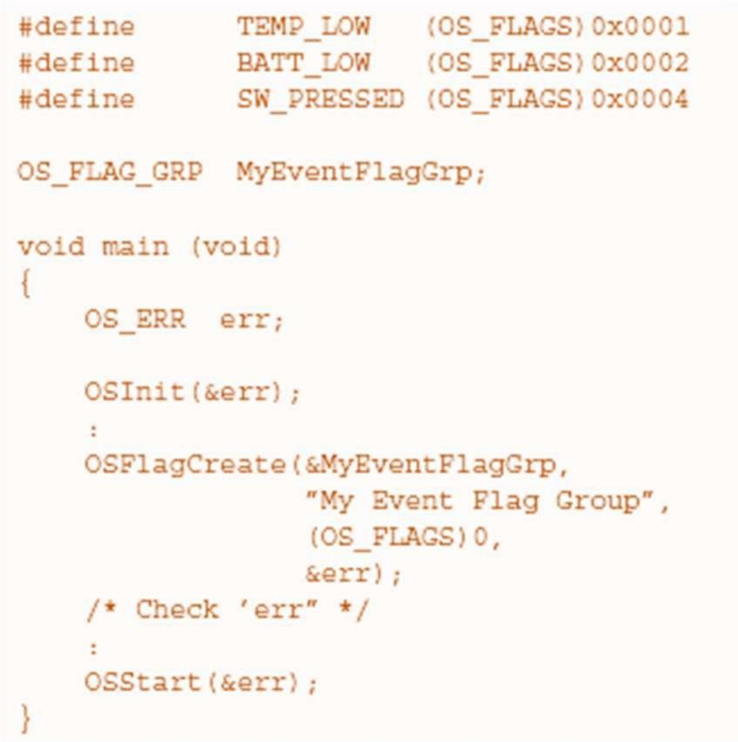
bit #0 -> temperature sensor is too low,

bit #1 -> low battery voltage,

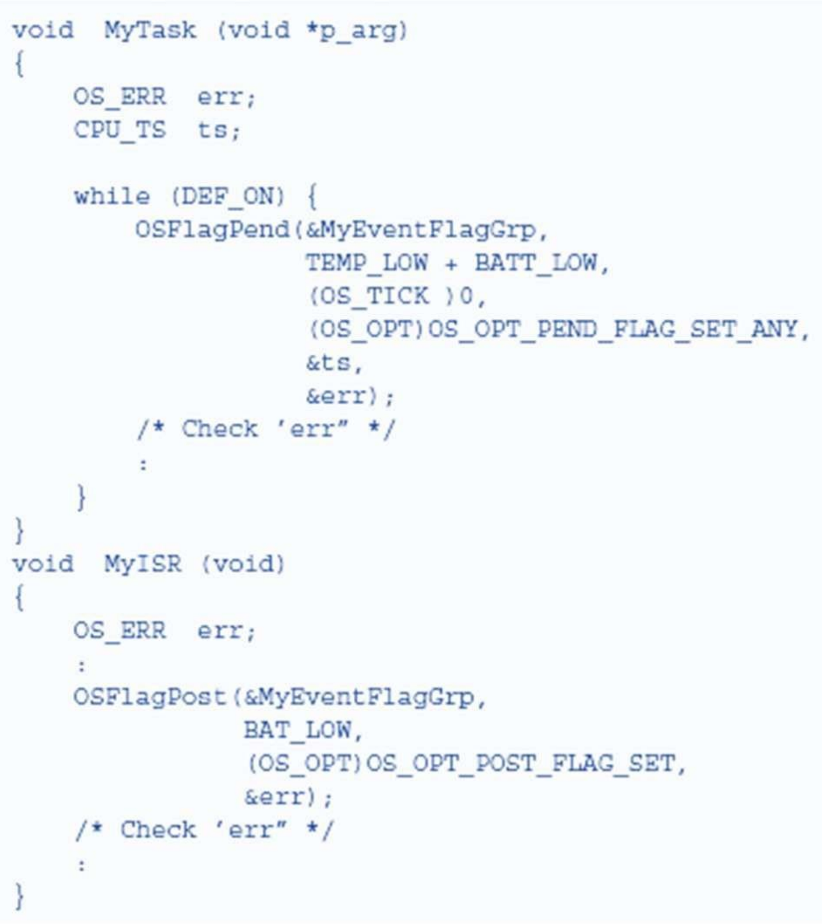
bit #2 -> a switch was pressed, etc.

The code (tasks or ISRs) that detects these conditions would set the appropriate event flag by calling OSFlagPost().

The task(s) that would respond to those conditions would call OSFlagPend()



```
#define      TEMP_LOW    (OS_FLAGS)0x0001
#define      BATT_LOW    (OS_FLAGS)0x0002
#define      SW_PRESSED  (OS_FLAGS)0x0004

OS_FLAG_GRP  MyEventFlagGrp;

void main (void)
{
    OS_ERR   err;

    OSInit(&err);
    :
    OSFlagCreate(&MyEventFlagGrp,
                 "My Event Flag Group",
                 (OS_FLAGS)0,
                 &err);
    /* Check 'err" */
    :
    OSStart(&err);
}
```

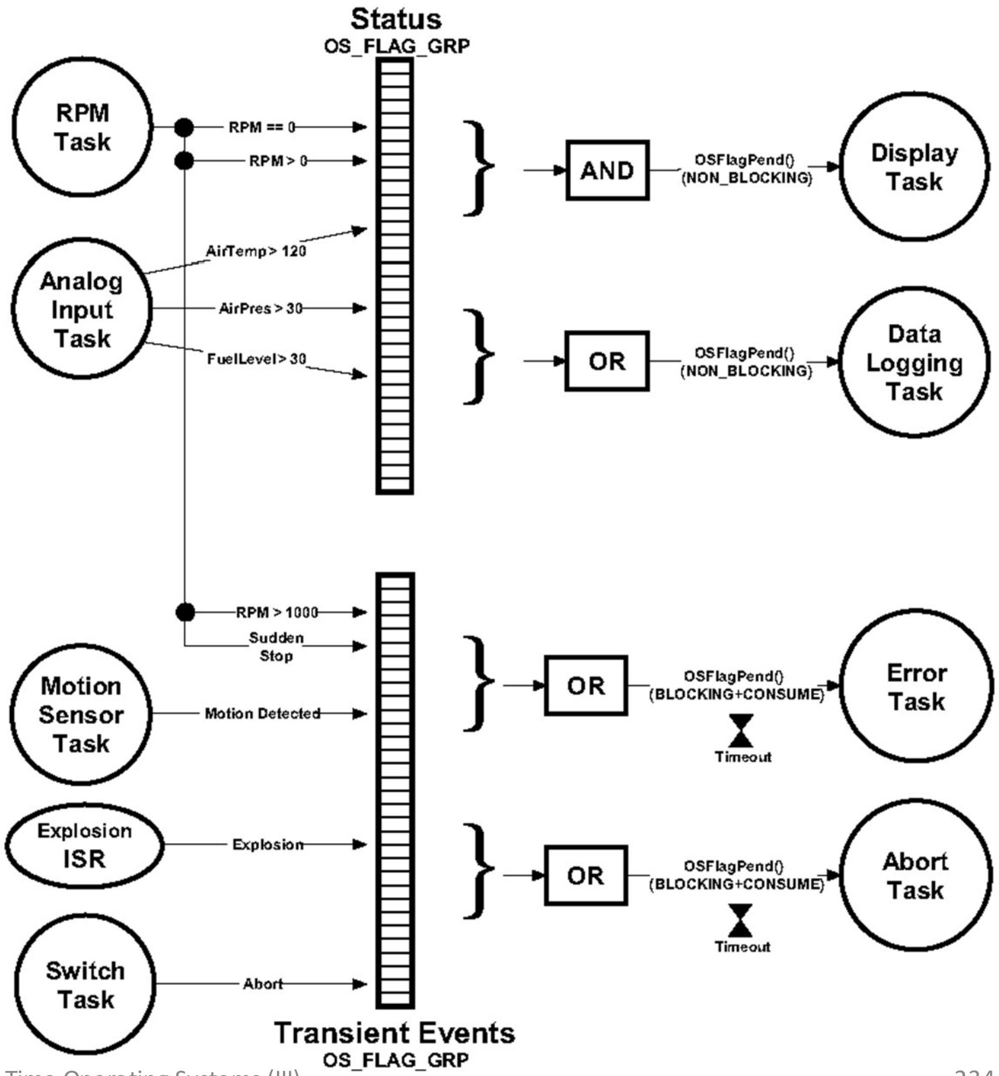


```
void  MyTask (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;

    while (DEF_ON) {
        OSFlagPend(&MyEventFlagGrp,
                   TEMP_LOW + BATT_LOW,
                   (OS_TICK )0,
                   (OS_OPT)OS_OPT_PEND_FLAG_SET_ANY,
                   &ts,
                   &err);
        /* Check 'err" */
        :
    }
}
void  MyISR (void)
{
    OS_ERR  err;
    :
    OSFlagPost(&MyEventFlagGrp,
               BAT_LOW,
               (OS_OPT)OS_OPT_POST_FLAG_SET,
               &err);
    /* Check 'err" */
    :
}
```

Event Flags Applications

Event flags are generally used for two purposes: status and transient events.

1. Used for status events
   - status information events are monitored by other tasks by using non-blocking wait calls.
2. Used for transient events
   - Task waiting for transient event will typically block waiting for any of those events to occur and “consume” the event

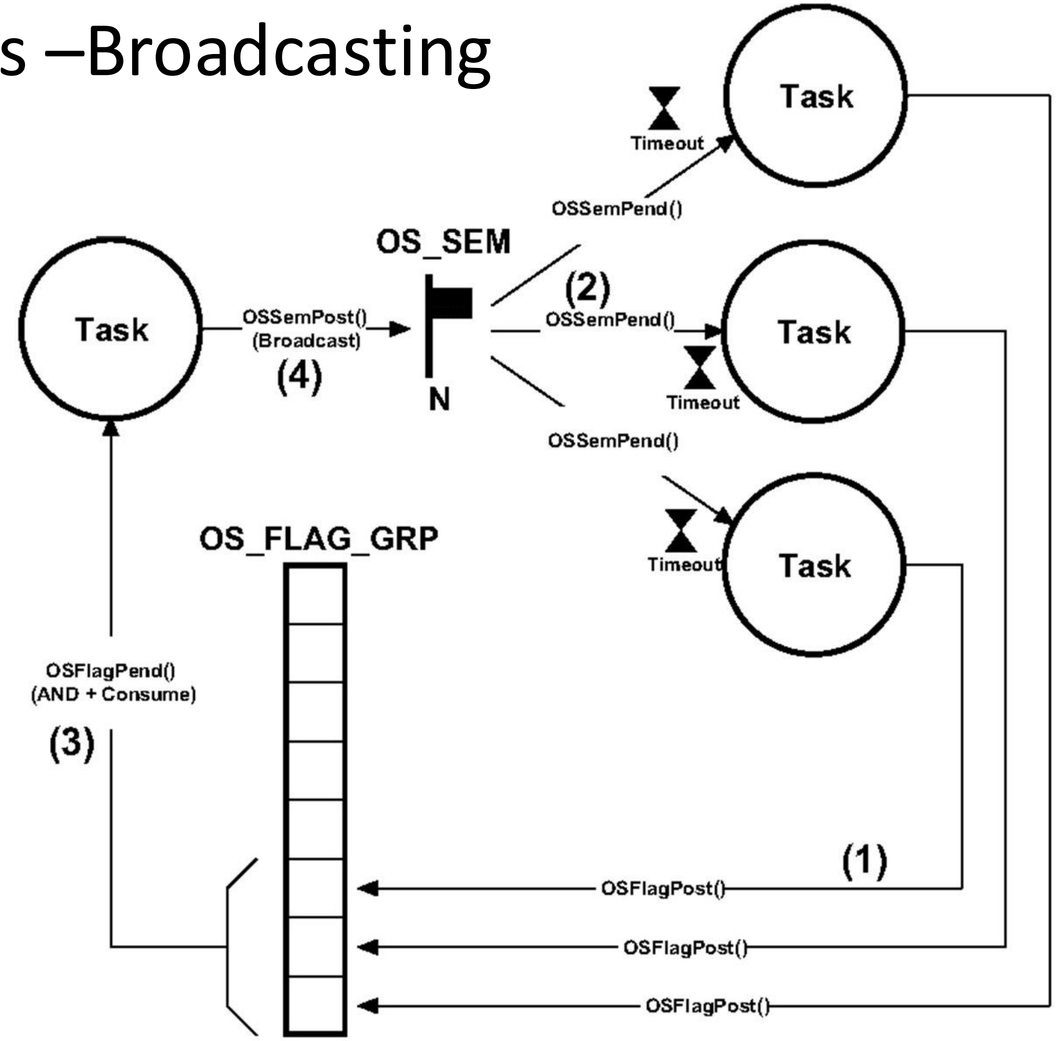


Synchronizing Multiple Tasks –Broadcasting

Problem: multiple task rendez-vous. However, some of the tasks synchronized might not be waiting for the semaphore when the broadcast is performed.

Solution: combine semaphores and flags

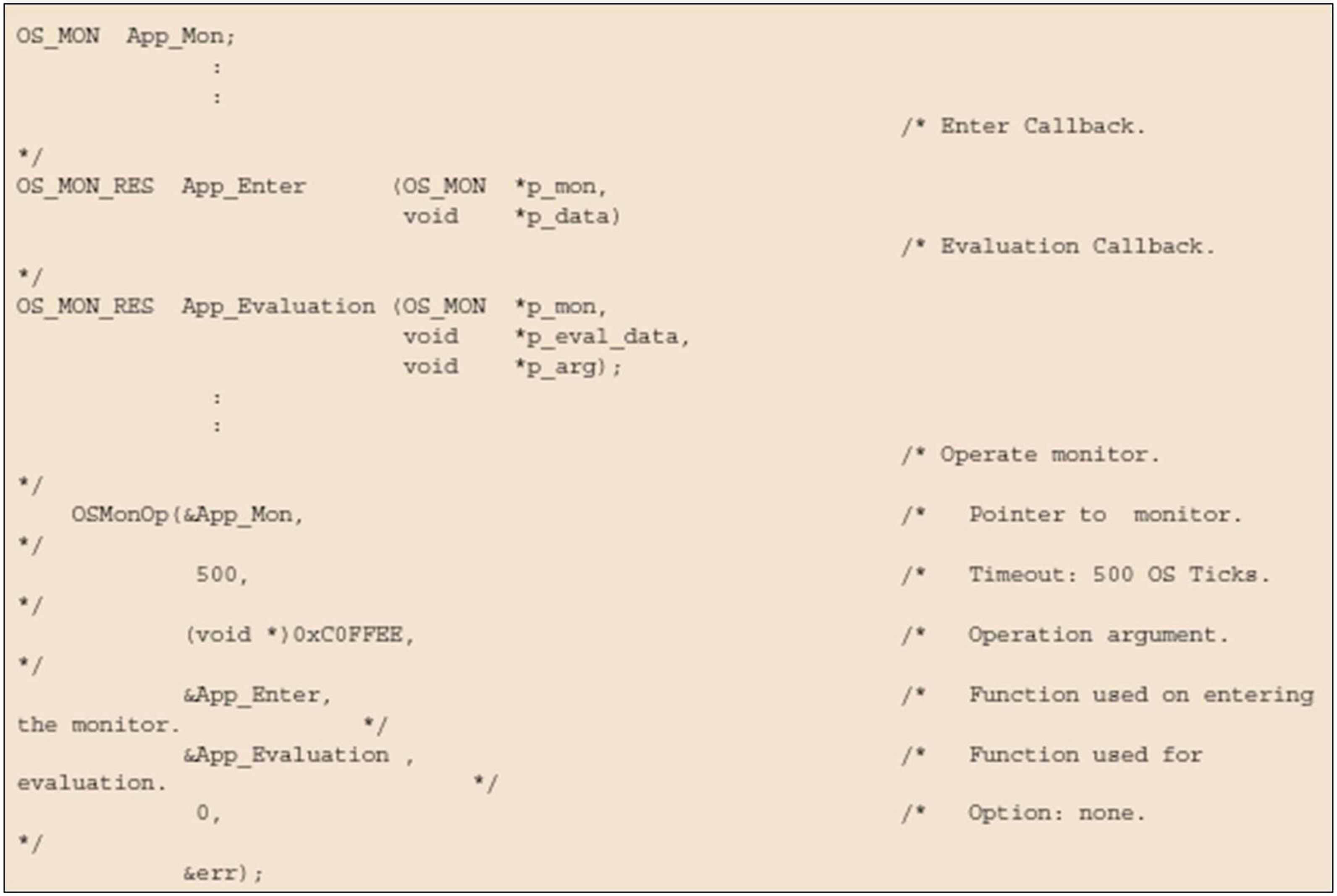
- (1) Each task that needs to synchronize at the rendez-vous needs to set an event flag bit and specify OS_OPT_POST_NO_SCHED, scheduler is not called on this post functions.
- (2) … wait for semaphore …
- (3) … wait for all task to be ready …
- (4) … broadcast to the semaphore …



Advanced Synchronization

- Monitor: an advanced synchronization mechanism.
  - It consists of a state variable and a list of waiting tasks.
  - A task can wait for a specific condition. This condition is evaluated, or checked, every time the monitor is changed.
- After a monitor is created with OSMonCreate(), tasks can operate it using the OSMonOp() function.
- Monitor operation uses two callbacks and their return values.
  - The callbacks are known as the enter callback and the evaluation callback.
  - The enter callback is responsible for blocking the calling task and the evaluation callback is responsible for readying a waiting task.

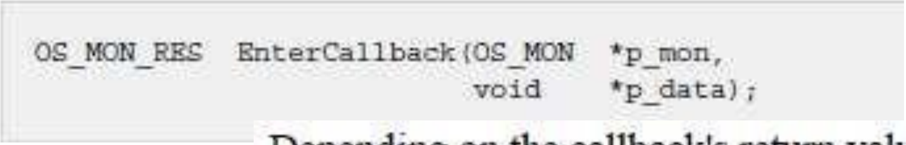
Operation of a Monitor



```
OS_MON  App_Mon;
              :
              :
                                                                /* Enter Callback.
*/
OS_MON_RES  App_Enter      (OS_MON  *p_mon,
                            void    *p_data)
                                                                /* Evaluation Callback.
*/
OS_MON_RES  App_Evaluation (OS_MON  *p_mon,
                            void    *p_eval_data,
                            void    *p_arg);
              :
              :
                                                                /* Operate monitor.
*/
    OSMonOp(&App_Mon,                                           /*   Pointer to  monitor.
*/
             500,                                               /*   Timeout: 500 OS Ticks.
*/
            (void *)0xC0FFEE,                                   /*   Operation argument.
*/
            &App_Enter,                                         /*   Function used on entering
the monitor.              */
            &App_Evaluation ,                                   /*   Function used for
evaluation.                       */
             0,                                                 /*   Option: none.
*/
            &err);
```

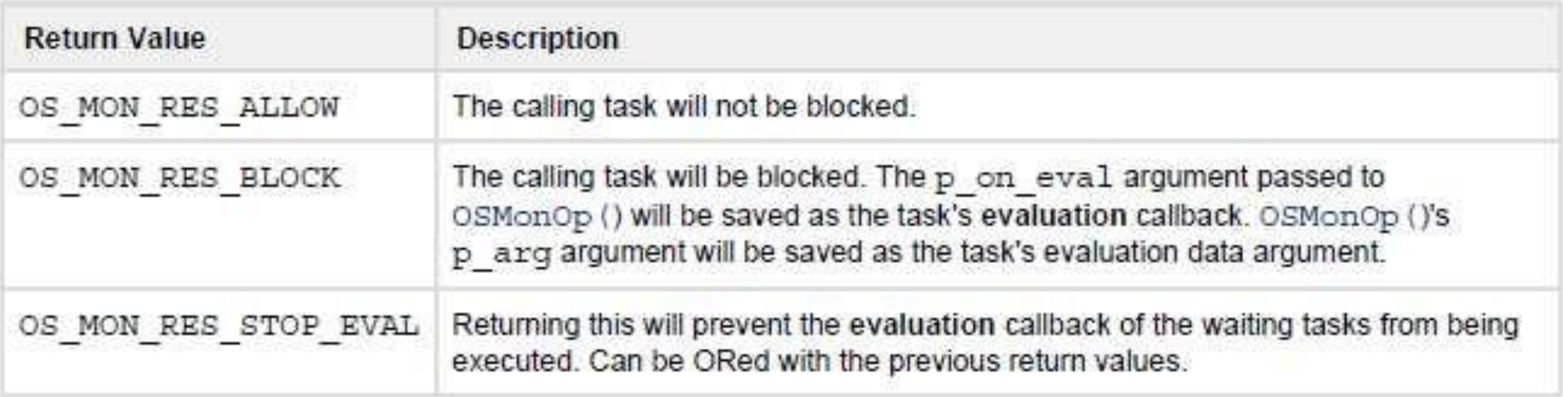
Enter callback

The **enter** callback has the following prototype:



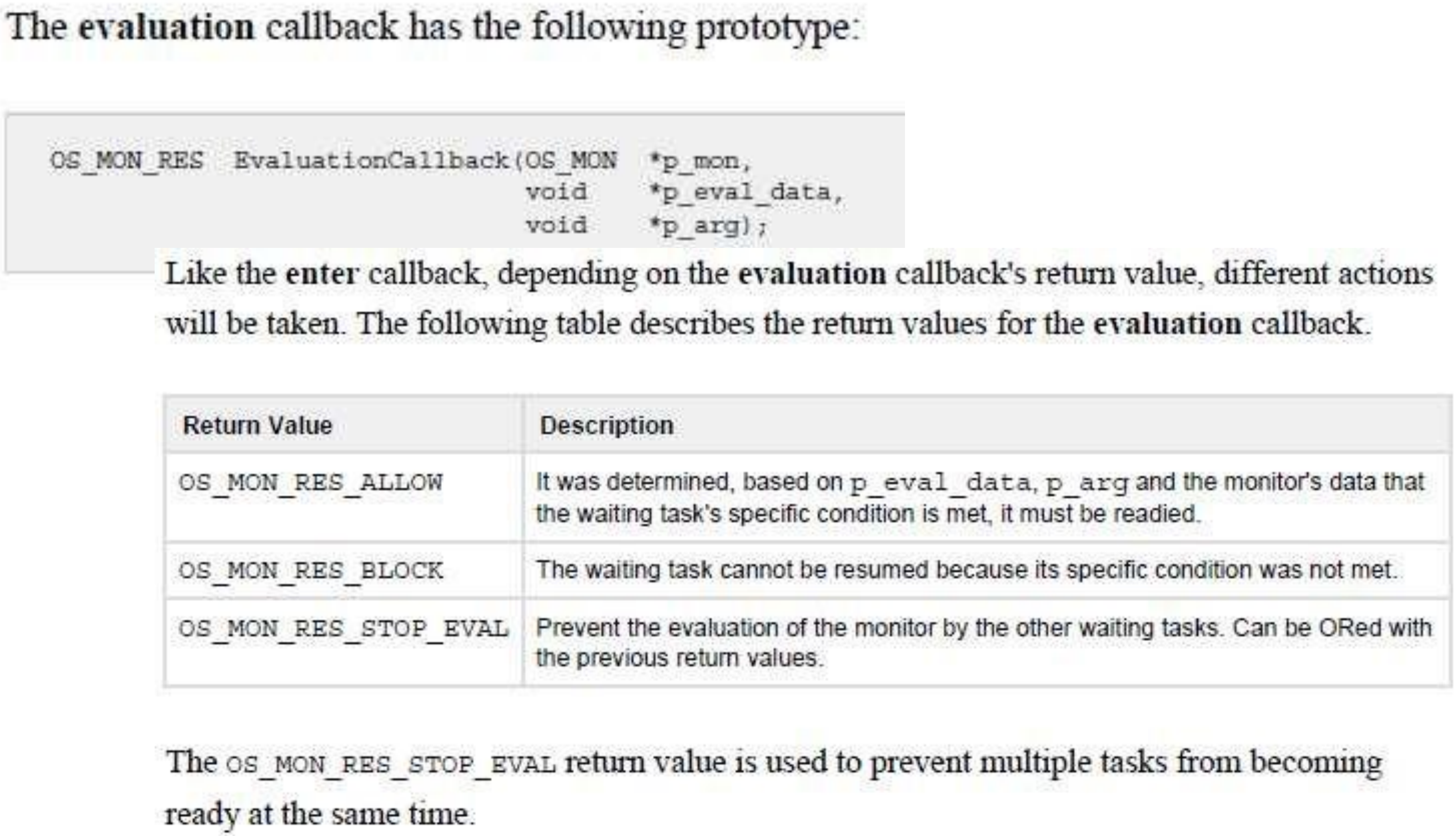
```
OS_MON_RES  EnterCallback(OS_MON   *p_mon,
                          void     *p_data);
```

Depending on the callback's return value, different actions will be taken. The following table describes the return values for the **enter** callback.



Return Value	Description
`OS_MON_RES_ALLOW`	The calling task will not be blocked.
`OS_MON_RES_BLOCK`	The calling task will be blocked. The `p_on_eval` argument passed to `OSMonOp ()` will be saved as the task's **evaluation** callback. `OSMonOp ()`'s `p_arg` argument will be saved as the task's evaluation data argument.
`OS_MON_RES_STOP_EVAL`	Returning this will prevent the **evaluation** callback of the waiting tasks from being executed. Can be ORed with the previous return values.

If the `p_enter` argument of the `OSMonOp ()` call is `DEF_NULL`, the default return value of `OS_MON_RES_BLOCK | OS_MON_RES_STOP_EVAL` will be used instead.



The **evaluation** callback has the following prototype:

```
OS_MON_RES  EvaluationCallback(OS_MON  *p_mon,
                               void    *p_eval_data,
                               void    *p_arg);
```

Like the **enter** callback, depending on the **evaluation** callback's return value, different actions will be taken. The following table describes the return values for the **evaluation** callback.

Return Value	Description
`OS_MON_RES_ALLOW`	It was determined, based on `p_eval_data`, `p_arg` and the monitor's data that the waiting task's specific condition is met, it must be readied.
`OS_MON_RES_BLOCK`	The waiting task cannot be resumed because its specific condition was not met.
`OS_MON_RES_STOP_EVAL`	Prevent the evaluation of the monitor by the other waiting tasks. Can be ORed with the previous return values.

The `OS_MON_RES_STOP_EVAL` return value is used to prevent multiple tasks from becoming ready at the same time.

Homework

- Study the examples related to monitors from the reference manual

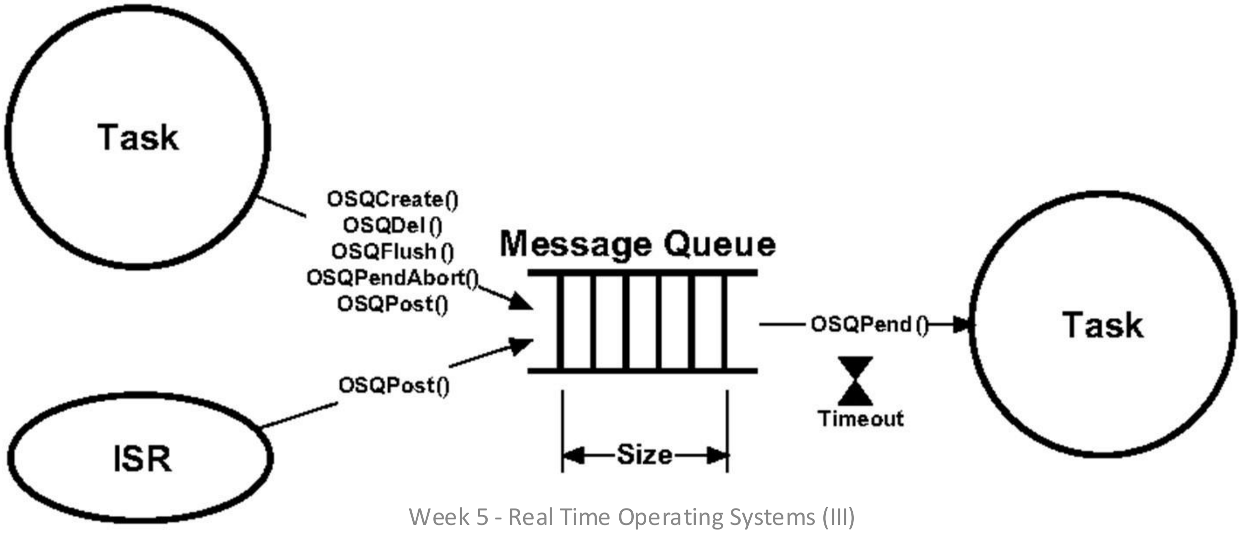
Inter-Task Communication, Message Passing

- How do we communicate information to other tasks?
- Ways to resolve inter-task communication:
  1. Global data … here we need to ensure exclusive access…
  2. Sending messages – messages can be sent to an intermediate object called a message queue or directly to a tasks since uC/OS-III implements built in task message queue
- What is a message?
  - A message consists of a pointer to data, a variable containing the size of the data pointed to, and a timestamp
  - The message contents must always remain in scope since the data is actually sent by reference instead of by value.

Message Queue

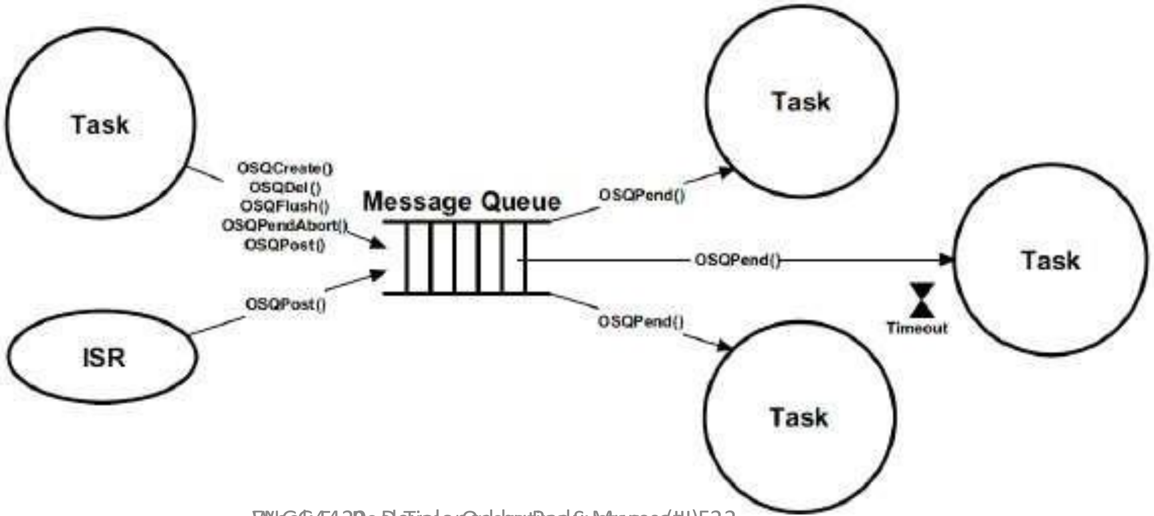
A message queue is a kernel object allocated by the application

- A message queue must be created first
- Multiple tasks can wait on a message queue



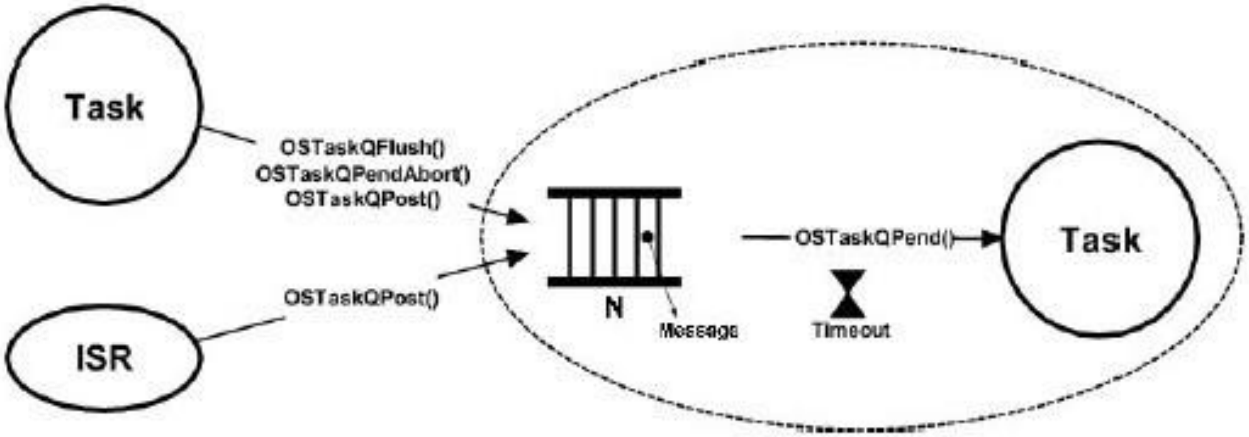
Multiple Tasks Waiting for a Message Queue

- When a message is sent to the message queue, what task will receive the message??
- The sender can broadcast a message to all tasks waiting on the message queue.
  - In this case, if any of the tasks receiving the message from the broadcast has a higher priority than the task sending the message it will execute after the message was posted



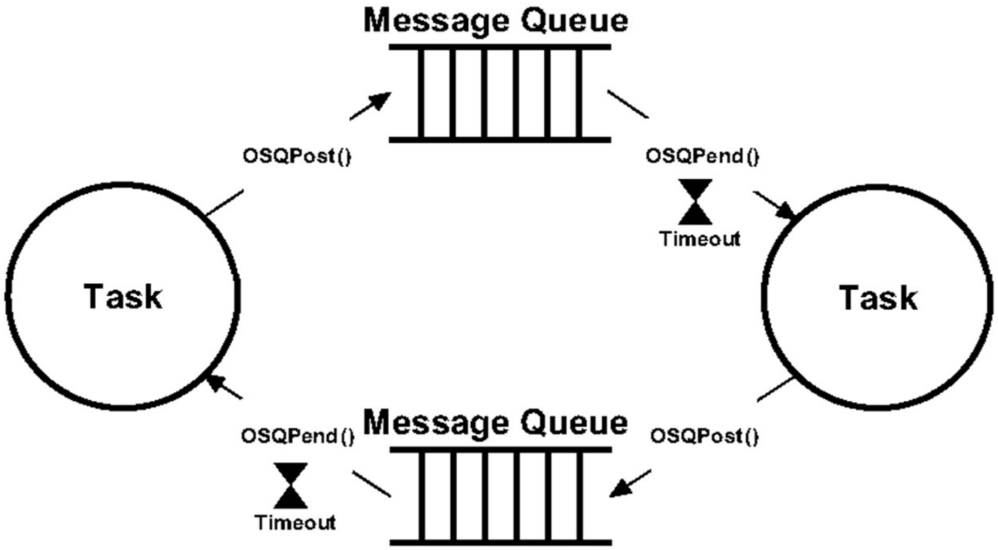
Task Message Queue

- A message queue is built into each task and a user can send messages directly to a task
- This feature can be more efficient than using a separate message queue object
- Setting OS_CFG_TASK_Q_EN to DEF_ENABLED in os_cfg.h enables task message queue services

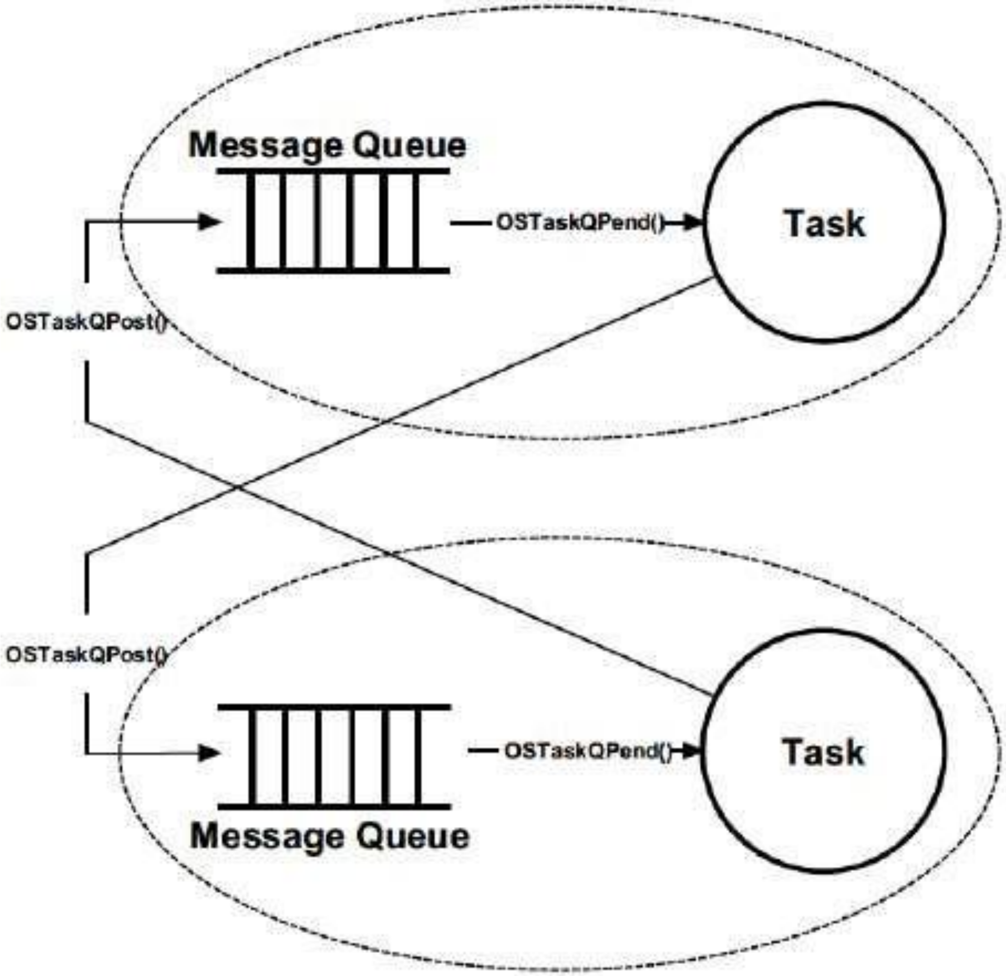


Example of Bilateral Rendez-Vous Using Message Queue

- Two tasks can synchronize their activities by using two message queues
- Each message queue holds a maximum of one message in this process
- Each task has its rendez-vous point:
  - Where it sends a message to the other task and waits for a message to come from that task
  - Notice the utilization of two queues

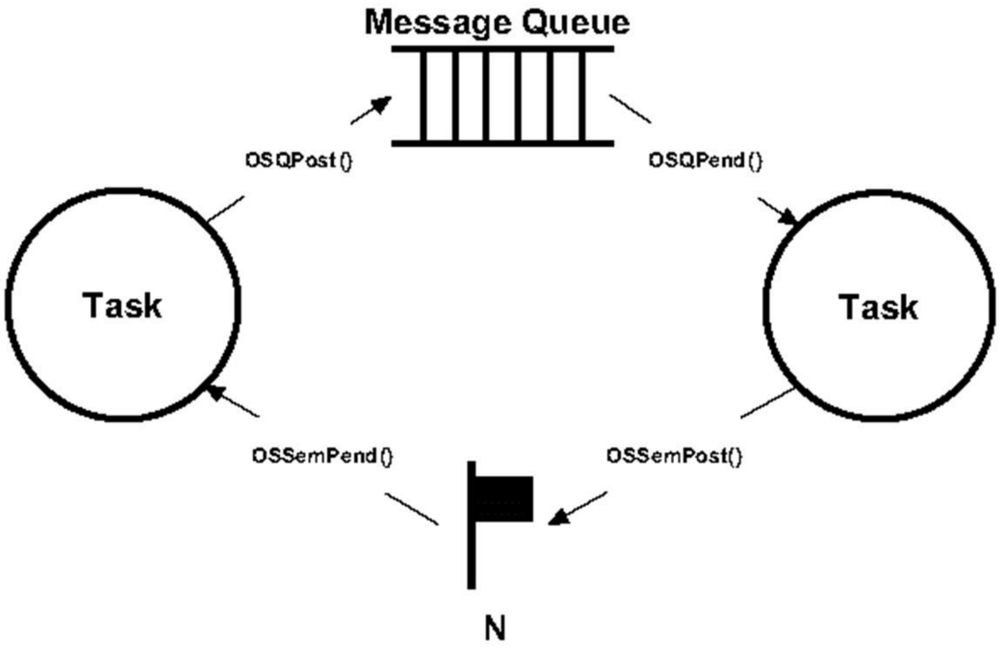


Task-Message Queues to Perform a Bilateral Rendez-Vous



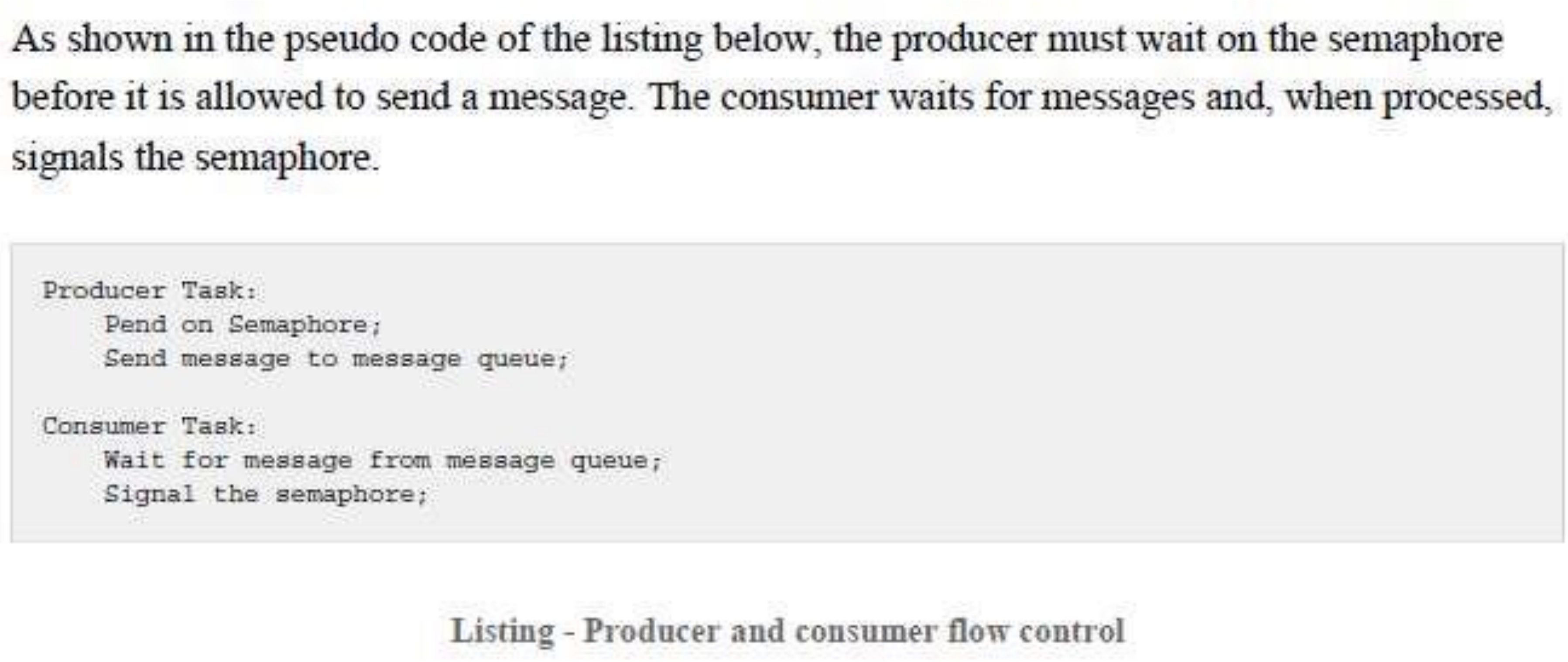
Flow Control

- Task-to-task communication can involve data transfers such that one task produces data while the other consumes it.
- Due to time differences in producing and processing times it is possible for the producer to overflow the message queue
  - Specifically, if a higher-priority task preempts the consumer.
- Solutions: add flow control in the process as shown …?
  - Message queue and counting semaphore
  - Built in task objects



Message queue and counting semaphore solution

Flow Control with Counting Semaphore



As shown in the pseudo code of the listing below, the producer must wait on the semaphore before it is allowed to send a message. The consumer waits for messages and, when processed, signals the semaphore.

```
Producer Task:
    Pend on Semaphore;
    Send message to message queue;

Consumer Task:
    Wait for message from message queue;
    Signal the semaphore;
```

Listing - Producer and consumer flow control

Flow Control with Built in Task Objects

In this case, OSTaskSemSet()

must be called immediately after creating the task to set the value of the task semaphore to the same value as the maximum number of allowable messages in the task message queue

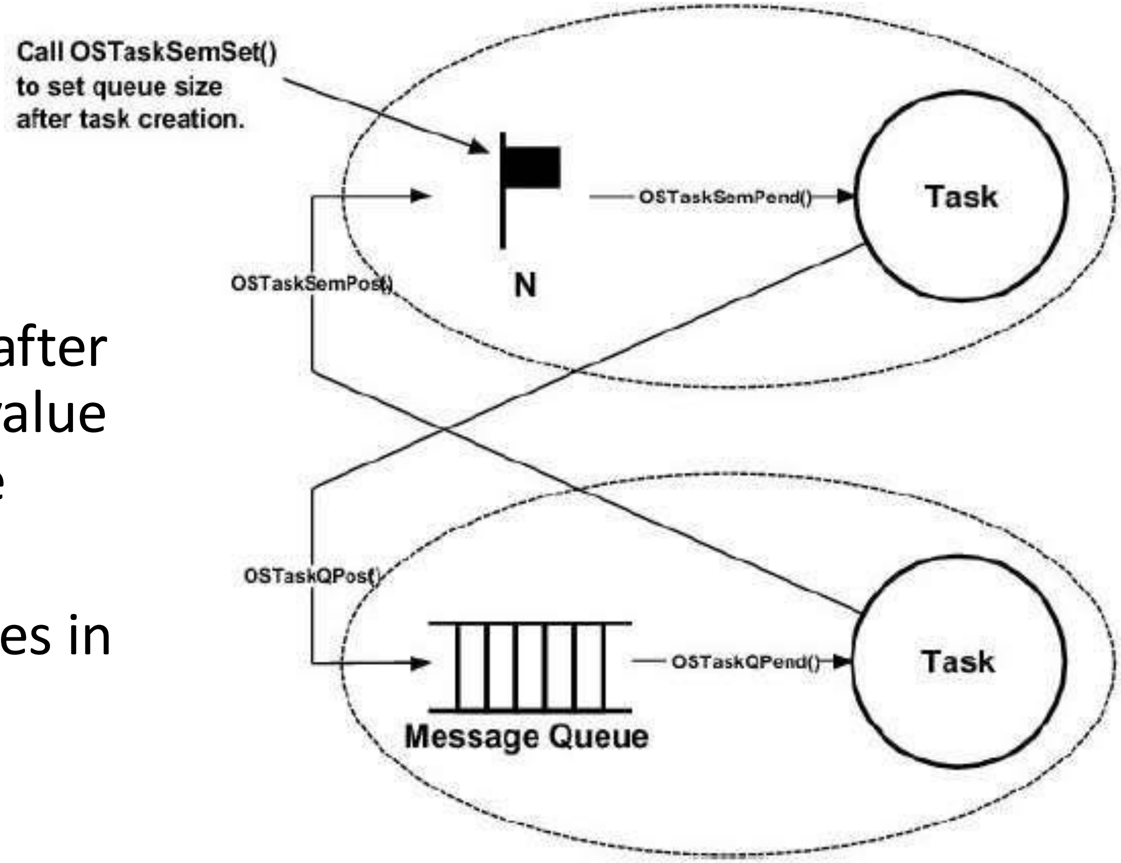
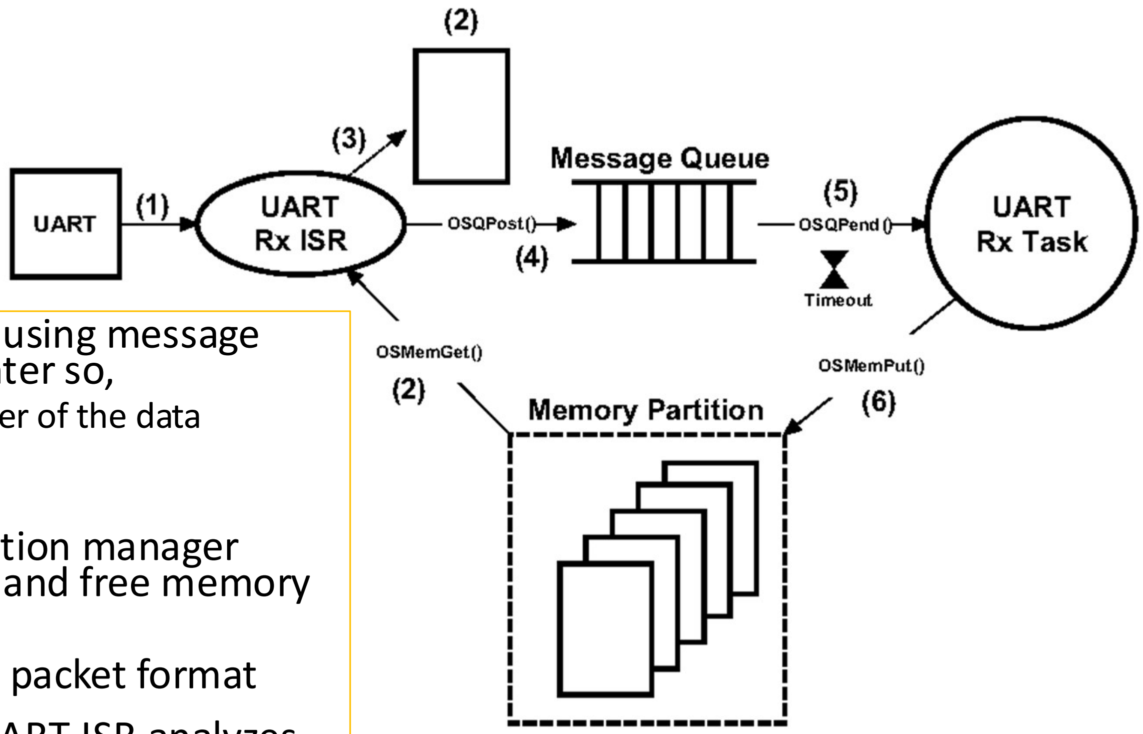


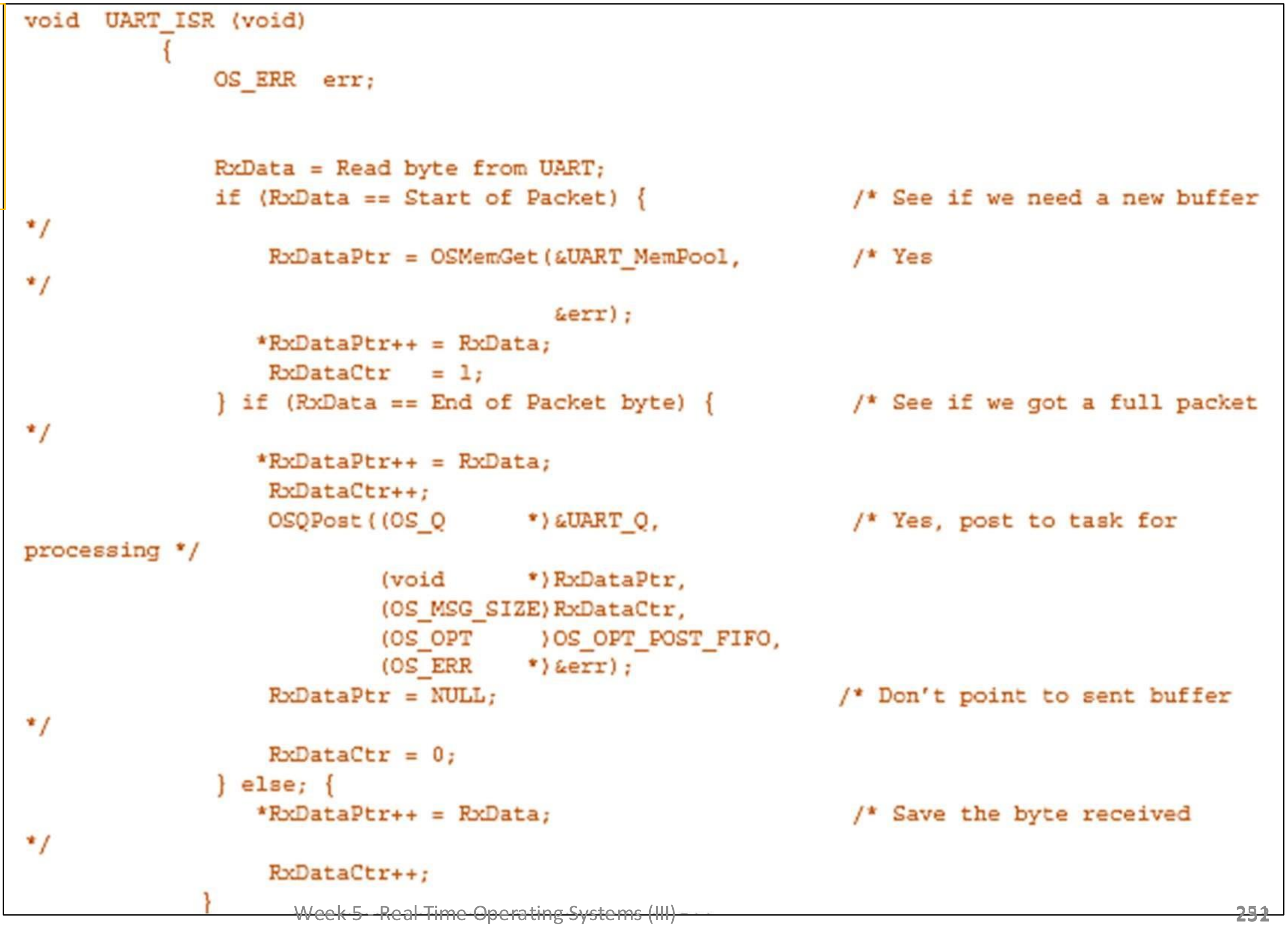
Figure - Flow control with task semaphore and task message queue

Keeping the Data in Scope



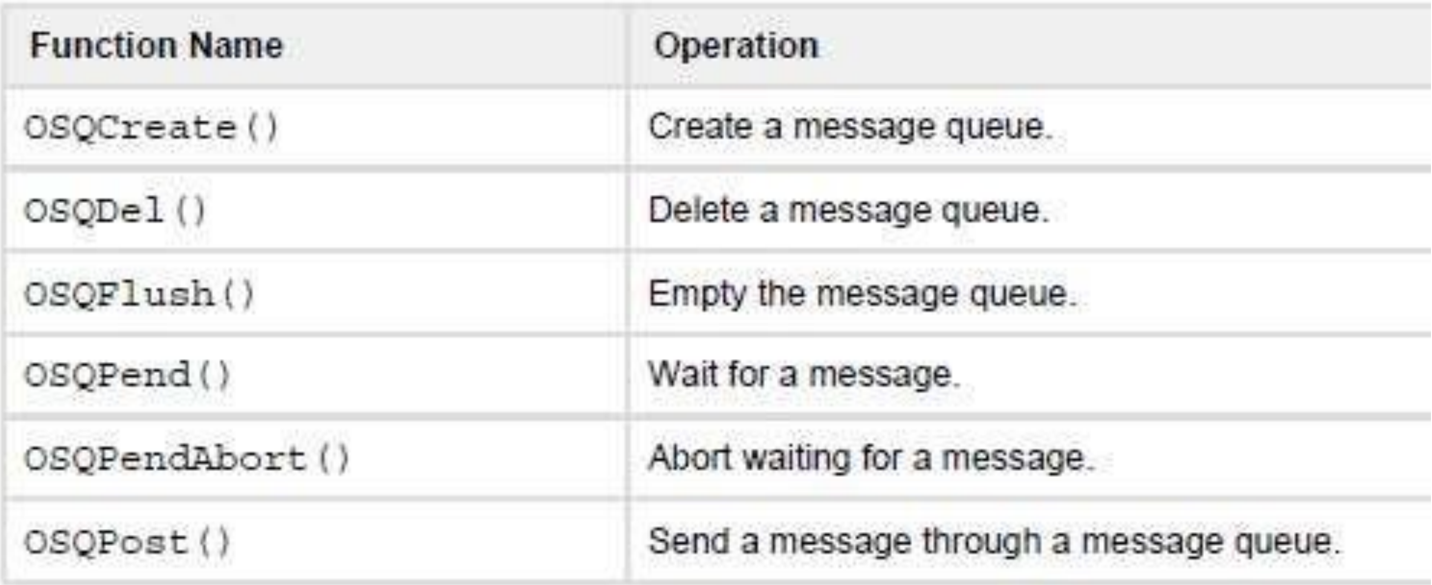
- In the case of inter-task communications using message queues, the messages are in form of pointer so,
  - the data must remain static until the receiver of the data completes its processing
  - The sender must not touch the sent data
- Solution: use the fixed-size memory partition manager provided with OS to dynamically allocate and free memory blocks used to pass the data
- Example: UART device sends messages in packet format
- (1) … a UART generates an interrupt … UART ISR analyzes the character for start-of-packet, end-of-packet or packet body character
  - (2): … start-of-packet … a new buffer is obtained …
  - (3) … packet character… place the byte in the buffer …
  - (4) … end-of-packet … post the address of the buffer …
- (5) … when the Rx task is HPT, retrieve the packet …
- (6) … after Rx task finished, return the buffer …

UART_ISR pseudo-code

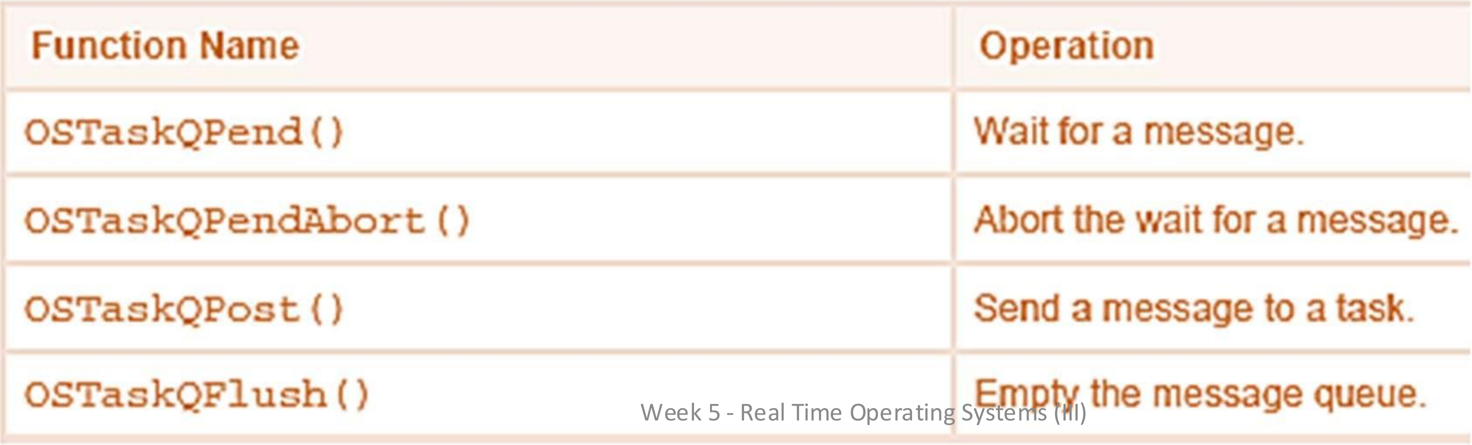


```
void  UART_ISR (void)
          {
              OS_ERR  err;


              RxData = Read byte from UART;
              if (RxData == Start of Packet) {                /* See if we need a new buffer
*/
                  RxDataPtr = OSMemGet(&UART_MemPool,         /* Yes
*/
                                       &err);
                 *RxDataPtr++ = RxData;
                  RxDataCtr   = 1;
              } if (RxData == End of Packet byte) {           /* See if we got a full packet
*/
                 *RxDataPtr++ = RxData;
                  RxDataCtr++;
                  OSQPost((OS_Q      *)&UART_Q,               /* Yes, post to task for
processing */
                          (void      *)RxDataPtr,
                          (OS_MSG_SIZE)RxDataCtr,
                          (OS_OPT     )OS_OPT_POST_FIFO,
                          (OS_ERR    *)&err);
                  RxDataPtr = NULL;                          /* Don't point to sent buffer
*/
                  RxDataCtr = 0;
              } else; {
                 *RxDataPtr++ = RxData;                       /* Save the byte received
*/
                  RxDataCtr++;
              }
```

Services for Message Queues and Task Message Queues



Function Name	Operation
OSQCreate()	Create a message queue.
OSQDel()	Delete a message queue.
OSQFlush()	Empty the message queue.
OSQPend()	Wait for a message.
OSQPendAbort()	Abort waiting for a message.
OSQPost()	Send a message through a message queue.



Function Name	Operation
OSTaskQPend()	Wait for a message.
OSTaskQPendAbort()	Abort the wait for a message.
OSTaskQPost()	Send a message to a task.
OSTaskQFlush()	Empty the message queue.

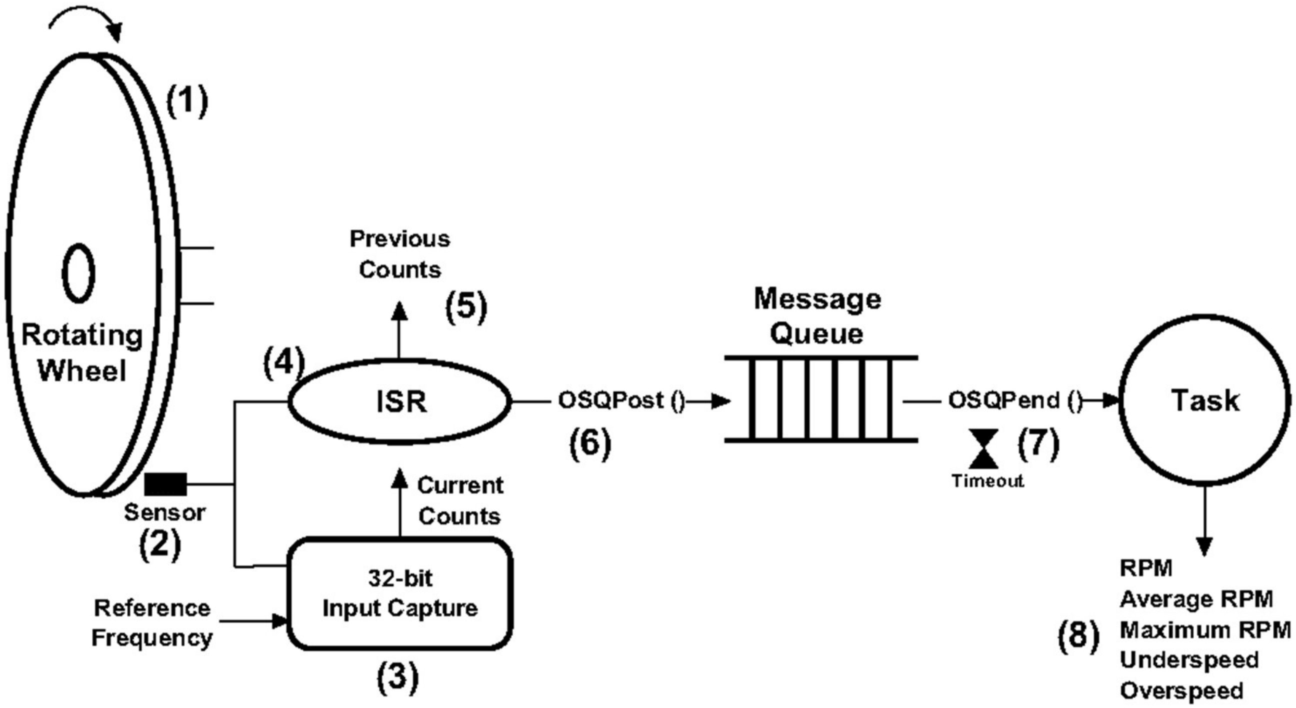
Example of Using Message Queue

- Application: RPM measurements on a rotating wheel

(1) … wheel rotates …

(2): … a sensor is used to detect the passage of a hole in the wheel …

(3) … 32-bit input capture …



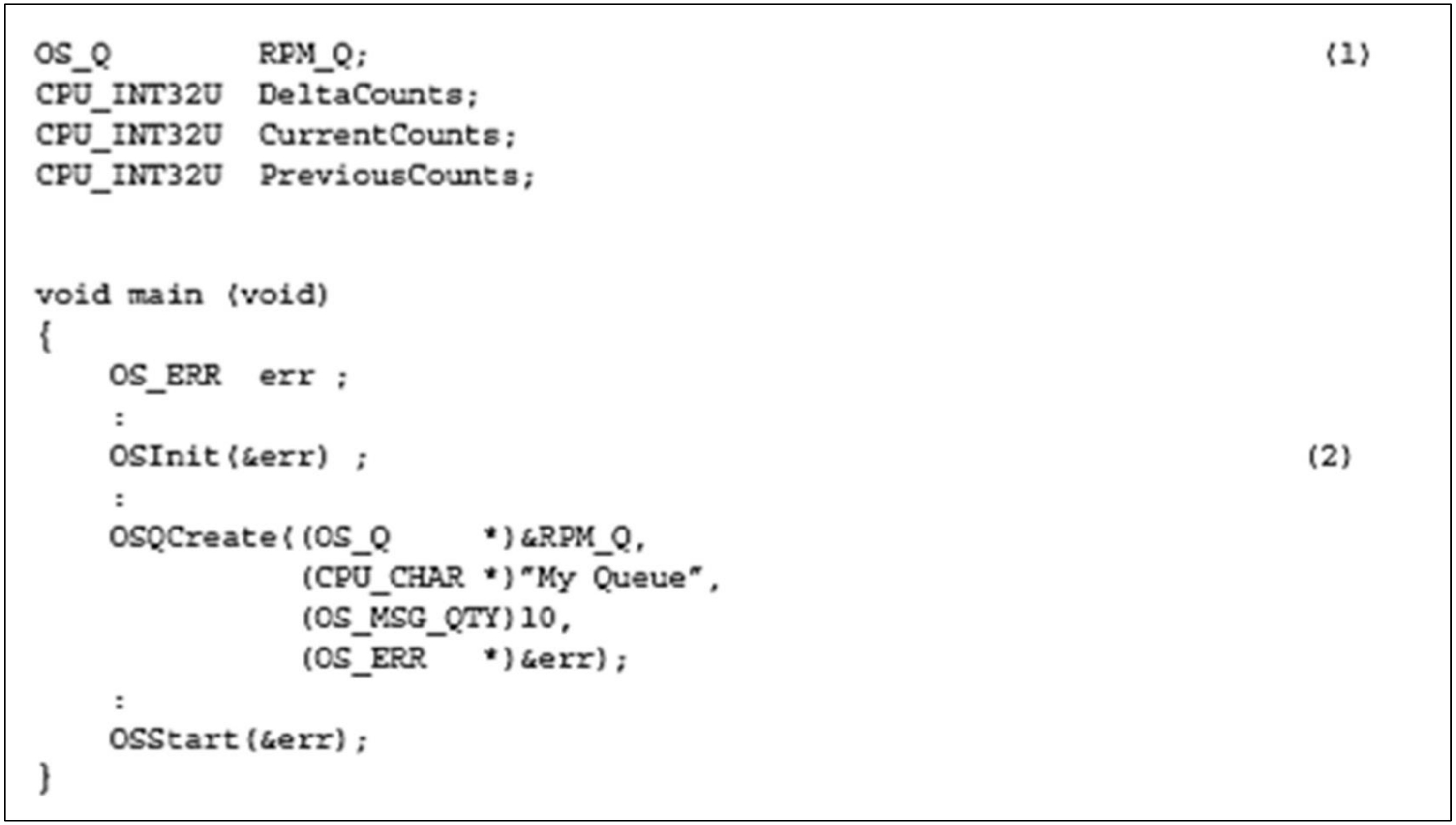
(4)… ISR reads the counter and calculates: ΔCounts = Current Counts – Previous Counts …

(5)-(6) … ΔCounts are sent to a message queue … in this case cast ΔCounts to a pointer …

(7): … RPM measurement task calculates: RPM = 60 * (Reference Frequency)/(Δcounts) …

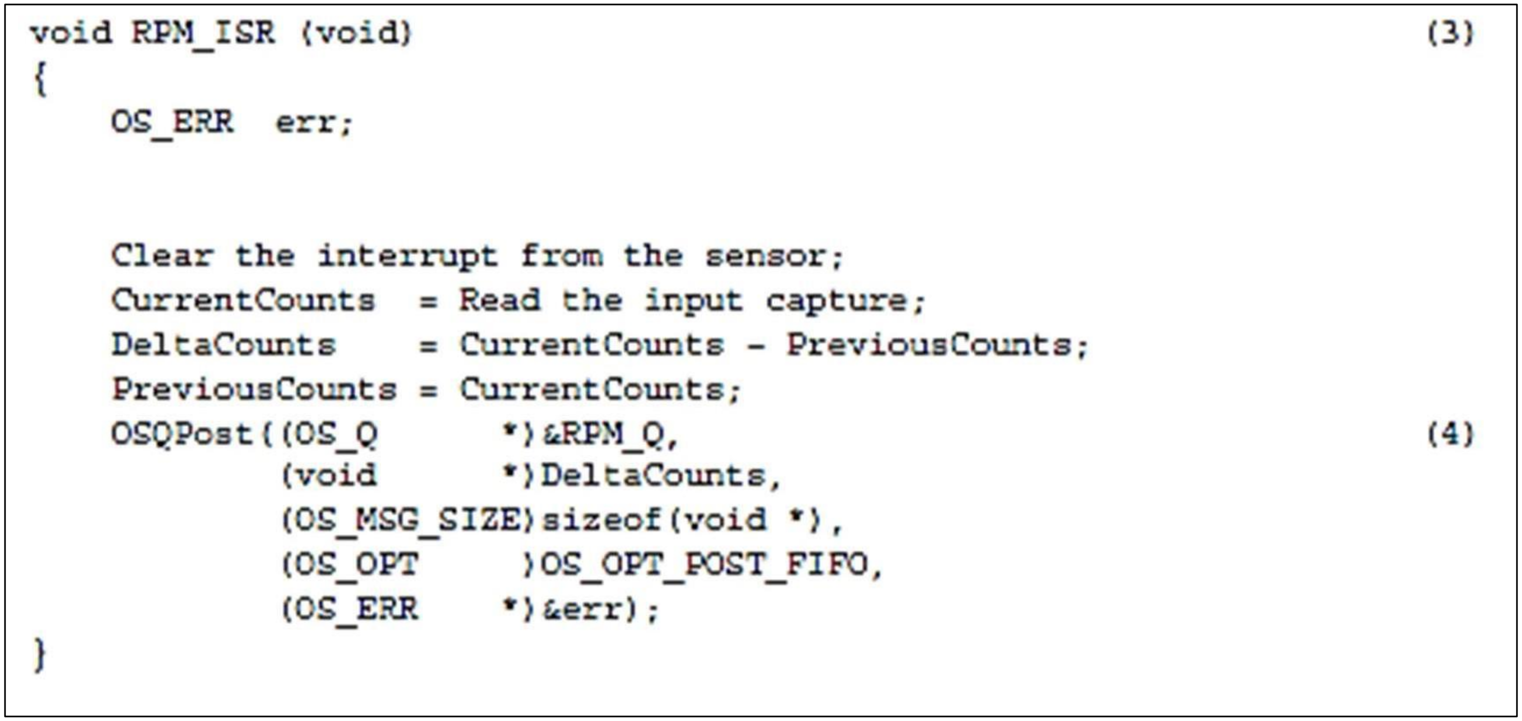
(8) … other metrics such as average RPM, max RPM, etc. can be calculated …

Pseudo Code main ()



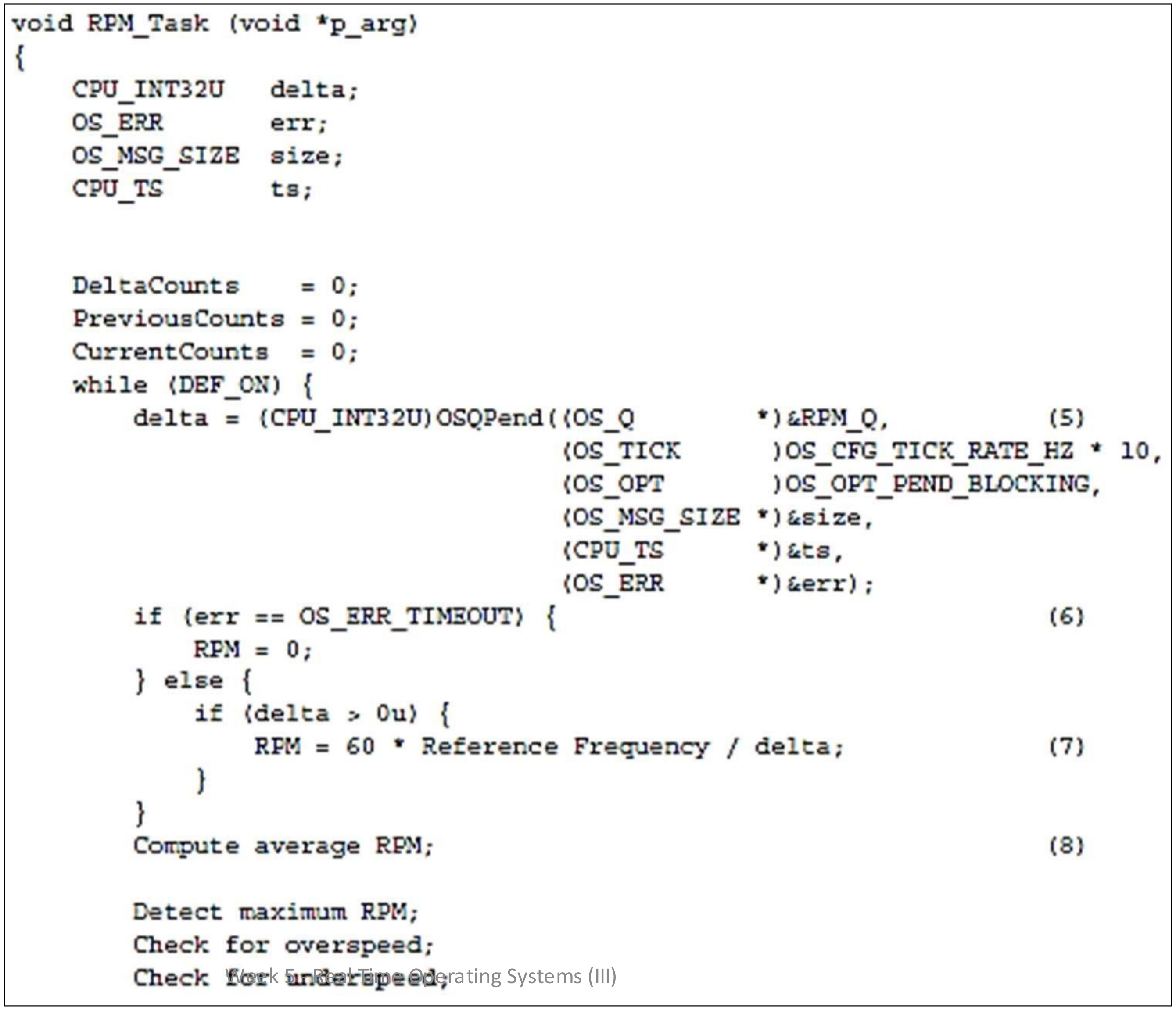
```
OS_Q        RPM_Q;                                                    (1)
CPU_INT32U  DeltaCounts;
CPU_INT32U  CurrentCounts;
CPU_INT32U  PreviousCounts;


void main (void)
{
    OS_ERR  err ;
    :
    OSInit(&err) ;                                                    (2)
    :
    OSQCreate((OS_Q      *)&RPM_Q,
              (CPU_CHAR *)"My Queue",
              (OS_MSG_QTY)10,
              (OS_ERR    *)&err);
    :
    OSStart(&err);
}
```

Code ISR



```
void RPM_ISR (void)                                                         (3)
{
    OS_ERR  err;


    Clear the interrupt from the sensor;
    CurrentCounts  = Read the input capture;
    DeltaCounts    = CurrentCounts - PreviousCounts;
    PreviousCounts = CurrentCounts;
    OSQPost((OS_Q      *)&RPM_Q,                                            (4)
            (void      *)DeltaCounts,
            (OS_MSG_SIZE)sizeof(void *),
            (OS_OPT     )OS_OPT_POST_FIFO,
            (OS_ERR    *)&err);
}
```

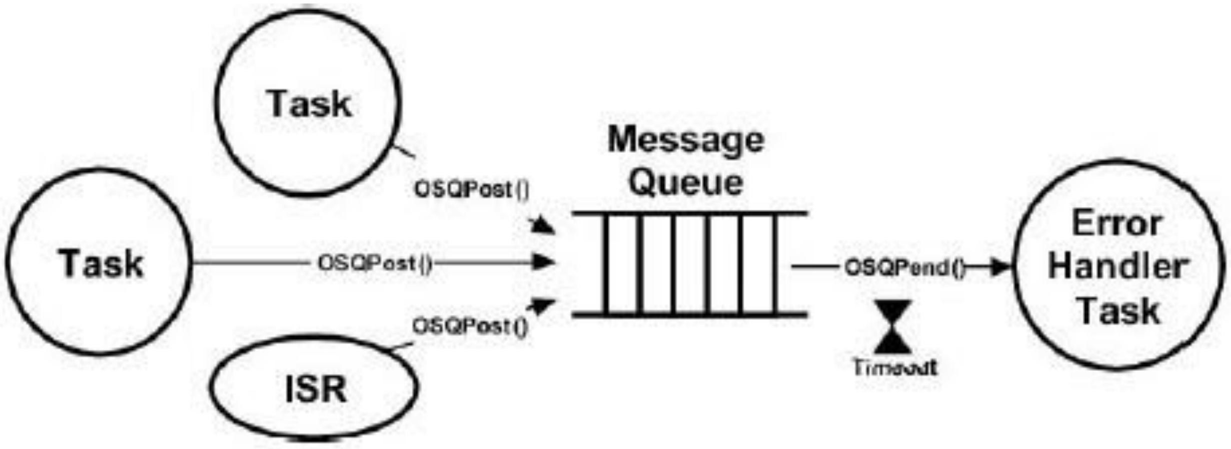
Code Task



```
void RPM_Task (void *p_arg)
{
    CPU_INT32U   delta;
    OS_ERR       err;
    OS_MSG_SIZE  size;
    CPU_TS       ts;


    DeltaCounts    = 0;
    PreviousCounts = 0;
    CurrentCounts  = 0;
    while (DEF_ON) {
        delta = (CPU_INT32U)OSQPend((OS_Q        *)&RPM_Q,              (5)
                                    (OS_TICK      )OS_CFG_TICK_RATE_HZ * 10,
                                    (OS_OPT       )OS_OPT_PEND_BLOCKING,
                                    (OS_MSG_SIZE *)&size,
                                    (CPU_TS      *)&ts,
                                    (OS_ERR      *)&err);
        if (err == OS_ERR_TIMEOUT) {                                    (6)
            RPM = 0;
        } else {
            if (delta > 0u) {
                RPM = 60 * Reference Frequency / delta;                 (7)
            }
        }
        Compute average RPM;                                            (8)

        Detect maximum RPM;
        Check for overspeed;
        Check for underspeed;
```

Message Queue for Clients And Servers Application

- Here, a task (the server) is used to monitor error conditions that are sent to it by other tasks or ISRs (clients) …
- When the clients detect error conditions, they send the message error through a message queue.



Message Queue Internals

A message consists of:

- a pointer to actual data,
- a variable indicating the size of the data being pointed to and
- a timestamp indicating when the message was sent.

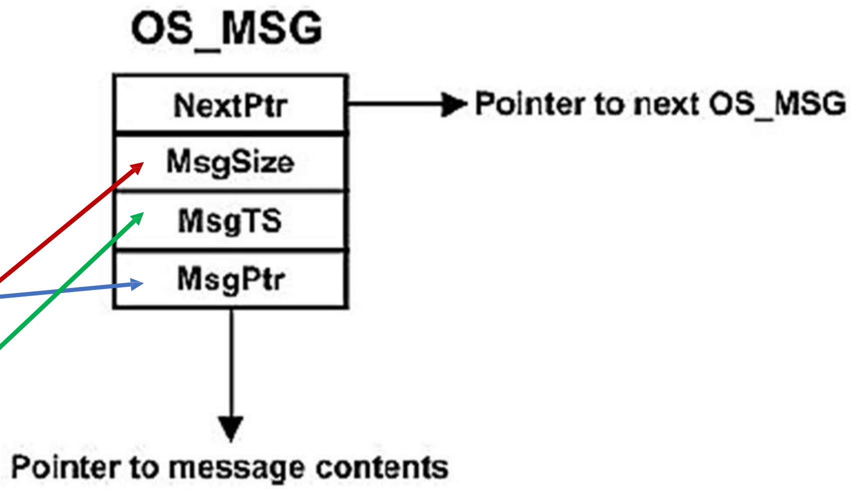


Figure - OS_MSG structure

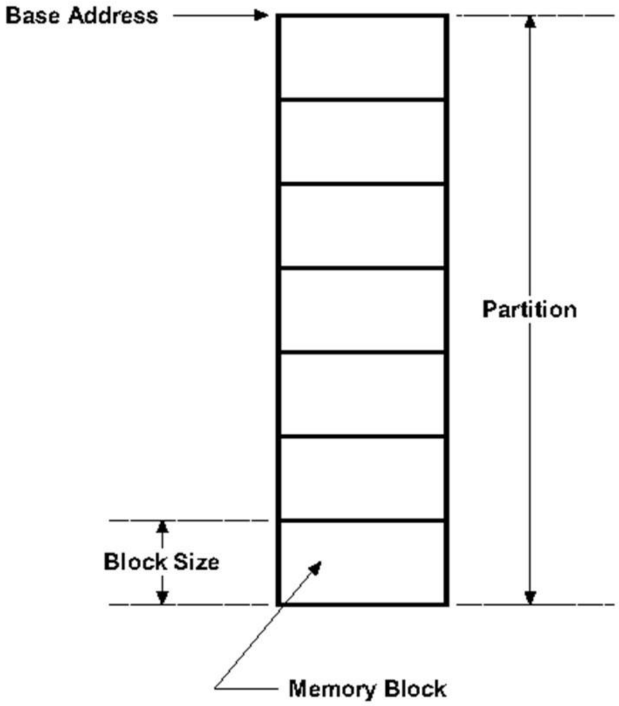
- µC/OS-III maintains a pool of free OS_MSGs.
- The total number of available messages in the pool is determined by the value of OS_CFG_MSG_POOL_SIZE found in os_cfg_app.h.
- When µC/OS-III is initialized, OS_MSGs are linked in a single linked list
- See the reference manual for the complete data structure for queues!

Homework

- Use the RPM program example from the lecture notes to build new applications such as: measure temperature, control the speed of a motor, measures the energy consumption of a device by measuring current consumption and voltage at a rate of 40 us, etc.
  - Provide the task diagram similar to the one in Slide 23
  - Provide the uC/OS-III code functionality for the application
- Note, this type of application could be a good test problem for the final exam

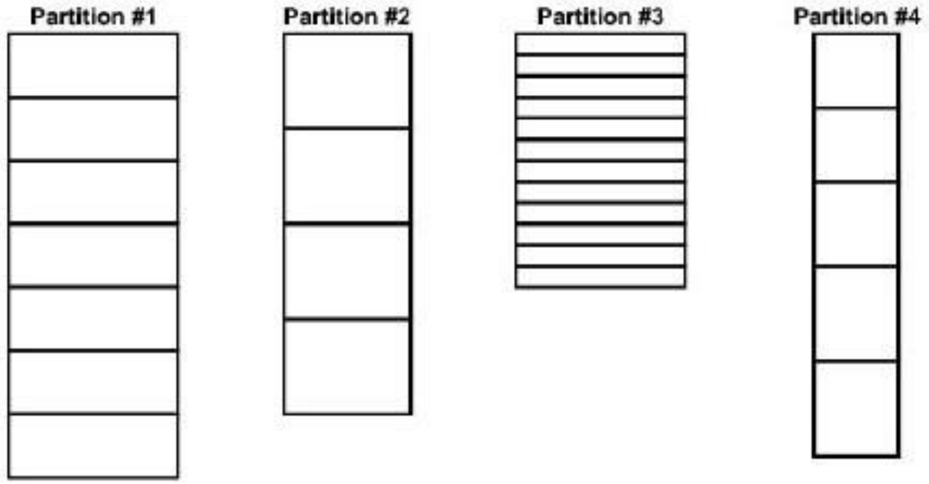
Memory Management

- uC/OS-III provides an alternative to malloc() and free() by allowing an application to obtain fixed-sized memory blocks from a partition made from contiguous memory area
- All memory blocks are the same size, and the partition contains an integral number of blocks.
    - Allocation and deallocation of these memory blocks is performed in constant time and is deterministic.
- The partition itself is typically allocated statically (as an array), but can also be allocated by using malloc() as long as it is never freed

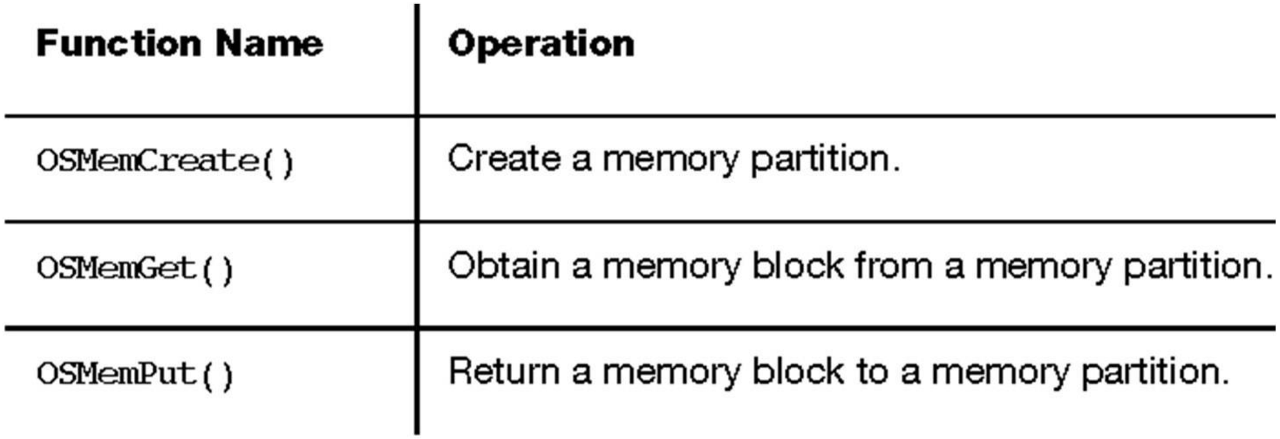


Multiple Memory Partitions

- More than one memory partition may exist in one application
- An application can obtain memory blocks of different sizes based upon requirements.
- But, a specific memory block must always be returned to the partition that it came from.



Using Memory Partitions



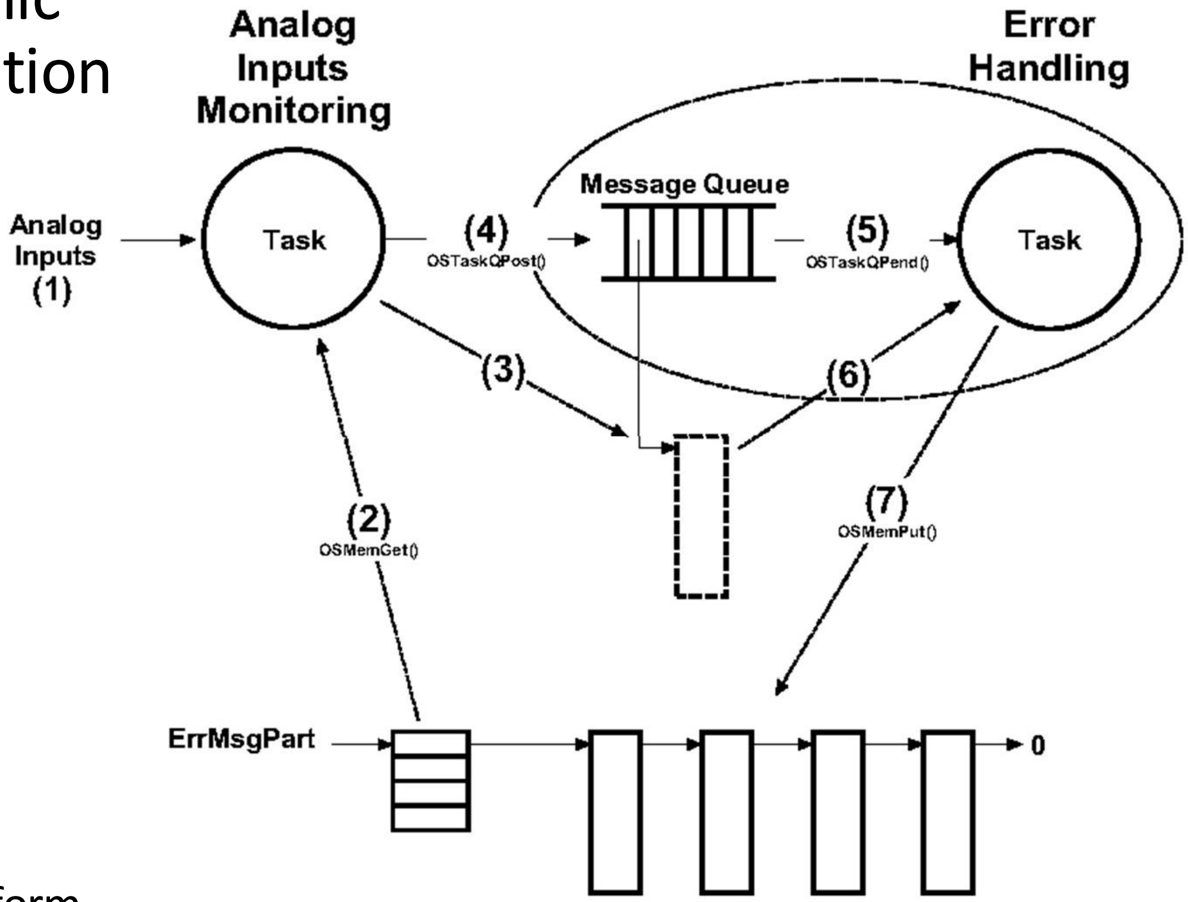
Function Name	Operation
OSMemCreate()	Create a memory partition.
OSMemGet()	Obtain a memory block from a memory partition.
OSMemPut()	Return a memory block to a memory partition.

- Memory management services are enabled at compile time by setting OS_CFG_MEM_EN to 1 in os_cfg_h
- We must create a memory partition first before we can use it
- OSMemCreate() can only be called from task-level code, but OSMemGet() and OSMemPut() can be called from ISR as well

Example of Using Dynamic Memory Allocation Partition

Application: A task monitors the value of analog inputs and sends an error message to an error handling task if any of the analog inputs exceed a threshold.

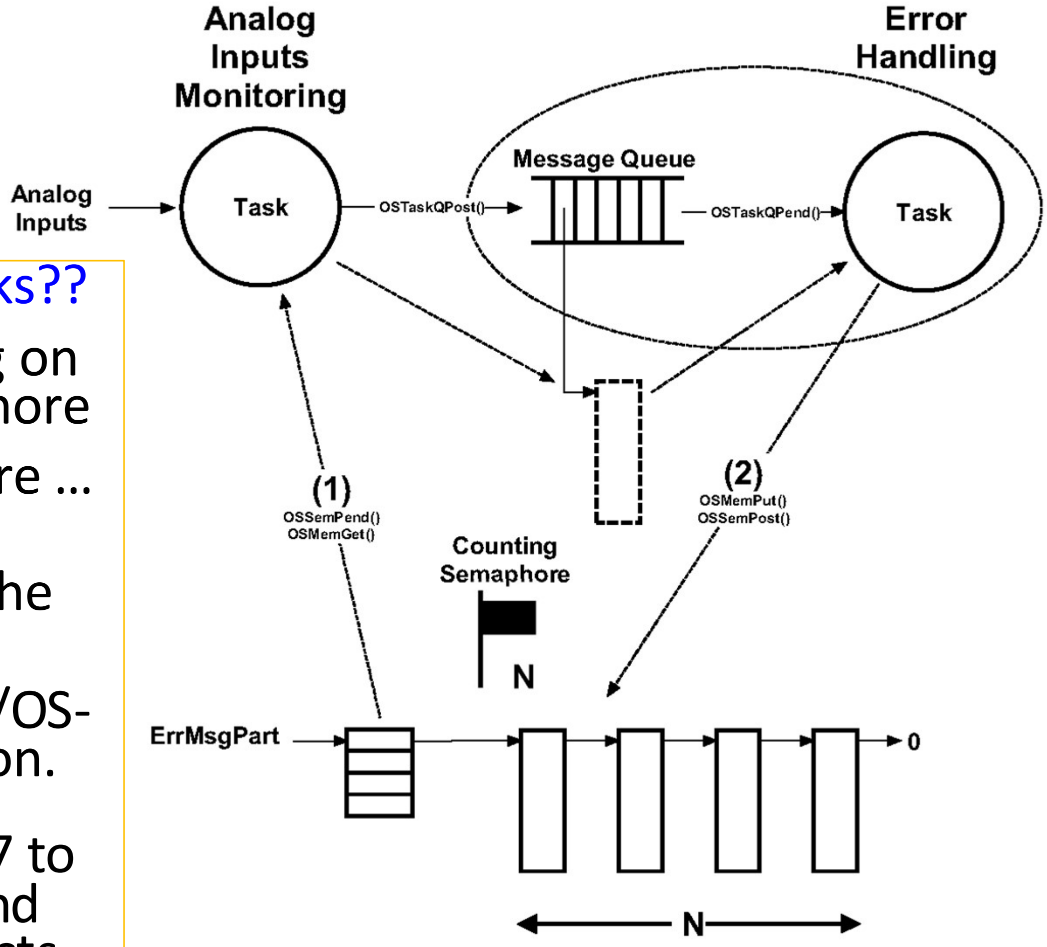
- Error handling is centralized.



- (1) … inputs read and analyzed …
- (2) … obtain a memory block …
- (3) … from the error message …
- (4) … post the error message …
- (5) … pend on error messages …
- (6) … analyze the message … perform actions required …
- (7) … return the memory block …

Task Waiting for a Memory Block

- What if a partition runs out of blocks??
- µC/OS-III does not support pending on partitions … add a counting semaphore
- (1) … obtain the partition semaphore … then call OSMenGet() …
- (2) … return the block and post to the semaphore …
- Homework: present a complete uC/OS-III functional code for this application. Use the example pp. 318 and the memory partition examples (pp 327 to 331). You need to show the tasks and all uC/OS-III function calls and objects utilized.



uC/OS-III Homework

1. Study all uC/OS-III topics, the examples and the posted homework throughout the lecture slides
2. Understand the initialization and start-up for using uC/OS-III and how to use tasks and objects (semaphores, mutex, flags, monitors, queues)
3. Understand task states, tick usage, interrupts, synchronization, inter-task communication memory management
4. Understand priority inversion protocol and deadlock,
5. Usage of: semaphore, mutex, event flags, queue, memory partitions
6. Complex applications using memory allocation services should be well understood well and how to utilize the uC/OS-III functions to build such applications!!!