Micriμm

Empowering Embedded Systems

μC/OS-II

Inter-Process Communication

Quick Start Guide

Application Note

AN-1005

www.Micrium.com

Table Of Contents

1.00 Introduction

Inter-process communication provides a mechanism for synchronizing and passing messages between tasks. The following IPC primitives are supported within µC/OS-II:

- Semaphores
- Message Boxes
- Message Queues
- Mutexes
- Event Flags

The use of Semaphores, Message Boxes and Message Queues, with design examples is described below. Please see AN1002 'uCOS-II-Mutex' and AN1007 'uCOS-II-EventFlags' for a full description of µC/OS-II Mutexes and Event Flags.

2.00 Semaphores

Semaphores may be used to 'signal' a task when an event occurs. Events may be in the form of an interrupt, or another task signaling a different task that a computation has been completed. Generally, it is best to keep Interrupt Service Routines (ISR's) as short as possible. Therefore in practice, if a lengthy ISR is necessary, then the system designing should signal a task when the interrupt occurs, exit the ISR, and allow the processing that would have occurred within the ISR to occur at task level.

Example 1: A packet is received on an Ethernet interface or a user pushes a user interface button. In both cases the application response requires a complex set of operations.

Solution 1: Create an Ethernet receive / user interface task and have it pend (block) on a semaphore. When the interrupt occurs, post the 'receive ready / button pushed' semaphore and exit the ISR. Posting the semaphore causes the receive / user interface task to enter the ready state. Depending on the priority of the task, µC/OS-II may schedule the task to run upon exiting the ISR if no other higher priority tasks are ready to run. Otherwise, the task will run when no other higher priority task is ready and waiting for processor time.

Step(s):

1. Declare an `OS_EVENT` type variable that is accessible 'C' scope wise from both the ISR handler code and the task that needs to pend on it. Of course, reference to the semaphore could be passed instead. In many cases, both the task to handle the event and the 'C' ISR handler are located in the same file, therefore, it is common practice to declare the semaphore as a static global variable at the top of that file.

```
static  OS_EVENT  *signal_sem;
```

2. Prior to entering the task body (loop), create a semaphore that will be used to wait for external events to occur.

```
signal_sem = OSSemCreate(0);
```

The argument to `OSSemCreate()` is an `INT16U` specifying the number of resources that are available. When the count is 0, the semaphore can be used like a communications signal. When a task pends on a semaphore that has a count of 0, it will be blocked until another task or ISR posts to the semaphore; thus the semaphore acts as a communication mechanism, or signal to wake up the pending task when the desired event occurs.
Notice that the return value of OSSemCreate() is reference to the newly created semaphore. Therefore, it is always GOOD PRACTICE to ensure that the semaphore was created successfully before proceeding. If an error occurs while creating the semaphore,

then the application requirements must dictate what action should be taken. In some cases, it's sufficient to delay the task for an arbitrary amount of time and try to create the semaphore again. This action may be performed in a loop until the semaphore is successfully created. Alternatively, a system reboot could be used if the error is determined to be of a critical nature for the application being developed. If no error occurs, then the application may continue normally, finish additional initialization of required objects and enter the task body (loop).

```
if (signal_sem == ((void *)0) {
   /* Failed to create semaphore */
}
```

3. Block the task by pending on the semaphore. This may be done prior to entering the task body if the desire is to prevent the task from running until initialization of another task is complete. However, very often the task will block waiting for an event, wake up to process the event, and then block itself again indefinitely within the context of the tasks body.

```
while (DEF_TRUE) {
    OSSemPend(&signal_sem, 0, &err);    (1)
    Perform_Work();                     (2)
}
```

1. Wait for a task or ISR to signal that an event has occurred.
2. Perform the necessary work to handle the event then loop and wait for the next event.

The arguments to OSSemPend() are a pointer to the semaphore, an optional timeout in units of clock ticks where specifying 0 blocks indefinitely until the event occurs, and a pointer to an error variable. The error variable should be declared at the top of your task function as a local variable.

```
CPU_INT08U  err;
```

As mentioned above, it is always GOOD PRACTICE to check return error codes from function calls that return errors either as return values or as a pointer to an error code. If your call to `OSSemPend()` is correct, meaning, you have not passed any NULL arguments, it is unlikely that the call will fail.

```
if (err != OS_ERR_NONE) {
    /* Perform error code here */
}
```

Of course other errors, some of which are not critical, may be returned and should be handled on a per case basis. In some cases, using a switch statement to handle the various possible return codes is desirable.

4. A separate task or ISR handler needs to wake up the task when an event occurs by posting to the semaphore that the task is waiting on. As always, error checking should be performed. `OSSemPost()`returns a `CPU_INT08U` that will be equivalent to `OS_ERR_NONE` if no errors are detected. Again, if the semaphore has been properly initialized and is non NULL, it is unlikely that the call to `OSSemPost()` will fail.

```
CPU_INT08U  result;

result = OSSemPost(&signal_sem);
if (result != OS_ERR_NONE) {
    /* Perform error handling */
}
```

Alternatively, semaphores can be used to guard resources against concurrent access.

Example 2: A serial port is available to the entire system. Any task may transmit or receive data from the serial port when desired. However, if the serial port is not protected, then data transmitted and received from / to multiple tasks may become interleaved in the same serial stream. Therefore, once a task has gained access to the serial port, a higher priority task MUST NOT be allowed to gain access to the port if and when preemption occurs. If a higher priority task preempts a lower priority task that owns the serial port resource, then the higher priority task must wait for the resource to become available.

Solution 2: Protect the serial port API functions from concurrent access by implementing a semaphore that the API functions pends on before accessing the serial port functionality. One semaphore may be used to protect an entire modules set of API functions. In this sense, the semaphore operates like a lock or gate only allowing one task to enter the serial ports critical code at a given time.

Step(s):

1. Declare an `OS_EVENT` type variable that is accessible 'C' scope wise to all of the serial port API functions. Similar to the signaling semaphore, it is common practice to declare the semaphore at the top of the file as a static variable.

```
static  OS_EVENT  *lock_sem;
```

2. Create a semaphore within the modules initialization function.

```
CPU_INT08U  Module_Init (void) {
    lock_sem = OSSemCreate(1);
    if (lock_sem == ((void *)0) {
        /* Failed to create semaphore     */
        /* Return error to caller?        */
    }
}
```

The argument to `OSSemCreate()` is an `INT16U` specifying the number of resources that are available. When a count greater than 0 is specified, the first caller to pend on the semaphore will NOT be blocked. However, the count of the semaphore will be decremented and when the count reaches 0, the next caller WILL be blocked.

3. Have the API function pend on the semaphore without timeout. All module API functions that cannot be executed concurrently should be protected in the following manner using the same lock.

```
void  Serial_Tx (CPU_CHAR *data) {
    OSSemPend(&lock_sem, 0, &err);    (1)
    Serial_TxData();                  (2)
    OSSemPost(&lock_sem);             (3)
}
```

```
void  Serial_SetBaud (CPU_INT32U baud) {
    OSSemPend(&lock_sem, 0, &err);    (1)
    Serial_UpdateBaudRegisters();     (2)
    OSSemPost(&lock_sem);             (3)
}
```

1. Obtain the lock semaphore. If another task is already executing within this routine, and the semaphore is initialized to 1, then the caller will be blocked. Otherwise, it will take as many calls as there are resources (determined by the initial count of the semaphore) before the caller of the API function is blocked by `OSSemPend()`.
2. Perform the necessary work to transmit the data.
3. Release the lock allowing the next highest priority task that is waiting on the semaphore to gain exclusive access to the resource.

3.00 Message Boxes

Message boxes are used to send messages between tasks. However, a message may perform the same function as a semaphore used for signaling since a message may be posted from an ISR to wake up a task. The difference between a semaphore used as a signal and a message posted to a box is that the signal is inherently a boolean value; that is, the event either occurred or it has not yet occurred. However, when using message boxes, the content of the message may be specified. Therefore, it is possible to not only signal a task, but also indicate what event has occurred and supply relevant data to the task that will be woken up.

Messages sent between tasks take the form of void pointers. This means that the task can either allocate a variable as simple as a CPU_INT08U, or as complex as a multi-member structure and pass reference to that structure between tasks. Alternatively, since a pointer in and of itself requires storage space, it is possible to send a message between tasks where the address of the pointer is the data itself. Which ever method is used, it is important that both the sender(s) and receiver(s) of messages agree on the format in which the message will be sent; e.g. by pointer, or by casting the pointer directly to a value.

Example 3: Several tasks are used to monitor various sensors. The application must respond to sensory data in an organized manner and output a computed result set to a display. There are several design possibilities:

1. The application designer divides the display into sections and each task writes only to their designated area. The display API functions are protected from concurrent access using a semaphore as a locking mechanism.
2. Each task obtains data and posts the data via a message to a control task which can then process the data, perform calculations on several pieces of data and update the display when complete.

Method number 2 is conducive to using message boxes to create a central point of authority for the entire system. All data can be analyzed by the control task and displayed when necessary to any portion of the display. The control task can also wait on user interface buttons to change the content of the entire LCD in a page or menu like fashion, whereas in option number 1, each task would have to communicate with all of the other tasks in order to synchronize access to larger portions of the screen if that functionality is required.

Solution 3: Create a message box and have each task pass either a single data value or structure indicating the type of message and message content to an application control task.

Step(s):

1. Declare an `OS_EVENT` type variable that is accessible 'C' scope wise to all of the tasks that will pend or post messages from or to the message box. Often times, it is easiest to declare the message box at the top of the file as a static variable. Of course, reference to the message box can be passed to various tasks during task creation by means of the task argument, or by using the **extern** 'C' language keyword.

```
static  OS_EVENT  *msgbox;
```

2. Create a message box BEFORE creating application tasks that utilize it. Otherwise, the µC/OS-II message box API functions will likely return an error, or unpredictable behavior may occur. Of course, other methods of preventing early use of OS objects can be devised, for example, each application task requiring use of a shared message box could wait for an initialization complete signal to be posted.

```
CPU_INT08U  Application_Init (void) {
    msgbox = OSMBoxCreate((void *)0);
    if (msgbox == ((void *)0) {
        /* Failed to create message box   */
        /* Return error to caller?        */
    }

    AppTaskCreate();
}
```

The argument to `OSMBoxCreate()` is a void pointer to an initial message to deposit in the message box. The initial message may be specified as (void *)0 or NULL, thus blocking the first task to pend on the message box via `OSMBoxPend()`. However, if a message is specified as a non NULL value, the first task to pend on the message box will not be blocked and the message will be consumed. The next call to `OSMBoxPend()` will cause the caller to be blocked unless a new message is deposited in the box.

3. Have the control task pend on the message box thus causing it to be suspending until a message is received.

```
void  AppControlTask (void *p_arg) {
    void        *msg;
    CPU_INT16U  data;
    CPU_INT08U  err;


    while (DEF_TRUE) {
        msg = OSMoxPend(msgbox, 0, &err);   (1)
        switch (err) {                      (2)
            case OS_ERR_NONE:
            case OS_ERR_TIMEOUT:
            case OS_ERR_PEND_ABORT:
                 break;

            default:
                 break;
        }

        data = (CPU_INT16U)msg;             (3)

        /* Process message content            */
        /* Update LCD if applicable           */
    }
}
```

1. Wait for a message to be received.

2. Be mindful of non critical errors such as timeouts and aborts. It may be desirable for a task to specify a timeout and not block indefinitely waiting for a message (in case an error occurs in the task that posts messages). Likewise, it may be desirable for a task to abort the timeout of the control task in which case you may want to execute some different code if a pend abort is detected. In most many cases, tasks wait indefinitely for messages to be received and most often receive the OS_ERR_NONE code when unblocked.
3. Cast the message back to the correct data type agreed upon by the sender and the receiver / application developer, then process the data.

4. Have ISR's and other tasks post messages to the control task's message box.

```
void  AppSensor1Task (void *p_arg) {
    CPU_INT08U  err;
     CPU_INT16U  data;


    while (DEF_TRUE) {
        data = ReadSensor();                          (1)
        err  = OSMBoxPost(msgbox, (void *)data);   (2)
        if (err != OS_ERR_NONE) {                     (3)
            /* Try again? */
        }
    }
}
```

1. Create data to be passed to another task.
2. Post the data to the specified message box.
   a. WARNING: In this particular example, the message is passed directly within the storage area of the pointer. Pointers require different amounts of storage space depending on the architecture. For example, if a pointer (address) requires 16-bits of storage on a particular architecture, then a 32 bit message can not be casted directly as the pointer address and passed to the receiving task. However, a pointer to a structure that has more than 32 bits of data may be passed instead. The negative to passing pointers to data types, is that both the receiver and sender may attempt to modify the same memory concurrently thus making the operation unsafe. In order to guarantee that the message passing is safe, the sender must allocate a message from either the heap (discouraged for various reasons) or from a memory pool of some sort. The sender must NOT try to modify the memory that is passed by reference to the receiving task until the receiving task has finished processing the message and returns the memory to the pool.
3. Check for errors and take appropriate action if an error occurs. In most cases, `OSMBoxPost()` will return `OS_ERR_NONE`. However, `OS_ERR_MBOX_FULL` may be returned if the receiving task has not processed the pending message in the box. In this case, the sending task should attempt to send the message again, or drop the message and attempt to send a new message at a later time. If the message box is full often, then a **µC/OS-II** Queue should be used instead.

4.00 Message Queues

Message Queues are similar to message boxes in that a task or ISR may post a message to a task waiting on a given queue. However, unlike message boxes, a message queue may receive more than one message at a time before returning `OS_ERR_MBOX_FULL`, or in the case of a queue, `OS_ERR_Q_FULL`. Once a task wakes up as a result of a message being posted, it will continue to process messages from the queue until the queue becomes empty. When the queue becomes empty, the last call to OSQPend() will cause the task to become blocked until a new message is arrived. This mechanism is preferred over message boxes when multiple tasks are posting messages to a central queue since it does not become full as easily. However, the size of the queue must be chosen appropriately by the application developer based on the frequency of messages being produced versus consumed.

Solution 3_Revised: Create a central control task that pends on a message queue. Each sensor task will post messages to the control tasks queue instead of a control task message box.

Step(s):

1. Declare an `OS_EVENT` type variable that is accessible 'C' scope wise to all of the tasks that will pend or post messages from or to the message queue. Often times, it is easiest to declare the message queue at the top of the file as a static variable. Of course, reference to the message box can be passed to various tasks during task creation by means of the task argument, or by using the **extern** 'C' language keyword.

```
static  OS_EVENT  *msgqueue;
```

   Next, declare an array of void pointers that will be used by the queue as place holders for received messages. `MAX_NBR_MSGS` must be declared by the user preferably in app_cfg.h.

```
static  void      *MessageStorage[MAX_NBR_MSGS];
```

2. Create a message queue BEFORE creating application tasks that utilize it. Otherwise, the µC/OS-II message queue API functions will likely return an error, or unpredictable behavior may occur. Of course, other methods of preventing early use of OS objects can be devised, for example, each application task requiring use of a shared message queue could wait for an initialization complete signal to be posted.

```
CPU_INT08U  Application_Init (void) {
    msgbox = OSQCreate(&MessageStorage, MAX_NBR_MSGS);
    if (msgbox == ((void *)0) {
        /* Failed to create message box   */
        /* Return error to caller?        */
    }

    AppTaskCreate();
}
```

3. Have the control task pend on the message queue thus causing it to be suspending until a message is received.

```
void  AppControlTask (void *p_arg) {
    void       *msg;
    CPU_INT16U  data;
    CPU_INT08U  err;


    while (DEF_TRUE) {
        msg = OSQPend(msgqueue, 0, &err);   (1)
        switch (err) {                      (2)
            case OS_ERR_NONE:
            case OS_ERR_TIMEOUT:
            case OS_ERR_PEND_ABORT:
                 break;

            default:
                 break;
        }

        data = (CPU_INT16U)msg;             (3)

        /* Process message content           */
        /* Update LCD if applicable          */
    }
}
```

4. Wait for a message to be received.
5. Be mindful of non critical errors such as timeouts and aborts. It may be desirable for a task to specify a timeout and not block indefinitely waiting for a message (in case an error occurs in the task that posts messages). Likewise, it may be desirable for a task to abort the timeout of the control task in which case you may want to execute some different code if a pend abort is detected. In most many cases, tasks wait indefinitely for messages to be received and most often receive the OS_ERR_NONE code when unblocked.
6. Cast the message back to the correct data type agreed upon by the sender and the receiver / application developer, process the data, loop and read the next message from the queue. If the queue is empty, the task will be blocked until a new message is received.

4. Have ISR's and other tasks post messages to the control task's message box.

```
void  AppSensor1Task (void *p_arg) {
    CPU_INT08U  err;
     CPU_INT16U  data;


    while (DEF_TRUE) {
        data = ReadSensor();                          (1)
        err  = OSQPost(msgbox, (void *)data);         (2)
        if (err != OS_ERR_NONE) {                     (3)
            /* Try again? */
        }
    }
}
```

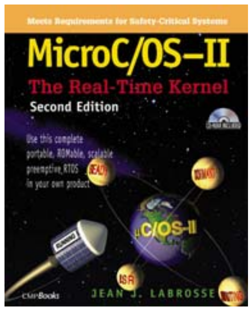
1. Create data to be passed to another task.
2. Post the data to the specified message box.
    a. WARNING: In this particular example, the message is passed directly within the storage area of the pointer. Pointers require different amounts of storage space depending on the architecture. For example, if a pointer (address) requires 16-bits of storage on a particular architecture, then a 32 bit message can not be casted directly as the pointer address and passed to the receiving task. However, a pointer to a structure that has more than 32 bits of data may be passed instead. The negative to passing pointers to data types, is that both the receiver and sender may attempt to modify the same memory concurrently thus making the operation unsafe. In order to guarantee that the message passing is safe, the sender must allocate a message from either the heap (discouraged for various reasons) or from a memory pool of some sort. The sender must NOT try to modify the memory that is passed by reference to the receiving task until the receiving task has finished processing the message and returns the memory to the pool.
3. Check for errors and take appropriate action if an error occurs. In most cases, `OSQPost()` will return `OS_ERR_NONE`. However, `OS_ERR_Q_FULL` may be returned if the receiving task has not processed a sufficient number of messages in the queue and the queue has become full.

Licensing

If you intend to use **µC/OS-II** in a commercial product, remember that you need to contact **Micriµm** to properly license its use in your product. The use of **µC/OS-II** in commercial applications is **NOT-FREE**. Your honesty is greatly appreciated.

References

MicroC/OS-II, The Real-Time Kernel, 2nd Edition
Jean J. Labrosse
CMP Technical Books, 2002
ISBN 1-5782-0103-9


Meets Requirements for Safety-Critical Systems
MicroC/OS-II
The Real-Time Kernel
Second Edition
Use this complete portable, ROMable, scalable preemptive RTOS in your own product
µC/OS-II
CMP Books
JEAN J. LABROSSE

Contacts

Micriµm
949 Crestview Circle
Weston, FL 33327
USA
954-217-2036
954-217-2037 (FAX)
e-mail: eric.shufro@Micrium.com
WEB: www.Micrium.com