# Micriμm, Inc.



# μC/OS-II
# and
# Event Flags

## Application Note
AN-1007A

**Jean J. Labrosse**
Jean.Labrosse@Micrium.com
www.Micrium.com

## Summary

Event flags are used when a task needs to synchronize with the occurrence of multiple events. The task can be synchronized when any of the events have occurred. This is called *disjunctive synchronization* (logical OR). A task can also be synchronized when all events have occurred. This is called *conjunctive synchronization* (logical AND). Disjunctive and conjunctive synchronization are shown in Figure 1.

This application note describes the Event Flag series of services which were added to µC/OS-II V2.05.

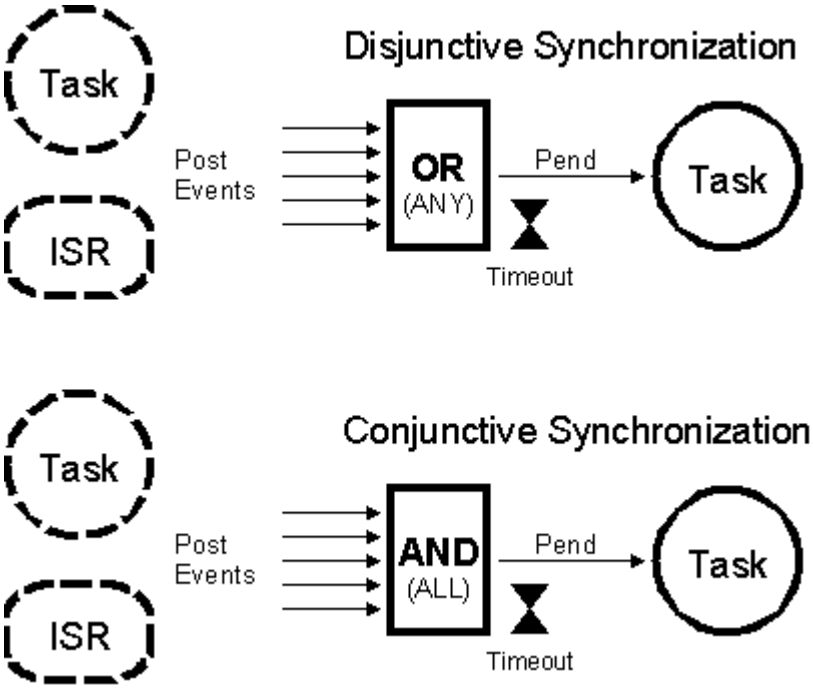


**Figure 1, Disjunctive and Conjunctive Synchronization using Event Flags.**

## Introduction

Common events can be used to signal multiple tasks, as shown in Figure 2. Events are typically grouped. Depending on the kernel, a group consists of 8, 16 or 32 events. µC/OS-II allows you to choose the number of bits in an event flag group at compile time. Tasks and ISRs can set or clear any event in a group. A task is resumed when all the events (i.e. bits in the event group) it requires are satisfied. The evaluation of which task will be resumed is performed when a new set of events occurs (i.e. during a POST operation).

µC/OS-II offer services to SET event flags, CLEAR event flags, and WAIT (or PEND) for event flags (conjunctively or disjunctively) to be either set or cleared. A task that waits for events to be SET can also clear those events once received. Similarly, a task that waits for events to be CLEARED can also set those events once received.

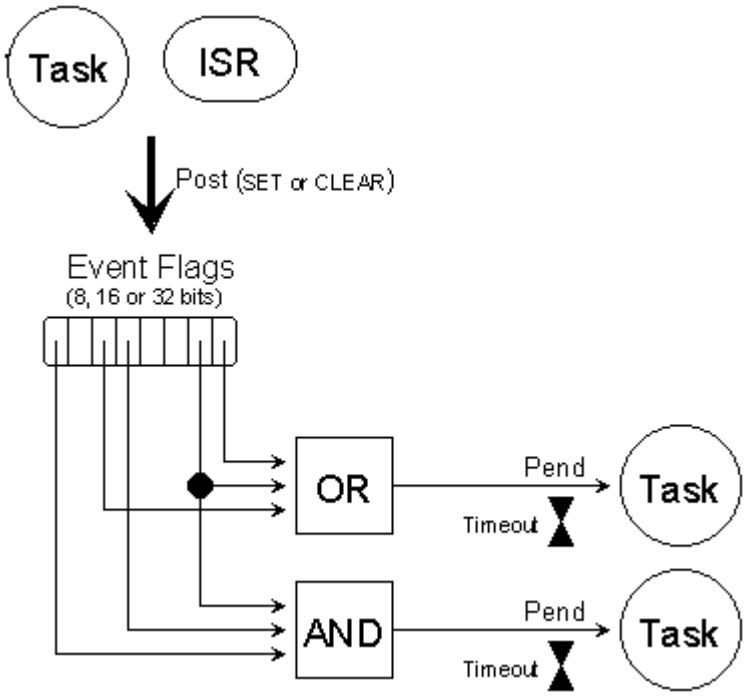


**Figure 2, Signaling events to tasks.**

Figure 3 shows the relationship between Tasks and ISRs and which services are provided by µC/OS-II. As you can see you can only create or delete from either task level code or startup code (i.e. code executed before starting µC/OS-II).

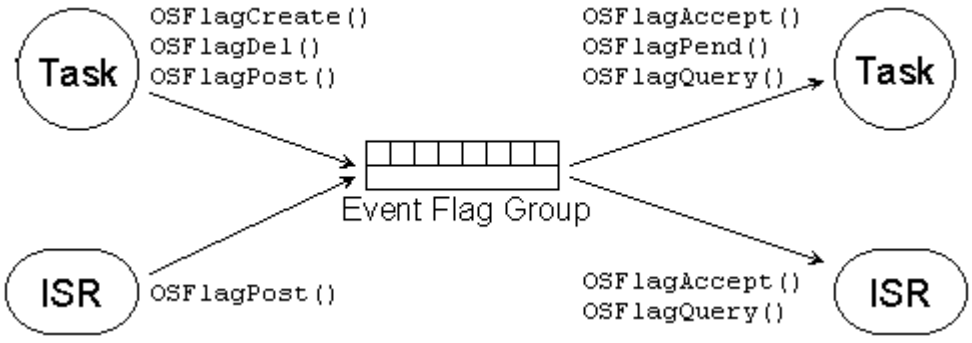


**Figure 3, µC/OS-II Event Flag services.**

## Event Flag Internals

A µC/OS-II's event flag group consist of three elements as shown in the `OS_FLAG_GRP` structure below. First, a type which is used to make sure that you are pointing to an event flag group. This field is the first field of the structure because it allows µC/OS-II services to 'validate' the type of structure being pointed to. For example, if you were to pass a pointer to an event flag group to `OSSemPend()`, µC/OS-II would return an error code indicating that you are not passing the proper 'object' to the semaphore pend call. The second field contains a series of flags (i.e. bits) which holds the current status of events. Finally, an event flag group contains a list of tasks waiting for events.

```
typedef struct {
    INT8U     OSFlagType;
    void     *OSFlagWaitList;
    OS_FLAGS  OSFlagFlags;
} OS_FLAG_GRP;
```

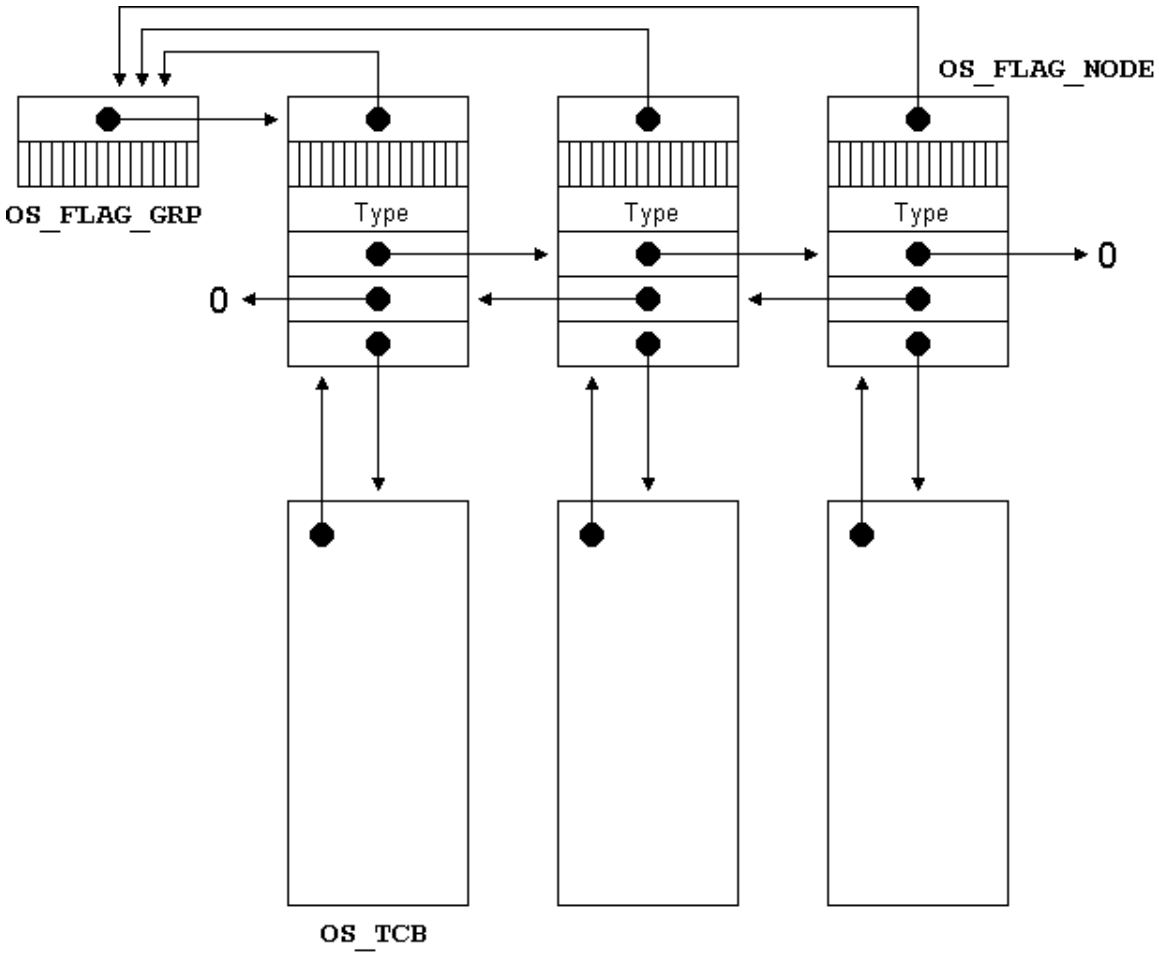


**Figure 4, Relationship between Event Flag Group, Event Flag Nodes and TCBs.**

You should note that the wait list for event flags is different than the other wait lists in µC/OS-II. With event flags, the wait list is accomplished through a doubly linked list as shown in figure 4. Three data structures are involved. `OS_FLAG_GRP` (mentioned above), `OS_TCB` which is the task control block and `OS_FLAG_NODE` which is used to keep track of which bits the task is waiting for as well as what type of wait (AND or OR). As you can see, there are a lot of pointers involved.

An `OS_FLAG_NODE` is created when a task desires to wait on bits of an event flag group and the node is 'destroyed' when the event(s) occur. In other words, a node is created by `OSFlagPend()` as we will see shortly. Before we discuss this, let's look at the `OS_FLAG_NODE` data structure.

```
typedef struct {
    void     *OSFlagNodeNext;
    void     *OSFlagNodePrev;
    void     *OSFlagNodeTCB;
    void     *OSFlagNodeFlagGrp;
    OS_FLAGS  OSFlagNodeFlags;
    INT8U     OSFlagNodeWaitType;
} OS_FLAG_NODE;
```

The `OSFlagNodeNext` and `OSFlagNodePrev` are used to maintain a doubly linked list of `OS_FLAG_NODE`s. The doubly linked list allows us to easily insert and especially remove nodes from the wait list.

`OSFlagNodeTCB` is used to point to the TCB of the task waiting on event flags belonging to the event flag group. This pointer thus allows us to know which tasks is waiting for the specified flags.

`OSFlagNodeFlagGrp` allows a link back to the event flag group. This pointer is used when removing the node from the doubly linked list and is there because a node might need to be removed because a task is deleted (see `OSTaskDel()`).

The `OSFlagNodeFlags` contains the bit-pattern of the flags that the task is waiting for. For example, your task might have performed an `OSFlagPend()` and specified that the task wants to wait for bits 0, 4, 6 and 7 (bit 0 is the rightmost bit). In this case, `OSFlagFlags` would contain `0xD1`. Depending on the size of the data type `OS_FLAGS`, `OSFlagFlags` is either 8, 16 or 32 bits. `OS_FLAGS` is specified in your application configuration file, i.e `OS_CFG.H`. Because µC/OS-II and the ports are provided in source form, you can easily change the number of bits in an event flag group to satisfy your requirements for a specific application or product. The reason you would limit the number of bits to 8 is to reduce both RAM and ROM for your application.

The last member of the `OS_FLAG_NODE` data structure is `OSFlagNodeWaitType` which determines whether the task is waiting for ALL (AND wait) the bits in the event flag group that matches `OSFlagNodeFlags` or, ANY (OR wait) of the bits in the event flag group that matches `OSFlagNodeFlags`. `OSFlagNodeWaitType` can be set to:

```
OS_FLAG_WAIT_CLR_ALL
OS_FLAG_WAIT_CLR_AND

OS_FLAG_WAIT_CLR_ANY
OS_FLAG_WAIT_CLR_OR

OS_FLAG_WAIT_SET_ALL
OS_FLAG_WAIT_SET_AND

OS_FLAG_WAIT_SET_ANY
OS_FLAG_WAIT_SET_OR
```

You should note that AND and ALL means the same thing and either one can be used. I prefer to use `OS_FLAG_WAIT_???_ALL` because it's more obvious but you are certainly welcomed to use `OS_FLAG_WAIT_???_AND`. Similarly, OR or ANY means the same thing and either one can be used. Again, I prefer to use `OS_FLAG_WAIT_???_ANY` because it's more obvious but again, you can use `OS_FLAG_WAIT_???_OR`. The other thing to notice is that you can wait for either bits to be SET or CLEARED.

## Creating an Event Flag Group, OSFlagCreate()

The code to create an event flag group is shown in listing 1.

```
OS_FLAG_GRP  *OSFlagCreate (OS_FLAGS flags, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                       /* Allocate storage for CPU status register           */
    OS_CPU_SR    cpu_sr;
#endif
    OS_FLAG_GRP *pgrp;


    if (OSIntNesting > 0) {                       /* (1) See if called from ISR ...                     */
        *err = OS_ERR_CREATE_ISR;                 /*     ... can't CREATE from an ISR                   */
        return ((OS_FLAG_GRP *)0);
    }
    OS_ENTER_CRITICAL();
    pgrp = OSFlagFreeList;                        /* (2) Get next free event flag                       */
    if (pgrp != (OS_FLAG_GRP *)0) {               /* (3) See if we have event flag groups available     */
                                                  /* (4) Adjust free list                               */
        OSFlagFreeList       = (OS_FLAG_GRP *)OSFlagFreeList->OSFlagWaitList;
        pgrp->OSFlagType     = OS_EVENT_TYPE_FLAG;/* (5) Set to event flag group type                   */
        pgrp->OSFlagFlags    = flags;             /* (6) Set to desired initial value                   */
        pgrp->OSFlagWaitList = (void *)0;         /* (7) Clear list of tasks waiting on flags           */
        OS_EXIT_CRITICAL();
        *err                 = OS_NO_ERR;
    } else {
        OS_EXIT_CRITICAL();
        *err                 = OS_FLAG_GRP_DEPLETED;
    }
    return (pgrp);                                /* (8) Return pointer to event flag group             */
}
```

**Listing 1, Creating an Event Flag Group.**

`OSFlagCreate()` starts by making sure it's not called from an ISR because that's not allowed L1(1).

`OSFlagCreate()` then attempts to get a free Event Flag Group (i.e. an `OS_FLAG_GRP`) from the free list L1(2). An non-NULL pointer indicates that an event flag group is available L1(3). Once a group is allocated, the free list pointer is adjusted L1(4). Note that the number of Event Flag Groups that you can create is determined by the `#define` constant `OS_MAX_FLAGS` which is defined in `OS_CFG.H` in your application.

`OSFlagCreate()` then fills in the fields in the event flag group. `OS_EVENT_TYPE_FLAG` indicates that this control block is an event flag group. Because this is the first field in the data structure, it's at offset zero. In µC/OS-II, the first byte of an event flag group or an event control block used for semaphores, mailboxes, queues and mutexes indicates the type which allows us to check that we are pointing to the proper object.

`OSFlagCreate()` then stores the initial value of the event flags into the event flag group L1(6). Typically, you would initialize the flags to all `0`s but, if you are checking for CLEARED bits then, you could initialize the flags to all `1`s.
Because we are creating the group, there are no tasks waiting on the group and thus, the wait list pointer is initialized to `NULL` L1(7).

The pointer to the created event flag group is returned. If there were no more groups available, `OSFlagCreate()` would return a `NULL` pointer L1(8).

## Deleting an Event Flag Group, OSFlagDel()

The code to delete an event flag group is shown in listing 2.

```
OS_FLAG_GRP  *OSFlagDel (OS_FLAG_GRP *pgrp, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                                 /*      Allocate storage for CPU status register */
    OS_CPU_SR     cpu_sr;
#endif
    BOOLEAN       tasks_waiting;
    OS_FLAG_NODE *pnode;


    if (OSIntNesting > 0) {                                 /*  (1) See if called from ISR ...               */
        *err = OS_ERR_DEL_ISR;                              /*      ... can't DELETE from an ISR             */
        return (pgrp);
    }
#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                         /*  (2) Validate 'pgrp'                          */
        *err = OS_FLAG_INVALID_PGRP;
        return (pgrp);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {           /*  (3) Validate event group type                */
        *err = OS_ERR_EVENT_TYPE;
        return (pgrp);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pgrp->OSFlagWaitList != (void *)0) {                /*  (4) See if any tasks waiting on event flags  */
        tasks_waiting = TRUE;                               /*      Yes                                      */
    } else {
        tasks_waiting = FALSE;                              /*      No                                       */
    }
    switch (opt) {
        case OS_DEL_NO_PEND:                                /*  (5) Delete group if no task waiting          */
             if (tasks_waiting == FALSE) {
                 pgrp->OSFlagType     = OS_EVENT_TYPE_UNUSED;
                 pgrp->OSFlagWaitList = (void *)OSFlagFreeList; /*  (6) Return group to free list           */
                 OSFlagFreeList       = pgrp;
                 OS_EXIT_CRITICAL();
                 *err                 = OS_NO_ERR;
                 return ((OS_FLAG_GRP *)0);                 /*  (7) Event Flag Group has been deleted        */
             } else {
                 OS_EXIT_CRITICAL();
                 *err                 = OS_ERR_TASK_WAITING;
                 return (pgrp);
             }

        case OS_DEL_ALWAYS:                                 /*  (8) Always delete the event flag group       */
             pnode = pgrp->OSFlagWaitList;
             while (pnode != (OS_FLAG_NODE *)0) {           /*  (9) Ready ALL tasks waiting for flags        */
                 OS_FlagTaskRdy(pnode, (OS_FLAGS)0);
                 pnode = pnode->OSFlagNodeNext;
             }
             pgrp->OSFlagType     = OS_EVENT_TYPE_UNUSED;
             pgrp->OSFlagWaitList = (void *)OSFlagFreeList;/* (10) Return group to free list                */
             OSFlagFreeList       = pgrp;
             OS_EXIT_CRITICAL();
             if (tasks_waiting == TRUE) {                   /* (11) Reschedule only if task(s) were waiting  */
                 OS_Sched();                                /*      Find highest priority task ready to run  */
             }
             *err = OS_NO_ERR;
             return ((OS_FLAG_GRP *)0);                     /* (12) Event Flag Group has been deleted        */

        default:
             OS_EXIT_CRITICAL();
             *err = OS_ERR_INVALID_OPT;
             return (pgrp);
    }
}
```

**Listing 2, Deleting an Event Flag Group.**

This is a function you should use with caution because multiple tasks could attempt to access a deleted event flag group. You should always use this function with great care. Generally speaking, before you would delete an event flag group, you would first delete all the tasks that access the event flag group.

`OSFlagDel()` starts by making sure that this function is not called from an ISR because that's not allowed L2(1).

We then validate the arguments passed to OSFlagDel(). First, we make sure that `pgrp` is not a `NULL` pointer L2(2) and `pgrp` points to point to an event flag group L2(3). Note that this code is conditionally compiled and thus, if `OS_ARG_CHK_EN` is set to `0` then this code is NOT compiled. This is done to allow you to reduce the amount of ROM needed by this module.

`OSFlagDel()` then determines whether there are any tasks waiting on the event flag group and sets the local `BOOLEAN tasks_waiting` accordingly L2(4).

Based on the option (i.e. `opt`) passed in the call, `OSFlagDel()` will either delete the event flag group only if no tasks are pending on the event flag group (`opt == OS_DEL_NO_PEND`) or, delete the event flag group even if tasks are waiting (`opt == OS_DEL_ALWAYS`).

> When `opt` is set to `OS_DEL_NO_PEND` and there is no task waiting on the event flag group L2(5), `OSFlagDel()` marks the group as unused and the event flag group is returned to the free list of groups L2(6). This will allow another event flag group to be created. You will note that `OSFlagDel()` returns a `NULL` pointer L2(7) since, at this point, the event flag group should no longer be accessed through the original pointer.
>
> When `opt` is set to `OS_DEL_ALWAYS` L2(8) then all tasks waiting on the event flag group will be readied L2(9). Each task will *think* the event(s) it was waiting for occurred. Once all pending tasks are readied, `OSFlagDel()` marks the event flag group as unused and the group is returned to the free list of groups L2(10). The scheduler is called only if there were tasks waiting on the event flag group L2(11). You will note that `OSFlagDel()` returns a `NULL` pointer L2(12) since, at this point, the event flag group should no longer be accessed through the original pointer.

## Waiting for event(s) of an Event Flag Group, OSFlagPend()

The code to wait for event(s) of an event flag group is shown in listing 3. This is a long function because a lot is happening.

Like all µC/OS-II pend calls, `OSFlagPend()` cannot be called from an ISR and thus, `OSFlagPend()` checks for this condition first L3(1).

Assuming that the configuration constant `OS_ARG_CHK_EN` is set to `1`, `OSFlagPend()` makes sure that the 'handle' `pgrp` is not a `NULL` pointer L3(2) and that `pgrp` points to an event flag group L3(3) that should have been created by `OSFlagCreate()`.

`OSFlagPend()` allows you to specify whether you will CLEAR or SET flags once they satisfy the condition you are waiting for. This is accomplished by ADDing or ORing `OS_FLAG_CONSUME` to the `wait_type` argument during the call to `OSFlagPend()`. For example, if you want to wait for BIT0 to be SET in the event flag group and if BIT0 is in fact SET, it will be CLEARED by `OSFlagPend()` if you ADD `OS_FLAG_CONSUME` to the type of wait desired as shown below:

```
OSFlagPend(OSFlagMyGrp,
           (OS_FLAGS)0x01,
           FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME,
           0,
           &err);
```

Because the 'consumption' of the flag(s) is done later in the code, `OSFlagPend()` saves the 'consume' option in the `BOOLEAN` variable called `consume` L3(4).

`OSFlagPend()` then executes code based on the wait type specified in the function called L3(5). There are four choices:

1) wait for ALL bits specified to be SET in the event flag group,
2) wait for ANY bit specified to be SET in the event flag group,
3) wait for ALL bits specified to be CLEARED in the event flag group,
4) wait for ANY bit specified to be CLEARED in the event flag group.

The last two choices are identical to the first two choices except that `OSFlagPend()` 'looks' for the bits specified to be CLEARED (i.e. 0) instead of them being SET (i.e. 1). For this reason, I will only discuss the first two choices. In fact, in order to conserve ROM, you may not need to look for bits to be cleared and thus, you can 'compile-out' all the corresponding code out by setting `OS_FLAG_WAIT_CLR_EN` to `0` in `OS_CFG.H`.

**Wait for ALL of the specified bits to be SET:**

When `wait_type` is set to either `OS_FLAG_WAIT_SET_ALL` or `OS_FLAG_WAIT_SET_AND`, `OSFlagPend()` will 'extract' the desired bits in the event flag group which are specified in the `flags` argument L3(6). If all the bits extracted matches the bits that you specified in the `flags` argument L3(7) then, the event flags that the task wants are all set and thus, the PEND call would return to the caller. Before we return, we need to determine whether we need to 'consume' the flags L3(8) and if so, we will CLEAR all the flags that satisfied the condition L3(9). The new value of the event flag group is obtained L3(10) and returned to the caller L3(11).

If ALL the desired bits in the event flag group were not SET then the calling task will block (i.e. suspend) until ALL the bits are either SET or a `timeout` occurs L3(12). Instead of repeating code for all four types of wait, I created a function (`OS_FlagBlock()`) to handle the details of blocking the calling task (described soon).

**Wait for ANY of the specified bits to be SET:**
When `wait_type` is set to either `OS_FLAG_WAIT_SET_ANY` or `OS_FLAG_WAIT_SET_OR`, `OSFlagPend()` will 'extract' the desired bits in the event flag group which are specified in the `flags` argument L3(13). If any of the bits extracted matches the bits that you specified in the `flags` argument L3(14) then the PEND call will return to the caller. Before we return, we need to determine whether we need to 'consume' the flag(s) L3(15) and if so, we need to CLEAR all the flag(s) that satisfied the condition L3(16). The new value of the event flag group is obtained L3(17) and returned to the caller L3(18).

If NONE of the desired bits in the event flag group were not SET then the calling task will block (i.e. suspend) until ANY of the bits is either SET or a `timeout` occurs L3(19).

As mentioned above, if the desired bits and conditions of a PEND call are not satisfied the the calling task is suspended until either the event or a timeout occurs. The task is suspended by `OS_FlagBlock()` (see Listing 4) which adds the calling task to the wait list of the event flag group. The process is shown in Figure 5.

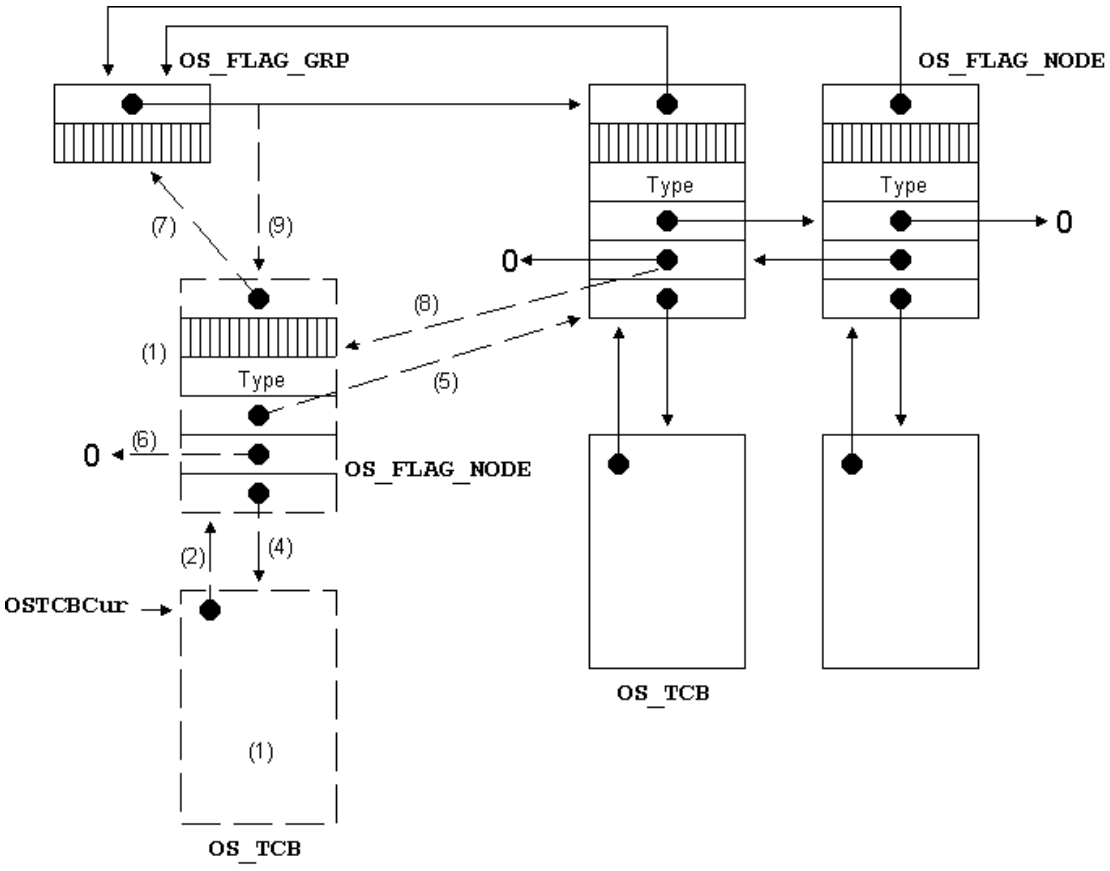


**Figure 5, Adding the current task to the wait list of the Event Flag Group.**

`OS_FlagBlock()` starts by setting the appropriate fields in the task control block L4-F5(1). You should note that an `OS_FLAG_NODE` is allocated on the stack of the calling task (see `OSFlagPend()`, L3). This means that we don't need to keep a separate 'free list' of `OS_FLAG_NODE` since these data structures can simply be allocated on the stack. That being said, the calling task must have sufficient stack space to allocate this structure on its stack.

We then link the `OS_FLAG_NODE` to the TCB, L4-F5(2) but only if `OS_TASK_DEL_EN` is set to `1`. This allows `OSTaskDel()` to remove the task being suspended from the wait list should a task decide to delete this task.

Next, `OS_FlagBlock()` saves the flags that the task is waiting for as well as the wait type in the `OS_FLAG_NODE` structure, L4-F5(3).

We then link the TCB to the `OS_FLAG_NODE`, L4-F5(4).

The `OS_FLAG_NODE` is then linked to the other `OS_FLAG_NODE`s in the wait list, L4-F5(5). You should note that the `OS_FLAG_NODE` is simply inserted at the beginning of the doubly-linked list for simplicity sake, L4-F5(6).

We then link the event flag group to the `OS_FLAG_NODE`, L4-F5(7). This is again done to allow us to delete the task that is being added to the wait list of the event flag group.

`OS_FlagBlock()` then links the previous 'first' node in the wait list to the new `OS_FLAG_NODE`, L4-F5(8).

Finally, the pointer of the beginning of the wait list is updated to point to the new `OS_FLAG_NODE`, L4-F5(9) and, the calling task is made NOT ready-to-run, L4-F5(10).

You should note that interrupts are disabled during the process of blocking the calling task.

We can now resume the discussion of listing 3. When `OS_FlagBlock()` returns, the scheduler is called because, of course, the calling task is no longer able to run since the event(s) it was looking for did not occur, L3(20).

When µC/OS-II resumes the calling task, `OSFlagPend()` checks HOW the task was readied, L3(21). If the status field in the TCB still indicate that the task is still waiting for event flags to be either set or cleared then, the task MUST have been readied because of a timeout. In this case, the `OS_FLAG_NODE` is removed from the wait list by calling `OS_FlagUnlink()`, L3(22) and, an error code is returned to the caller indicating the outcome of the call. The code for `OS_FlagUnlink()` is shown in Listing 5 and should be quite obvious since we are simply removing a node from a doubly linked list.

If the calling task is NOT resumed because of a timeout then, it MUST have been resumed because the event flags that it was waiting for have been either set or cleared. In this case, we determine whether the calling task wanted to consume the event flags. If this is the case, the appropriate flags are either set or cleared based on the wait type, L3(24).

Finally, `OSFlagPend()` obtains the current value of the event flags in the group in order to return this information to the caller, L3(25).

```
OS_FLAGS  OSFlagPend (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                                 /*      Allocate storage for CPU status register */
    OS_CPU_SR     cpu_sr;
#endif
    OS_FLAG_NODE  node;
    OS_FLAGS      flags_cur;
    OS_FLAGS      flags_rdy;
    BOOLEAN       consume;


    if (OSIntNesting > 0) {                                 /*  (1) See if called from ISR ...              */
        *err = OS_ERR_PEND_ISR;                             /*      ... can't PEND from an ISR              */
        return ((OS_FLAGS)0);
    }
#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                         /*  (2) Validate 'pgrp'                         */
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {           /*  (3) Validate event block type               */
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    if (wait_type & OS_FLAG_CONSUME) {                      /*  (4) See if we need to consume the flags     */
        wait_type &= ~OS_FLAG_CONSUME;
        consume    = TRUE;
    } else {
        consume    = FALSE;
    }
    OS_ENTER_CRITICAL();
    switch (wait_type) {                                    /*  (5)                                         */
        case OS_FLAG_WAIT_SET_ALL:                          /*      See if all required flags are set       */
             flags_rdy = pgrp->OSFlagFlags & flags;         /*  (6) Extract only the bits we want           */
             if (flags_rdy == flags) {                      /*  (7) Must match ALL the bits that we want    */
                 if (consume == TRUE) {                     /*  (8) See if we need to consume the flags     */
                     pgrp->OSFlagFlags &= ~flags_rdy;       /*  (9) Clear ONLY the flags that we wanted     */
                 }
                 flags_cur = pgrp->OSFlagFlags;             /* (10) Will return the state of the group      */
                 OS_EXIT_CRITICAL();                        /*      Yes, condition met, return to caller    */
                 *err      = OS_NO_ERR;
                 return (flags_cur);                        /* (11)                                         */
             } else {                                       /* (12) Block task until events occur or timeout */
                 OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                 OS_EXIT_CRITICAL();
             }
             break;

        case OS_FLAG_WAIT_SET_ANY:
             flags_rdy = pgrp->OSFlagFlags & flags;         /* (13) Extract only the bits we want           */
             if (flags_rdy != (OS_FLAGS)0) {                /* (14) See if any flag set                     */
                 if (consume == TRUE) {                     /* (15) See if we need to consume the flags     */
                     pgrp->OSFlagFlags &= ~flags_rdy;       /* (16) Clear ONLY the flags that we got        */
                 }
                 flags_cur = pgrp->OSFlagFlags;             /* (17) Will return the state of the group      */
                 OS_EXIT_CRITICAL();                        /*      Yes, condition met, return to caller    */
                 *err      = OS_NO_ERR;
                 return (flags_cur);                        /* (18)                                         */
             } else {                                       /* (19) Block task until events occur or timeout */
                 OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                 OS_EXIT_CRITICAL();
             }
             break;

#if OS_FLAG_WAIT_CLR_EN > 0
        case OS_FLAG_WAIT_CLR_ALL:                          /*     See if all required flags are cleared    */
             flags_rdy = ~pgrp->OSFlagFlags & flags;        /*     Extract only the bits we want            */
             if (flags_rdy == flags) {                      /*     Must match ALL the bits that we want     */
                 if (consume == TRUE) {                     /*     See if we need to consume the flags      */
                     pgrp->OSFlagFlags |= flags_rdy;        /*     Set ONLY the flags that we wanted        */
                 }
                 flags_cur = pgrp->OSFlagFlags;             /*     Will return the state of the group       */
                 OS_EXIT_CRITICAL();                        /*     Yes, condition met, return to caller     */
                 *err      = OS_NO_ERR;
                 return (flags_cur);
             } else {                                       /*     Block task until events occur or timeout */
                 OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                 OS_EXIT_CRITICAL();
             }
             break;

        case OS_FLAG_WAIT_CLR_ANY:
             flags_rdy = ~pgrp->OSFlagFlags & flags;        /*     Extract only the bits we want            */
```

```
            if (flags_rdy != (OS_FLAGS)0) {                /*     See if any flag cleared                  */
                if (consume == TRUE) {                     /*     See if we need to consume the flags      */
                    pgrp->OSFlagFlags |= flags_rdy;        /*     Set ONLY the flags that we got           */
                }
                flags_cur = pgrp->OSFlagFlags;             /*     Will return the state of the group       */
                OS_EXIT_CRITICAL();                        /*     Yes, condition met, return to caller     */
                *err      = OS_NO_ERR;
                return (flags_cur);
            } else {                                       /*     Block task until events occur or timeout */
                OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                OS_EXIT_CRITICAL();
            }
            break;
#endif

        default:
            OS_EXIT_CRITICAL();
            flags_cur = (OS_FLAGS)0;
            *err      = OS_FLAG_ERR_WAIT_TYPE;
            return (flags_cur);
    }
    OS_Sched();                                             /* (20) Find next HPT ready to run              */
    OS_ENTER_CRITICAL();
    if (OSTCBCur->OSTCBStat & OS_STAT_FLAG) {              /* (21) Have we timed-out?                      */
        OS_FlagUnlink(&node);                               /* (22)                                         */
        OSTCBCur->OSTCBStat = OS_STAT_RDY;                 /*      Yes, make task ready-to-run             */
        OS_EXIT_CRITICAL();
        flags_cur           = (OS_FLAGS)0;
        *err                = OS_TIMEOUT;                  /*      Indicate that we timed-out waiting      */
    } else {
        if (consume == TRUE) {                             /* (23) See if we need to consume the flags     */
            switch (wait_type) {
                case OS_FLAG_WAIT_SET_ALL:
                case OS_FLAG_WAIT_SET_ANY:                 /* (24) Clear ONLY the flags we got             */
                     pgrp->OSFlagFlags &= ~OSTCBCur->OSTCBFlagsRdy;
                     break;

                case OS_FLAG_WAIT_CLR_ALL:
                case OS_FLAG_WAIT_CLR_ANY:                 /*      Set   ONLY the flags we got             */
                     pgrp->OSFlagFlags |= OSTCBCur->OSTCBFlagsRdy;
                     break;
            }
        }
        flags_cur = pgrp->OSFlagFlags;                     /* (25)                                         */
        OS_EXIT_CRITICAL();
        *err      = OS_NO_ERR;                             /*      Event(s) must have occurred             */
    }
    return (flags_cur);
}
```

**Listing 3, Waiting for event(s) of an event flag group.**

```
static  void  OS_FlagBlock (OS_FLAG_GRP  *pgrp,  OS_FLAG_NODE  *pnode,  OS_FLAGS  flags,  INT8U  wait_type,  INT16U
timeout)
{
    OS_FLAG_NODE  *pnode_next;


    OSTCBCur->OSTCBStat      |= OS_STAT_FLAG;         /*  (1)                                                 */
    OSTCBCur->OSTCBDly        = timeout;              /*      Store timeout in task's TCB                     */
#if OS_TASK_DEL_EN > 0
    OSTCBCur->OSTCBFlagNode   = pnode;                /*  (2) TCB to link to node                             */
#endif
    pnode->OSFlagNodeFlags    = flags;                /*  (3) Save the flags that we need to wait for         */
    pnode->OSFlagNodeWaitType = wait_type;            /*      Save the type of wait we are doing              */
    pnode->OSFlagNodeTCB      = (void *)OSTCBCur;     /*  (4) Link to task's TCB                              */
    pnode->OSFlagNodeNext     = pgrp->OSFlagWaitList; /*  (5) Add node at beginning of event flag wait list   */
    pnode->OSFlagNodePrev     = (void *)0;            /*  (6)                                                 */
    pnode->OSFlagNodeFlagGrp  = (void *)pgrp;         /*  (7) Link to Event Flag Group                        */
    pnode_next                = pgrp->OSFlagWaitList;
    if (pnode_next != (void *)0) {                    /*      Is this the first NODE to insert?               */
        pnode_next->OSFlagNodePrev = pnode;           /*  (8) No, link in doubly linked list                  */
    }
    pgrp->OSFlagWaitList = (void *)pnode;             /*  (9)                                                 */
                                                      /* (10) Suspend current task until flag(s) received     */
    if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
}
```

**Listing 4, Adding a task to the event flag group wait list.**

```
static  void  OS_FlagUnlink (OS_FLAG_NODE *pnode)
{
    OS_TCB       *ptcb;
    OS_FLAG_GRP  *pgrp;
    OS_FLAG_NODE *pnode_prev;
    OS_FLAG_NODE *pnode_next;


    pnode_prev = pnode->OSFlagNodePrev;                         /* Get pointers to neighboring nodes   */
    pnode_next = pnode->OSFlagNodeNext;
    if (pnode_prev == (OS_FLAG_NODE *)0) {                      /* Is it first node in wait list?      */
        pgrp                = pnode->OSFlagNodeFlagGrp;         /* Yes, Point to event flag group      */
        pgrp->OSFlagWaitList = (void *)pnode_next;              /*      Update list for new 1st node   */
        if (pnode_next != (OS_FLAG_NODE *)0) {
            pnode_next->OSFlagNodePrev = (OS_FLAG_NODE *)0;     /*      Link new 1st node PREV to NULL */
        }
    } else {                                                    /* No,  A node somewhere in the list   */
        pnode_prev->OSFlagNodeNext = pnode_next;                /*      Link around the node to unlink */
        if (pnode_next != (OS_FLAG_NODE *)0) {                  /*      Was this the LAST node?        */
            pnode_next->OSFlagNodePrev = pnode_prev;            /*      No, Link around current node   */
        }
    }
    ptcb                = (OS_TCB *)pnode->OSFlagNodeTCB;
#if OS_TASK_DEL_EN > 0
    ptcb->OSTCBFlagNode = (void *)0;
#endif
}
```

**Listing 5, Removing an OS_FLAG_NODE from the wait list.**

## Setting or Clearing event(s) in an Event Flag Group, OSFlagPost()

The code to either setting or clearing bits in an event flag group is done by calling `OSFlagPost()` and the code for this function is shown in listing 6.

Assuming that the configuration constant `OS_ARG_CHK_EN` is set to `1`, `OSFlagPost()` makes sure that the 'handle' `pgrp` is not a `NULL` pointer L6(1) and that `pgrp` points to an event flag group L6(2) that should have been created by `OSFlagCreate()`.

Depending on the option you specified in the `opt` argument of `OSFlagPost()` L6(3), the flags specified in the `flags` argument will either be SET (when `opt == OS_FLAG_SET`) L6(4) or CLEARED (when `opt == OS_FLAG_CLR`) L6(5). If `opt` is not one of the two choices, the call is aborted and an error code is returned to the caller.

We next start by assuming that POSTing doesn't make a higher priority task ready-to-run and thus, we set the `BOOLEAN` variable `sched` to `FALSE`, L6(7). If this assumption is not verified because we will make a higher-priority-task ready-to-run then `sched` will simply be set to `TRUE`.

We then go through the wait list to see if any tasks is waiting on one or more events. If the wait list is empty (L6(8)), we simply get the current state of the event flag bits (L6(16)) and return this information to the caller (L6(17)).

If there are tasks waiting on the event flag group, we go through the list of `OS_FLAG_NODE`s to see if the new event flag bits now satisfy any of the waiting tasks conditions. Each one of the tasks can be waiting for one of four conditions, L6(9):

1) ALL of the bits specified in the PEND call to be set.
2) ANY of the bits specified in the PEND call to be set.
3) ALL of the bits specified in the PEND call to be cleared.
4) ANY of the bits specified in the PEND call to be cleared.

Note that the last two condition can be 'compiled-out' by setting `OS_FLAG_WAIT_CLR_EN` to `0` (see `OS_CFG.H`). You would do this if you didn't need the functionality of waiting for cleared bits and/or you need to reduce the amount of ROM in your product. When a waiting task's condition is satisfied (L6(10)), the waiting task is readied by calling `OS_FlagTaskRdy()` (L6(11)) (see Listing 7). Because a task is made ready-to-run, the scheduler will be called, L6(12). However, we will only do this after going through all waiting tasks because, there is no need to call the scheduler every time a task is made ready-to-run.

```
OS_FLAGS  OSFlagPost (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                          /*     Allocate storage for CPU status register       */
    OS_CPU_SR     cpu_sr;
#endif
    OS_FLAG_NODE *pnode;
    BOOLEAN       sched;
    OS_FLAGS      flags_cur;
    OS_FLAGS      flags_rdy;


#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                  /*  (1) Validate 'pgrp'                               */
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {    /*  (2) Make sure we are pointing to an event flag grp */
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
```

```
    OS_ENTER_CRITICAL();
    switch (opt) {                                  /*  (3) Determine whether we need to SET or CLEAR bits */
        case OS_FLAG_CLR:
             pgrp->OSFlagFlags &= ~flags;           /*  (4) Clear the flags specified in the group         */
             break;

        case OS_FLAG_SET:
             pgrp->OSFlagFlags |=  flags;           /*  (5) Set   the flags specified in the group         */
             break;

        default:
             OS_EXIT_CRITICAL();                    /*  (6) INVALID operation!                             */
             *err = OS_FLAG_INVALID_OPT;
             return ((OS_FLAGS)0);
    }
    sched = FALSE;                                  /*  (7) Indicate that we don't need rescheduling       */
    pnode = pgrp->OSFlagWaitList;
    while (pnode != (OS_FLAG_NODE *)0) {            /*  (8) Go through all tasks waiting on event flag(s)  */
        switch (pnode->OSFlagNodeWaitType) {
            case OS_FLAG_WAIT_SET_ALL:              /*  (9) See if all req. flags are set for current node */
                 flags_rdy = pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                 if (flags_rdy == pnode->OSFlagNodeFlags) {         /* (10)                                */
                     if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {/* (11) Make task RTR, event(s) Rx'd    */
                         sched = TRUE;                              /* (12) When done we will reschedule    */
                     }
                 }
                 break;

            case OS_FLAG_WAIT_SET_ANY:              /*     See if any flag set                            */
                 flags_rdy = pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                 if (flags_rdy != (OS_FLAGS)0) {
                     if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {/*     Make task RTR, event(s) Rx'd    */
                         sched = TRUE;                              /*     When done we will reschedule    */
                     }
                 }
                 break;

#if OS_FLAG_WAIT_CLR_EN > 0
            case OS_FLAG_WAIT_CLR_ALL:              /*     See if all req. flags are set for current node */
                 flags_rdy = ~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                 if (flags_rdy == pnode->OSFlagNodeFlags) {
                     if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {/*     Make task RTR, event(s) Rx'd    */
                         sched = TRUE;                              /*     When done we will reschedule    */
                     }
                 }
                 break;

            case OS_FLAG_WAIT_CLR_ANY:              /*     See if any flag set                            */
                 flags_rdy = ~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                 if (flags_rdy != (OS_FLAGS)0) {
                     if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {/*     Make task RTR, event(s) Rx'd    */
                         sched = TRUE;                              /*     When done we will reschedule    */
                     }
                 }
                 break;
#endif
        }
        pnode = pnode->OSFlagNodeNext;              /* (13) Point to next task waiting for event flag(s)  */
    }
    OS_EXIT_CRITICAL();
    if (sched == TRUE) {                            /* (14)                                               */
        OS_Sched();                                 /* (15)                                               */
    }
    OS_ENTER_CRITICAL();
    flags_cur = pgrp->OSFlagFlags;                  /* (16)                                               */
    OS_EXIT_CRITICAL();
    *err      = OS_NO_ERR;
    return (flags_cur);                             /* (17)                                               */
}
```

## Listing 6, Setting or Clearing bits (i.e. events) in an Event Flag Group.

We of course proceed to the next node by following the linked list, L6(13).

When we have gone through the whole waiting list, we examine the `sched` flag (L6(14)) to see if we need to run the scheduler (L6(15)) and thus possibly perform a context switch to the higher priority task that just received the event flag(s) it was waiting for. `OSFlagPost()` returns the current state of the event flag group, L6(17).

You should note that interrupts are disabled while we are going through the wait list. The implication is that `OSFlagPost()` can potentially disable interrupts for a long period of time, especially if multiple tasks are made ready-to-run.

As previously mentioned, the code in listing 7 is executed to make a task ready-to-run. This is a standard procedure in µC/OS-II except for the fact that the `OS_FLAG_NODE` needs to be unlinked from the waiting list of the event flag group, L7(3). Note that even though this function 'removes' the waiting task from the event flag group wait list, the task could still be suspended and may not be ready-to-run. This is why the `BOOLEAN` variable `sched` is used and returned to the caller.

```
static  BOOLEAN  OS_FlagTaskRdy (OS_FLAG_NODE *pnode, OS_FLAGS flags_rdy)
{
    OS_TCB   *ptcb;
    BOOLEAN   sched;


    ptcb               = (OS_TCB *)pnode->OSFlagNodeTCB;  /*     Point to TCB of waiting task             */
    ptcb->OSTCBDly     = 0;
    ptcb->OSTCBFlagsRdy = flags_rdy;
    ptcb->OSTCBStat    &= ~OS_STAT_FLAG;
    if (ptcb->OSTCBStat == OS_STAT_RDY) {                 /*     Put task into ready list                 */
        OSRdyGrp               |= ptcb->OSTCBBitY;
        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        sched                   = TRUE;                   /* (1)                                          */
    } else {
        sched                   = FALSE;                  /* (2)                                          */
    }
    OS_FlagUnlink(pnode);                                 /* (3)                                          */
    return (sched);
}
```

**Listing 7, Make a waiting Task Ready-to-Run.**

## Looking for event(s) of an Event Flag Group, OSFlagAccept()

The code to look for desired event(s) from an event flag group without waiting is shown in listing 8. This function is quite similar to `OSFlagPend()` except that the caller will not be suspended (i.e. blocked) should the event(s) not be present. The only two things that are different are:

1) `OSFlagAccept()` can be called from an ISR unlike some of the other calls.
2) If the conditions are NOT met, the call does not block and simply returns an error code that the caller should check.

```
OS_FLAGS  OSFlagAccept (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                                 /* Allocate storage for CPU status register */
    OS_CPU_SR     cpu_sr;
#endif
    OS_FLAGS      flags_cur;
    OS_FLAGS      flags_rdy;
    BOOLEAN       consume;


#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                         /* Validate 'pgrp'                          */
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {           /* Validate event block type                */
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    if (wait_type & OS_FLAG_CONSUME) {                      /* See if we need to consume the flags      */
        wait_type &= ~OS_FLAG_CONSUME;
        consume    = TRUE;
    } else {
        consume    = FALSE;
    }

    OS_ENTER_CRITICAL();
    switch (wait_type) {
        case OS_FLAG_WAIT_SET_ALL:                          /* See if all required flags are set        */
             flags_rdy = pgrp->OSFlagFlags & flags;         /* Extract only the bits we want            */
             if (flags_rdy == flags) {                      /* Must match ALL the bits that we want     */
                 if (consume == TRUE) {                     /* See if we need to consume the flags      */
                     pgrp->OSFlagFlags &= ~flags_rdy;       /* Clear ONLY the flags that we wanted      */
                 }
                 flags_cur = pgrp->OSFlagFlags;             /* Will return the state of the group       */
                 OS_EXIT_CRITICAL();                        /* Yes, condition met, return to caller     */
                 *err      = OS_NO_ERR;
             } else {
                 flags_cur = pgrp->OSFlagFlags;
                 OS_EXIT_CRITICAL();
                 *err      = OS_FLAG_ERR_NOT_RDY;
             }
             break;

        case OS_FLAG_WAIT_SET_ANY:
             flags_rdy = pgrp->OSFlagFlags & flags;         /* Extract only the bits we want            */
             if (flags_rdy != (OS_FLAGS)0) {                /* See if any flag set                      */
                 if (consume == TRUE) {                     /* See if we need to consume the flags      */
                     pgrp->OSFlagFlags &= ~flags_rdy;       /* Clear ONLY the flags that we got         */
                 }
                 flags_cur = pgrp->OSFlagFlags;             /* Will return the state of the group       */
                 OS_EXIT_CRITICAL();                        /* Yes, condition met, return to caller     */
                 *err      = OS_NO_ERR;
             } else {
                 flags_cur = pgrp->OSFlagFlags;
                 OS_EXIT_CRITICAL();
                 *err      = OS_FLAG_ERR_NOT_RDY;
             }
             break;

#if OS_FLAG_WAIT_CLR_EN > 0
        case OS_FLAG_WAIT_CLR_ALL:                          /* See if all required flags are cleared    */
             flags_rdy = ~pgrp->OSFlagFlags & flags;        /* Extract only the bits we want            */
             if (flags_rdy == flags) {                      /* Must match ALL the bits that we want     */
                 if (consume == TRUE) {                     /* See if we need to consume the flags      */
```

```
                    pgrp->OSFlagFlags |= flags_rdy;       /* Set ONLY the flags that we wanted        */
                }
                flags_cur = pgrp->OSFlagFlags;            /* Will return the state of the group       */
                OS_EXIT_CRITICAL();                       /* Yes, condition met, return to caller     */
                *err      = OS_NO_ERR;
            } else {
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_FLAG_ERR_NOT_RDY;
            }
            break;

        case OS_FLAG_WAIT_CLR_ANY:
            flags_rdy = ~pgrp->OSFlagFlags & flags;       /* Extract only the bits we want            */
            if (flags_rdy != (OS_FLAGS)0) {               /* See if any flag cleared                  */
                if (consume == TRUE) {                    /* See if we need to consume the flags      */
                    pgrp->OSFlagFlags |= flags_rdy;       /* Set ONLY the flags that we got           */
                }
                flags_cur = pgrp->OSFlagFlags;            /* Will return the state of the group       */
                OS_EXIT_CRITICAL();                       /* Yes, condition met, return to caller     */
                *err      = OS_NO_ERR;
            } else {
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_FLAG_ERR_NOT_RDY;
            }
            break;
#endif

        default:
            OS_EXIT_CRITICAL();
            flags_cur = (OS_FLAGS)0;
            *err      = OS_FLAG_ERR_WAIT_TYPE;
            break;
    }
    return (flags_cur);
}
```

**Listing 8, Looking for Event Flags without waiting.**

## Querying an Event Flag Group, OSFlagQuery()

`OSFlagQuery()` allows your code to get the current value of the event flag group. The code for this function is shown in listing 9.

`OSFlagQuery()` is passed two arguments: `pgrp` contains a pointer to the event flag group which is returned what `OSFlagCreate()` returns when the event flag group is created and, `err` which is a pointer to an error code that will let the caller know whether the call was successful or not.

As with all µC/OS-II calls, `OSFlagQuery()` performs argument checking if this feature is enabled by OS_ARG_CHK_EN L9(1) and L9(2).

If there are no errors, `OSFlagQuery()` obtains the current state of the event flags L9(3) and returns this to the caller, L9(4).

```
OS_FLAGS  OSFlagQuery (OS_FLAG_GRP *pgrp, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                       /*    Allocate storage for CPU status register           */
    OS_CPU_SR  cpu_sr;
#endif
    OS_FLAGS   flags;


#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {               /* (1) Validate 'pgrp'                                   */
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) { /* (2) Validate event block type                         */
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    OS_ENTER_CRITICAL();
    flags = pgrp->OSFlagFlags;                    /* (3) Get current event flags                           */
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return (flags);                               /* (4) Return the current value of the event flags       */
}
```

**Listing 9, Obtaining the current flags of an event flag group.**

# OSFlagAccept()

```
OS_FLAGS OSFlagAccept(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT8U *err);
```

| File | Called from | Code enabled by |
|---|---|---|
| OS_FLAG.C | Task | OS_FLAG_EN && OS_FLAG_ACCEPT_EN |

`OSFlagAccept()` allows you to check the status of a combination of bits to be either set or cleared in an event flag group. Your application can check for ANY bit to be set/cleared or ALL bits to be set/cleared. This function behaves exactly as `OSFlagPend()` except that the caller will NOT block if the desired event flags are not present.

**Arguments**

`pgrp` is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created (see `OSFlagCreate()`).

`flags` is a bit pattern indicating which bit(s) (i.e. flags) you wish to check. The bits you want are specified by setting the corresponding bits in `flags`.

`wait_type` specifies whether you want ALL bits to be set/cleared or ANY of the bits to be set/cleared. You can specify the following argument:

| | |
|---|---|
| `OS_FLAG_WAIT_CLR_ALL` | You will check ALL bits in 'flags' to be clear (0) |
| `OS_FLAG_WAIT_CLR_ANY` | You will check ANY bit in 'flags' to be clear (0) |
| `OS_FLAG_WAIT_SET_ALL` | You will check ALL bits in 'flags' to be set (1) |
| `OS_FLAG_WAIT_SET_ANY` | You will check ANY bit in 'flags' to be set (1) |

You can add `OS_FLAG_CONSUME` if you want the event flag(s) to be 'consumed' by the call. For example, to wait for ANY flag in a group and then clear the flags that are present, set `wait_type` to:

```
OS_FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME
```

`err` a pointer to an error code and can be:

| | |
|---|---|
| `OS_NO_ERR` | No error |
| `OS_ERR_EVENT_TYPE` | You are not pointing to an event flag group |
| `OS_FLAG_ERR_WAIT_TYPE` | You didn't specify a proper 'wait_type' argument. |
| `OS_FLAG_INVALID_PGRP` | You passed a NULL pointer instead of the event flag handle. |
| `OS_FLAG_ERR_NOT_RDY` | The desired flags you are waiting for are not available. |

**Returned Value**

The state of the flags in the event flag group.

**Notes/Warnings**

1) The event flag group must be created before it is used.
2) This function does NOT block if the desired flags are not present.

## Example

```
#define  ENGINE_OIL_PRES_OK   0x01
#define  ENGINE_OIL_TEMP_OK   0x02
#define  ENGINE_START         0x04


OS_FLAG_GRP *EngineStatus;


void Task (void *pdata)
{
    INT8U     err;
    OS_FLAGS  value;


    pdata = pdata;
    for (;;) {
        value  =  OSFlagAccept(EngineStatus,  ENGINE_OIL_PRES_OK  +  ENGINE_OIL_TEMP_OK,  OS_FLAG_WAIT_SET_ALL,
&err);
        switch (err) {
            case OS_NO_ERR:
                 /* Desired flags are available */
                 break;

            case OS_FLAG_ERR_NOT_RDY:
                 /* The desired flags are NOT available */
                 break;
        }
        .
        .
    }
}
```

# OSFlagCreate()

**OS_FLAG_GRP *OSFlagCreate(OS_FLAGS flags, INT8U *err);**

| File | Called from | Code enabled by |
|---|---|---|
| OS_FLAG.C | Task or startup code | OS_FLAG_EN |

`OSFlagCreate()` is used to create and initialize an event flag group.

### Arguments

`flags` contains the initial value to store in the event flag group.

`err` is a pointer to a variable which will be used to hold an error code. The error code can be one of the following:

| | |
|---|---|
| `OS_NO_ERR` | if the call was successful and the event flag group was created. |
| `OS_ERR_CREATE_ISR` | if you attempted to create an event flag group from an ISR. |
| `OS_FLAG_GRP_DEPLETED` | if there are no more event flag groups available. You will need to increase the value of `OS_MAX_FLAGS` in `OS_CFG.H`. |

### Returned Value

A pointer to the event flag group if a free one is available. If no event flag group is available, `OSFlagCreate()` will return a `NULL` pointer.

### Notes/Warnings

1) Event flag groups must be created by this function before they can be used by the other services.

### Example

```
OS_FLAG_GRP *EngineStatus;


void main (void)
{
    INT8U  err;

    .
    .
    OSInit();                                /* Initialize µC/OS-II                                          */
    .
    .
    EngineStatus = OSFlagCreate(0x00, &err); /* Create an event flag group containing the engine's status    */
    .
    .
    OSStart();                               /* Start Multitasking                                           */
}
```

# OSFlagDel()

```
OS_FLAG_GRP *OSFlagDel(OS_FLAG_GRP *pgrp, INT8U opt, INT8U *err);
```

| File | Called from | Code enabled by |
|---|---|---|
| OS_FLAG.C | Task | OS_FLAG_EN and OS_FLAG_DEL_EN |

`OSFlagDel()` is used to delete an event flag group. This is a dangerous function to use because multiple tasks could be relying on the presence of the event flag group. You should always use this function with great care. Generally speaking, before you would delete an event flag group, you would first delete all the tasks that access the event flag group.

### Arguments

`pgrp` is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created (see `OSFlagCreate()`).

`opt` specifies whether you want to delete the event flag group only if there are no pending tasks (`OS_DEL_NO_PEND`) or whether you always want to delete the event flag group regardless of whether tasks are pending or not (`OS_DEL_ALWAYS`). In this case, all pending task will be readied.

`err` is a pointer to a variable which will be used to hold an error code. The error code can be one of the following:

| | |
|---|---|
| `OS_NO_ERR` | if the call was successful and the event flag group was deleted. |
| `OS_ERR_DEL_ISR` | if you attempted to delete an event flag group from an ISR. |
| `OS_ERR_EVENT_TYPE` | if `pgrp` is not pointing to an event flag group. |
| `OS_ERR_INVALID_OPT` | if you didn't specify one of the two options mentioned above. |
| `OS_ERR_TASK_WAITING` | if one or more task were waiting on the event flag group and and you specified `OS_DEL_NO_PEND`. |
| `OS_FLAG_INVALID_PGRP` | if you passed a `NULL` pointer in `pgrp`. |

### Returned Value

A `NULL` pointer if the event flag group is deleted or `pgrp` if the event flag group was not deleted. In the latter case, you would need to examine the error code to determine the reason.

### Notes/Warnings

1) You should use this call with care because other tasks may expect the presence of the event flag group.
2) This call can potentially disable interrupts for a long time. The interrupt disable time is directly proportional to the number of tasks waiting on the event flag group.

**Example**

```
OS_FLAG_GRP *EngineStatusFlags;


void Task (void *pdata)
{
    INT8U        err;
    OS_FLAG_GRP *pgrp;


    pdata = pdata;
    while (1) {
        .
        .
        pgrp = OSFlagDel(EngineStatusFlags, OS_DEL_ALWAYS, &err);
        if (pgrp == (OS_FLAG_GRP *)0) {
            /* The event flag group was deleted */
        }
        .
        .
    }
}
```

# OSFlagPend()

```
OS_FLAGS OSFlagPend(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout, INT8U *err);
```

| File | Called from | Code enabled by |
|---|---|---|
| OS_FLAG.C | Task only | OS_FLAG_EN |

`OSFlagPend()` is used to have a task wait for a combination of conditions (i.e. events or bits) to be set (or cleared) in an event flag group. You application can wait for ANY condition to be set (or cleared) or, ALL conditions to be either set or cleared. If the events that the calling task desires are not available then, the calling task will be blocked until the desired conditions are satisfied or, the specified timeout expires.

**Arguments**

`pgrp` is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created (see `OSFlagCreate()`).

`flags` is a bit pattern indicating which bit(s) (i.e. flags) you wish to check. The bits you want are specified by setting the corresponding bits in `flags`.

`wait_type` specifies whether you want ALL bits to be set/cleared or ANY of the bits to be set/cleared. You can specify the following argument:

| | |
|---|---|
| `OS_FLAG_WAIT_CLR_ALL` | You will check ALL bits in 'flags' to be clear (0) |
| `OS_FLAG_WAIT_CLR_ANY` | You will check ANY bit in 'flags' to be clear (0) |
| `OS_FLAG_WAIT_SET_ALL` | You will check ALL bits in 'flags' to be set (1) |
| `OS_FLAG_WAIT_SET_ANY` | You will check ANY bit in 'flags' to be set (1) |

You can also specify whether the flags will be 'consumed' by adding `OS_FLAG_CONSUME` to the `wait_type`. For example, to wait for ANY flag in a group and then CLEAR the flags that satisfy the condition, set `wait_type` to:

```
OS_FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME
```

`err` a pointer to an error code and can be:

| | |
|---|---|
| `OS_NO_ERR` | No error |
| `OS_ERR_PEND_ISR` | You tried to call `OSFlagPend` from an ISR which is not allowed. |
| `OS_ERR_EVENT_TYPE` | You are not pointing to an event flag group |
| `OS_FLAG_ERR_WAIT_TYPE` | You didn't specify a proper 'wait_type' argument. |
| `OS_FLAG_INVALID_PGRP` | You passed a NULL pointer instead of the event flag handle. |
| `OS_FLAG_ERR_NOT_RDY` | The desired flags you are waiting for are not available. |
| `OS_TIMEOUT` | The flags were not available within the specified amount of time. |

**Returned Value**

The value of the flags in the event flag group after they are consumed (if `OS_FLAG_CONSUME` is specified) or, the state of the flags just before `OSFlagPend()` returns. `OSFlagPend()` returns `0` if a timeout occurs.

**Notes/Warnings**

1) The event flag group must be created before it's used.

**Example**

```
#define  ENGINE_OIL_PRES_OK   0x01
#define  ENGINE_OIL_TEMP_OK   0x02
#define  ENGINE_START         0x04


OS_FLAG_GRP *EngineStatus;


void Task (void *pdata)
{
    INT8U     err;
    OS_FLAGS  value;


    pdata = pdata;
    for (;;) {
        value = OSFlagPend(EngineStatus,
                           ENGINE_OIL_PRES_OK   + ENGINE_OIL_TEMP_OK,
                           OS_FLAG_WAIT_SET_ALL + OS_FLAG_CONSUME,
                           10,
                           &err);
        switch (err) {
            case OS_NO_ERR:
                 /* Desired flags are available */
                 break;

            case OS_TIMEOUT:
                 /* The desired flags were NOT available before 10 ticks occurred */
                 break;
        }
        .
        .
    }
}
```

# OSFlagPost()

```
OS_FLAGS OSFlagPost(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *err);
```

| File | Called from | Code enabled by |
|---|---|---|
| OS_FLAG.C | Task or ISR | OS_FLAG_EN |

You set or clear event flag bits by calling `OSFlagPost()`. The bits set or cleared are specified in a 'bit mask'. `OSFlagPost()` will ready each task that has it's desired bits satisfied by this call. You can set or clear bits that are already set or cleared.

**Arguments**

`pgrp` is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created (see `OSFlagCreate()`).

`flags` specifies which bits you want set or cleared. If `opt` (see below) is `OS_FLAG_SET`, each bit that is set in `'flags'` will set the corresponding bit in the event flag group. e.g. to set bits 0, 4 and 5 you would set `flags` to 0x31 (note, bit 0 is least significant bit). If `opt` (see below) is `OS_FLAG_CLR`, each bit that is set in `flags` will CLEAR the corresponding bit in the event flag group. e.g. to clear bits 0, 4 and 5 you would specify `'flags'` as 0x31 (note, bit 0 is least significant bit).

`opt` indicates whether the flags will be set (`OS_FLAG_SET`) or cleared (`OS_FLAG_CLR`).

`err` is a pointer to an error code and can be:

| | |
|---|---|
| `OS_NO_ERR` | The call was successfull |
| `OS_FLAG_INVALID_PGRP` | You passed a `NULL` pointer |
| `OS_ERR_EVENT_TYPE` | You are not pointing to an event flag group |
| `OS_FLAG_INVALID_OPT` | You specified an invalid option |

**Returned Value**

The new value of the event flags.

**Notes/Warnings**

1) Event flag groups must be created before they are used.
2) The execution time of this function depends on the number of tasks waiting on the event flag group. However, the execution time is deterministic.
3) The amount of time interrupts are DISABLED also depends on the number of tasks waiting on the event flag group.

**Example**

```
#define  ENGINE_OIL_PRES_OK   0x01
#define  ENGINE_OIL_TEMP_OK   0x02
#define  ENGINE_START         0x04


OS_FLAG_GRP  *EngineStatusFlags;


void  TaskX (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    for (;;) {
        .
        .
        err = OSFlagPost(EngineStatusFlags, ENGINE_START, OS_FLAG_SET, &err);
        .
        .
    }
}
```

# OSFlagQuery()

```
OS_FLAGS OSFlagQuery(OS_FLAG_GRP *pgrp, INT8U *err);
```

| File | Called from | Code enabled by |
|---|---|---|
| OS_FLAG.C | Task or ISR | OS_FLAG_EN && OS_FLAG_QUERY_EN |

`OSFlagQuery()` is used to obtain the current value of the event flags in a group. At this time, this function does NOT return the list of tasks waiting for the event flag group.

### Arguments

`pgrp` is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created (see `OSFlagCreate()`).

`err` is a pointer to an error code and can be:

| | |
|---|---|
| `OS_NO_ERR` | The call was successfull |
| `OS_FLAG_INVALID_PGRP` | You passed a `NULL` pointer |
| `OS_ERR_EVENT_TYPE` | You are not pointing to an event flag group |

### Returned Value

The state of the flags in the event flag group.

### Notes/Warnings

1) The event flag group to query must be created.
2) You can call this function from an ISR.

### Example

In this example, we check the contents of the mutex to determine the highest priority task that is waiting for it.

```
OS_FLAG_GRP *EngineStatusFlags;


void Task (void *pdata)
{
    OS_FLAGS flags;
    INT8U    err;


    pdata = pdata;
    for (;;) {
        .
        .
        flags = OSFlagQuery(EngineStatusFlags, &err);
        .
        .
    }
}
```

# References

***µC/OS-II, The Real-Time Kernel***
Jean J. Labrosse
R&D Technical Books, 1998
ISBN 0-87930-543-6

# Contacts

**Micriμm, Inc.**
949 Crestview Circle
Weston, FL 33327
954-217-2036
954-217-2037 (FAX)
e-mail: Jean.Labrosse@Micrium.com
WEB: www.Micrium.com

**R&D Books, Inc.**
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-9950
(785) 841-1631
(785) 841-2624 (FAX)
WEB: http://www.rdbooks.com
e-mail: rdorders@rdbooks.com