uC/OS-II Part 4:
Task Management

Prof. Li-Pin Chang
Embedded Software and Storage Lab.

Objectives

- To understand how the following kernel services are implemented:
  - creating a task,
  - deleting a task,
  - changing the priority of a task,
  - suspending a task,
  - resuming a task, and
  - obtaining information of a task.

Tasks

- A task is either periodic or aperiodic

```
void YourTask (void *pdata)
{
   for (;;) {
      /* USER CODE */
      Call one of uC/OS-II's services:
      OSMboxPend();
      OSQPend();
      OSSemPend();
      OSTaskDel(OS_PRIO_SELF);
      OSTaskSuspend(OS_PRIO_SELF);
      OSTimeDly();
      OSTimeDlyHMSM();
      /* USER CODE */
      }
}
```

A task periodically executes user code

```
void YourTask (void *pdata)
{
   /* USER CODE */
   OSTaskDel(OS_PRIO_SELF);
}
```

A job is created on demand.

Creating a Task

- You must create at least one task before multitasking is started.
  - Calling OSInit() and OSStatInit() will implicitly create 2 tasks (it must be done before OSStart())
- Necessary data structures for a new tasks
  - A Task Control Block (TCB)
  - A stack
  - An entry of the priority table
- Do not create tasks in ISR
  - May cause deadlocks because task creation will involve user-level context switch

OSTaskCreate()

```
INT8U  OSTaskCreate (void (*task)(void *pd),
           void *pdata, OS_STK *ptos, INT8U prio)
{
   OS_STK    *psp;
   INT8U     err;

   OS_ENTER_CRITICAL();
   if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {
      OSTCBPrioTbl[prio] = (OS_TCB *)1;

      OS_EXIT_CRITICAL();
      psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, 0);
```

Occupying a priority table slot and re-enable interrupts immediately.

Push the entry address onto the stack as a “return” address

Stack initialization is a hardware-dependant implementation. (Because of the growing direction)

OSTaskCreate()

```
        err = OS_TCBInit(prio, psp, (OS_STK *)0, 0, 0, (void *)0, 0);
        if (err == OS_NO_ERR) {
            OS_ENTER_CRITICAL();
            OSTaskCtr++;
            OS_EXIT_CRITICAL();
            if (OSRunning == TRUE) {
                OS_Sched();
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;
            OS_EXIT_CRITICAL();
        }
        return (err);
    }
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
```

Create a corresponding TCB and connect it to the priority table.

Call the scheduler if multitasking is enabled

Q: What does the stack of a new task look like?

OSTaskCreateExt()

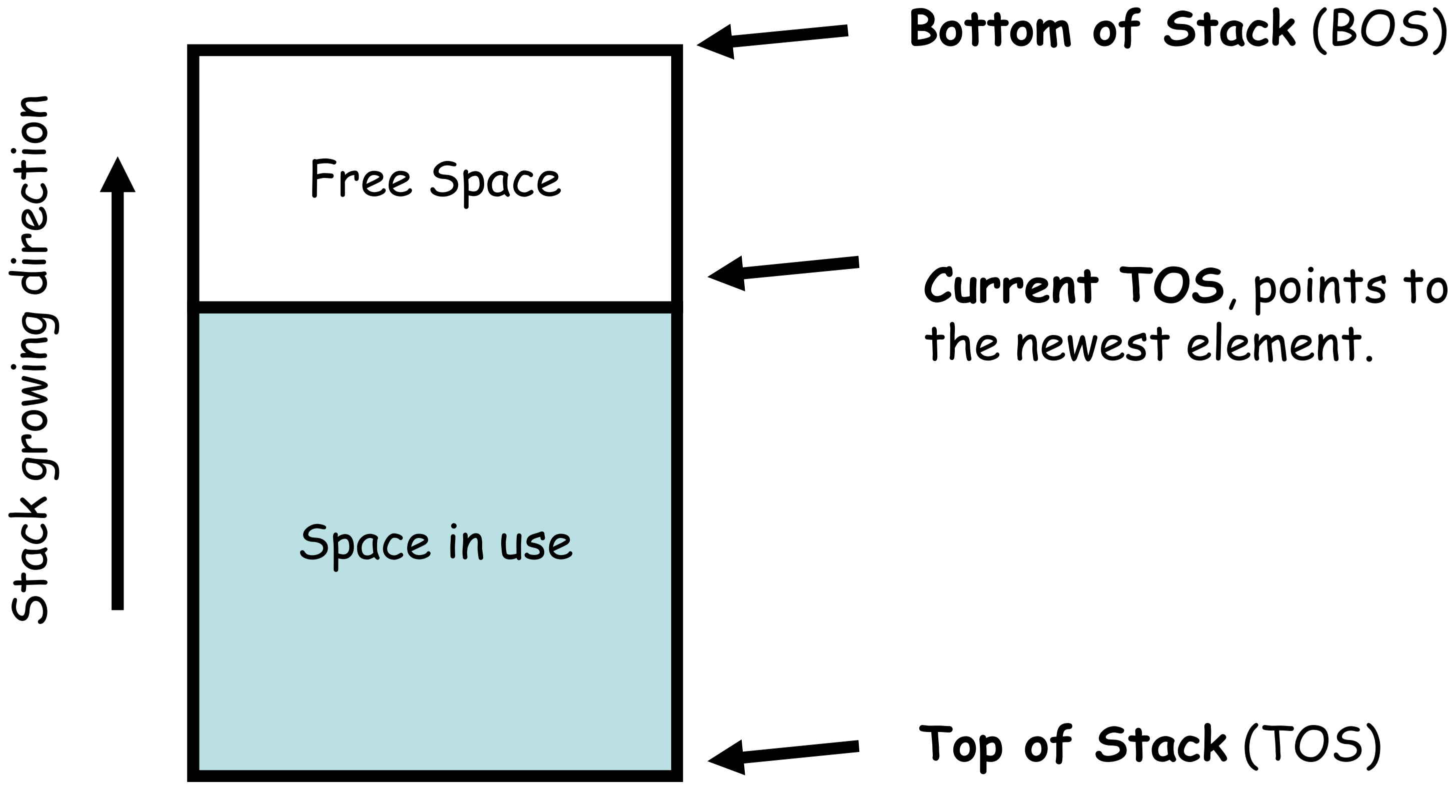
```
INT8U  OSTaskCreateExt (void   (*task)(void *pd),  void    *pdata,
                  OS_STK  *ptos, INT8U    prio, INT16U   id, OS_STK  *pbos,
                  INT32U   stk_size, void    *pext, INT16U   opt)
{
   OS_STK    *psp;
   INT8U      err;


   OS_ENTER_CRITICAL();
   if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {
      OSTCBPrioTbl[prio] = (OS_TCB *)1;

      OS_EXIT_CRITICAL();

      if (((opt & OS_TASK_OPT_STK_CHK) != 0x0000) ||
         ((opt & OS_TASK_OPT_STK_CLR) != 0x0000)) {
         #if OS_STK_GROWTH == 1
         (void)memset(pbos, 0, stk_size * sizeof(OS_STK));
         #else
         (void)memset(ptos, 0, stk_size * sizeof(OS_STK));
         #endif
      }
```

The stack is required to be cleared

The stack grows toward low address, so the starting address is bos.

The stack grows toward high address, so the starting address is tos.

OSTaskCreateExt()


Stack growing direction
Free Space
Space in use
Bottom of Stack (BOS)
Current TOS, points to the newest element.
Top of Stack (TOS)

OSTaskCreateExt()

- **task**: a pointer-to-function points to the entry point of a task (note the syntax).
- **pdata**: a parameter passed to the task.
- **ptos**: a pointer points to the top-of-stack.
- **prio**: task priority.
- **id**: task id, for future extension.
- **pbos**: a pointer points to the bottom-of-stack.
- **stk_size**: the stack size in the number of elements (OS_STK bytes each) 1 word under x86
- **pext**: an user-defined extension to the TCB.
- **opt**: the options specified to create the task.

OSTaskCreateExt()

```
        psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, opt);
        err = OS_TCBInit(prio, psp, pbos, id, stk_size, pext, opt);
        if (err == OS_NO_ERR) {
            OS_ENTER_CRITICAL();
            OSTaskCtr++;
            OS_EXIT_CRITICAL();
            if (OSRunning == TRUE) {
                OS_Sched();
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;
            OS_EXIT_CRITICAL();
        }
        return (err);
    }
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
```

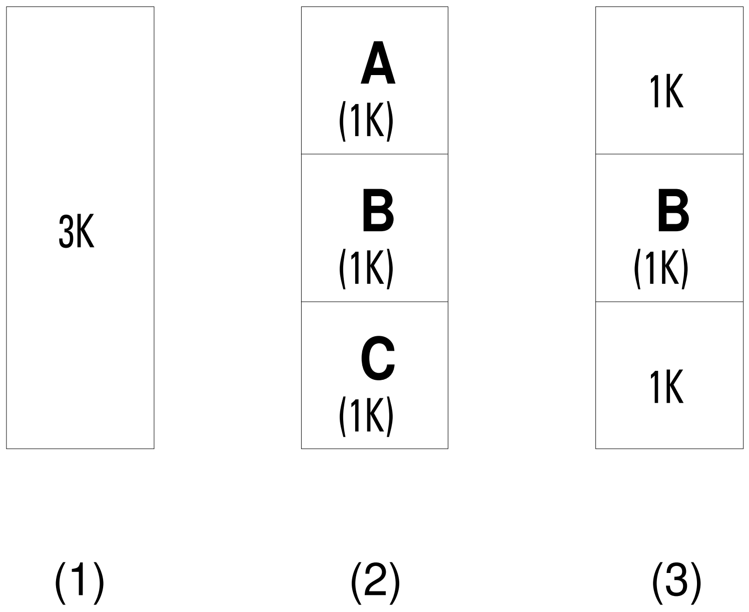
OSTaskCreate:
(task, pdata, ptos, 0);
(prio, psp, (OS_STK *)0, 0, 0, (void *)0, 0);

OSTaskCreateExt:
(task, pdata, ptos, opt);
(prio, psp, pbos, id, stk_size, pext, opt);

Task Stacks

- The stack is a contiguous memory space
  - Allocated from global arrays
- The element size is hardware dependent
  - 16 bits in x86
  - Defined by a macro OS_STK
- Stack may grow upward or downward
  - Toward “low” memory addresses in x86

Task Stacks



•OS_STK_GROWTH == 0
Stacks grow toward high addresses.

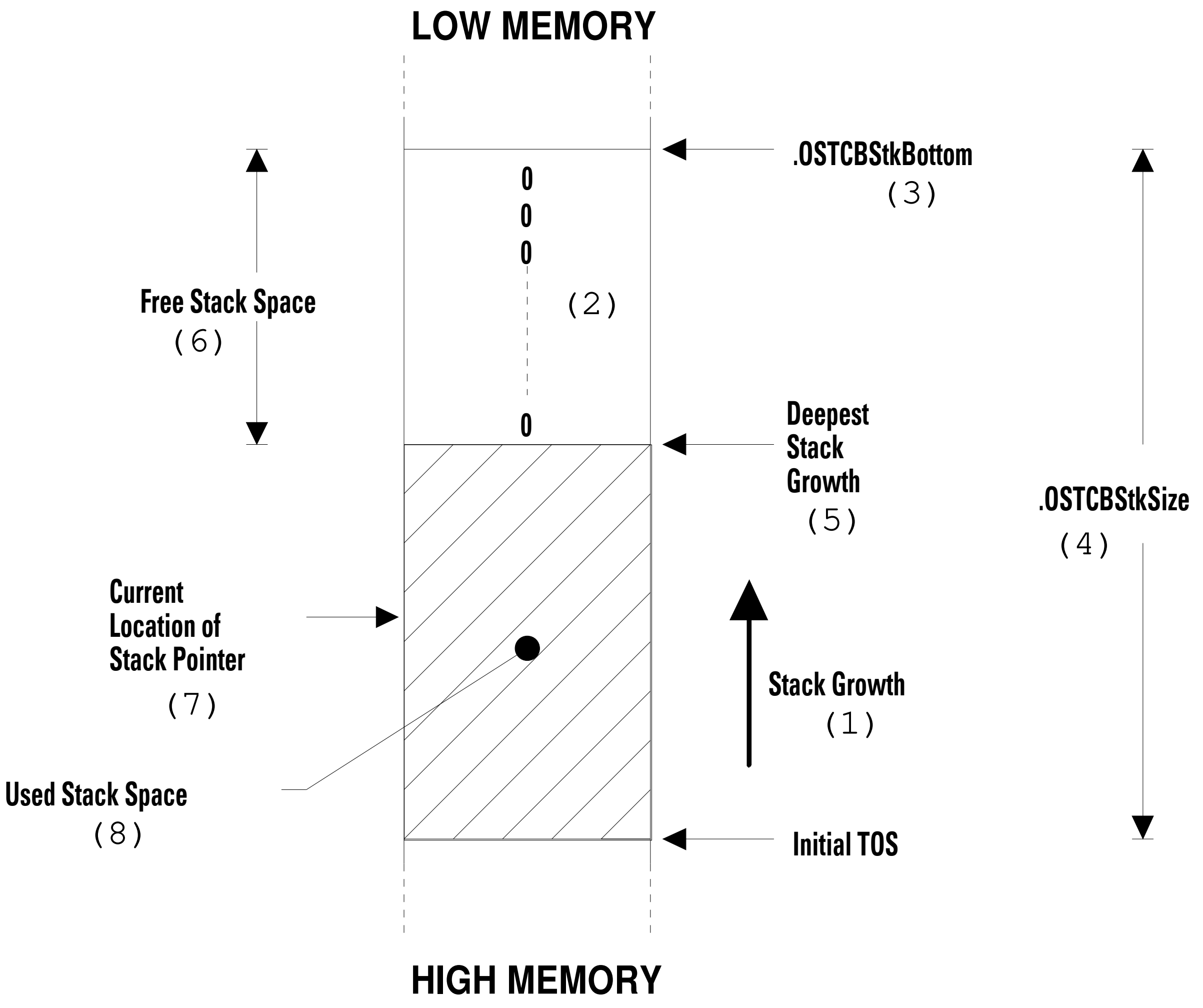
•OS_STK_GROWTH == 1
Stacks grow toward low addresses.

```
OS_STK  TaskStack[TASK_STACK_SIZE];
#if OS_STK_GROWTH == 0
   OSTaskCreate(task, pdata, &TaskStack[0], prio);
#else
   OSTaskCreate(task, pdata, &TaskStack[TASK_STACK_SIZE-1], prio);
#endif
```

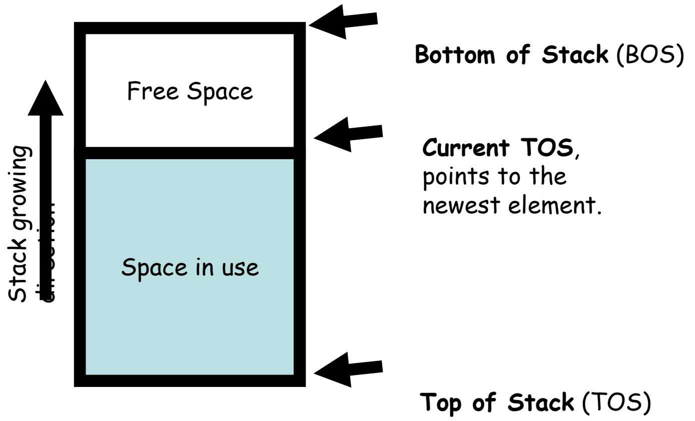
Stack Checking

- Stack checking intends to determine the maximum run-time usage of stacks
- To do stack checking:
  - Set OS_TASK_CREATE_EXT to 1 in OS_CFG.h
  - Create your tasks by using OSTaskCreateExt() with options OS_TASK_OPT_STK_CHK + OS_TASK_OPT_SRK_CLR and give the tasks reasonably large stacks
  - Call OSTaskStkChk() to determine the stack usage of a certain task
  - Reduce the stack size if possible, once you think you had run enough simulation

Stack Checking



```
INT8U  OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata)
{
    OS_TCB    *ptcb;
    OS_STK    *pchk;
    INT32U     free, size;

    pdata->OSFree = 0;
    pdata->OSUsed = 0;
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb == (OS_TCB *)0) {
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_EXIST);
    }
    if ((ptcb->OSTCBOpt & OS_TASK_OPT_STK_CHK) == 0) {
        OS_EXIT_CRITICAL();
        return (OS_TASK_OPT_ERR);
    }
```



```
    free = 0;
    size = ptcb->OSTCBStkSize;
    pchk = ptcb->OSTCBStkBottom;
    OS_EXIT_CRITICAL();
#if OS_STK_GROWTH == 1
    while (*pchk++ == (OS_STK)0) {
        free++;
    }
#else
    while (*pchk-- == (OS_STK)0) {
        free++;
    }
#endif
    pdata->OSFree = free * sizeof(OS_STK);
    pdata->OSUsed = (size - free) * sizeof(OS_STK);
    return (OS_NO_ERR);
}
```

For either stack growing direction…

Counting from BOS until a non-zero element is encountered.

Deleting a Task

- Task deletion releases all data structures (e.g., TCB) associate with the deleted task
  - The code resides in ROM are still there
- Deleting a task is slightly more complicated than creating it since every resources/objects held by the task must be returned to the operating system

Procedures

1. Prevent from deleting an idle task
   - OS_ARG_CHK_EN > 0
2. Prevent from deleting a task from within an ISR
   - Because OS_Sched() is called after deletion
   - Because the task to be deleted may be the interrupted task
3. Verify that the task to be deleted does exist
4. Remove the OS_TCB
5. Remove the task from ready list or other lists
6. Set .OSTCBStat to OS_STAT_RDY
7. Call OSDummy()
8. Call OSTaskDelHook()
9. Remove the OS_TCB from priority table
10. Call OSSched()

```
INT8U  OSTaskDel (INT8U prio)
{
    OS_EVENT     *pevent;
    OS_FLAG_NODE *pnode;
    OS_TCB       *ptcb;
    BOOLEAN       self;

    if (OSIntNesting > 0) {
        return (OS_TASK_DEL_ISR);
    }
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {
        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
            OSRdyGrp &= ~ptcb->OSTCBBitY;
        }
        pevent = ptcb->OSTCBEventPtr;
        if (pevent != (OS_EVENT *)0) {
            if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
                pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
            }
        }
        pnode = ptcb->OSTCBFlagNode;
        if (pnode != (OS_FLAG_NODE *)0) {
            OS_FlagUnlink(pnode);
        }
```

We do not allow to delete a task within ISR's, because the ISR might currently interrupts that task.

Clear the corresponding bit of the task-to-delete in the ready list.

If the row are all 0's, then clear the RdyGrp bit also.

Remove the task from the event control block since we no longer wait for the event.

```
    ptcb->OSTCBDly  = 0;
    ptcb->OSTCBStat = OS_STAT_RDY;
    if (OSLockNesting < 255) {
        OSLockNesting++;
    }
    OS_EXIT_CRITICAL();
    OS_Dummy();
    OS_ENTER_CRITICAL();
    if (OSLockNesting > 0) {
        OSLockNesting--;
    }
    OSTaskDelHook(ptcb);
    OSTaskCtr--;
    OSTCBPrioTbl[prio] = (OS_TCB *)0;
    if (ptcb->OSTCBPrev == (OS_TCB *)0) {
        ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0;
        OSTCBList                  = ptcb->OSTCBNext;
    } else {
        ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;
        ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;
    }
    ptcb->OSTCBNext = OSTCBFreeList;
    OSTCBFreeList   = ptcb;
    OS_EXIT_CRITICAL();
    OS_Sched();
    return (OS_NO_ERR);
}
OS_EXIT_CRITICAL();
return (OS_TASK_DEL_ERR);
}
```

Prevent tickISR from making this task ready when interrupts are re-enabled later.

Prevent this task from being resumed since we are not in the ready list now (a "ready" task can not be resumed).

Lock the scheduler.

Interrupts are re-enabled for a while (note that the scheduler is locked).
*What does OS_dummy() do?

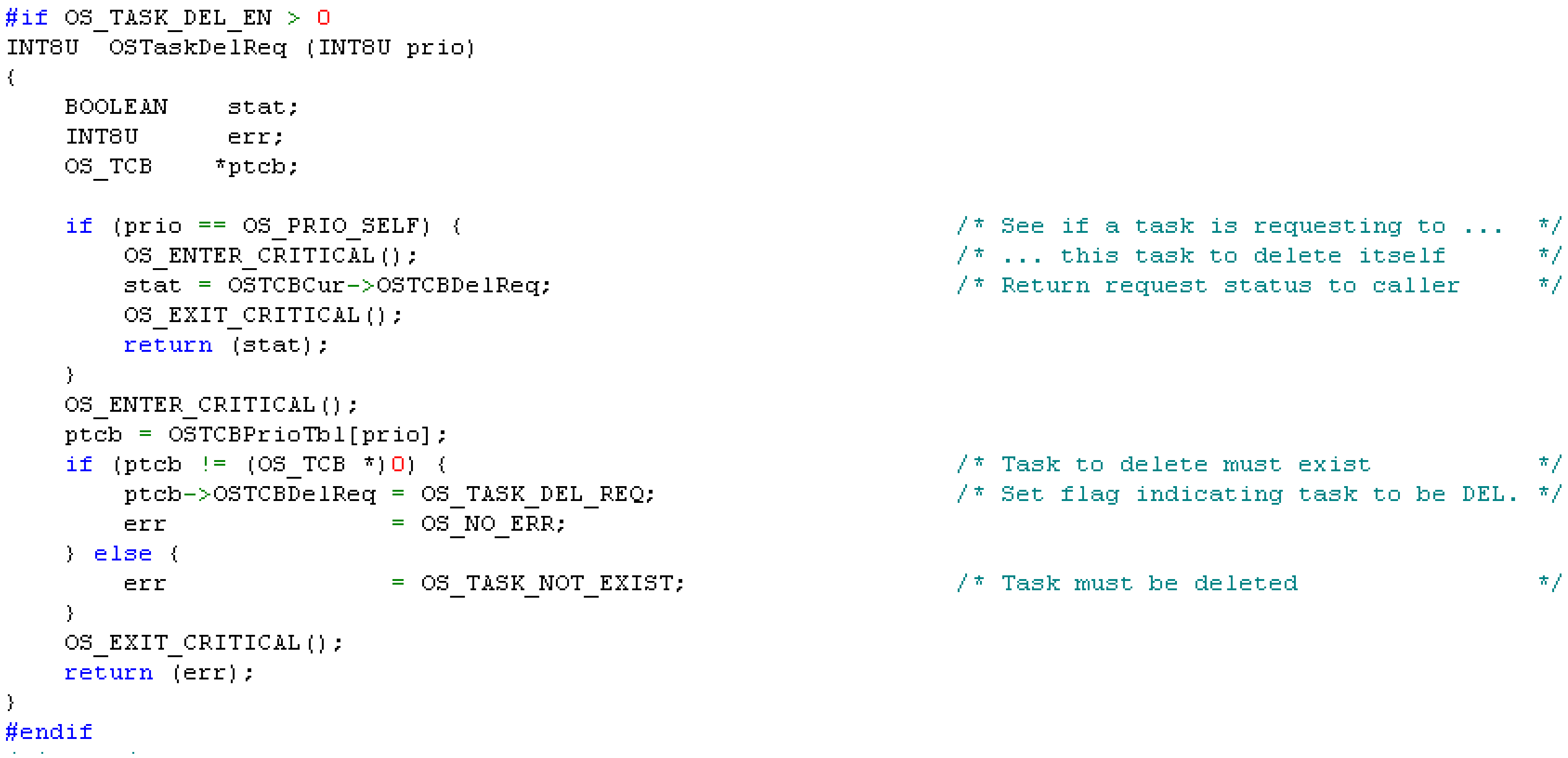
Necessary if the task is deleting itself

OSTaskDelReq()

- A task can safely delete itself after it explicitly releases all resources
  - OSTaskDel(OS_PRIO_SELF)
- It is, however, dangerous to delete a task asynchronously
  - May leave resources unreleased
  - A safer approach is to send a deletion request to a task
  - The task can delete itself after releasing all resources

Requesting a Task to Delete Itself

```
void RequestorTask(void *pdata) {
    for (;;) {
    /*application code*/
        if (the task “TaskToBeDeleted()” needs to be deleted) {
               while(OSTaskDelReq(TASK_TO_DEL_PRIO) !=
                                      OS_TASK_NOT_EXIT)
                       OSTimeDly(1);
    }
}

void TaskToBeDeleted(void *pdata) {
    while(1) {
        /*application code*/
        if (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ) {
            /*release any owned resources;
            de-allocated any dynamic memory; */
            OSTaskDel(OS_PRIO_SELF);
        } else {
            /*application code)*/
        }
    }
}
```



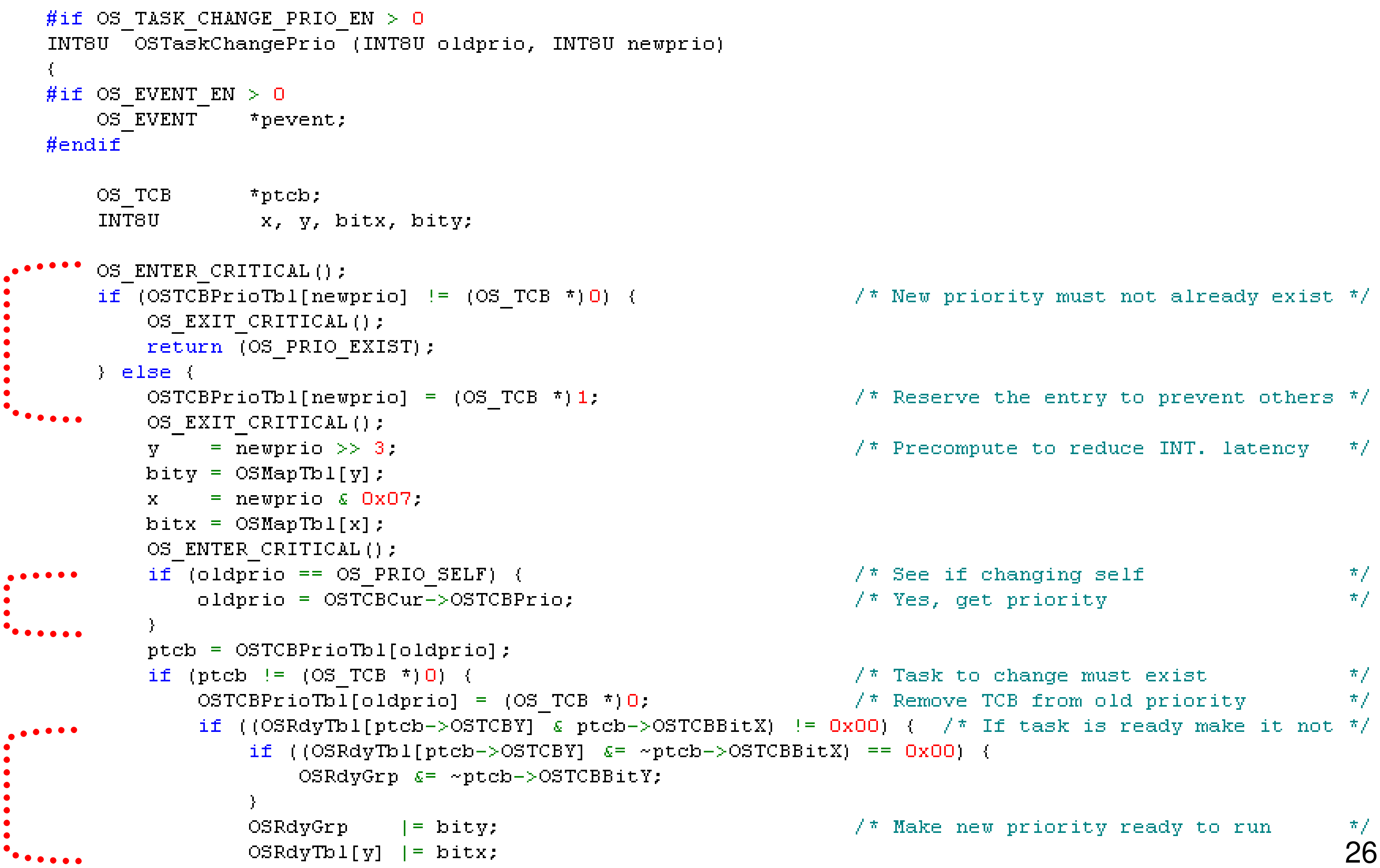
```c
#if OS_TASK_DEL_EN > 0
INT8U  OSTaskDelReq (INT8U prio)
{
    BOOLEAN    stat;
    INT8U      err;
    OS_TCB    *ptcb;

    if (prio == OS_PRIO_SELF) {                                 /* See if a task is requesting to ...  */
        OS_ENTER_CRITICAL();                                    /* ... this task to delete itself      */
        stat = OSTCBCur->OSTCBDelReq;                           /* Return request status to caller     */
        OS_EXIT_CRITICAL();
        return (stat);
    }
    OS_ENTER_CRITICAL();
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {                                  /* Task to delete must exist           */
        ptcb->OSTCBDelReq = OS_TASK_DEL_REQ;                    /* Set flag indicating task to be DEL. */
        err               = OS_NO_ERR;
    } else {
        err               = OS_TASK_NOT_EXIST;                  /* Task must be deleted                */
    }
    OS_EXIT_CRITICAL();
    return (err);
}
#endif
```

Changing a Task’s Priority

- When you create a new task, you assign the task a priority
- During run time, you can change this priority using OSTaskChangePrio(oldPrio, newPrio)
- Cannot change the priority of the idle task

Procedures

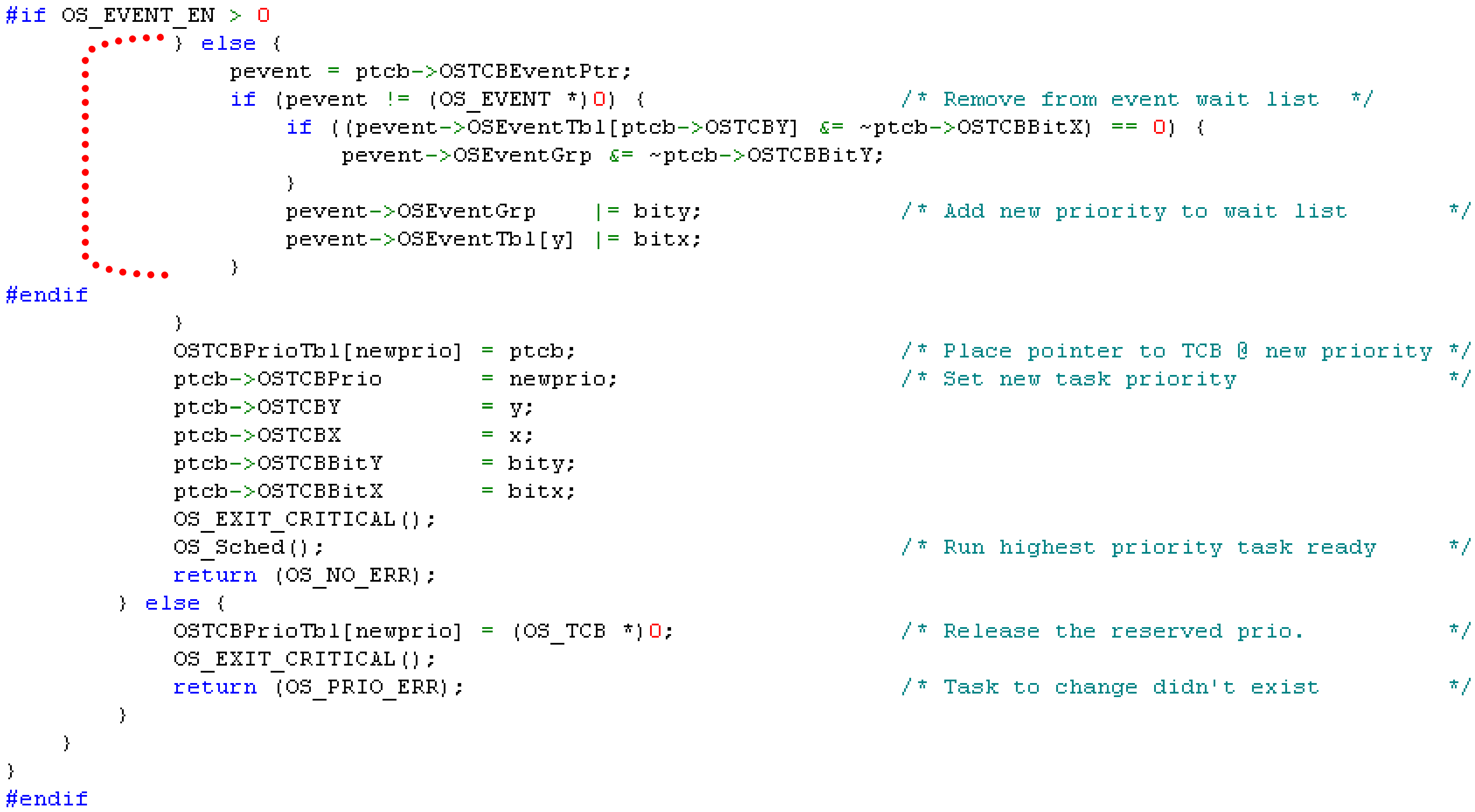
1. Reserve the new priority by OSTCBPrioTbl[newprio] = (OS_TCB *) 1;
2. Remove the task from the priority table; insert the task to the new location in the priority table
3. Adjust the ready list (task is ready)
4. Adjust the event waiting list (task is waiting)
5. Change the OS_TCB of the task
6. Call OSSched()



```
#if OS_TASK_CHANGE_PRIO_EN > 0
INT8U  OSTaskChangePrio (INT8U oldprio, INT8U newprio)
{
#if OS_EVENT_EN > 0
    OS_EVENT    *pevent;
#endif

    OS_TCB      *ptcb;
    INT8U        x, y, bitx, bity;

    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[newprio] != (OS_TCB *)0) {                 /* New priority must not already exist */
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    } else {
        OSTCBPrioTbl[newprio] = (OS_TCB *)1;                    /* Reserve the entry to prevent others */
        OS_EXIT_CRITICAL();
        y    = newprio >> 3;                                    /* Precompute to reduce INT. latency   */
        bity = OSMapTbl[y];
        x    = newprio & 0x07;
        bitx = OSMapTbl[x];
        OS_ENTER_CRITICAL();
        if (oldprio == OS_PRIO_SELF) {                          /* See if changing self                */
            oldprio = OSTCBCur->OSTCBPrio;                      /* Yes, get priority                   */
        }
        ptcb = OSTCBPrioTbl[oldprio];
        if (ptcb != (OS_TCB *)0) {                              /* Task to change must exist           */
            OSTCBPrioTbl[oldprio] = (OS_TCB *)0;                /* Remove TCB from old priority        */
            if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) {  /* If task is ready make it not */
                if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
                    OSRdyGrp &= ~ptcb->OSTCBBitY;
                }
                OSRdyGrp    |= bity;                            /* Make new priority ready to run      */
                OSRdyTbl[y] |= bitx;
```

If the task is ready, modify the ready list.

The task is not ready but on a waiting list. Modify the waiting list accordingly



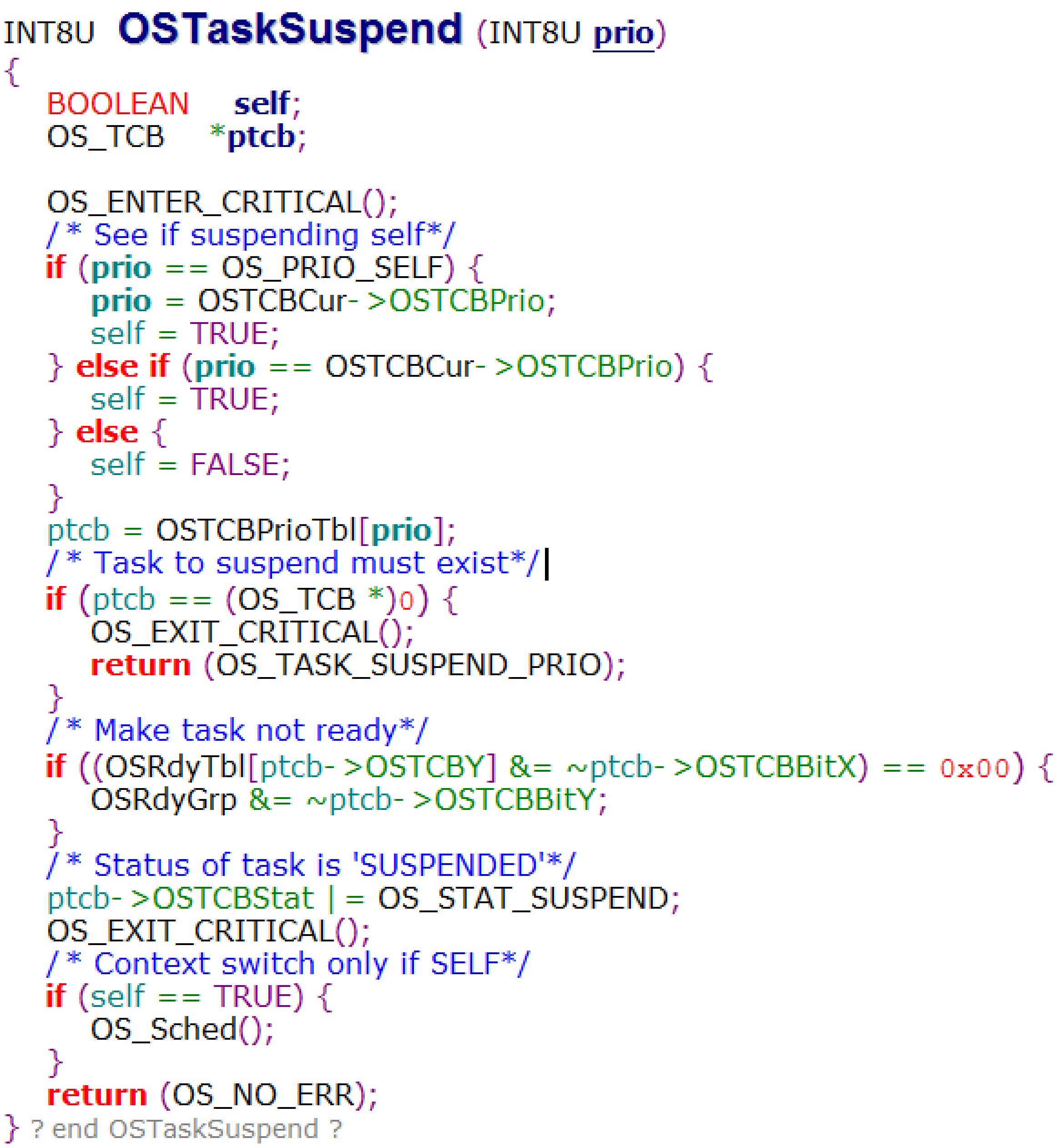
```
#if OS_EVENT_EN > 0
            } else {
                pevent = ptcb->OSTCBEventPtr;
                if (pevent != (OS_EVENT *)0) {                  /* Remove from event wait list  */
                    if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
                        pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
                    }
                    pevent->OSEventGrp    |= bity;              /* Add new priority to wait list       */
                    pevent->OSEventTbl[y] |= bitx;
                }
#endif
            }
            OSTCBPrioTbl[newprio] = ptcb;                       /* Place pointer to TCB @ new priority */
            ptcb->OSTCBPrio       = newprio;                    /* Set new task priority               */
            ptcb->OSTCBY          = y;
            ptcb->OSTCBX          = x;
            ptcb->OSTCBBitY       = bity;
            ptcb->OSTCBBitX       = bitx;
            OS_EXIT_CRITICAL();
            OS_Sched();                                         /* Run highest priority task ready     */
            return (OS_NO_ERR);
        } else {
            OSTCBPrioTbl[newprio] = (OS_TCB *)0;                /* Release the reserved prio.          */
            OS_EXIT_CRITICAL();
            return (OS_PRIO_ERR);                               /* Task to change didn't exist         */
        }
    }
}
#endif
```

Suspending a Task

- A task put suspended is neither ready nor waiting
- A suspended task can only be resumed by OSTaskResume()
- Suspending a task waiting for an event may deadlock a system!
- INT8U OSTaskSuspend(INT8U prio)
- INT8U OSTaskResume(INT8U prio)

Procedure – OSTaskSuspend()

1. Check the input priority
2. Remove the task from the ready list
3. Set the OS_STAT_SUSPEND flag in OS_TCB
4. Call OSSched(), if self suspension



```
INT8U  OSTaskSuspend (INT8U prio)
{
    BOOLEAN    self;
    OS_TCB    *ptcb;

    OS_ENTER_CRITICAL();
    /* See if suspending self*/
    if (prio == OS_PRIO_SELF) {
        prio = OSTCBCur- >OSTCBPrio;
        self = TRUE;
    } else if (prio == OSTCBCur- >OSTCBPrio) {
        self = TRUE;
    } else {
        self = FALSE;
    }
    ptcb = OSTCBPrioTbl[prio];
    /* Task to suspend must exist*/
    if (ptcb == (OS_TCB *)0) {
        OS_EXIT_CRITICAL();
        return (OS_TASK_SUSPEND_PRIO);
    }
    /* Make task not ready*/
    if ((OSRdyTbl[ptcb- >OSTCBY] &= ~ptcb- >OSTCBBitX) == 0x00) {
        OSRdyGrp &= ~ptcb- >OSTCBBitY;
    }
    /* Status of task is 'SUSPENDED'*/
    ptcb- >OSTCBStat | = OS_STAT_SUSPEND;
    OS_EXIT_CRITICAL();
    /* Context switch only if SELF*/
    if (self == TRUE) {
        OS_Sched();
    }
    return (OS_NO_ERR);
} ? end OSTaskSuspend ?
```

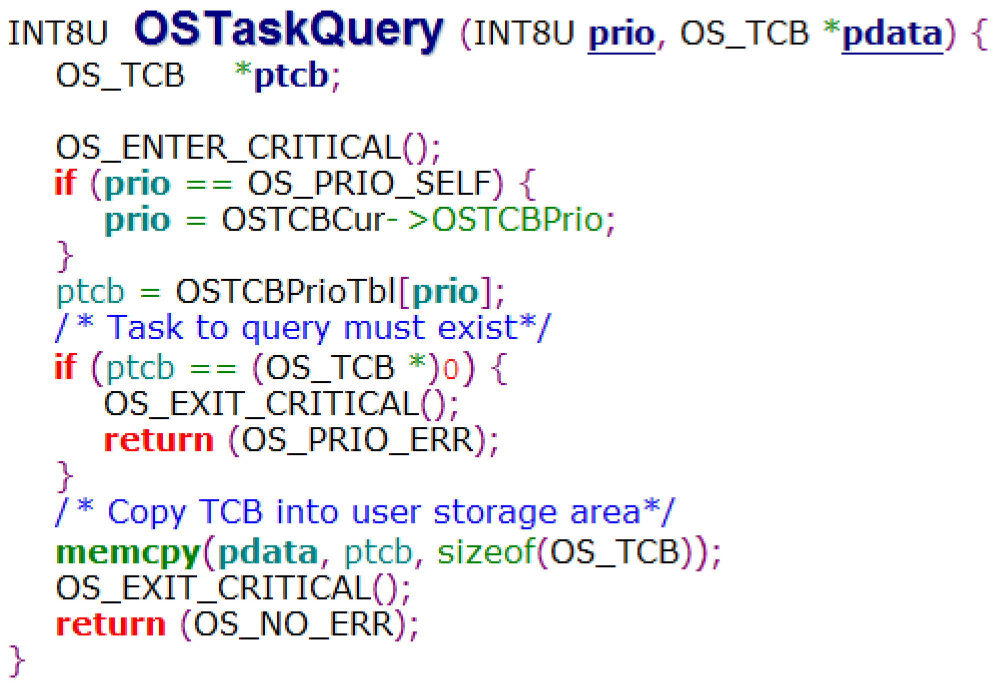
Procedure – OSTaskResume()

1. Check the input priority
2. Clear the OS_STAT_SUSPEND bit in the OSTCBStat field
3. Add the task back to the ready list
4. Call OSSched()

```
INT8U  OSTaskResume (INT8U prio) {
    OS_TCB    *ptcb;

    OS_ENTER_CRITICAL();
    ptcb = OSTCBPrioTbl[prio];
    /* Task to suspend must exist*/
    if (ptcb == (OS_TCB *)0) {
        OS_EXIT_CRITICAL();
        return (OS_TASK_RESUME_PRIO);
    }
    /* Task must be suspended   */
    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) != 0x00) {
      /* Remove suspension         */
        if (((ptcb->OSTCBStat &= ~OS_STAT_SUSPEND) == OS_STAT_RDY) &&
             /* Must not be delayed      */
             (ptcb->OSTCBDly  == 0)) {
            /* Make task ready to run   */
            OSRdyGrp               |= ptcb->OSTCBBitY;
            OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
            OS_EXIT_CRITICAL();
            OS_Sched();
        } else {
            OS_EXIT_CRITICAL();
        }
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();
    return (OS_TASK_NOT_SUSPENDED);
} ? end OSTaskResume ?
```

Getting Information About a Task

- OSTaskQuery return a copy of the contents of the desired task's OS_TCB
- To call OSTaskQuery, your application must allocate storage for an OS_TCB
- Only you this function to SEE what a task is doing
  - don't modify the contains (OSTCBNext, OSTCBPrev)



```
INT8U  OSTaskQuery (INT8U prio, OS_TCB *pdata) {
    OS_TCB     *ptcb;

    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {
        prio = OSTCBCur- >OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    / * Task to query must exist*/
    if (ptcb == (OS_TCB *)0) {
        OS_EXIT_CRITICAL();
        return (OS_PRIO_ERR);
    }
    / * Copy TCB into user storage area*/
    memcpy(pdata, ptcb, sizeof(OS_TCB));
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

Summary

- In this part, you should have learned how to
  - create, delete, suspend, and resume tasks
  - change the priority of a task

- Materials in this part require in-depth understanding of kernel structures
  - Ready list
  - TCBs