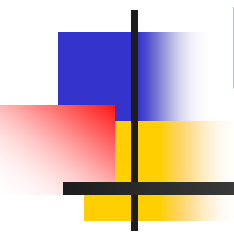
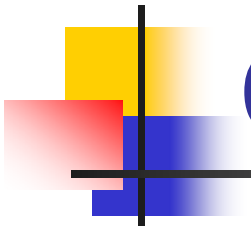
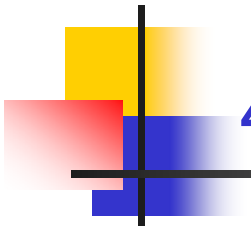


# MicroC/OS-II Chapter 4

中興資科所 盧慶達
學號:79256022
指導教授 張軒彬



# CHAPTER 4 Task Management



# 4.00 Creating a Task[OSTaskCreate()]

- you create a task by passing its address and other arguments to one of two functions:OSTaskCreate(),OSTaskCreateExt().
- OSTaskCreate() is backward compatible with μC/OS.
- OSTaskCreateExt() is an extended version of the OSTaskCreate(),providing additional features.
- OSTaskCreate requires four arguments:

  1.task is a pointer to the task code.

  2.pdata is a pointer to an that is passed to your task when it start executing.

  3.ptos is a pointer to the top of the stack that is assigned to the task.

  4.prio is the desired task priority.

# 4.00 Creating a Task[OSTaskCreate()]

```
INT8U  OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)
{
#if OS_CRITICAL_METHOD == 3                /* Allocate storage for CPU status register           */
    OS_CPU_SR  cpu_sr;
#endif
    OS_STK    *psp;
    INT8U      err;
#if OS_ARG_CHK_EN > 0
    if (prio > OS_LOWEST_PRIO) {           /* Make sure priority is within allowable range       */
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) { /* Make sure task doesn't already exist at this priority  */
        OSTCBPrioTbl[prio] = (OS_TCB *)1;    /* Reserve the priority to prevent others from doing ...  */
                                             /* ... the same thing until task is created.              */
        OS_EXIT_CRITICAL();
        psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, 0);    /* Initialize the task's stack        */
        err = OS_TCBInit(prio, psp, (OS_STK *)0, 0, 0, (void *)0, 0);
```

# 4.00 Creating a Task[OSTaskCreate()]

```
if (err == OS_NO_ERR) {
        OS_ENTER_CRITICAL();
        OSTaskCtr++;                          /* Increment the #tasks counter        */
        OS_EXIT_CRITICAL();
        if (OSRunning == TRUE) {        /* Find highest priority task if multitasking has started */
            OS_Sched();
        }

} else {
        OS_ENTER_CRITICAL();
        OSTCBPrioTbl[prio] = (OS_TCB *)0;/* Make this priority available to others                */
        OS_EXIT_CRITICAL();
    }
    return (err);
  }
  OS_EXIT_CRITICAL();
  return (OS_PRIO_EXIST);
}
```

# 4.00 Creating a Task[OSTaskCreate()]

- the priority of idle task is reserved by OSInit().
- task's priority must unique priority.
- μC/OS-II reserves the priority by placing a non_NULL pointer in OSTCBPrioTbl[].
- function *OSTaskStkInit(task,pdata,ptos,0)* responsible for setting up the task's stack.
- *OSTaskStkInit()* return the new top_of _stack which saved in the task's OS_TCB.
- *OS_STACK_GROWTH* reserves the stack grow.
- call *OSTCBInit()* to obtain and initialize an *OS_TCB* from the pool of free *OS_TCBs*.
- *OSTCBInit(prio,psp,(void*)0,0,0,(void*)0,0)* return OS_NO_ERR if task get tcb, OS_NO_MORE_TCB if no tcb in TCBFreeList.

# 4.01 Creating a Task[OSTaskCreateExt()]

- OSTaskCreateExt() offers more flexibility but at the expense of additional overload.

```
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)
{
#if OS_CRITICAL_METHOD == 3          /* Allocate storage for CPU status register           */
    OS_CPU_SR  cpu_sr;
#endif
    OS_STK    *psp;
    INT8U      err;
```

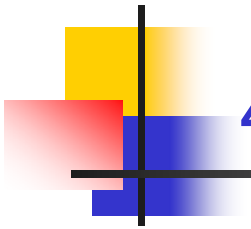
# 4.01 Creating a Task[OSTaskCreateExt()]

```
#if OS_ARG_CHK_EN > 0
    if (prio > OS_LOWEST_PRIO) {           /* Make sure priority is within allowable range          */
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) { /* Make sure task doesn't already exist at this priority  */
        OSTCBPrioTbl[prio] = (OS_TCB *)1;    /* Reserve the priority to prevent others from doing ...  */
                                             /* ... the same thing until task is created.              */
        OS_EXIT_CRITICAL();

        if (((opt & OS_TASK_OPT_STK_CHK) != 0x0000) ||   /* See if stack checking has been enabled     */
            ((opt & OS_TASK_OPT_STK_CLR) != 0x0000)) {   /* See if stack needs to be cleared           */
            #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
        }
```

# 4.01 Creating a Task[OSTaskCreateExt()]

```
psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, opt); /* Initialize the task's stack        */
        err = OS_TCBInit(prio, psp, pbos, id, stk_size, pext, opt);
        if (err == OS_NO_ERR) {
            OS_ENTER_CRITICAL();
            OSTaskCtr++;                                /* Increment the #tasks counter       */
            OS_EXIT_CRITICAL();
            if (OSRunning == TRUE) {                    /* Find HPT if multitasking has started */
                OS_Sched();
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;           /* Make this priority avail. to others  */
            OS_EXIT_CRITICAL();
        }
        return (err);
    }
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
```



# 4.02 Task Stacks

- a stack must declared as being of type OS_STK and must consist of contiguous memory locations.
- you can allocate stack space either statically or dynamically.

## static stack

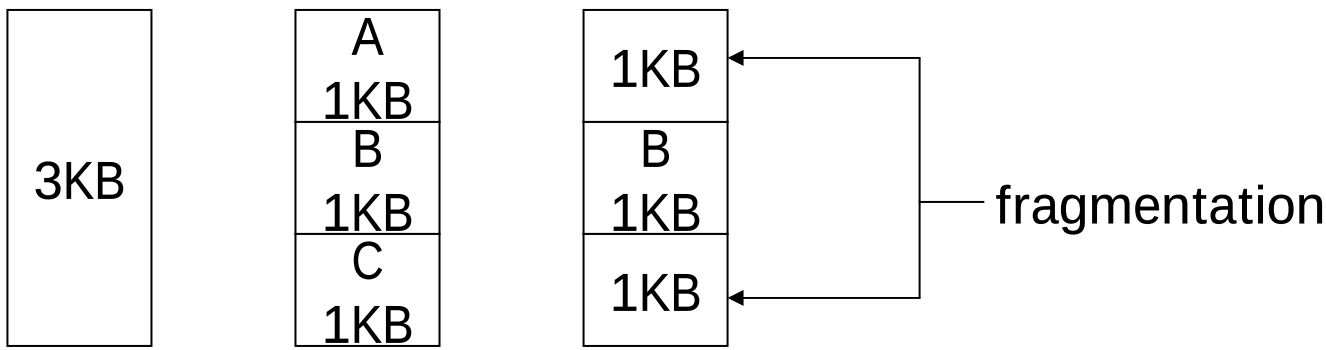
```
Static OS_STK MyTaskStack[stack_size];
or
OS_STK MyTaskStack[stack_size];
```

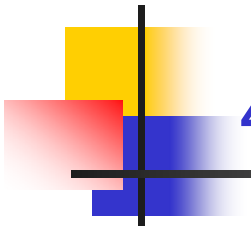
## dynamically stack

```
OS_STK *pstk;
pstk=(OS_STK*)malloc(stack_size); /*C compiler's malloc() function*/
if(pstk !=(OS_STK*)0){
Create the task;
}
```

# 4.02 Task Stacks

-dynamically allocate must be careful with fragmentation.





# 4.02 Task Stacks

-you must know how ths stack grows when you call either OSTaskCreate() or OSTaskCreateExt() because you need to pass the task's top_of_stack to these functions.

```
OS_STK TaskStk[TASK_STK_SIZE];
#if OS_STACK_GROWTH==0
OSTaskCreate(task,pdata,&Task[0],prio)
#else
OSTaskCreate(task,pdata,&Task[TASK_STK_SIZE-1],prio)
#endif
```

-the size of task needed by your task is application specific.

# 4.03 Stack Checking[OSTaskStkChk()]

-Stack checking allow you to reduce the amount of RAM needed by your application code.

- μC/OS-II provides *OSTaskStkChk()*.

-in order to use the μC/OS-II stack checking facilities you must do the following:

1.Set *OS_TASK_CREATE_EXT=1* in OS_CFG.H.

2.Create task use *OSTaskCreateExt()* and give more space than you think it really needs.

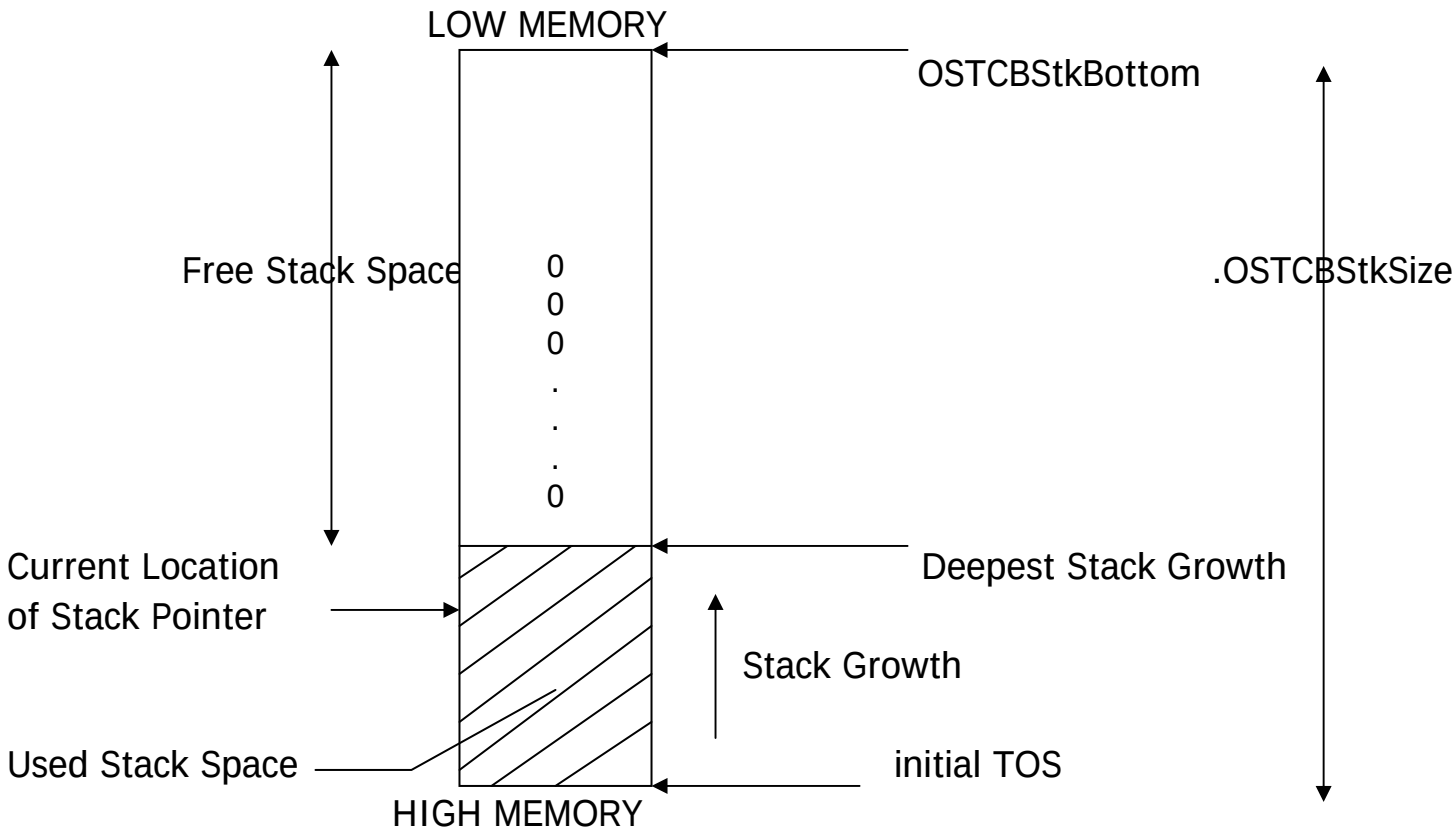
3.Set the *opt* argument in *OSTaskCreateExt()* to *OS_TASK_OPT_STK_CHK+OS_TASK_OPT_STK_CLR*

# 4.03 Stack Checking[OSTaskStkChk()]

- if your start code clears all RAM and you never delete tasks after they are created,you don’t need to set the *OS_TASK_OPT_STK_CLR.*

4.call *OSTaskChk()* from a task by specifying the priority of the task you want to check.

# 4.03 Stack Checking[OSTaskStkChk()]



# 4.03 Stack Checking[OSTaskStkChk()]

- µC/OS-II determine stack growth by looking at the contents of the stack itself.
- µC/OS-II requires that the stack be filled with zeros when the task is created.
- if task created with *OSTaskCreateExt()*, the location of the bottom_of_stack and the size of the stack are stored in the task's OS_TCB.
- *OSTaskStkChk()* computes the location of the bottom of stack and counting the number of zero-value entries on the stack until a nonzero value is found.

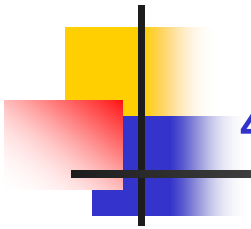
# 4.03 Stack Checking[OSTaskStkChk()]

- used stack=total size-zero-value entries.
- *OSTaskCreateExt()* places the number of bytes free and the number of bytes used in a data structure of type *OS_STK_DATA*.

```
typedef struct {
    INT32U  OSFree;                /* Number of free bytes on the stack                */
    INT32U  OSUsed;                /* Number of bytes used on the stack                */
} OS_STK_DATA;
```

- you may get a different value for amount of free space on the stack until your task has reached its deepest growth.

-you have to run the application long enough, to get worst case requirement, then you can go back to set the final size of your stack.



# 4.03 Stack Checking[OSTaskStkChk()]

- you should accommodate system expansion, so make sure you allocate between 10 and 100 percent more than what OSTaskStkChk() report.

# 4.03 Stack Checking[OSTaskStkChk()]

```
INT8U  OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3                  /* Allocate storage for CPU status register           */
    OS_CPU_SR  cpu_sr;
#endif
    OS_TCB    *ptcb;
    OS_STK    *pchk;
    INT32U     free;
    INT32U     size;


#if OS_ARG_CHK_EN > 0
    if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {        /* Make sure task priority is valid    */
        return (OS_PRIO_INVALID);
    }
#endif
    pdata->OSFree = 0;                                  /* Assume failure, set to 0 size       */
    pdata->OSUsed = 0;
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                  /* See if check for SELF                        */
        prio = OSTCBCur->OSTCBPrio;
    }
```

# 4.03 Stack Checking[OSTaskStkChk()]

```
ptcb = OSTCBPrioTbl[prio];
    if (ptcb == (OS_TCB *)0) {                    /* Make sure task exist                      */
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_EXIST);}
    if ((ptcb->OSTCBOpt & OS_TASK_OPT_STK_CHK) == 0) { /* Make sure stack checking option is set  */
        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) {                /* Compute the number of zero entries on the stk */
        free++;
    }
#else
    while (*pchk-- == (OS_STK)0) {
        free++;}
#endif
    pdata->OSFree = free * sizeof(OS_STK);        /* Compute number of free bytes on the stack     */
    pdata->OSUsed = (size - free) * sizeof(OS_STK);   /* Compute number of bytes used on the stack     */
   return (OS_NO_ERR);}
```

# 4.04 Deleting a Task[OSTaskDel()]

-Deleting a task means that the task is returned to the dormant state, the task code is simply no longer scheduled by μC/OS-II.
-you delete a task by calling OSTaskDel().

```
 INT8U  OSTaskDel (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3                  /* Allocate storage for CPU status register          */
    OS_CPU_SR     cpu_sr;
#endif

#if OS_EVENT_EN > 0
    OS_EVENT     *pevent;
#endif
#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
    OS_FLAG_NODE *pnode;
#endif
    OS_TCB       *ptcb;
    BOOLEAN       self;
```

# 4.04 Deleting a Task[OSTaskDel()]

```
if (OSIntNesting > 0) {            (1)                    /* See if trying to delete from ISR    */
        return (OS_TASK_DEL_ISR); }
#if OS_ARG_CHK_EN > 0  (2)
    if (prio == OS_IDLE_PRIO) {                          /* Not allowed to delete idle task     */
        return (OS_TASK_DEL_IDLE);}
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {       /* Task priority valid ?               */
        return (OS_PRIO_INVALID);}
#endif
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                          /* See if requesting to delete self    */
        prio = OSTCBCur->OSTCBPrio;}                       /* Set priority to delete to current   */
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {                            /* Task to delete must exist      */
        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {  /* Make task not ready           */
            OSRdyGrp &= ~ptcb->OSTCBBitY;}
#if OS_EVENT_EN > 0
        pevent = ptcb->OSTCBEventPtr;
        if (pevent != (OS_EVENT *)0) {                    /* If task is waiting on event         */
            if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) { /* ... remove task from */
                pevent->OSEventGrp &= ~ptcb->OSTCBBitY; }}                    /* ... event ctrl block */
```

# 4.04 Deleting a Task[OSTaskDel()]

```
#endif
#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
    pnode = ptcb->OSTCBFlagNode;
    if (pnode != (OS_FLAG_NODE *)0) {          /* If task is waiting on event flag    */
        OS_FlagUnlink(pnode);                  /* Remove from wait list               */
    }
#endif
    ptcb->OSTCBDly  = 0;                       /* Prevent OSTimeTick() from updating  */
    ptcb->OSTCBStat = OS_STAT_RDY;             /* Prevent task from being resumed     */
        if (OSLockNesting < 255) {
        OSLockNesting++;                                          (3)
        }
    OS_EXIT_CRITICAL();                        /* Enabling INT. ignores next instruc. */
    OS_Dummy();                                /* ... Dummy ensures that INTs will be */
    OS_ENTER_CRITICAL();                       /* ... disabled HERE!                  */
        if (OSLockNesting > 0) {
        OSLockNesting--;
        }
```

# 4.04 Deleting a Task[OSTaskDel()]

```
OSTaskDelHook(ptcb);                                /* Call user defined hook                */
        OSTaskCtr--;                                /* One less task being managed           */
        OSTCBPrioTbl[prio] = (OS_TCB *)0;           /* Clear old priority entry              */
        if (ptcb->OSTCBPrev == (OS_TCB *)0) {       /* Remove from TCB chain                 */
            ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0;
            OSTCBList                  = ptcb->OSTCBNext;
        } else {
            ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;
            ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;
        }
        ptcb->OSTCBNext = OSTCBFreeList;            /* Return TCB to free TCB list           */
        OSTCBFreeList   = ptcb;
        OS_EXIT_CRITICAL();
        OS_Sched();                                 /* Find new highest priority task        */
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();
    return (OS_TASK_DEL_ERR);
}
```

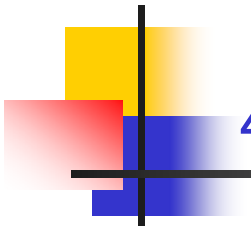
# 4.04 Deleting a Task[OSTaskDel()]

- you can't delete a task from within an ISR. (1)
- if you enable *OS_ARG_CHK_EN*, it will check,

  1.if you try to delete idle task *OSTaskDel()* return error message *OS_TASK_DEL_IDLE.*

  2.if you give the argument prio larger than *OS_LOWEST_PRIO* or try to delete task which priority not the same as caller *OSTaskDel()* return error message *OS_PRIO_INVALID.*(2)
- The call can delete itself by specifying *OS_PRIO_SELF as argument.Ex: OSTaskDel(OS_PRIO_SELF)*

-OSTaskDel() verifies that the task to delete does in fact exist.



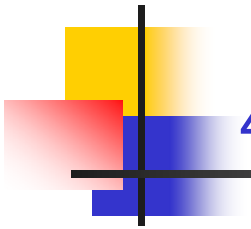
# 4.04 Deleting a Task[OSTaskDel()]

-if task in the ready list, it is removed.

-if task in a list waiting for a mutex, mailbox,... it removed from that list.

-if task in a list waiting for event flag, it is removed from that list.

-OSTaskDel() sets the task's OSTCBStat flag to $OS\_STAT\_READY$ it is preventing another task or an ISR from resuming this task.



# 4.04 Deleting a Task[OSTaskDel()]

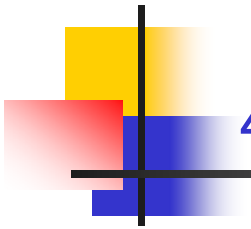
-OSTaskDel() must prevent the scheduler from switching to another task because if the current task is almost deleted, it could not be rescheduled.(3)

-to reduce interrupt latency OSTaskDel() re-enable interrupt.

-we need to unlink from OS_TCB from OS_TCB chain and return the OS_TCB to free OS_TCB list.

-dummy function *OS_Dummy* is called because some CPU to have interrupt disabled until the end of instruction .

- *OS_Dummy* ensure that I executed a call and a return instruction before re-disabling interrupts.

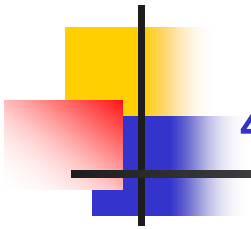


# 4.05 Requesting to Delete a Task[OSTaskDelReq()]

-call OSTaskDelReq() to delete task which owns resources.

-Both the requester and the task to be deleted need to call OSTaskReq().

```
void RequesttorTask(void *pdata)
{
INT8U err;
pdata=pdata;
for(;;){
/* Application code*/
if('TaskToBeDeleted()' need to be deleted) {
While(OSTask(TASK_TO_DEL_PRIO) !=OS_TASK_NOT_EXIST) {
OSTimeDly(1);
}
}
/*Application code*/
}}
```



# 4.05 Requesting to Delete a Task[OSTaskDelReq()]

-task’s request need to determine what conditions can cause a request for the task to be deleted.

-the request task need to delay a certain of time.

-if requested task has delete itself exit the loop.

# 4.05 Requesting to Delete a Task[OSTaskDelReq()]

```
void TaskToBeDeleted(void *pdata)
{
INT8U err;
pdata=pdata;
for(;;){
/*Application code*/
if(OSTaskDelReq(OS_PRIO_SELF)==OS_TASK_DEL_REQ) {
Release any owned resources;
De-allocate any dynamic memory;
OSTaskDel(OS_PRIO_SELF);
}else{
/*Application code*/
}
}
}
```

# 4.05 Requesting to Delete a Task[OSTaskDelReq()]

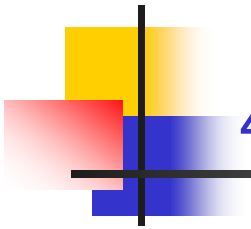
-task release its own resource and call OSTaskDel(OS_PRIO_SELF).

-you can recreate task by calling  either OSTaskCreate() or OSTaskCreateExt().

```
INT8U  OSTaskDelReq (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3                 /* Allocate storage for CPU status register          */
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    stat;
    INT8U      err;
    OS_TCB     *ptcb;
#if OS_ARG_CHK_EN > 0
    if (prio == OS_IDLE_PRIO) {                     /* Not allowed to delete idle task     */
        return (OS_TASK_DEL_IDLE);
```

# 4.05 Requesting to Delete a Task[OSTaskDelReq()

```
}
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {       /* Task priority valid ?              */
        return (OS_PRIO_INVALID);
    }
#endif
    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);}
```



# 4.05 Requesting to Delete a Task[OSTaskDelReq()

```
#define OS_TASK_DEL_ERR         60
#define OS_TASK_DEL_IDLE        61
#define OS_TASK_DEL_REQ         62
#define OS_TASK_DEL_ISR         63
```

# 4.06 Changing a Task's Priority[OSTaskChangePrio()]

-you can change priority of any task by calling OSTaskCahngePrio() at runtime.

```
INT8U  OSTaskChangePrio (INT8U oldprio, INT8U newprio)
{
#if OS_CRITICAL_METHOD == 3                  /* Allocate storage for CPU status register           */
    OS_CPU_SR    cpu_sr;
#endif

#if OS_EVENT_EN > 0
    OS_EVENT    *pevent;
#endif

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

# 4.06 Changing a Task's Priority[OSTaskChangePrio()]

```
#if OS_ARG_CHK_EN > 0
    if ((oldprio >= OS_LOWEST_PRIO && oldprio != OS_PRIO_SELF)  ||
         newprio >= OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    }
#endif
    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();
```

# 4.06 Changing a Task's Priority[OSTaskChangePrio()]

```
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 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;}
```

# 4.06 Changing a Task's Priority[OSTaskChangePrio()]

```
#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         */
        }
    }
}
```

# 4.07 Suspending a Task[OSTaskSuspend()]

-OSTaskSuspend() is called to suspend the execution of a task.

-a suspended task can only call OSTaskResume() to be resumed.

-a suspended task is waiting for time expire.

-a task can suspend either itself or another task.

```
INT8U  OSTaskSuspend (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3                  /* Allocate storage for CPU status register          */
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    self;
    OS_TCB    *ptcb;
```

# 4.07 Suspending a Task[OSTaskSuspend()]

```
#if OS_ARG_CHK_EN > 0
    if (prio == OS_IDLE_PRIO) {                          /* Not allowed to suspend idle task    */
        return (OS_TASK_SUSPEND_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {       /* Task priority valid ?               */
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                          /* See if suspend SELF                 */
        prio = OSTCBCur->OSTCBPrio;
        self = TRUE;
    } else if (prio == OSTCBCur->OSTCBPrio) {                 /* See if suspending self              */
        self = TRUE;
    } else {
        self = FALSE;                                    /* No suspending another task          */
    }
    ptcb = OSTCBPrioTbl[prio];
```

# 4.07 Suspending a Task[OSTaskSuspend()]

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

# 4.08 Resuming a Task[OSTaskResume()

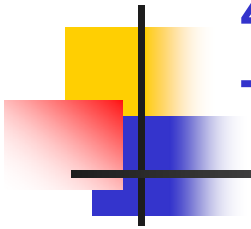
-a suspended task can only resumed by calling OSTaskResume().

Code:

```
INT8U  OSTaskResume (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
 OS_CPU_SR  cpu_sr;
#endif
 OS_TCB     *ptcb;
#if OS_ARG_CHK_EN > 0
if (prio >= OS_LOWEST_PRIO) {
return (OS_PRIO_INVALID);
 }
#endif
OS_ENTER_CRITICAL();
ptcb = OSTCBPrioTbl[prio];
if (ptcb == (OS_TCB *)0) {
OS_EXIT_CRITICAL();
 return (OS_TASK_RESUME_PRIO);
```

# 4.08 Resuming a Task[OSTaskResume()

```
}
if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) != OS_STAT_RDY) {
if (((ptcb->OSTCBStat &= ~OS_STAT_SUSPEND) == OS_STAT_RDY) &&(ptcb->OSTCBDly  == 0)) {
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);
```



# 4.09 Getting Information about a Task[OSTaskQuery()]

-your application can obtain information about itself or other application tasks by calling OSTaskQuery().

-in fact OSTaskQuery() obtains copy of the contents of the desired task's OS_TCB.