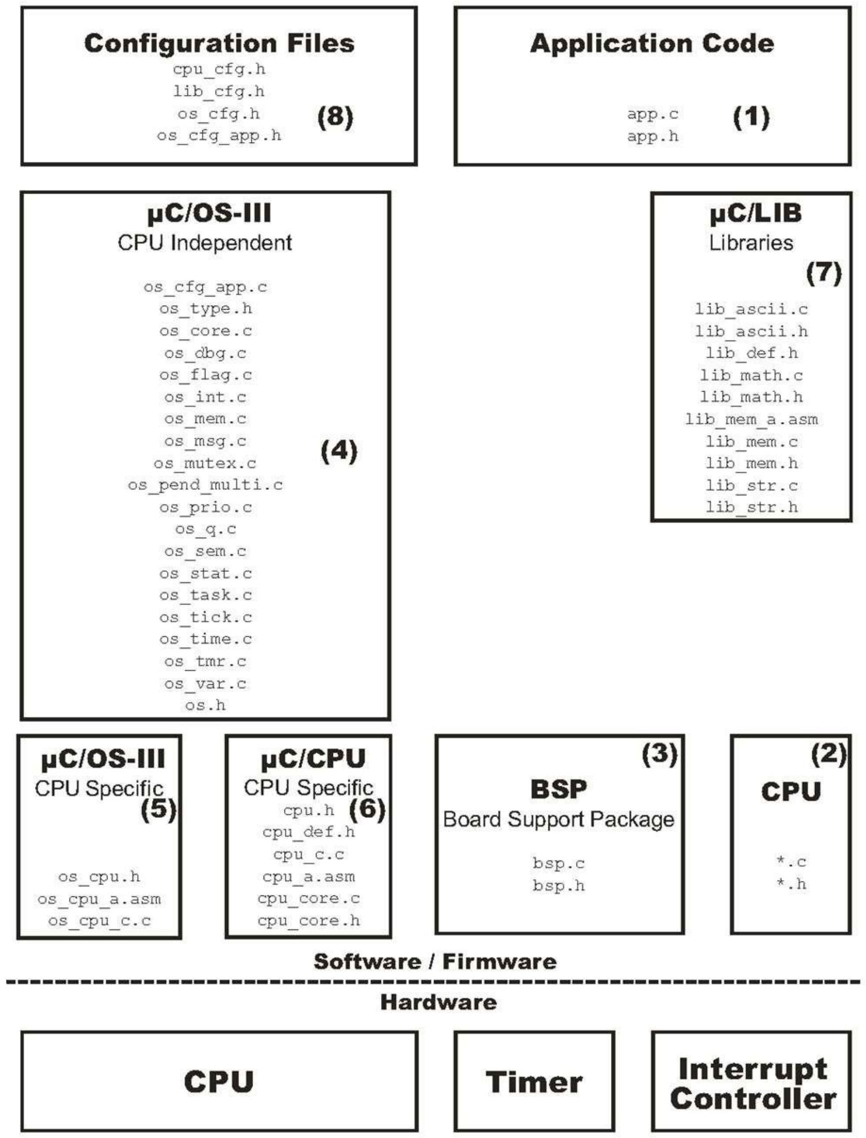
Features Provided by uC/OS-III

uC/OS-III Topic follows the Reference: uC/OS-III the Real-Time Kernel, User’s Manual, by J. J. Labrosse … for further notes consult the reference …

- **uC/OS-II and –III** are currently maintained by Micrium Inc., a subsidiary of Silicon Labs.
- **Source Code:** µC/OS-III is provided in ANSI-C source form
- **Intuitive Application Programming Interface (API):** µC/OS-III is highly intuitive
- **Preemptive multitasking:** µC/OS-III is a preemptive multi-tasking kernel
- **Round robin scheduling of tasks at equal priority:** µC/OS-III allows multiple tasks to run at the same priority level
- **Deterministic:** Interrupt response with µC/OS-III is deterministic
- **Scalable:** The footprint (both code and data) can be adjusted based on the requirements of the application (between 5 to 24 Kbytes)
- **Portable:** µC/OS-III can be ported to many microprocessors and DSP processors
- Commercial usage: avionics, medical equipment, data communications, appliances, industrial control, consumer electronics, automotive, etc.

uC/OS-III Architecture and Its Relationship with the Hardware

See pp. 31 to 33 in the reference …



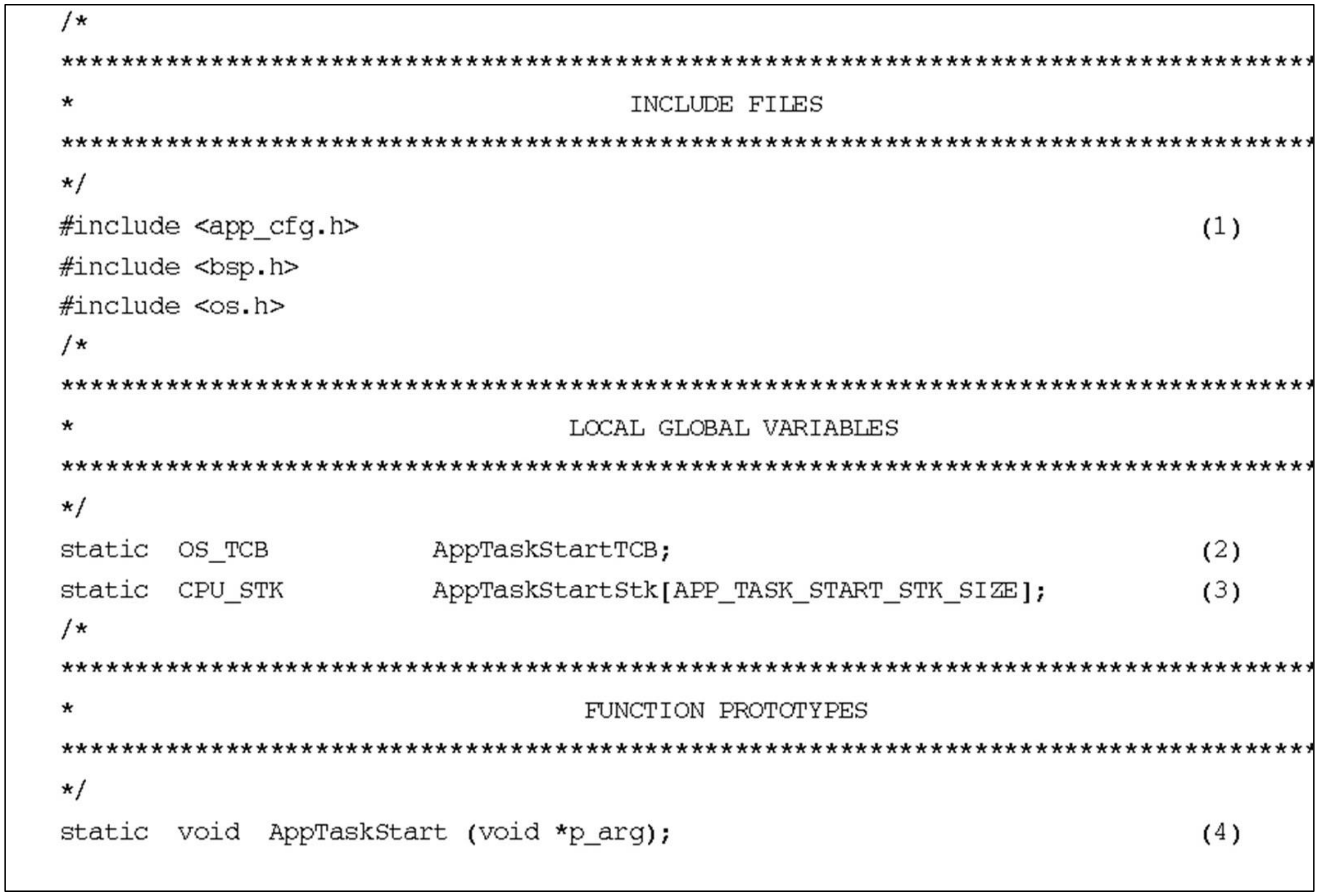
Getting Started with uC/OS-III

Refer to pp. 51 to 65 in the reference book …

- uC/OS-III provides a set of functions for the development of application code
- uC/OS-III functions offer services to manage tasks, semaphores, message queues, mutual exclusion semaphores, etc.
- An application can call uC/OS-III functions as any other function
- We discuss two types of application structures that apply to uC/OS-III:
    - Single task application
    - Multiple tasks application

Single Task Application: app.c

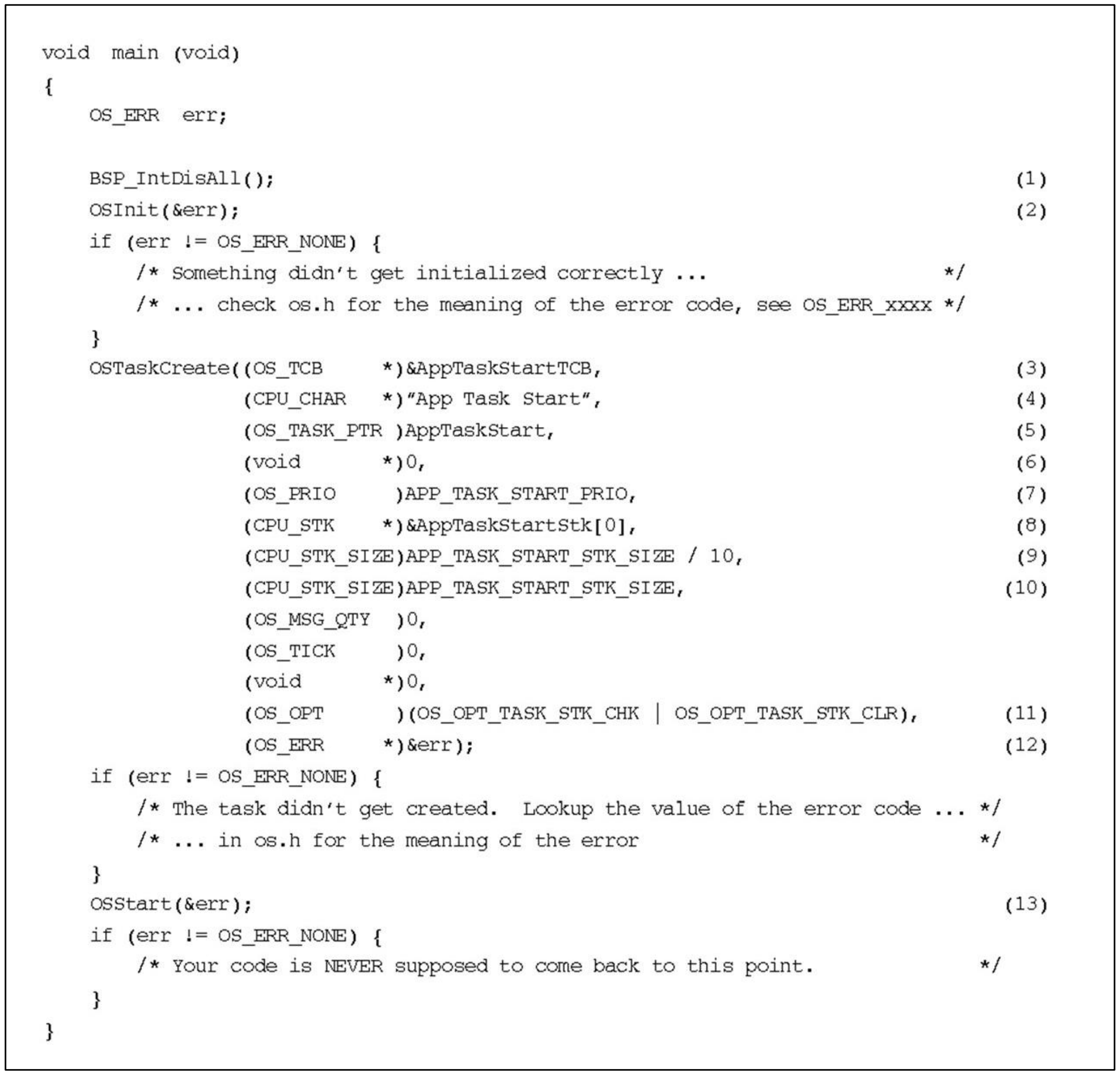
- What's happening in this code?
- (1) …
- (2) …
- (3) …
- (4) …



```
/*
*************************************************************************************
*                                    INCLUDE FILES
*************************************************************************************
*/
#include <app_cfg.h>                                                            (1)
#include <bsp.h>
#include <os.h>
/*
*************************************************************************************
*                                 LOCAL GLOBAL VARIABLES
*************************************************************************************
*/
static  OS_TCB          AppTaskStartTCB;                                        (2)
static  CPU_STK         AppTaskStartStk[APP_TASK_START_STK_SIZE];               (3)
/*
*************************************************************************************
*                                  FUNCTION PROTOTYPES
*************************************************************************************
*/
static  void  AppTaskStart (void *p_arg);                                       (4)
```

The Main Program

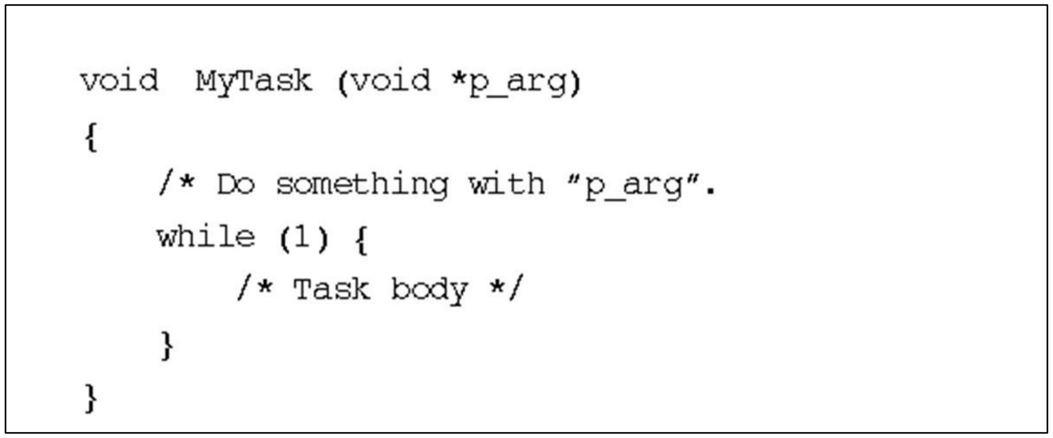
- The code lines
- (1) …
- (2) …
- OSTaskCreate() creates a task and requires 13 arguments …



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

    BSP_IntDisAll();                                                      (1)
    OSInit(&err);                                                         (2)
    if (err != OS_ERR_NONE) {
        /* Something didn't get initialized correctly ...              */
        /* ... check os.h for the meaning of the error code, see OS_ERR_xxxx */
    }
    OSTaskCreate((OS_TCB      *)&AppTaskStartTCB,                         (3)
                 (CPU_CHAR    *)"App Task Start",                         (4)
                 (OS_TASK_PTR )AppTaskStart,                              (5)
                 (void        *)0,                                        (6)
                 (OS_PRIO      )APP_TASK_START_PRIO,                      (7)
                 (CPU_STK     *)&AppTaskStartStk[0],                      (8)
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE / 10,              (9)
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,                  (10)
                 (OS_MSG_QTY  )0,
                 (OS_TICK      )0,
                 (void        *)0,
                 (OS_OPT       )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), (11)
                 (OS_ERR      *)&err);                                   (12)
    if (err != OS_ERR_NONE) {
        /* The task didn't get created.  Lookup the value of the error code ... */
        /* ... in os.h for the meaning of the error                         */
    }
    OSStart(&err);                                                       (13)
    if (err != OS_ERR_NONE) {
        /* Your code is NEVER supposed to come back to this point.          */
    }
}
```

Typical Task Implementation

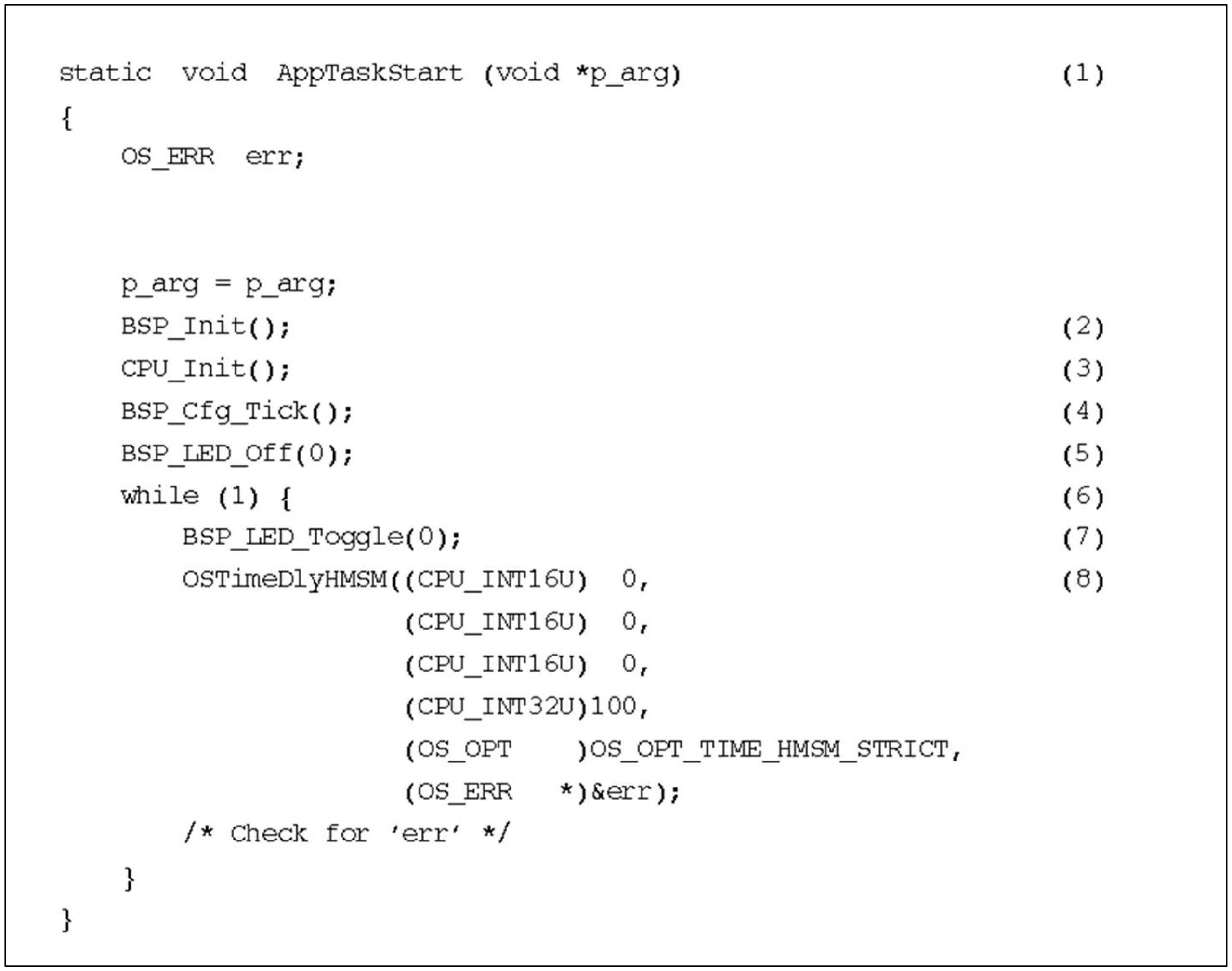
- A task is typically implemented as an infinite loop as shown in this code.
- Another implementation of a task could be as run to completion, where while(1) statement is not used.



```
void  MyTask (void *p_arg)
{
    /* Do something with "p_arg".
    while (1) {
        /* Task body */
    }
}
```

AppTaskStart()

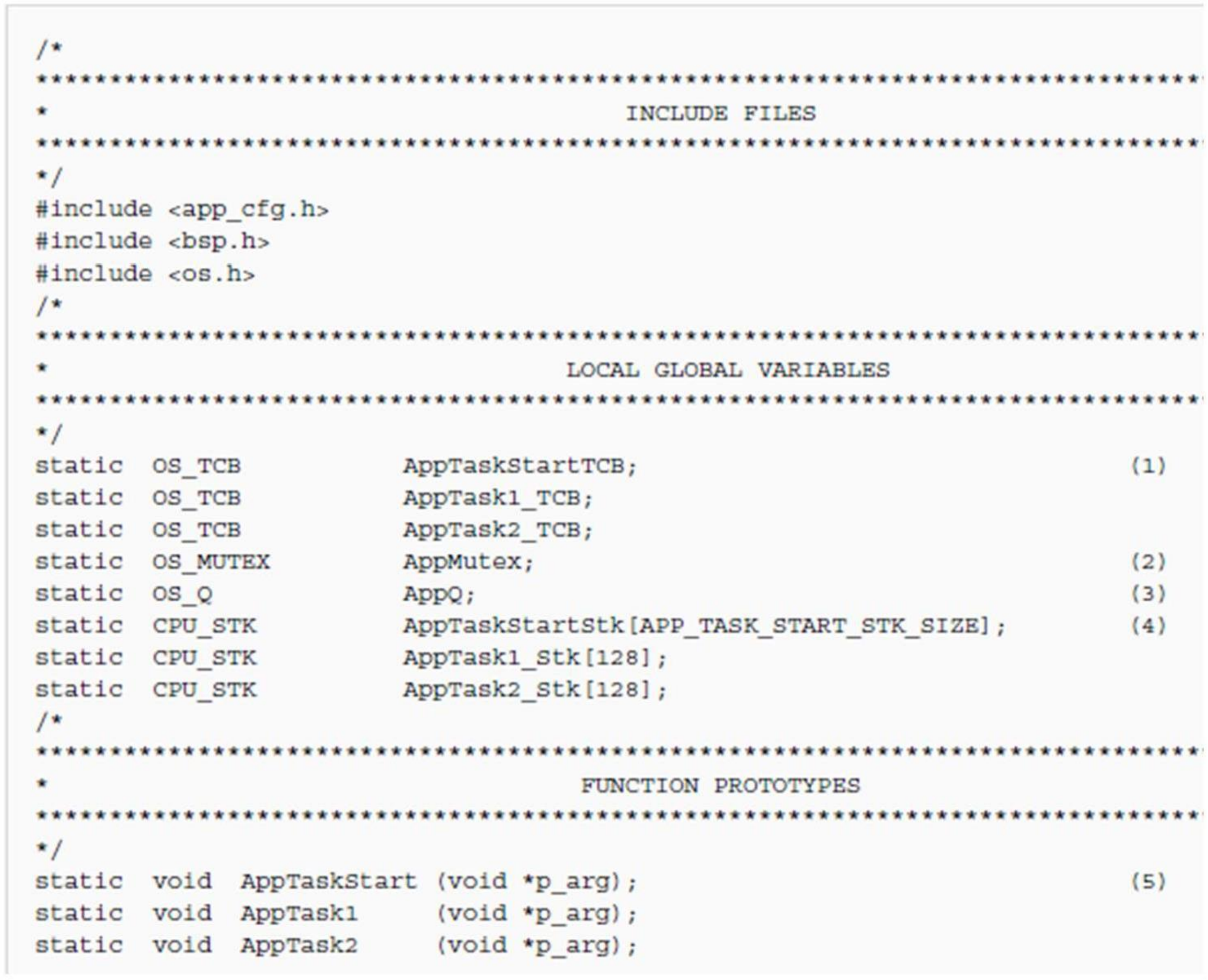
- Code lines:
  - (1) …
  - (2) …
  - (3) …
  - (4) …
  - (5) …
  - (6) …
  - (7) …
  - OSTimeDlyHMSM() …



```
static  void  AppTaskStart (void *p_arg)                          (1)
{
    OS_ERR  err;


    p_arg = p_arg;
    BSP_Init();                                                   (2)
    CPU_Init();                                                   (3)
    BSP_Cfg_Tick();                                               (4)
    BSP_LED_Off(0);                                               (5)
    while (1) {                                                   (6)
        BSP_LED_Toggle(0);                                        (7)
        OSTimeDlyHMSM((CPU_INT16U)  0,                            (8)
                      (CPU_INT16U)  0,
                      (CPU_INT16U)  0,
                      (CPU_INT32U)100,
                      (OS_OPT    )OS_OPT_TIME_HMSM_STRICT,
                      (OS_ERR   *)&err);
        /* Check for 'err' */
    }
}
```

Multiple Tasks Applications with Kernel Objects

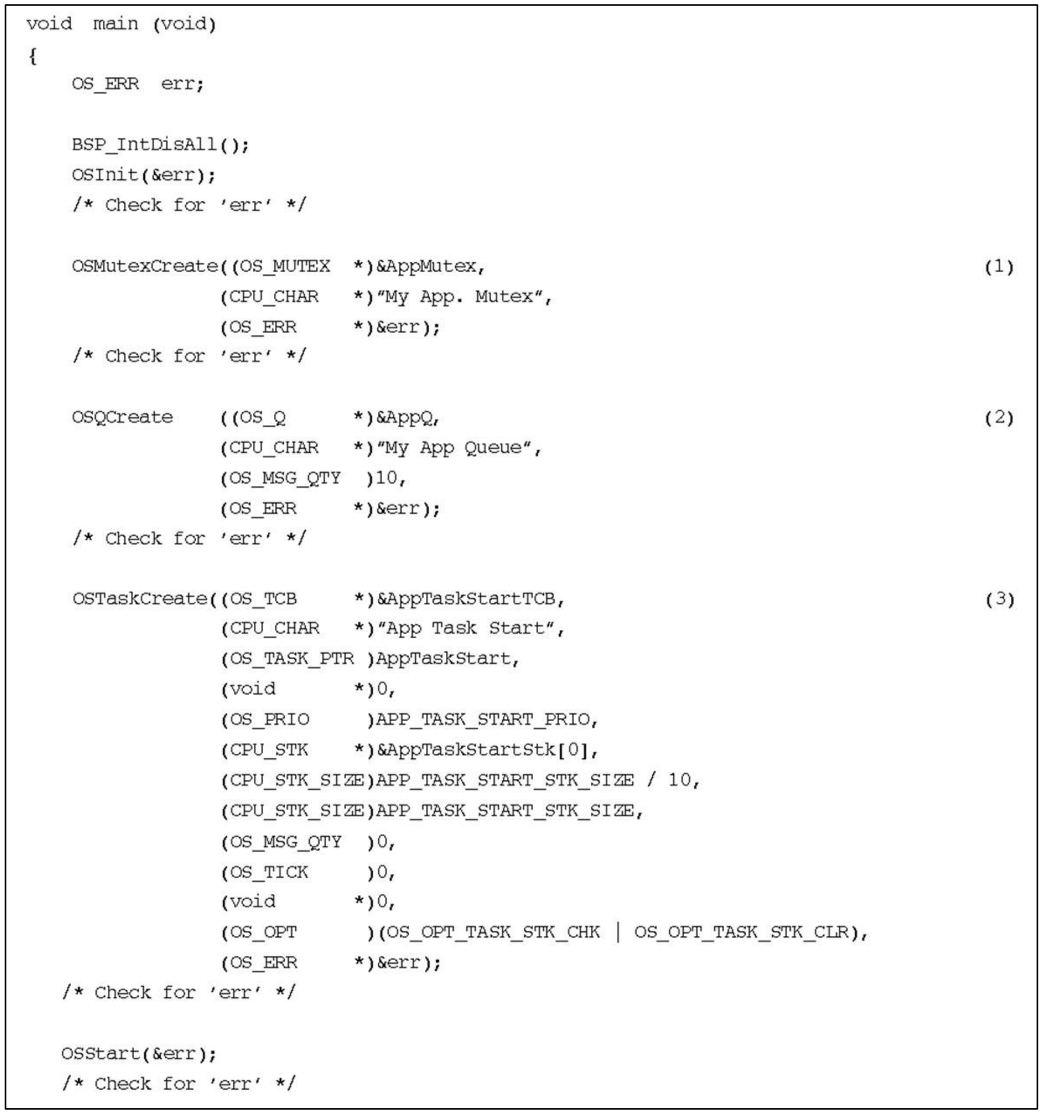
- Code lines:
  - (1) …
  - (2) …
  - (3) …
  - (4) …
  - (5) …



```
/*
**************************************************************************
*                                   INCLUDE FILES
**************************************************************************
*/
#include <app_cfg.h>
#include <bsp.h>
#include <os.h>
/*
**************************************************************************
*                               LOCAL GLOBAL VARIABLES
**************************************************************************
*/
static  OS_TCB          AppTaskStartTCB;                              (1)
static  OS_TCB          AppTask1_TCB;
static  OS_TCB          AppTask2_TCB;
static  OS_MUTEX        AppMutex;                                     (2)
static  OS_Q            AppQ;                                         (3)
static  CPU_STK         AppTaskStartStk[APP_TASK_START_STK_SIZE];     (4)
static  CPU_STK         AppTask1_Stk[128];
static  CPU_STK         AppTask2_Stk[128];
/*
**************************************************************************
*                                FUNCTION PROTOTYPES
**************************************************************************
*/
static  void  AppTaskStart (void *p_arg);                             (5)
static  void  AppTask1     (void *p_arg);
static  void  AppTask2     (void *p_arg);
```

The Main Program

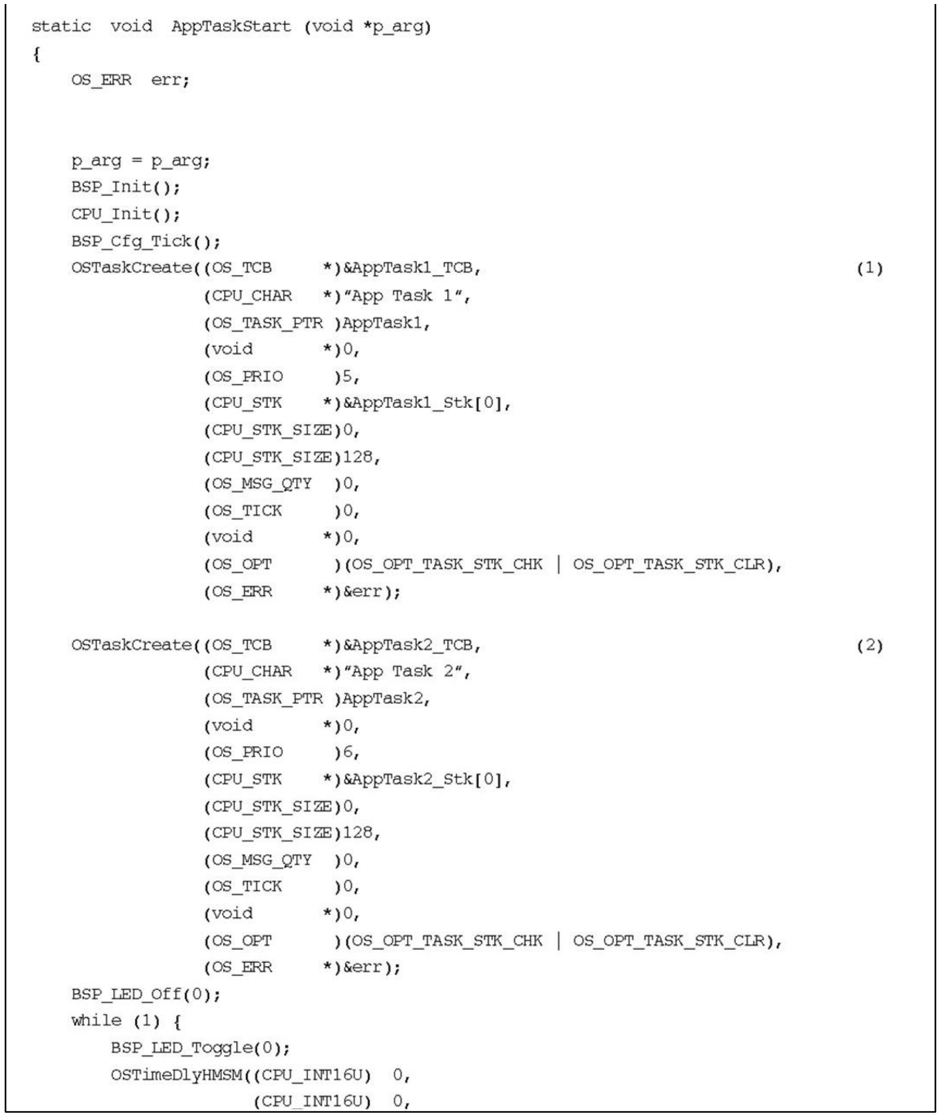
- OSMutexCreate() …
- OSQCreate() …
- OSTaskCreate() …



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

    BSP_IntDisAll();
    OSInit(&err);
    /* Check for 'err' */

    OSMutexCreate((OS_MUTEX  *)&AppMutex,                                         (1)
                  (CPU_CHAR   *)"My App. Mutex",
                  (OS_ERR     *)&err);
    /* Check for 'err' */

    OSQCreate     ((OS_Q      *)&AppQ,                                            (2)
                  (CPU_CHAR   *)"My App Queue",
                  (OS_MSG_QTY  )10,
                  (OS_ERR     *)&err);
    /* Check for 'err' */

    OSTaskCreate((OS_TCB     *)&AppTaskStartTCB,                                  (3)
                  (CPU_CHAR   *)"App Task Start",
                  (OS_TASK_PTR )AppTaskStart,
                  (void       *)0,
                  (OS_PRIO     )APP_TASK_START_PRIO,
                  (CPU_STK    *)&AppTaskStartStk[0],
                  (CPU_STK_SIZE)APP_TASK_START_STK_SIZE / 10,
                  (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,
                  (OS_MSG_QTY  )0,
                  (OS_TICK     )0,
                  (void       *)0,
                  (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                  (OS_ERR     *)&err);
   /* Check for 'err' */

   OSStart(&err);
   /* Check for 'err' */
```

Application Start Task

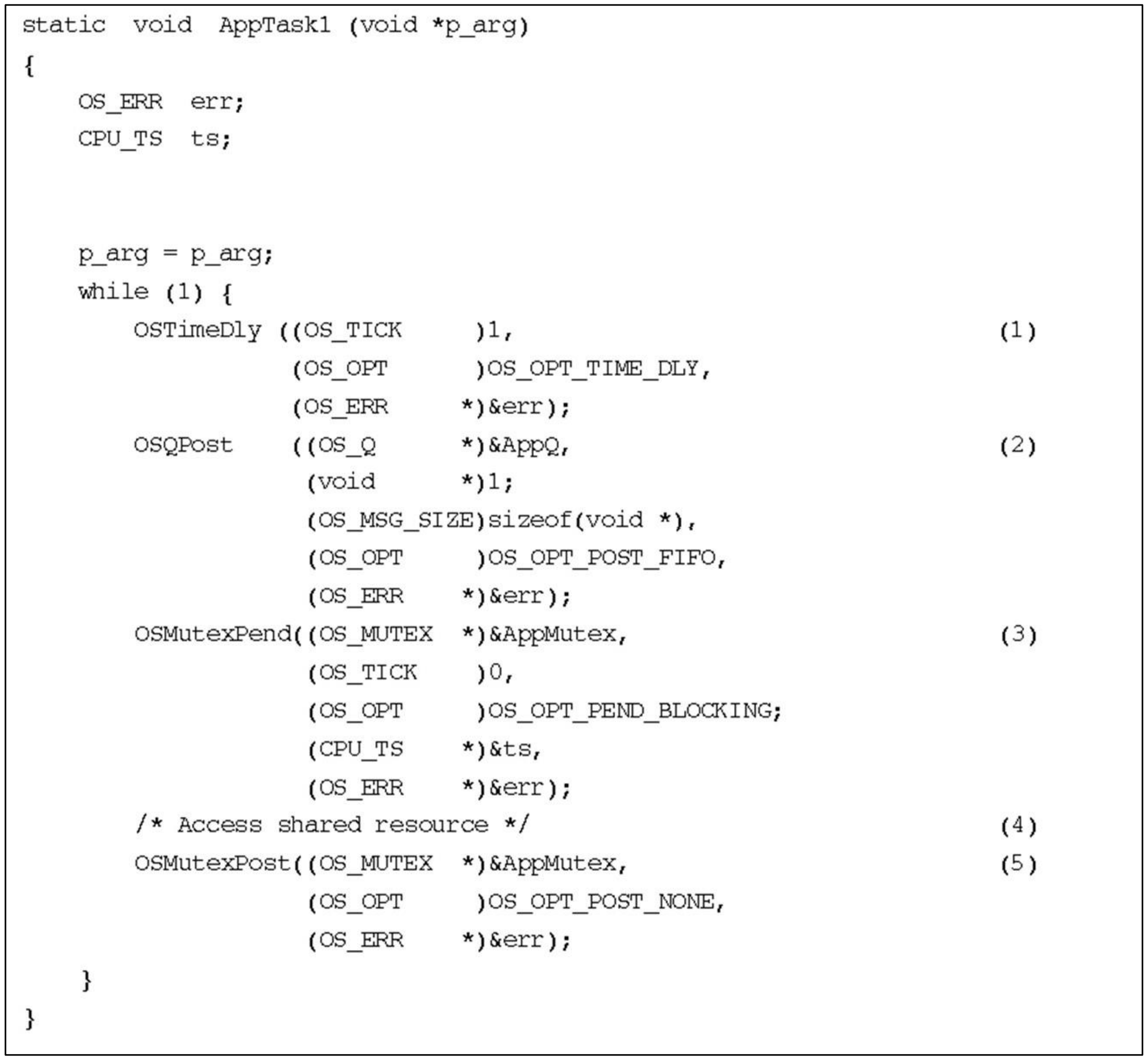
- OSTaskCreate() …
- OSTaskCreate() …
- …



```
static  void  AppTaskStart (void *p_arg)
{
    OS_ERR  err;


    p_arg = p_arg;
    BSP_Init();
    CPU_Init();
    BSP_Cfg_Tick();
    OSTaskCreate((OS_TCB     *)&AppTask1_TCB,                                    (1)
                 (CPU_CHAR   *)"App Task 1",
                 (OS_TASK_PTR )AppTask1,
                 (void       *)0,
                 (OS_PRIO     )5,
                 (CPU_STK    *)&AppTask1_Stk[0],
                 (CPU_STK_SIZE)0,
                 (CPU_STK_SIZE)128,
                 (OS_MSG_QTY  )0,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR     *)&err);

    OSTaskCreate((OS_TCB     *)&AppTask2_TCB,                                    (2)
                 (CPU_CHAR   *)"App Task 2",
                 (OS_TASK_PTR )AppTask2,
                 (void       *)0,
                 (OS_PRIO     )6,
                 (CPU_STK    *)&AppTask2_Stk[0],
                 (CPU_STK_SIZE)0,
                 (CPU_STK_SIZE)128,
                 (OS_MSG_QTY  )0,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR     *)&err);
    BSP_LED_Off(0);
    while (1) {
        BSP_LED_Toggle(0);
        OSTimeDlyHMSM((CPU_INT16U)  0,
                      (CPU_INT16U)  0,
```

Application Task 1

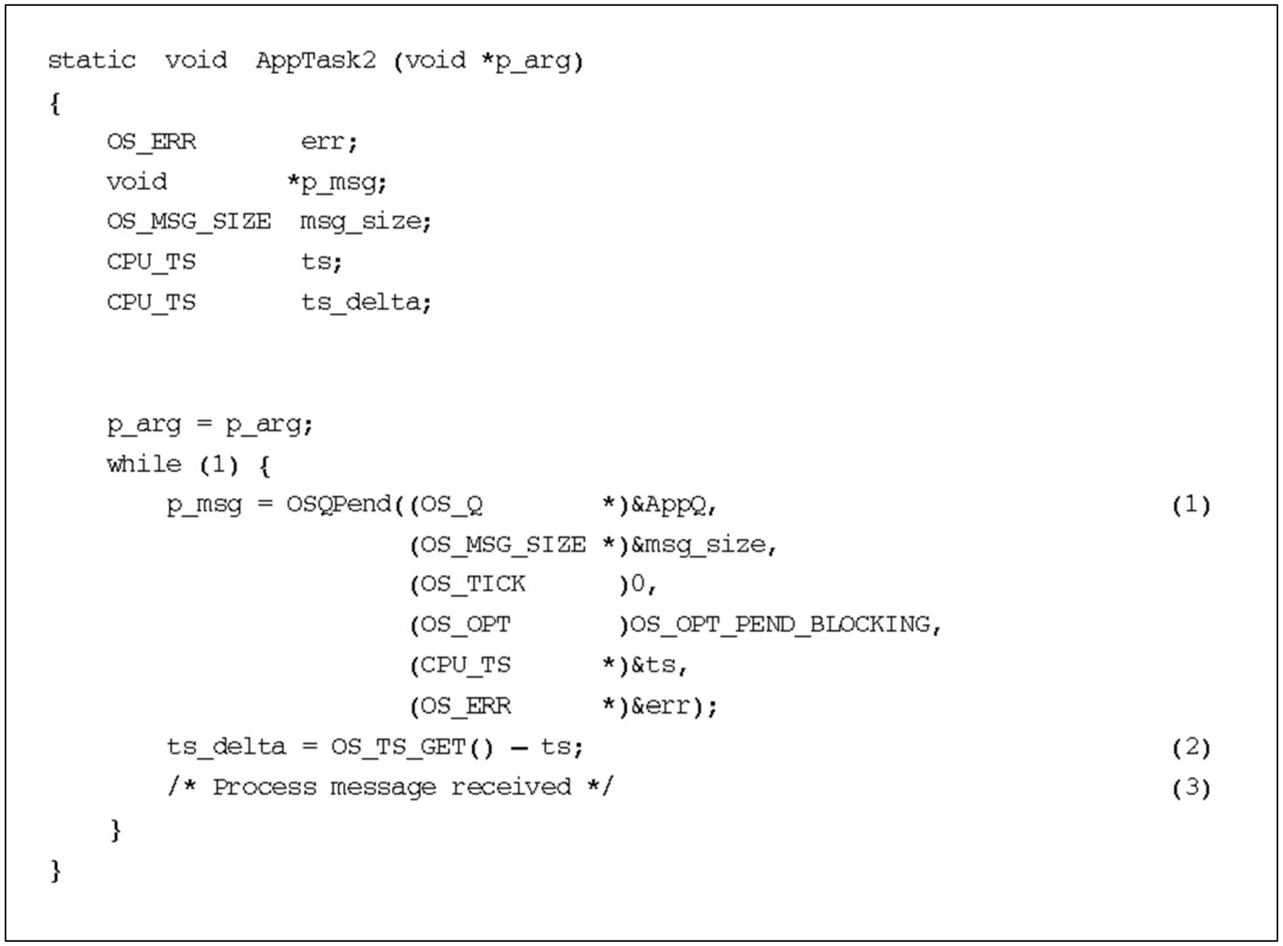
- Code lines:
  - (1) …
  - (2) …
  - (3) …
  - (4) …
  - (5) …



```
static  void  AppTask1 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;


    p_arg = p_arg;
    while (1) {
        OSTimeDly ((OS_TICK     )1,                                     (1)
                   (OS_OPT      )OS_OPT_TIME_DLY,
                   (OS_ERR     *)&err);
        OSQPost    ((OS_Q      *)&AppQ,                                 (2)
                    (void      *)1;
                    (OS_MSG_SIZE)sizeof(void *),
                    (OS_OPT     )OS_OPT_POST_FIFO,
                    (OS_ERR    *)&err);
        OSMutexPend((OS_MUTEX  *)&AppMutex,                             (3)
                    (OS_TICK    )0,
                    (OS_OPT     )OS_OPT_PEND_BLOCKING;
                    (CPU_TS    *)&ts,
                    (OS_ERR    *)&err);
        /* Access shared resource */                                    (4)
        OSMutexPost((OS_MUTEX  *)&AppMutex,                             (5)
                    (OS_OPT     )OS_OPT_POST_NONE,
                    (OS_ERR    *)&err);
    }
}
```

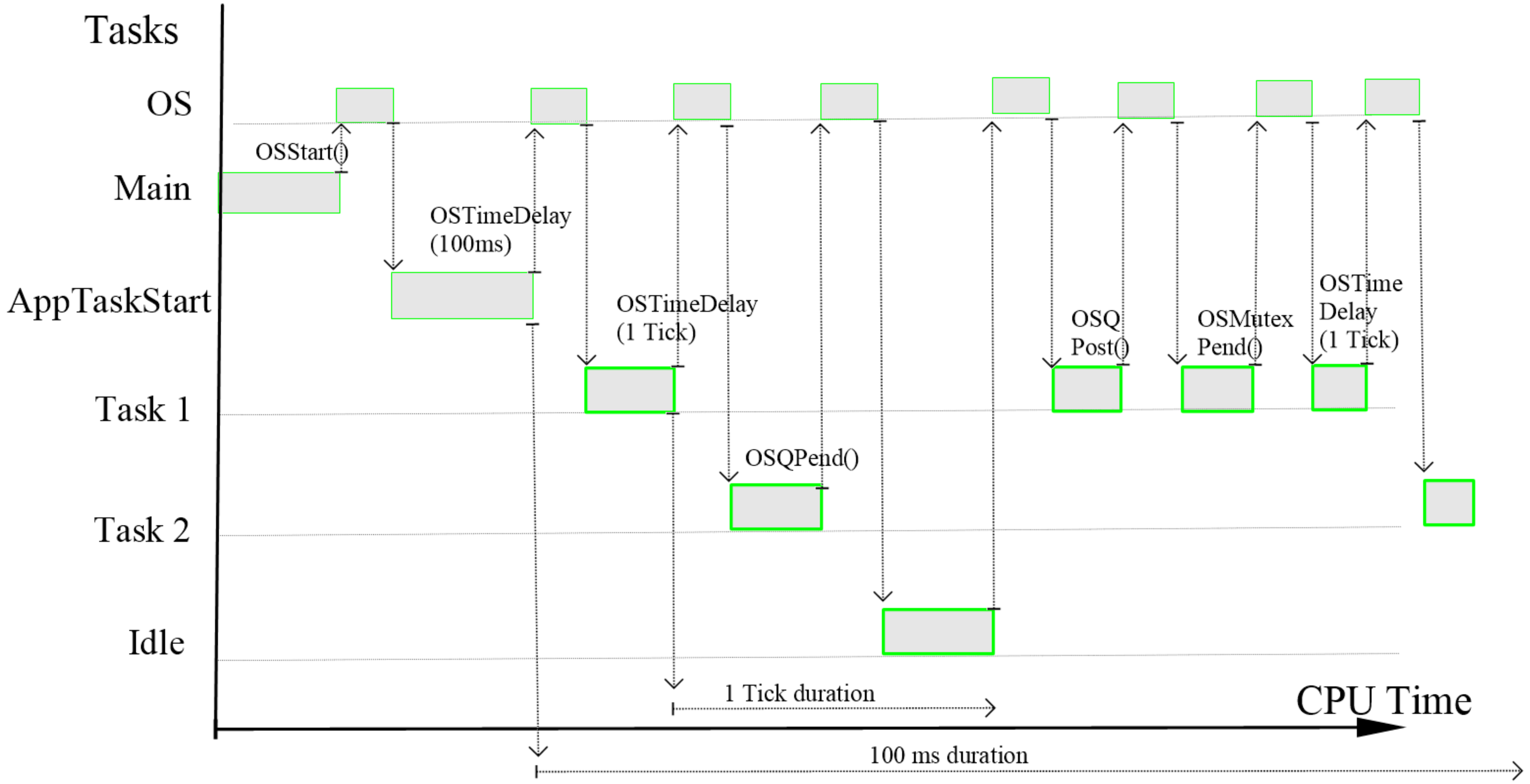
Application Task 2

- Code lines:
  - (1) …
  - (2) …
  - (3) …



```
static  void  AppTask2 (void *p_arg)
{
    OS_ERR       err;
    void        *p_msg;
    OS_MSG_SIZE  msg_size;
    CPU_TS       ts;
    CPU_TS       ts_delta;


    p_arg = p_arg;
    while (1) {
        p_msg = OSQPend((OS_Q        *)&AppQ,                                   (1)
                        (OS_MSG_SIZE *)&msg_size,
                        (OS_TICK      )0,
                        (OS_OPT       )OS_OPT_PEND_BLOCKING,
                        (CPU_TS      *)&ts,
                        (OS_ERR      *)&err);
        ts_delta = OS_TS_GET() - ts;                                            (2)
        /* Process message received */                                          (3)
    }
}
```

SLIDE 121 NOTES ... page 1 ...
Draw a time execution diagram of the tasks for the example above!
Assume that the AppTaskStart has priority 4.



Critical Sections or Critical Regions

Refer to pp. 66 to 71 in the reference book …

- Critical section is code that needs to be treated indivisibly
- There are many critical section of code contained in a kernel
- Critical Section scenarios:
  - If a critical section is accessible by an ISR and a task, then protecting the critical region is done by disabling interrupts
  - If a critical section is only accessible by task level code, the critical section may be protected through use of a preemption lock.
- Note: disabling the interrupts or locking the preemption for too long defeats the purpose of real-time scheduling
- uC/OS-III provides ways to measure the time duration while the interrupts are disabled

Critical Sections or Critical Regions

- uC/OS-III will disable interrupts when accessing internal critical sections
- The OS uses uC/CPU macros for entering and leaving a critical section:
  - CPU_CRITICAL_ENTER() and
  - CPU_CRITICAL_EXIT()

Critical Sections or Critical Regions

- A tasks can lock the scheduler and thus preventing the preemption process of the OS
- By locking the scheduler, the current task will be executing and thus the method will prevent higher priority task to execute while the scheduler is locked
- Functions to lock and unlock scheduler are:
    - OSSchedLock()
    - OSSchedUnlock()
- Interrupts are still recognized during scheduler locked state, but the kernel will not switch to higher priority tasks.

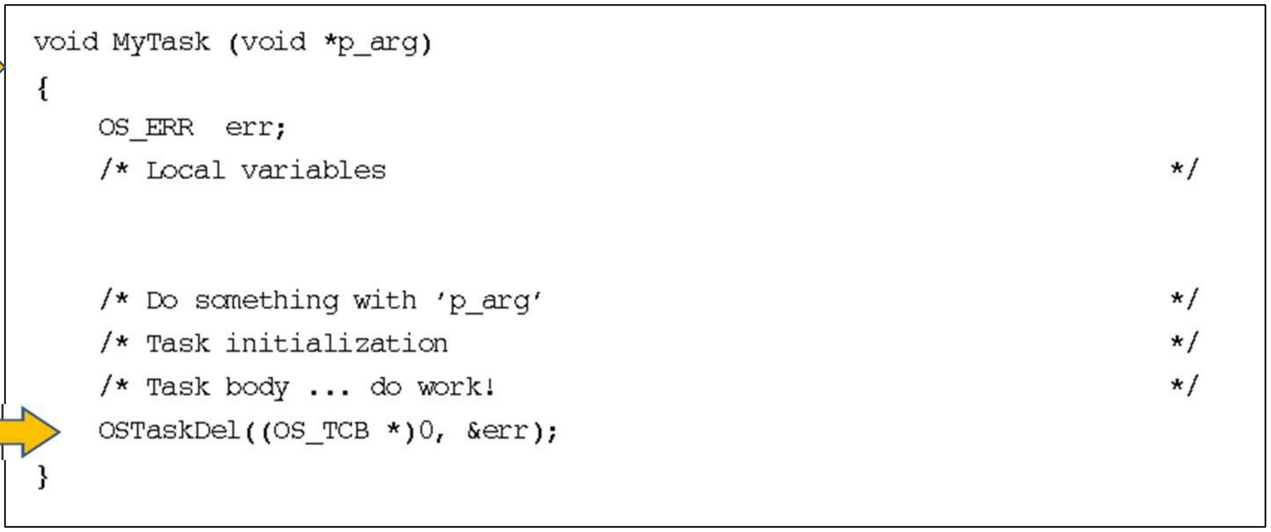
Task Management

Refer to pp. 72 to 127 in the reference book …

- uC/OS-III supports multitasking and supports any number of tasks
- A task (also called a thread) is a simple program that thinks it has the CPU all to itself.
- Task usage:
  - Monitoring inputs
  - Updating outputs
  - Performing computations
  - Control loops
  - Update one or more displays
  - Reading buttons and keyboards
  - Communicating with other systems
  - Etc.

Types of Tasks in uC/OS-III

- Run-to-completion
- Infinite loop
- Example of run-to-completion task

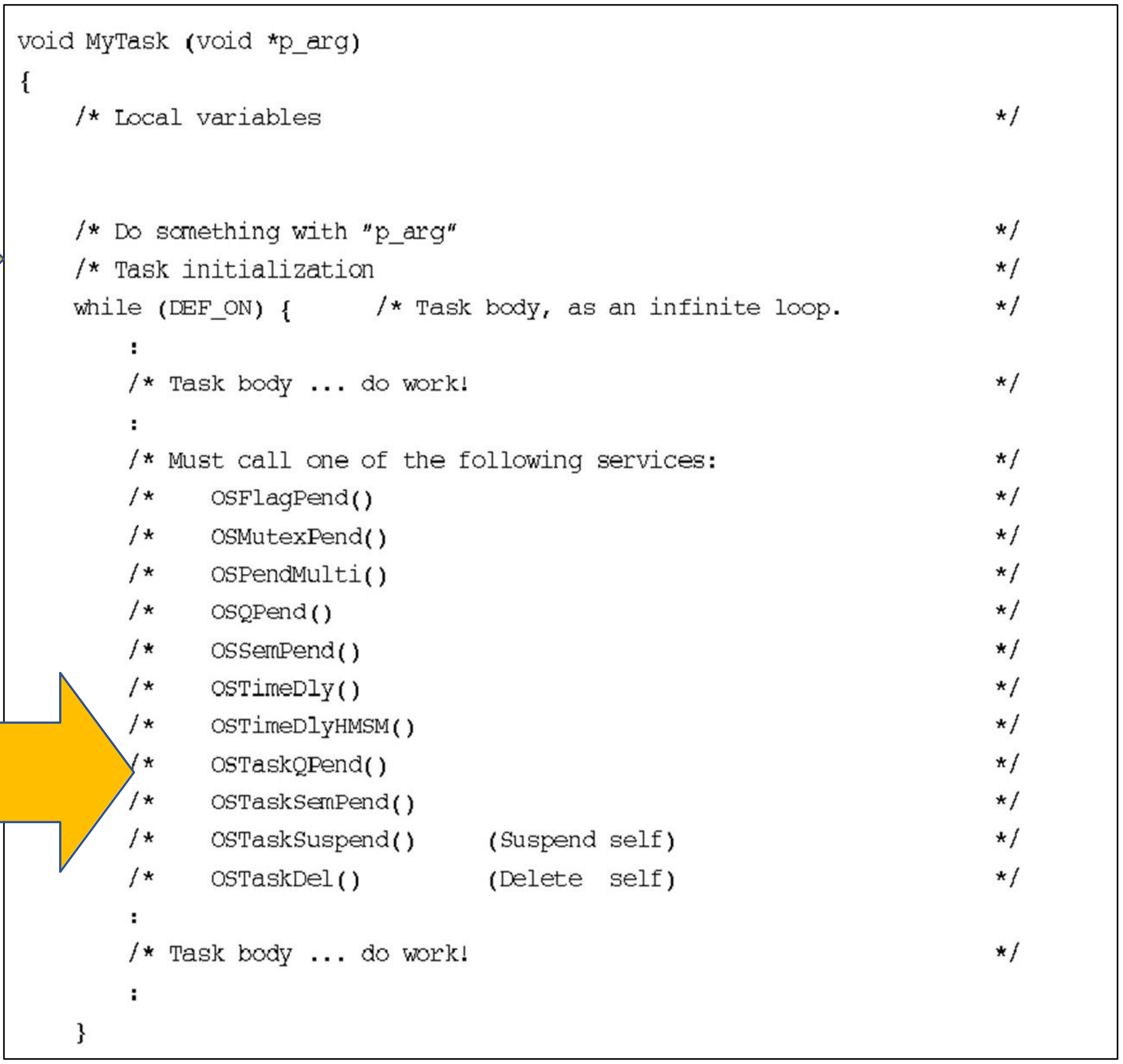


```
void MyTask (void *p_arg)
{
    OS_ERR  err;
    /* Local variables                                                              */


    /* Do something with 'p_arg'                                                    */
    /* Task initialization                                                          */
    /* Task body ... do work!                                                       */
    OSTaskDel((OS_TCB *)0, &err);
}
```

- Note: a run-to completion task must delete itself

Types of Tasks in uC/OS-III

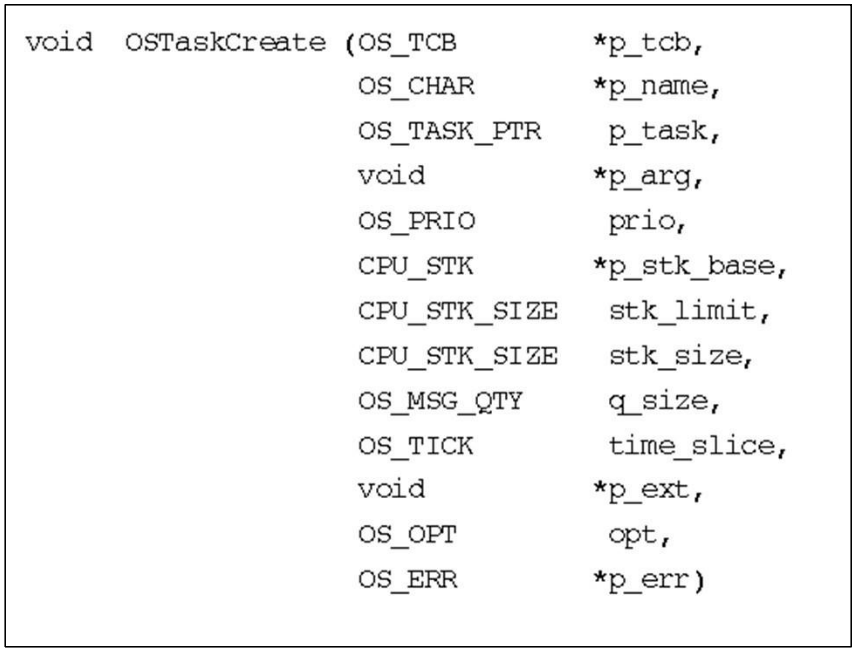
- The infinite loop task is common in embedded systems because of the repetitive work needed;
- Reading inputs, updating outputs, control operations etc.
- Important that each task must make a service call to OS to support multitasking process.



```
void MyTask (void *p_arg)
{
    /* Local variables                                                    */


    /* Do something with "p_arg"                                          */
    /* Task initialization                                                */
    while (DEF_ON) {        /* Task body, as an infinite loop.            */
        :
        /* Task body ... do work!                                         */
        :
        /* Must call one of the following services:                       */
        /*     OSFlagPend()                                               */
        /*     OSMutexPend()                                              */
        /*     OSPendMulti()                                              */
        /*     OSQPend()                                                  */
        /*     OSSemPend()                                                */
        /*     OSTimeDly()                                                */
        /*     OSTimeDlyHMSM()                                            */
        /*     OSTaskQPend()                                              */
        /*     OSTaskSemPend()                                            */
        /*     OSTaskSuspend()     (Suspend self)                         */
        /*     OSTaskDel()         (Delete  self)                         */
        :
        /* Task body ... do work!                                         */
        :
    }
```

OSTaskCreate() Function

- OS knows about the tasks after they are created by:
  - OSTaskCreate() function
- The function needs 13 parameters to be initialized: see uC/OS-III API Reference for arguments description
- Common parameters to note are task address, task priority, task TCB, task stack, etc.

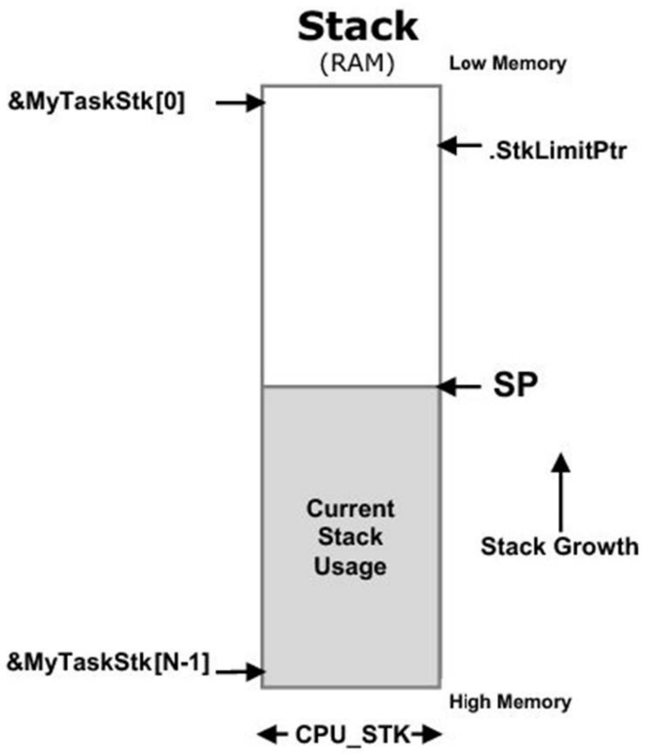


```
void   OSTaskCreate (OS_TCB          *p_tcb,
                     OS_CHAR         *p_name,
                     OS_TASK_PTR      p_task,
                     void            *p_arg,
                     OS_PRIO          prio,
                     CPU_STK         *p_stk_base,
                     CPU_STK_SIZE     stk_limit,
                     CPU_STK_SIZE     stk_size,
                     OS_MSG_QTY       q_size,
                     OS_TICK          time_slice,
                     void            *p_ext,
                     OS_OPT           opt,
                     OS_ERR          *p_err)
```

The stack and its Descriptive Arguments

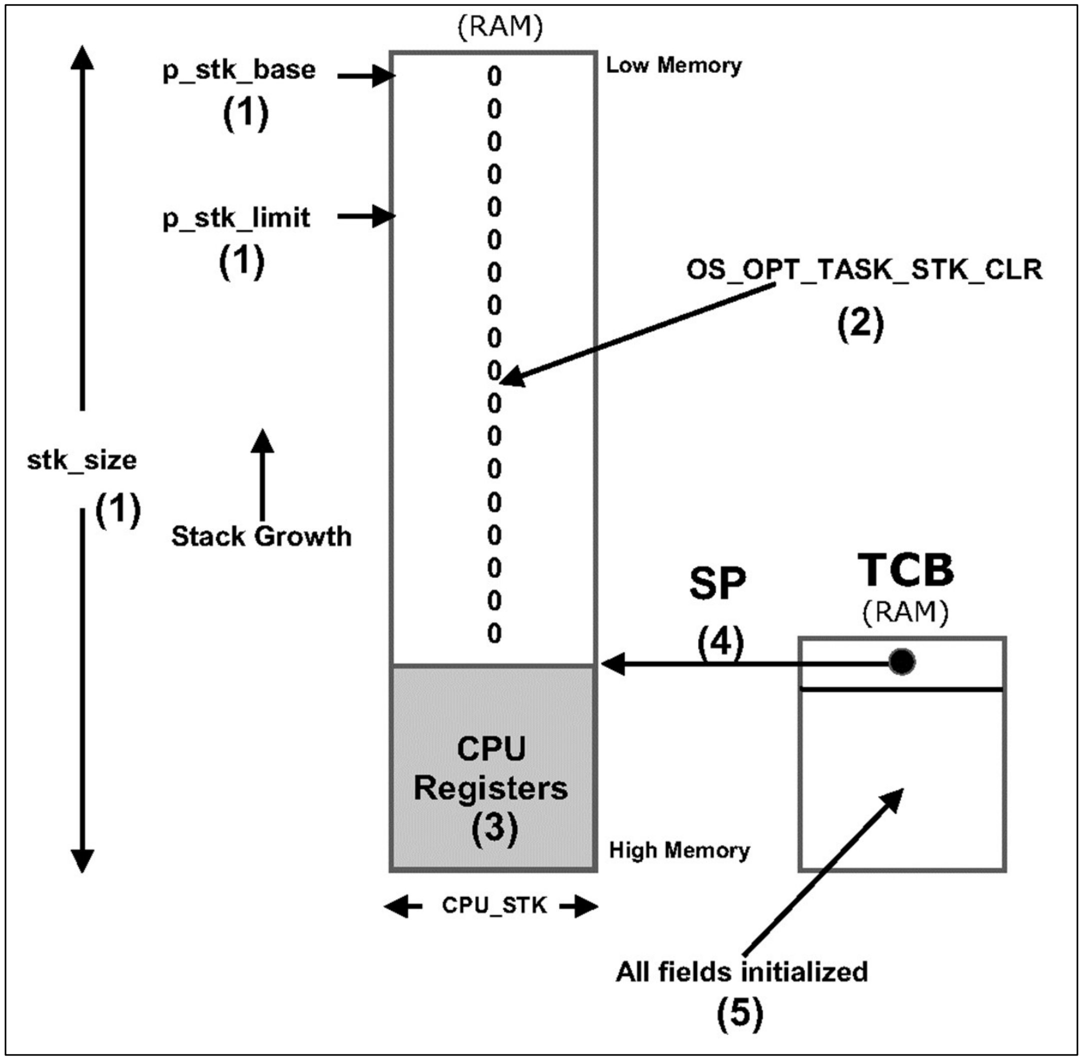
The stack is used to store local variables, function parameters, return addresses and registers during an interrupt.
p_stk_base …
stk_limit …
stk_size …

```
OS_TCB  MtTaskTCB;
CPU_STK MyTaskStk[1000];


OSTaskCreate(&MyTaskTCB,
             "MyTaskName",
              MyTask,
             &MyTaskArg,
              MyPrio,
             &MyTaskStk[0],   /* Stack base address
               100,           /* Used to set .StkLimitPtr to trigger exception
                              /* ... at stack usage > 90%
              1000,           /* Total stack size (in CPU_STK elements)
              MyTaskQSize,
```

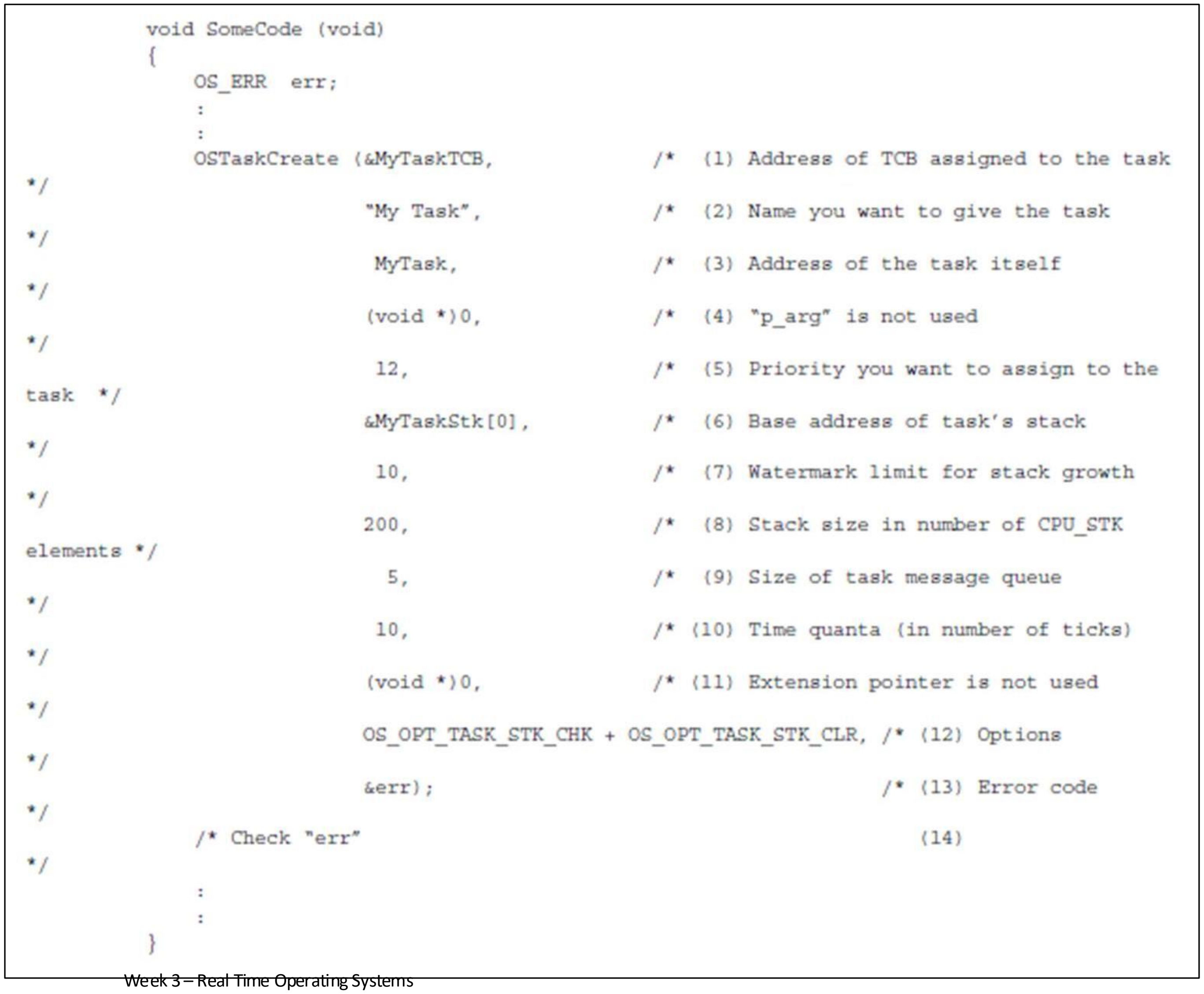


OSTaskCreate() Role

- When we call OSTaskCreate (…) what happens??
- Step 1:
  - (1) …; (2) …; (3) …; (4) …; (5) …
- Step 2:
  - Call OSTaskHook() …
- Step 3:
  - Task placed in ready list …



Example of OSTaskCreate () Usage



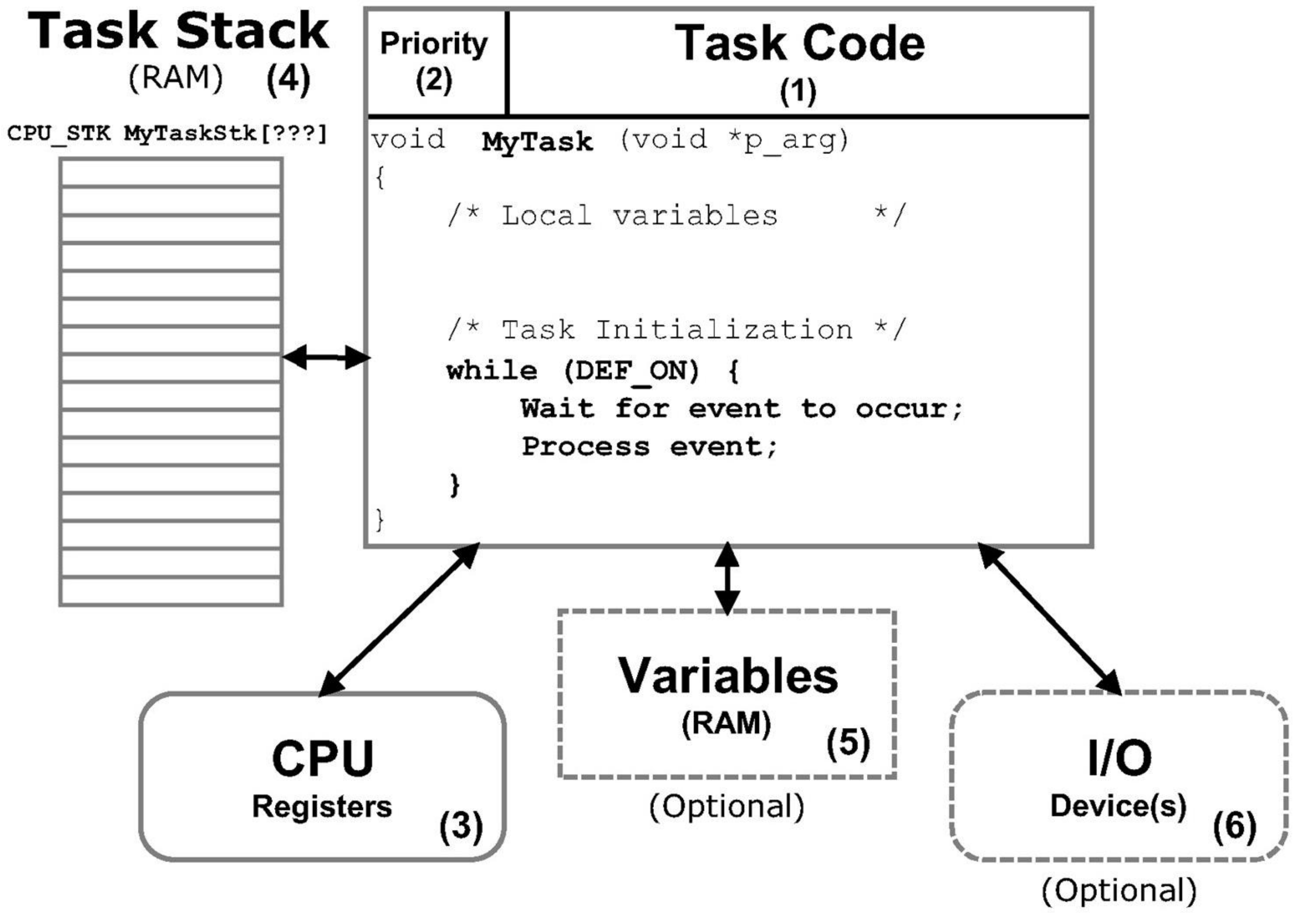
```
          void SomeCode (void)
          {
              OS_ERR  err;
              :
              :
              OSTaskCreate (&MyTaskTCB,             /*  (1) Address of TCB assigned to the task
*/
                            "My Task",              /*  (2) Name you want to give the task
*/
                             MyTask,                /*  (3) Address of the task itself
*/
                            (void *)0,              /*  (4) "p_arg" is not used
*/
                             12,                    /*  (5) Priority you want to assign to the
task  */
                            &MyTaskStk[0],          /*  (6) Base address of task's stack
*/
                             10,                    /*  (7) Watermark limit for stack growth
*/
                            200,                    /*  (8) Stack size in number of CPU_STK
elements */
                              5,                    /*  (9) Size of task message queue
*/
                             10,                    /* (10) Time quanta (in number of ticks)
*/
                            (void *)0,              /* (11) Extension pointer is not used
*/
                            OS_OPT_TASK_STK_CHK + OS_OPT_TASK_STK_CLR, /* (12) Options
*/
                            &err);                                       /* (13) Error code
*/
              /* Check "err"                                                 (14)
*/
              :
              :
          }
```

Functionality of a Task

- The task body can invoke other services, specifically:
  - Create another tasks
  - Suspend and resume other tasks
  - Post signals or messages to other tasks (OS???Post)
  - Share resources with other tasks.
  - Wait for events
  - Etc.

Typical Resources with which a Task Interacts

- (1) …
- (2) …
- (3) …
- (4) …



```
void  MyTask (void *p_arg)
{
    /* Local variables     */


    /* Task Initialization */
    while (DEF_ON) {
        Wait for event to occur;
        Process event;
    }
}
```

Assigning Task Priorities

- One method to assign task priorities is called rate monotonic scheduling (RMS) which assign priorities based on how often the tasks execute
- The tasks with the highest rate ($1/T_i$) of execution are given the highest priority
- For a set of n tasks that use RMS priorities all task hard real-time deadlines are met if the following inequality holds:

$$\sum \frac{E_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

Determining the Size of a Stack for a Task

- The size of the stack required by a task is application specific and accounts for:
  - The nesting of all functions called by the task;
  - The number of local variables to be allocated by all functions called by the task;
  - The stack requirements for all nested ISRs
  - In addition, the stack will store all CPU registers and possible floating-point unit registers
- You could manually figure out (tedious) the stack space needed by adding:
  - all the memory required by all function call nesting (1 pointer each function call for the return address),
  - plus all the memory required by all the arguments passed in those function calls,
  - plus storage for a full CPU context (depends on the CPU),
  - plus another full CPU context for each nested ISRs (if the CPU doesn’t have a separate stack to handle ISRs),
  - plus whatever stack space is needed by those ISRs.
- Most likely you would not make the stack size that precise so you should multiply your result by some safety factor, possibly 1.5 to 2.0

Determining the Size of a Stack for a Task

- Another method is to use the support of compilers/linkers data provided in the link map.
  - For each function, the link map will indicate the worst-case stack usage
- We still need to add to that information:
  - The stack space for a full CPU context switch
  - Plus, another full CPU context for each nested ISR,
  - Plus, whatever space is needed by those ISRs
- Then allow for a safety factor of 1.5 to 2.0.

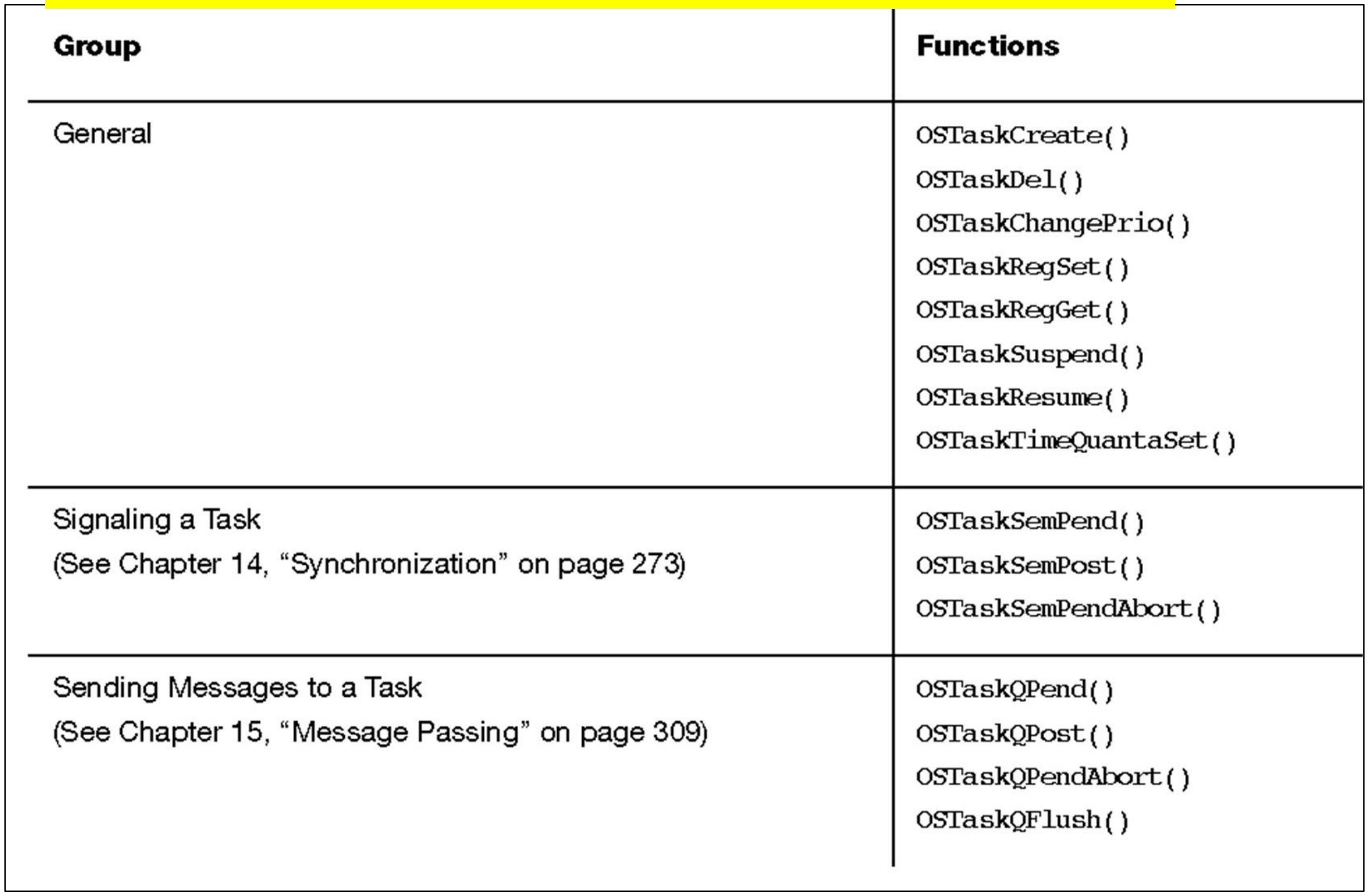
Stack Monitoring

- Monitor stack usage at run-time while developing and testing the product as stack overflows can occur and affect system behavior
- Detecting task stack overflows can be done by (see reference):
  - Using Memory Management Unit (MMU) or a Memory Protection Unit (MPU)
    - hardware support to detect a task attempts to access invalid memory locations
  - Using a CPU with stack overflow detection
    - Supported by some CPU architectures to check some limits for the stack pointer register
  - Custom software-based stack overflow detection
    - Utilizes the OSTaskHook() function to implement this feature
  - Redzone stack overflow detection
    - This is a configurable feature in uC/OS-III version
  - Counting the amount of free stack space
    - Allocate a larger stack space, initialize it with zeros, then use a low priority task to count how much space has been used.

Task Management Services

These services are found in os_task.c and they all start with OSTask???().

See uC/OS-III API Reference Manual for the complete description!



Group	**Functions**

General	OSTaskCreate()
	OSTaskDel()
	OSTaskChangePrio()
	OSTaskRegSet()
	OSTaskRegGet()
	OSTaskSuspend()
	OSTaskResume()
	OSTaskTimeQuantaSet()

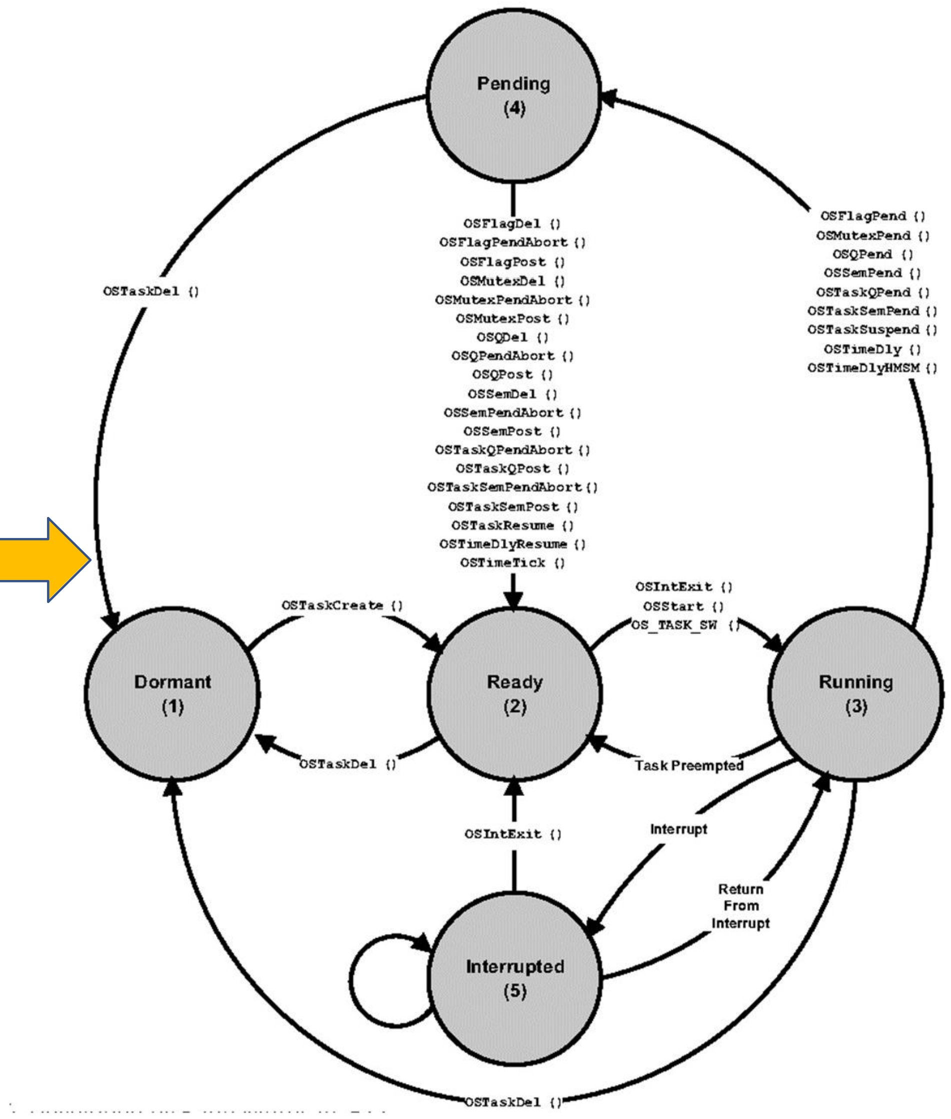
Signaling a Task
(See Chapter 14, "Synchronization" on page 273)
	OSTaskSemPend()
	OSTaskSemPost()
	OSTaskSemPendAbort()

Sending Messages to a Task
(See Chapter 15, "Message Passing" on page 309)
	OSTaskQPend()
	OSTaskQPost()
	OSTaskQPendAbort()
	OSTaskQFlush()

Task States From User Point of View

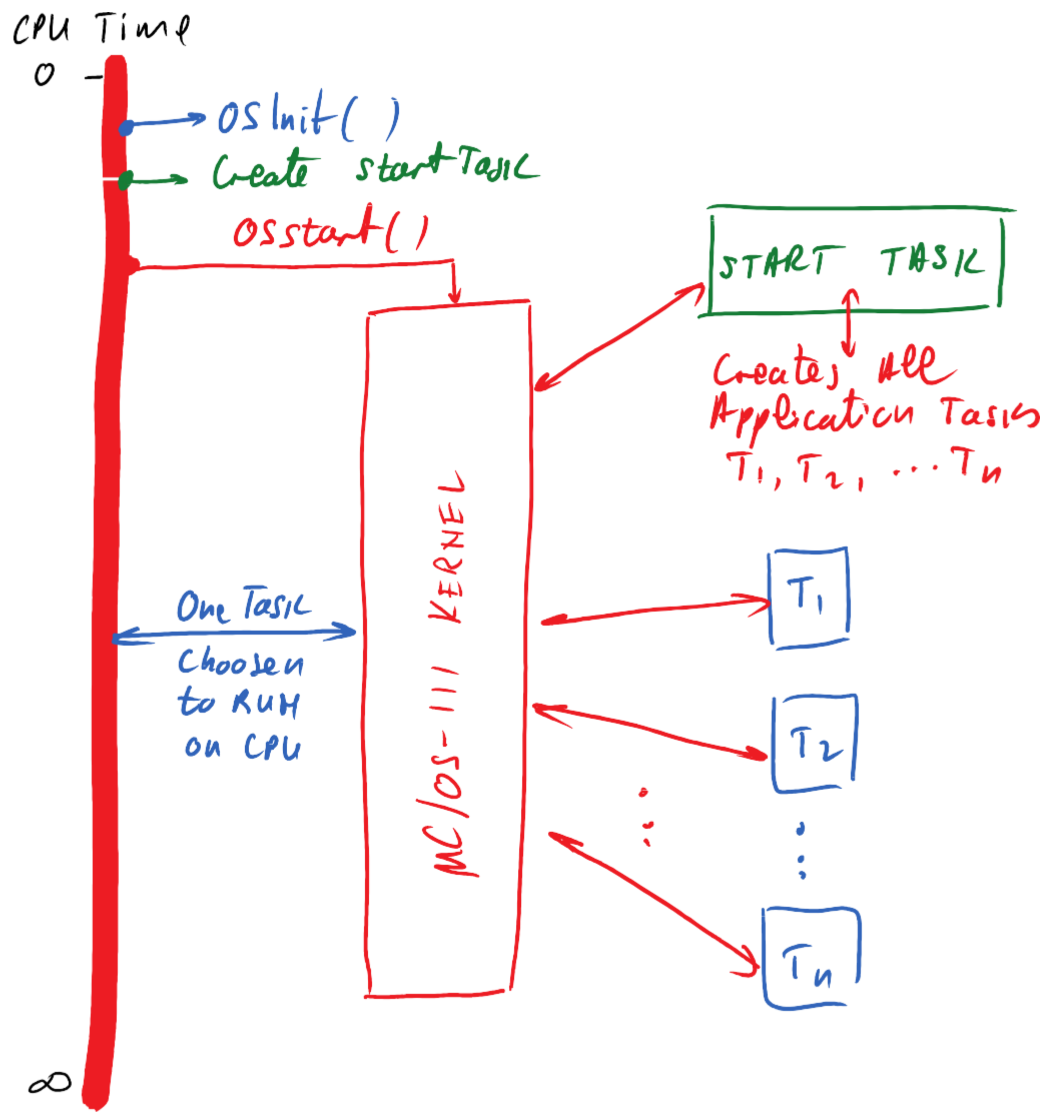
From a user point of view a task can be in any of five states as shown in this state diagram:

(1) …
(2) …
(3) …
(4) …
(5) …



How uC/OS-III Manages CPU Execution Time?

Explain …



Task Control Block (TCB)

- TCB is a data structure that is part of os.h and used by the kernel to keep all the information about a task
- Each task requires its own TCB
  - The user assigns the TCB in user memory space (RAM)
- The address of the TCB is provided to uC/OS-III kernel when calling task related services (i.e., OSTask???())
- The TCB data structure has a long list of fields (See Reference manual pp. 97); however, important fields to notice in the TCB data structure are:
  - .StkPtr …
  - .PendOn …
  - .TaskState …
  - .Prio…
  - .TS …

SLIDE 141 NOTES ... page 1 ...

.PendOn

This field indicates what the task is pending on and contains one of the following values declared in os.h:

-> OS_TASK_PEND_ON_NOTHING
-> OS_TASK_PEND_ON_FLAG
-> OS_TASK_PEND_ON_TASK_Q
-> OS_TASK_PEND_ON_MUTEX
-> OS_TASK_PEND_ON_Q
-> OS_TASK_PEND_ON_SEM
-> OS_TASK_PEND_ON_TASK_SEM
-> OS_TASK_PEND_ON_COND_VAR

.TaskState

This field indicates the current state of a task and contains one of the eight (8) task states that a task can be in.

These states are declared in os.h and reflect the internal states not the user states:

-> OS_TASK_STATE_RDY
-> OS_TASK_STATE_DLY
-> OS_TASK_STATE_PEND
-> OS_TASK_STATE_PEND_TIMEOUT
-> OS_TASK_STATE_SUSPENDED
-> OS_TASK_STATE_DLY_SUSPENDED
-> OS_TASK_STATE_PEND_SUSPENDED
-> OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED

Internal Tasks

- During initialization, µC/OS-III creates between 0 and 4 internal tasks depending on configuration constants found in os_cfg.h:
  - Idle task: OS_IdleTask() – enabled by: OS_CFG_TASK_IDLE_EN
    - The priority of the idle task is always set to OS_CFG_PRIO_MAX-1
    - It runs when no other tasks are ready to run …
  - Tick task: OS_TickTask() – enabled by: OS_CFG_TASK_TICK_EN
  - Statistics task: OS_StatTask() – enabled by: OS_CFG_TASK_STAT_EN
  - Timer task: OS_TmrTask() – enabled by: OS_CFG_TMR_EN

The Tick Task

- Every RTOS will require a periodic time source *called clock* tick or *system tick*
- OS_TickTask() is a task created by µC/OS-III and its priority is configurable, it's priority should be set high, but lower than your most important tasks
- OS_TickTask() is a periodic tasks, waits for signals from the Tick ISR and is used to:
  - keep track of tasks waiting for time to expire or,
  - for tasks that are pending on kernel objects with a timeout

(1) …
(2) …
(3) …
(4) …

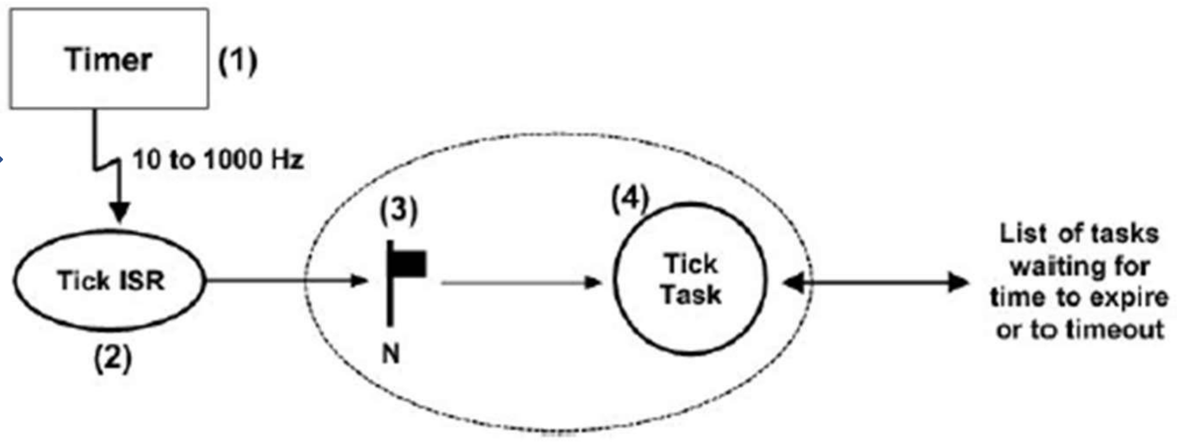


Figure - Tick ISR and Tick Task relationship

Tick Management

- The TickISR () calls OSTimeTick () which does most of the work.
- OS maintains a tick list that is split into two separate ‘delta-lists’
  - (1) This list contains all the tasks that are waiting for time to expire
  - (2) This list contains all the tasks that are pending on an object with a timeout
- Tasks are automatically inserted in the proper tick list when the application calls OSTimeDly???() Or OS???Pend() functions.

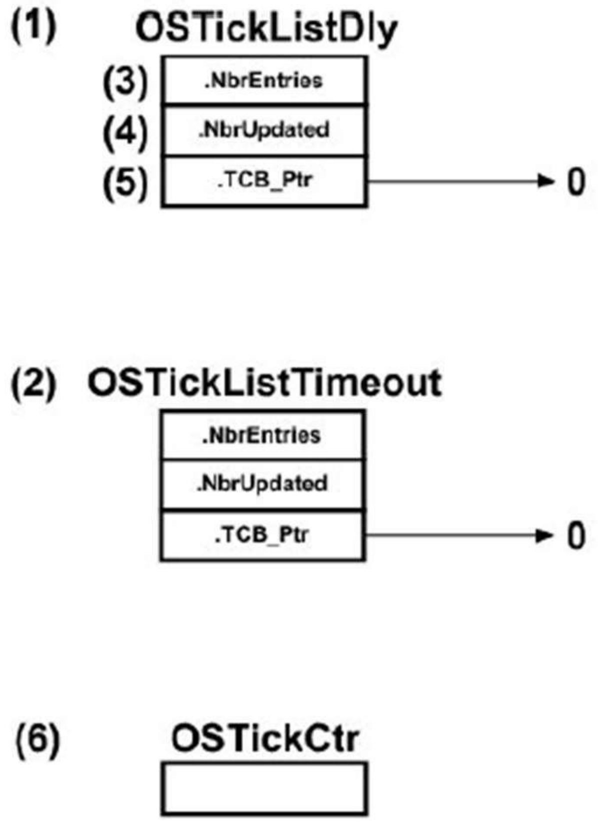


Figure - Empty Tick Lists

Example of Tick Task Working

- Assume that the OSTickListDly list is completely empty
- Then a task uses OSTimeDly() as follows:

  :

  OSTimeDly(10, OS_OPT_TIME_DLY, &err);

  :
- This task will be the first task entered in the tick list as shown

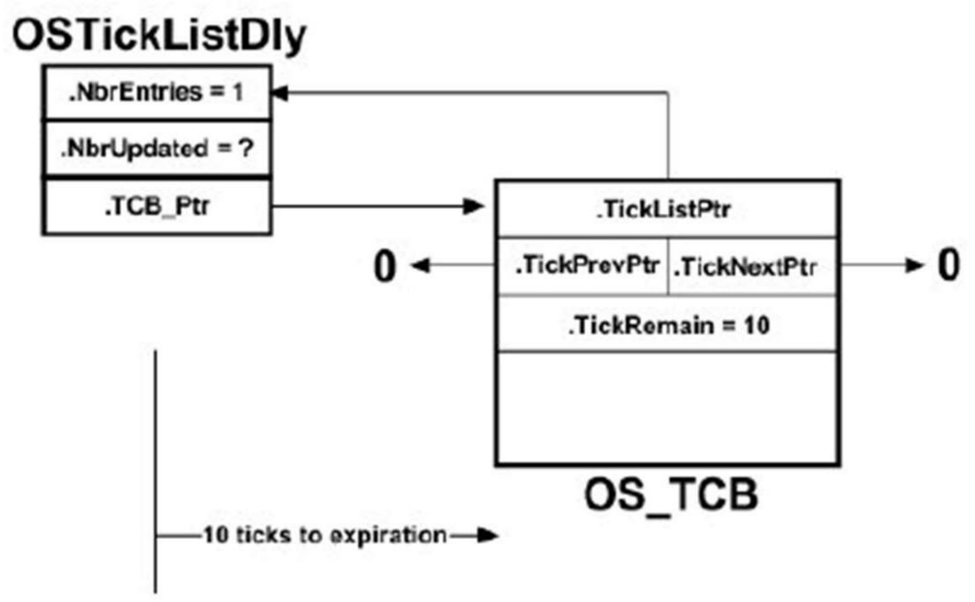


Figure - Inserting a task in the delayed tasks tick list

Example Cont…

- If the next task to run will also call OSTimeDly() “before” the next tick arrives, then the list will update as shown:

  :

  OSTimeDly(7, OS_OPT_TIME_DLY, &err);

Note: the timeout for the last task in the list is the sum of the previuos values in the .TickRemain fields.

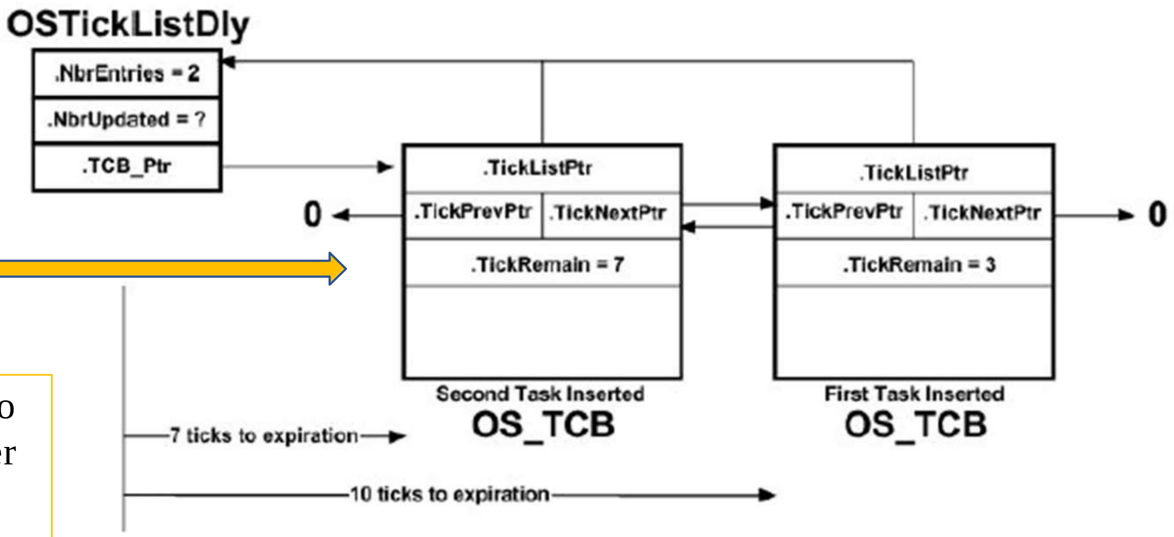


Figure - Inserting a second task in the delayed tasks tick list

The standard method used for the Tick task is to wake up `OS_CFG_TICK_RATE_HZ` times per second as triggered by the Tick ISR.

- Then, the tick task increments `OSTickCtr` by one, and
- Updates the timeout and delay tick lists, reading all tasks that have an expired delay or timeout

Dynamic Tick Mode

- OS offers a dynamic tick mode where the Tick task is not run periodically every tick, this mode will save power consumption
- In this mode the Tick task is awakened when the smallest needed delay of any task of either tick list has elapsed.
- This minimum delay is called a tick step
- The Tick Step, is calculated:
  - every time a task needs to be delayed,
  - every time a task pends on a kernel object with a timeout and
  - every time the Tick task is run.
- This Tick Step is applied to the Dynamic tick timer by calling the BSP_OS_TickNextSet() function

Statistics Task

- This is an optional task and is controlled by a compile-time configuration constant OS_CFG_STAT_TASK_EN defined in os_cfg.h
- This task provides run-time statistics such as overall CPU utilization, per-task CPU utilization, and per-task stack usage.
- As of V3.03.00, CPU utilization is represented as an integer from 0 to 10,000 (0.00% to 100.00%).

- HOMEWORK: Study the use of the statistics task (See reference ebook pp. 121-124)

Timer Task: OS_TmrTask()

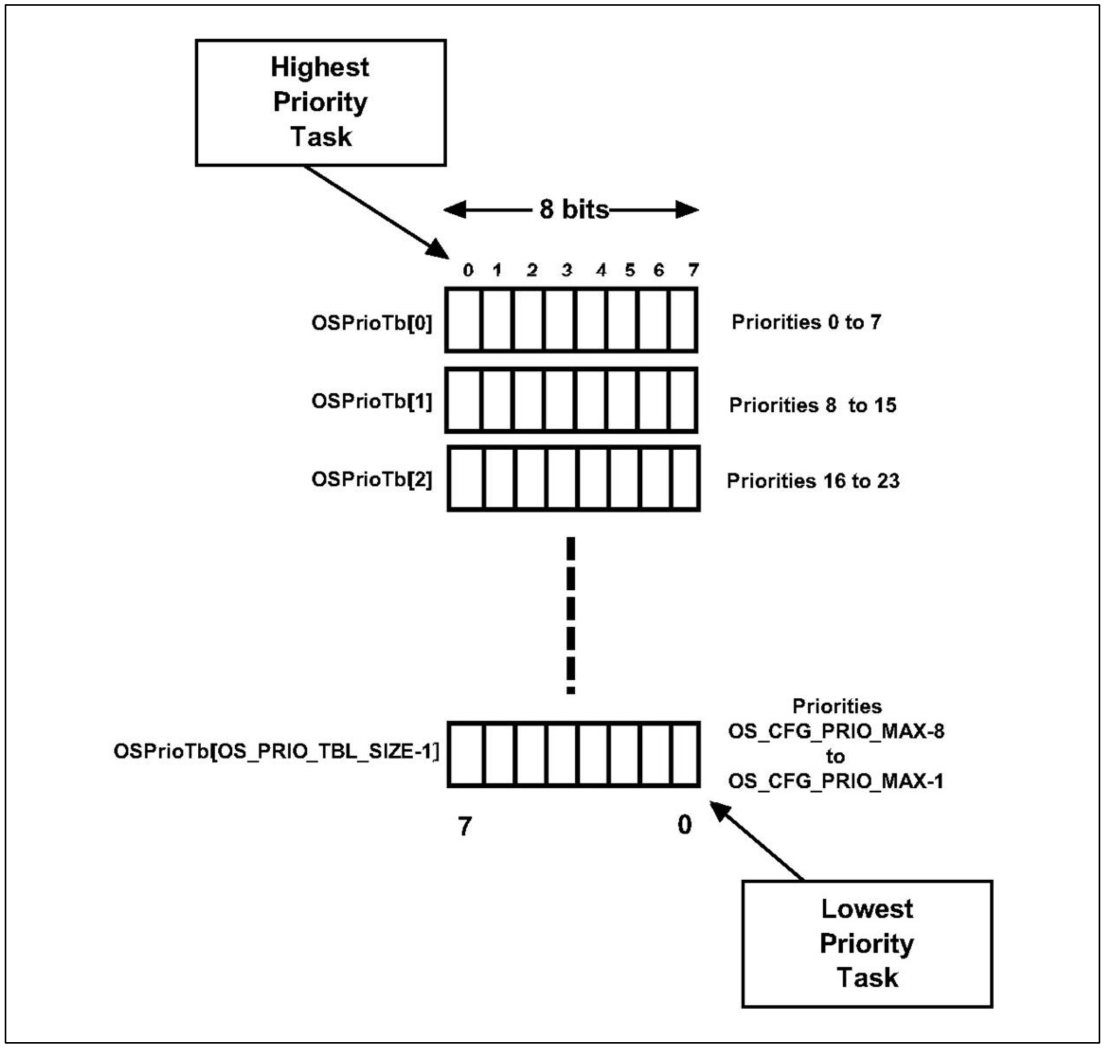
- This is also an optional task enabled at compile time; and it is a periodic task controlled by TickISR(), but updated at a lower rate.
    - Its priority should be set to a low to medium in regard to the other tasks
- Timers are countdown counters that perform an action when the counter reaches zero.
    - The action is provided by the user through a callback function.
- A callback function is a function that the user declares and that will be called when the timer expires.
    - The callback can thus be used to turn on or off a light, a motor, or perform whatever action needed. It is important to
- Note that the callback function is called from the context of the timer task

uC/OS-III Ready List

- The uC/OS-III kernel maintains a ready list which holds all the tasks that are ready to execute
- The ready list consists of two parts:
  - A bitmap which contains the priority levels of the ready tasks
  - A table containing pointers to all the tasks ready.

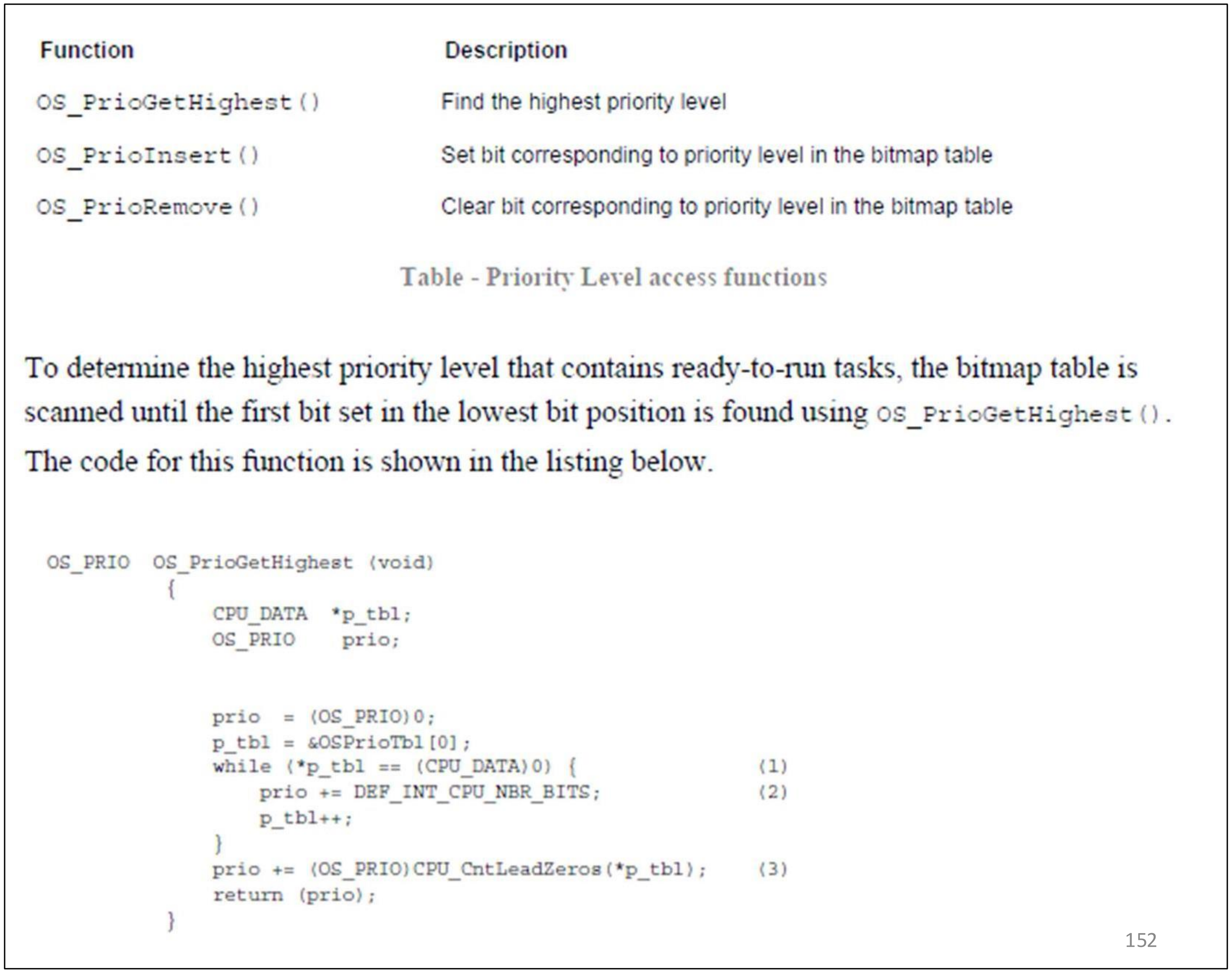
Bitmap of Priority Levels

- The width of the bitmap table depends on the data type CPU_DATA that is found in cpu.h, which can either be 8-, 16-, or 32-bits.
- The example here is an 8-bit bitmap table
- Tasks ready-to-run at a given priority level will have their bit set (i.e., 1) in the bitmap table



Priority Level Access

Homework: Think about a numeric example for this function See pp. 132



Function	Description
`OS_PrioGetHighest()`	Find the highest priority level
`OS_PrioInsert()`	Set bit corresponding to priority level in the bitmap table
`OS_PrioRemove()`	Clear bit corresponding to priority level in the bitmap table

Table - Priority Level access functions

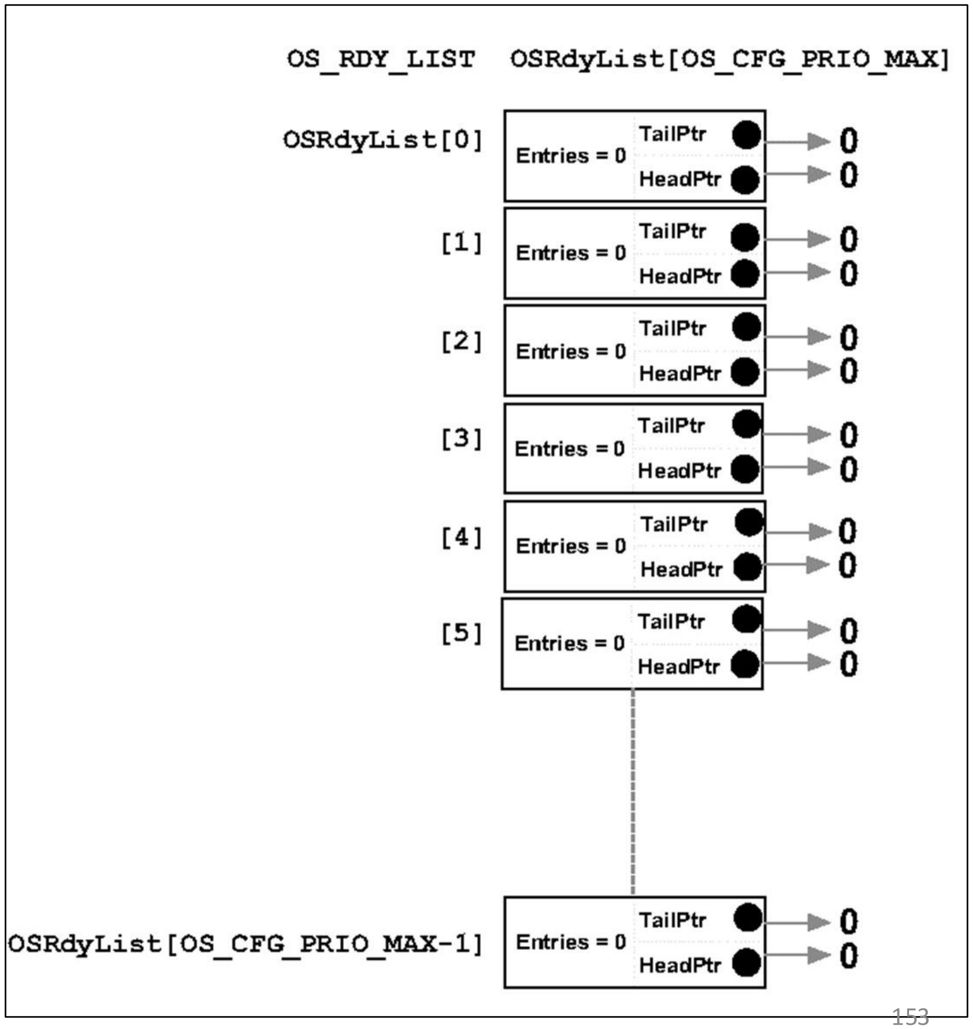
To determine the highest priority level that contains ready-to-run tasks, the bitmap table is scanned until the first bit set in the lowest bit position is found using `OS_PrioGetHighest()`. The code for this function is shown in the listing below.

```
OS_PRIO  OS_PrioGetHighest (void)
         {
             CPU_DATA  *p_tbl;
             OS_PRIO    prio;


             prio  = (OS_PRIO)0;
             p_tbl = &OSPrioTbl[0];
             while (*p_tbl == (CPU_DATA)0) {                  (1)
                 prio += DEF_INT_CPU_NBR_BITS;                (2)
                 p_tbl++;
             }
             prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);       (3)
             return (prio);
         }
```

152

The Ready List Structure

- The ready list is an array called OSRdyList[] containing OS_CFG_PRIO_MAX entries with each entry defined by the data type OS_RDY_LIST (see os.h)
- The OS_RDY_LIST entry consists of three fields:
  - *Entries* …
  - *TailPtr* and *HeadPtr* …
  - The *index* in the array …



After OSInit()

Homework:

See Example of adding a task in the ready list pp. 136-137

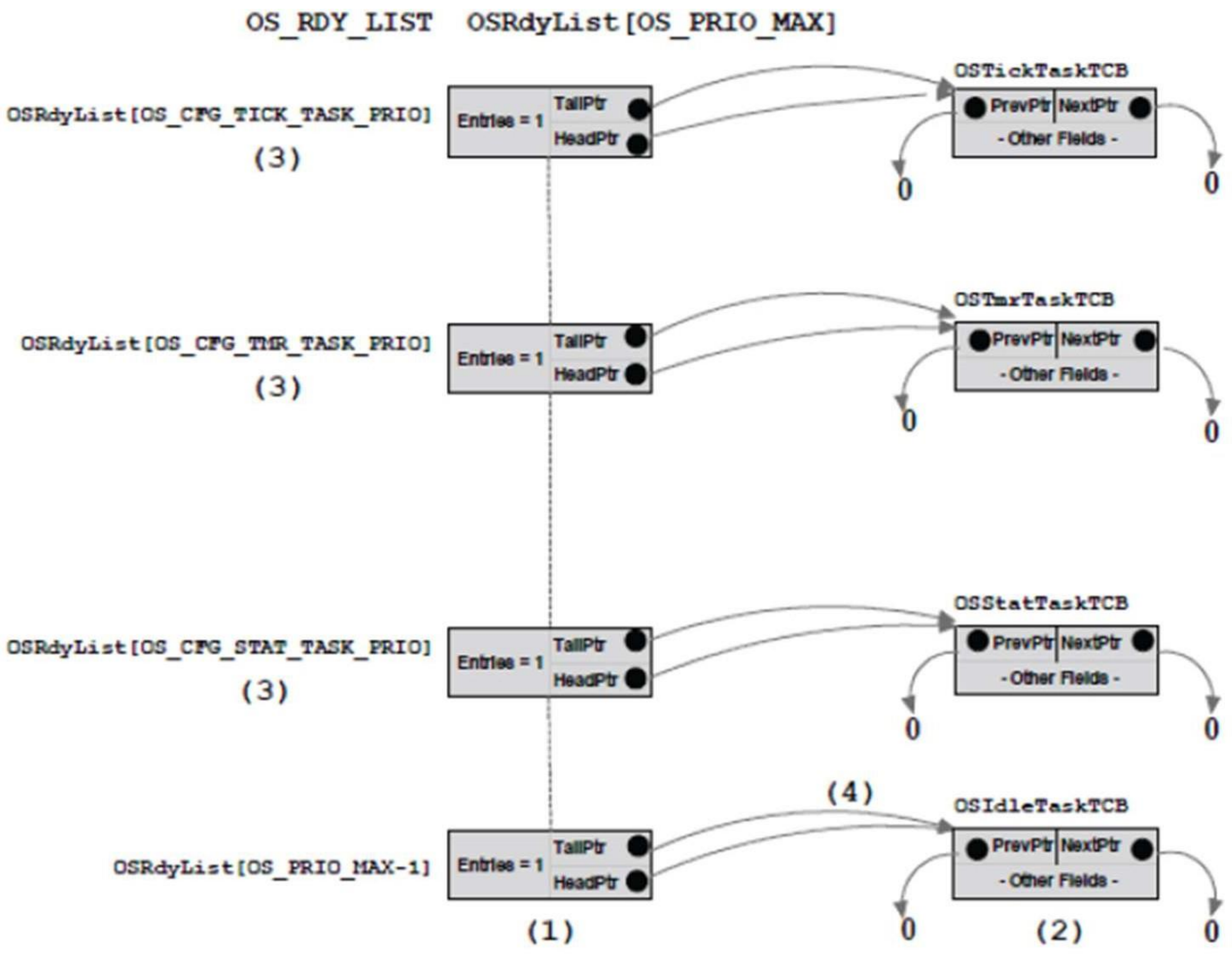


Figure - Ready List after calling OSInit()

154

Scheduling in uC/OS-III

- The scheduler is also called dispatcher
- Preemptive priority based kernel means that if an event occurs, and that event makes a more important task ready-to-run, then uC/OS-III will immediately give control of the CPU to that task
- We have some typical scenarios here:

  Case 1: a task signals or sends a message to a higher-priority task:

  - The current task is suspended, and the higher priority task will run

  Case 2: an Interrupt Service Routine (ISR) signals or sends a message to a higher priority task than the one interrupted:

  - The interrupted task remains suspended and the new task runs
  - Here, µC/OS-III handles event posting from interrupts using two different methods: Direct and Deferred Post

Preemptive Scheduling Workings

- (1) …?
- (2) …
- (3) …
- (4) …
- (5) …
- (6) …
- (7) …
- (8) …
- (9) …
- (10) …
- (11) …

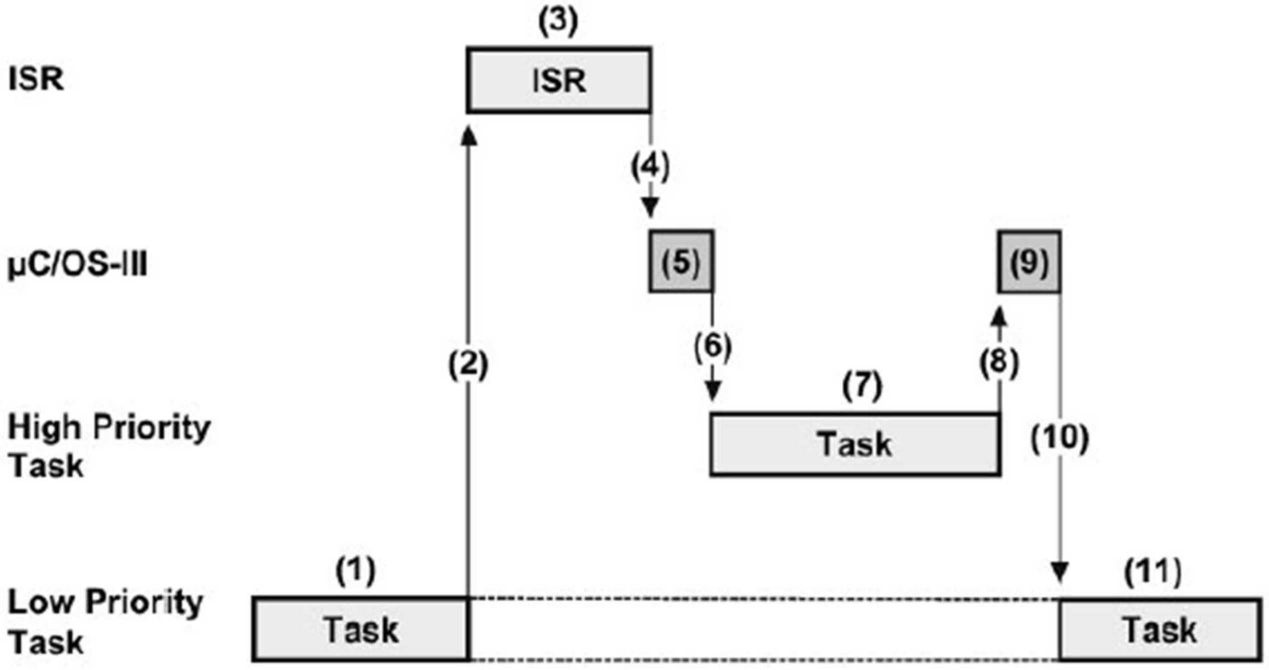


Figure - Preemptive scheduling

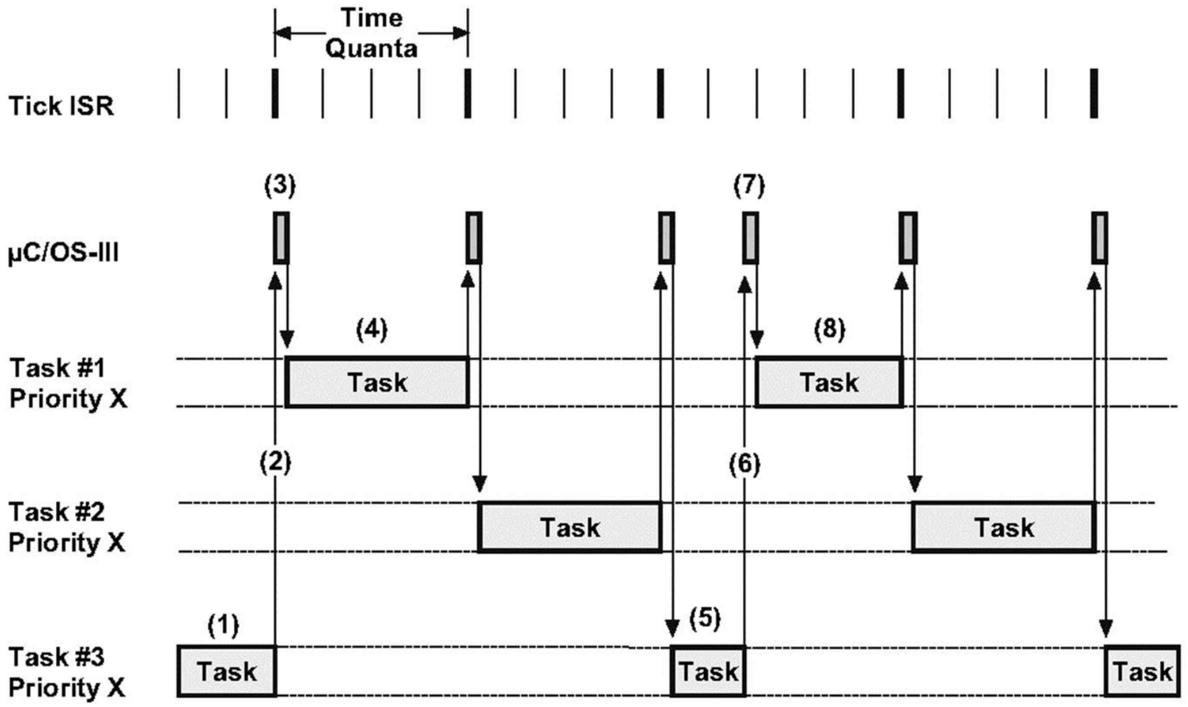
Scheduling Points

Scheduling occurs at scheduling points and nothing special must be done in the application code since scheduling occurs automatically based on conditions happening at scheduling points:

1. A task signals or sends a message to another task: OS???Post()
2. A task calls OSTimeDly() or OSTimeDlyHMSM()
3. A task waits for an event to occur, and the event has not yet occurred: OS???Pend()
4. If a task aborts a pend -- by calling OS???PendAbort()
5. If a task is created or deleted
6. If a kernel object is deleted -- an event flag group, a semaphore, a message queue, or a mutual exclusion semaphore.
8. A task changes the priority of itself or another task.
9. A task suspends itself by calling OSTaskSuspend()
10. A task resumes another task that was suspended by OSTaskSuspend()
11. At the end of all nested ISRs -- the scheduling is performed by OSIntExit
12. The scheduler is unlocked by calling OSSchedUnlock()
13. A task gives up its time quanta by calling OSScedRoundRobinYield()
14. The user calls OSSched …

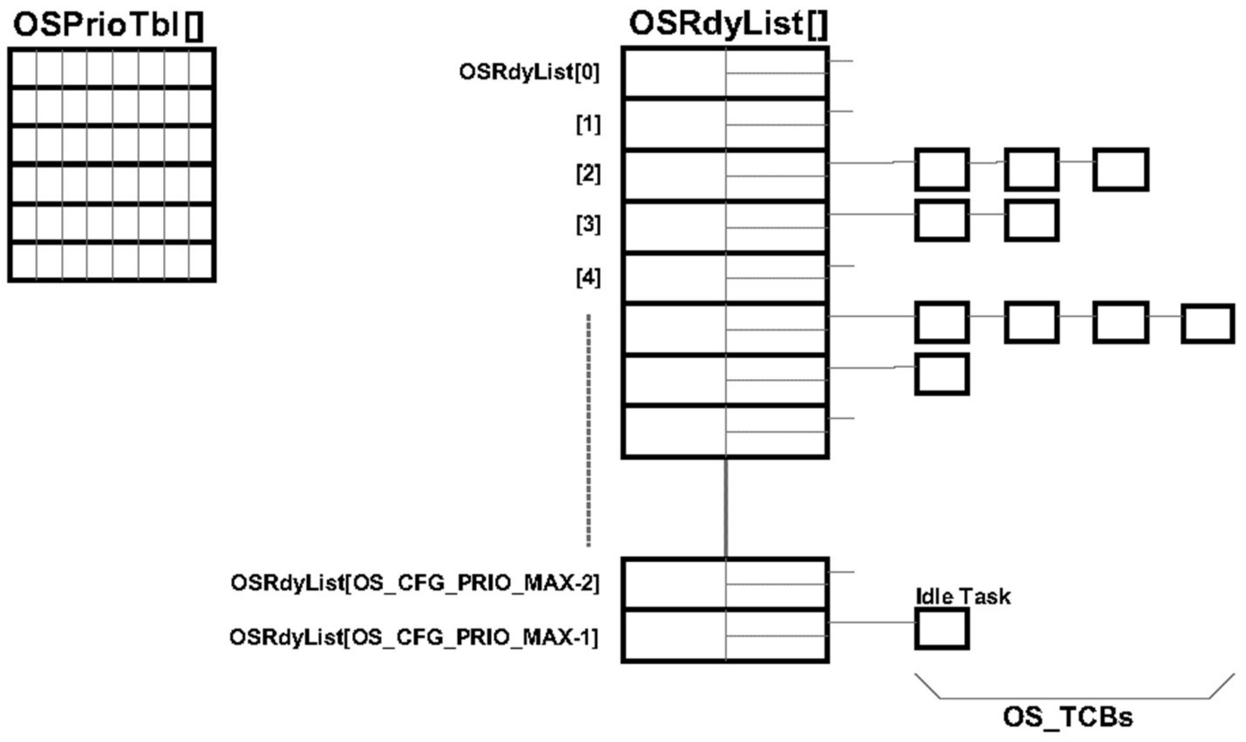
Round-Robin or Time Slicing Scheduling

- When more tasks have the same priority: a task will run for a *Time Quanta* □ round-robin
- A task can give up voluntarily its *Time Quanta – Yielding*
- (1) …; (8)
- OS_SchedRoundRobin() is the code used to select the task to run.



Scheduling Internals

- Scheduling is performed by two functions:
  - OSSched(): called by task level code
  - OSIntExit(): called by ISR level code
- Data structures used by the scheduler are:
  - The priority ready bitmap table
  - The array ready list


OSPrioTbl[]
OSRdyList[]
OSRdyList[0]
[1]
[2]
[3]
[4]
OSRdyList[OS_CFG_PRIO_MAX-2]
OSRdyList[OS_CFG_PRIO_MAX-1]
Idle Task
OS_TCBs