# Chapter-3 Kernel Structure

Dr. Li-Pin Chang
Real-Time and Embedded System Lab.
National Taiwan University

# Objectives

- To understand what a task is.
- To learn how uC/OS-2 manages tasks.
- To know how an ISR works.
- To learn how to determine the percent CPU your application is using.

# The uC/OS-2 File Structure

Application Code (Your Code!)

**Processor independent implementations**

- Scheduling policy
- Event flags
- Semaphores
- Mailboxes
- Event queues
- Task management
- Time management
- Memory management

**Application Specific Configurations**

OS_CFG.H

- Max # of tasks
- Max Queue length
- …

uC/OS-2 port for processor specific codes

Software

---

Hardware

CPU

Timer

# Source Availability

- Download the source code of uC/OS-2 from the “course” section of the web site http://140.112.28.99/me

- The password for extraction is “tzuchiang”

# Critical Sections

- A critical section is a portion of code that is not safe from race conditions.
  - Because of the use of shared resources.
- They can be protected by interrupt disabling/enabling interrupts or semaphores.
  - However, the use of semaphores imposes a more significant amount of overheads.
  - A RTOS kernel itself mostly use interrupts disabling/enabling to protect critical sections. (why?)
- Once interrupts are disabled, neither context switches nor any other ISR's can occur.

# Critical Sections

- The interrupt latency is a vital specification of an RTOS.
  - Interrupts should be disabled as short as possible to improve the responsiveness.
  - It must be accounted as a blocking time in the schedulability analysis.

- Interrupt disabling must be used carefully:
  - E.g., if OSTimeDly() is called with interrupt disabled, the machine might hang!

```
{
    .
    OS_ENTER_CRITICAL();
    .     /* Critical Section */
    OS_EXIT_CRITICAL();
    .
}
```

# Critical Sections

- The states of the processor must be carefully maintained across multiple calls of OS_ENTER_CRITICAL() / OS_EXIT_CRITICAL().

- There are three possible implementations for the maintenance of process states:
  - Interrupt enabling/disabling instructions.
  - Interrupt status save/restore onto/from stacks.
  - Processor Status Word (PSW) save/restore onto/from memory variables.

- Interrupt enabling/disabling can be done by various way:
  - In-line assembly.
  - Compiler extension for specific processors.

# Critical Sections

- OS_CRITICAL_METHOD=1
- Interrupt enabling/disabling instructions.
- The simplest way, however, this approach does not have the sense of “save” and “restore”.
- Interrupt statuses might not be consistent across kernel services/function calls!!

```
{
    .
    disable_interrupt();
    a_kernel_service();
    .
    .
}
```

```
{
    .
    disable_interrupt();
    critical section
    enable_interrupt();
    .
}
```

Interrupts are now implicitly re-enabled!

# Critical Sections

- OS_CRITICAL_METHOD=2
- Processor Status Word (PSW) can be saved/restored onto/from stacks.
  - PSW's of nested interrupt enable/disable operations can be exactly recorded in stacks.

```
#define OS_ENTER_CRITICAL() \
        asm(“PUSH   PSW”);
        asm(“DI”);


#define OS_EXIT_CRITICAL() \
        asm(“POP    PSW”);
```

Some compilers might not be smart enough to adjust the stack pointer after the processing of in-line assembly.

# Critical Sections

- OS_CRITICAL_METHOD=3
- The compiler and processor allow the PSW to be saved/restored to/from a memory variable.

```
void foo(arguments)
{
    OS_CPU_SR cpu_sr;

    .
    cpu_sr = get_processor_psw();
    disable_interrupts();
    .
    /* critical section */
    .
    set_processor_psw(cpu_sr);
    .
}
```

OS_ENTER_CRITICAL()

OS_EXIT_CRITICAL()

# Tasks

- A task is an active entity which could do some computations.
- Under real-time systems, a task is typically an infinite loop.

```
void YourTask (void *pdata)                                (1)
{
   for (;;) {                                              (2)
      /* 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 */
   }
}
```

Delay itself for next event/period, so that other tasks can run.

# Tasks

- uC/OS-2 can have up to 64 priorities.
  - Each task must associate with an unique priority.
  - 63 and 62 are reserved (idle, stat).

- Insufficient number of priority will damage the schedulability of a real-time scheduler.
  - The number of schedulable task would be reduced.
    - Because there is no distinction among the tasks with the same priority.
    - For example, under RMS, tasks have different periods but are assigned with the same priority.
    - It is possible that all other tasks with the same priority are always issued before a particular task.
  - Fortunately, most embedded systems have a limited number of tasks to run.

# Tasks

- A task is created by OSTaskCreate() or OSTaskCreateExt().
- The priority of a task can be changed by OSTaskChangePrio().
- A task could delete itself when done.

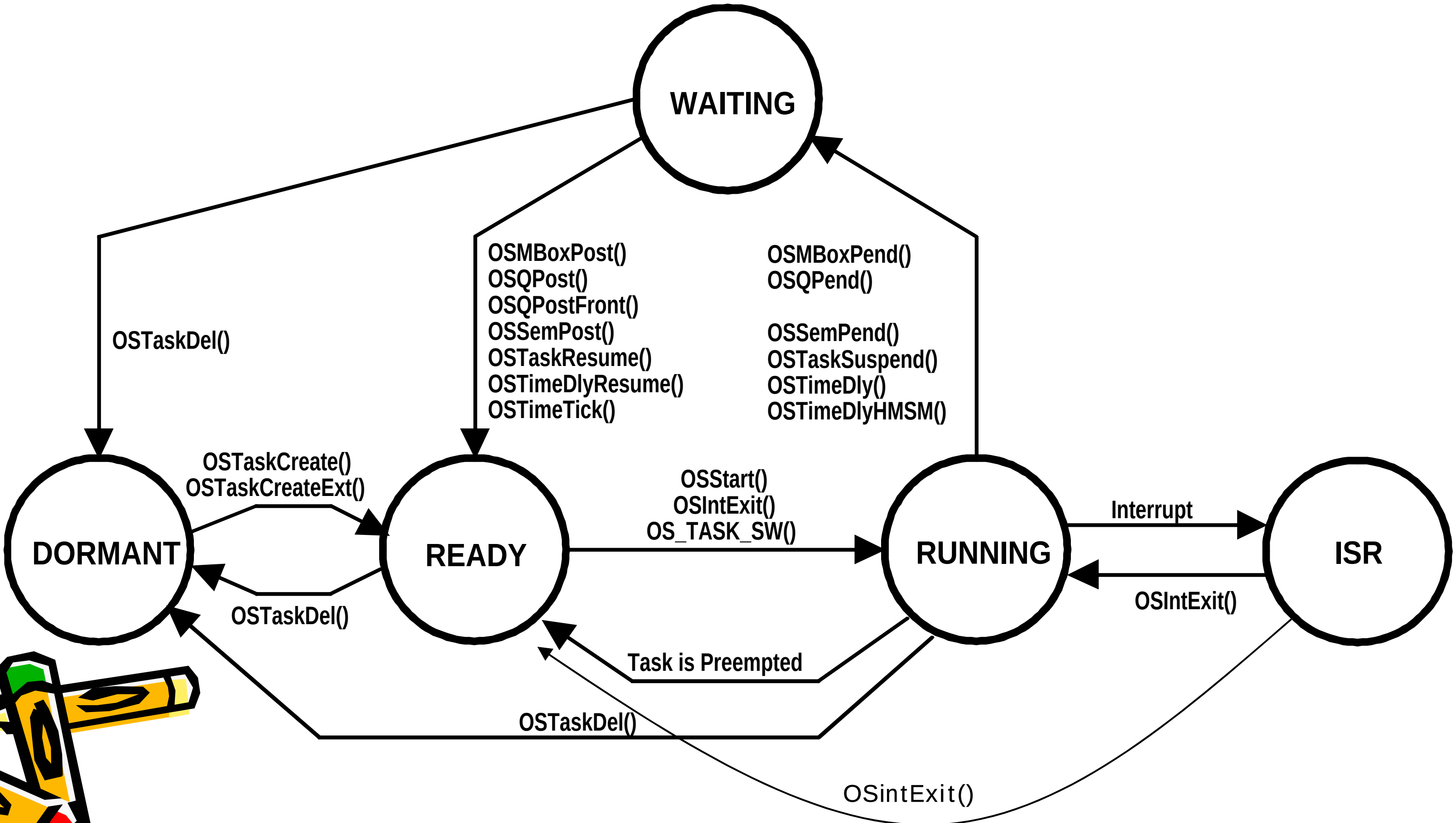
```
void YourTask (void *pdata)
{
   /* USER CODE */
   OSTaskDel(OS_PRIO_SELF);
}
```

The priority of the current task

# Task States

- Dormant: Procedures residing on RAM/ROM is not an task unless you call OSTaskCreate() to execute them.
  - Actually no tasks correspond to the codes.
- Ready: A task is neither delayed nor waiting for any event to occur.
  - A task is ready once it is created.
- Running: A ready task is scheduled to run on the CPU .
  - There must be only one running task.
  - The task running might be preempted and become ready.
- Waiting: A task is waiting for certain events to occur.
  - Timer expiration, signaling of semaphores, messages in mailboxes, and etc.
- ISR: A task is preempted by an interrupt.
  - The stack of the interrupted task is utilized by the ISR.

# Task States


WAITING
OSTaskDel()
OSMBoxPost()
OSQPost()
OSQPostFront()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
OSMBoxPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
OSTaskCreate()
OSTaskCreateExt()
DORMANT
READY
OSStart()
OSIntExit()
OS_TASK_SW()
RUNNING
Interrupt
ISR
OSTaskDel()
OSIntExit()
Task is Preempted
OSTaskDel()
OSintExit()

# Task States

- A task can delay itself by calling OSTimeDly() or OSTimeDlyHMSM().
  - The task is placed in the waiting state.
  - The task will be made ready by OSTimeTick().
    - It is the clock ISR, you don't have to call it explicitly from your code.

- A task can wait for an event by OSFlagPend(), OSSemPend(), OSMboxPend(), or OSQPend().
  - The task remains waiting until the occurrence of the desired event. (or timeout)

- The running task is always preempted by ISR's, unless interrupts are disabled.
  - ISR's could make one or more tasks ready by signaling events.
  - On the return of an ISR, the scheduler will check if rescheduling is needed.

- Once new tasks become ready, the next highest priority ready task is scheduled to run (due to occurrences of events, timer expirations).

- If no task is running and all tasks are not in the ready state, the idle task executes.

# Task Control Blocks (TCB)

- A TCB is a main-memory-resident data structure used by to maintain the state of a task when it is preempted.
- Each task is associated with a TCB.
  - All valid TCB's are doubly linked.
  - Free TCB's are linked in a free list.
- The contents of a TCB is saved/restored when a context-switch occurs.
  - Task priority, delay counter, event to wait, location of the stack.
  - CPU registers are stored in the stack rather than in the TCB.

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;
#if OS_TASK_CREATE_EXT_EN
    void            *OSTCBExtPtr;
    OS_STK          *OSTCBStkBottom;
    INT32U           OSTCBStkSize;
    INT16U           OSTCBOpt;
    INT16U           OSTCBId;
#endif
    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
    OS_EVENT        *OSTCBEventPtr;
#endif
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    void            *OSTCBMsg;
#endif

    INT16U           OSTCBDly;
    INT8U            OSTCBStat;
    INT8U            OSTCBPrio;
    INT8U            OSTCBX;
    INT8U            OSTCBY;
    INT8U            OSTCBBitX;
    INT8U            OSTCBBitY;
#if OS_TASK_DEL_EN
    BOOLEAN          OSTCBDelReq;
#endif
} OS_TCB;
```

# Task Control Blocks (TCB)

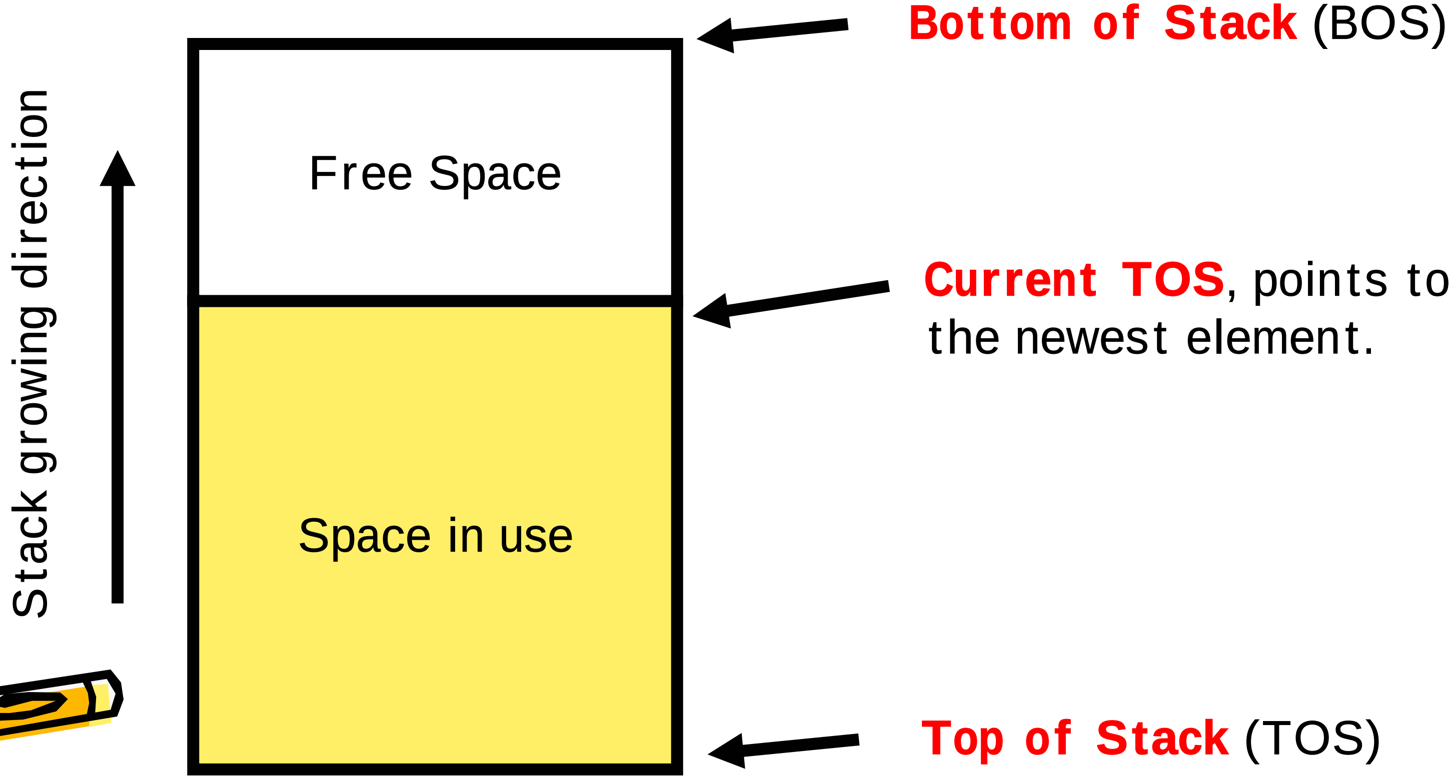
- **.OSTCBStkPtr** contains a pointer to the current TOS for the task.
  - It is the first entry of TCB so that it can be accessed directly from assembly language. (offset=0)

- **.OSTCBExtPtr** is a pointer to a user-definable task control block extension.
  - Set OS_TASK_CREATE_EXT_EN to 1.
  - The pointer is set when OSTaskCreateExt( ) is called
  - The pointer is ordinarily cleared in the hook OSTaskDelHook().

- **.OSTCBStkBottom** is a pointer to the bottom of the task's stack.

- **.OSTCBStkSize** holds the size of the stack in number of elements instead of bytes.
  - The element size is the macro OS_STK.
  - Total stack size is OSTCBStkSize*OS_STK bytes
  - .OSTCBStkBottom and .OSTCBStkSize are used to check stack.

# Task Control Blocks (TCB)



# Task Control Blocks (TCB)

- **.OSTCBOpt** holds “options” that can be passed to OSTaskCreateExt( )
  - OS_TASK_OPT_STK_CHK: stack checking is enable for the task being created.
  - OS_TASK_OPT_STK_CLR: indicates that the stack needs to be cleared when the task is created.
  - OS_TASK_OPT_SAVE_FP: tells OSTaskCreateExt( ) that the task will be doing floating-point computations. Floating point processor’s registers must be saved to the stack on context-switches.
- **.OSTCBId**: holds an identifier for the task.
- **.OSTCBNext** and **.OSTCBPrev** are used to double link OS_TCBs
- **.OSTCBEVEventPtr** is pointer to an event control block.
- **.OSTCBMsg** is a pointer to a message that is sent to a task.
- **.OSTCBFlagNode** is a pointer to a flagnode.
- **.OSTCBFlagsRdy** maintains which event flags make the task ready.
- **.OSTCBDly** is used when:
  - a task needs to be delayed for a certain number of clock ticks, or
  - a task needs to pend for an event to occur with a timeout.
- **.OSTCBStat** contains the state of the task. ( 0 is ready to run)
- **.OSTCBPrio** contains the task priority.

# Task Control Blocks (TCB)

- **.OSTCBX .OSTCBY .OSTCBBitX** and **.OSTCBBitY**
  - They are used to accelerate the process of making a task ready to run or make a task wait for an event.

```
OSTCBY  = priority >> 3;
OSTCBBitY       = OSMapTbl[priority >> 3];
OSTCBX  = priority & 0x07;
OSTCBBitX       = OSMapTbl[priority & 0x07];
```

- **.OSTCBDelReq** is boolean used to indicate whether or not a task request that the current task to be deleted.

- OS_MAX_TASKS is specified in OS_CFG.H
  - # OS_TCBs allocated by µC/OS-II

- **OSTCBTbl**[ ] : where all OS_TCBs are placed.

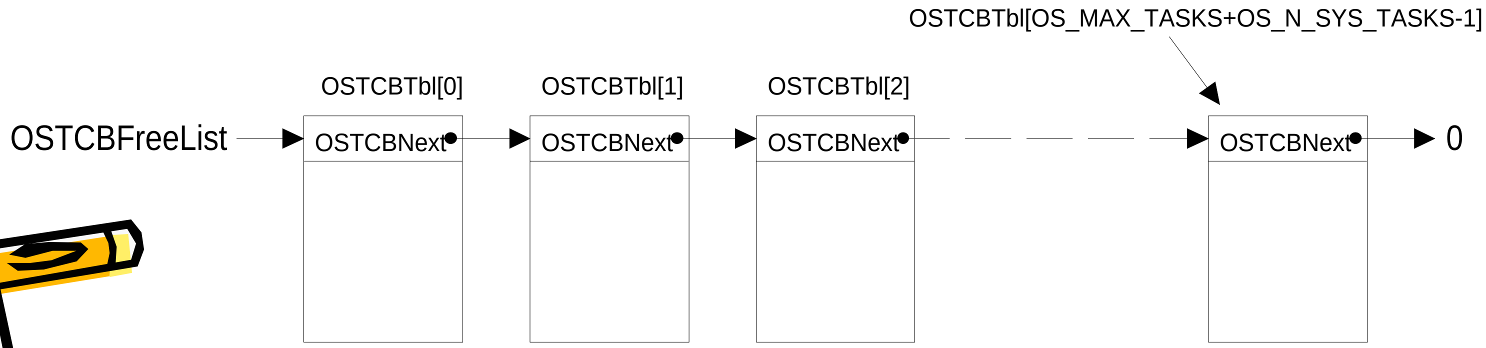
- When µC/OS-II is initialized, all OS_TCBs in the table are linked in a singly linked list of free OS_TCBs

# Task Control Blocks (TCB)

- When a task is created, the OS_TCB pointed to by OSTCBFreeList is assigned to the task, and OSTCBFreeList is adjusted to point the next OS_TCB in the chain.

- When a task is deleted, its OS_TCB is returned to the list of free OS_TCB.

- An OS_TCB is initialized by the function OS_TCBInit(), which is called by OSTaskCreate().



```
INT8U  OS_TCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos, INT16U id, 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_TCB    *ptcb;


    OS_ENTER_CRITICAL();
    ptcb = OSTCBFreeList;                                 /* Get a free TCB from the free TCB list    */
    if (ptcb != (OS_TCB *)0) {
        OSTCBFreeList          = ptcb->OSTCBNext;         /* Update pointer to free TCB list          */
        OS_EXIT_CRITICAL();
        ptcb->OSTCBStkPtr    = ptos;                      /* Load Stack pointer in TCB                */
        ptcb->OSTCBPrio      = (INT8U)prio;               /* Load task priority into TCB              */
        ptcb->OSTCBStat      = OS_STAT_RDY;               /* Task is ready to run                     */
        ptcb->OSTCBDly       = 0;                         /* Task is not delayed                      */

#if OS_TASK_CREATE_EXT_EN > 0
        ptcb->OSTCBExtPtr    = pext;                      /* Store pointer to TCB extension           */
        ptcb->OSTCBStkSize   = stk_size;                  /* Store stack size                         */
        ptcb->OSTCBStkBottom = pbos;                      /* Store pointer to bottom of stack         */
        ptcb->OSTCBOpt       = opt;                       /* Store task options                       */
        ptcb->OSTCBId        = id;                        /* Store task ID                            */
#else
        pext                 = pext;                      /* Prevent compiler warning if not used     */
        stk_size             = stk_size;
        pbos                 = pbos;
        opt                  = opt;
        id                   = id;
#endif

#if OS_TASK_DEL_EN > 0
        ptcb->OSTCBDelReq    = OS_NO_ERR;
#endif

        ptcb->OSTCBY         = prio >> 3;                 /* Pre-compute X, Y, BitX and BitY          */
        ptcb->OSTCBBitY      = OSMapTbl[ptcb->OSTCBY];
        ptcb->OSTCBX         = prio & 0x07;
        ptcb->OSTCBBitX      = OSMapTbl[ptcb->OSTCBX];
```

Get a free TCB from the free list

```
#if OS_EVENT_EN > 0
        ptcb->OSTCBEventPtr  = (OS_EVENT *)0;              /* Task is not pending on an event          */
#endif

#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0) && (OS_TASK_DEL_EN > 0)
        ptcb->OSTCBFlagNode  = (OS_FLAG_NODE *)0;          /* Task is not pending on an event flag     */
#endif

#if (OS_MBOX_EN > 0) || ((OS_Q_EN > 0) && (OS_MAX_QS > 0))
        ptcb->OSTCBMsg       = (void *)0;                  /* No message received                      */
#endif

#if OS_VERSION >= 204
        OSTCBInitHook(ptcb);
#endif

        OSTaskCreateHook(ptcb);                            /* Call user defined hook                   */

        OS_ENTER_CRITICAL();
        OSTCBPrioTbl[prio] = ptcb;
        ptcb->OSTCBNext    = OSTCBList;                    /* Link into TCB chain                      */
        ptcb->OSTCBPrev    = (OS_TCB *)0;
        if (OSTCBList != (OS_TCB *)0) {
            OSTCBList->OSTCBPrev = ptcb;
        }
        OSTCBList               = ptcb;
        OSRdyGrp               |= ptcb->OSTCBBitY;         /* Make task ready to run                   */
        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        OS_EXIT_CRITICAL();
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();
    return (OS_NO_MORE_TCB);
}
```

User-defined hook is called here.

Priority table

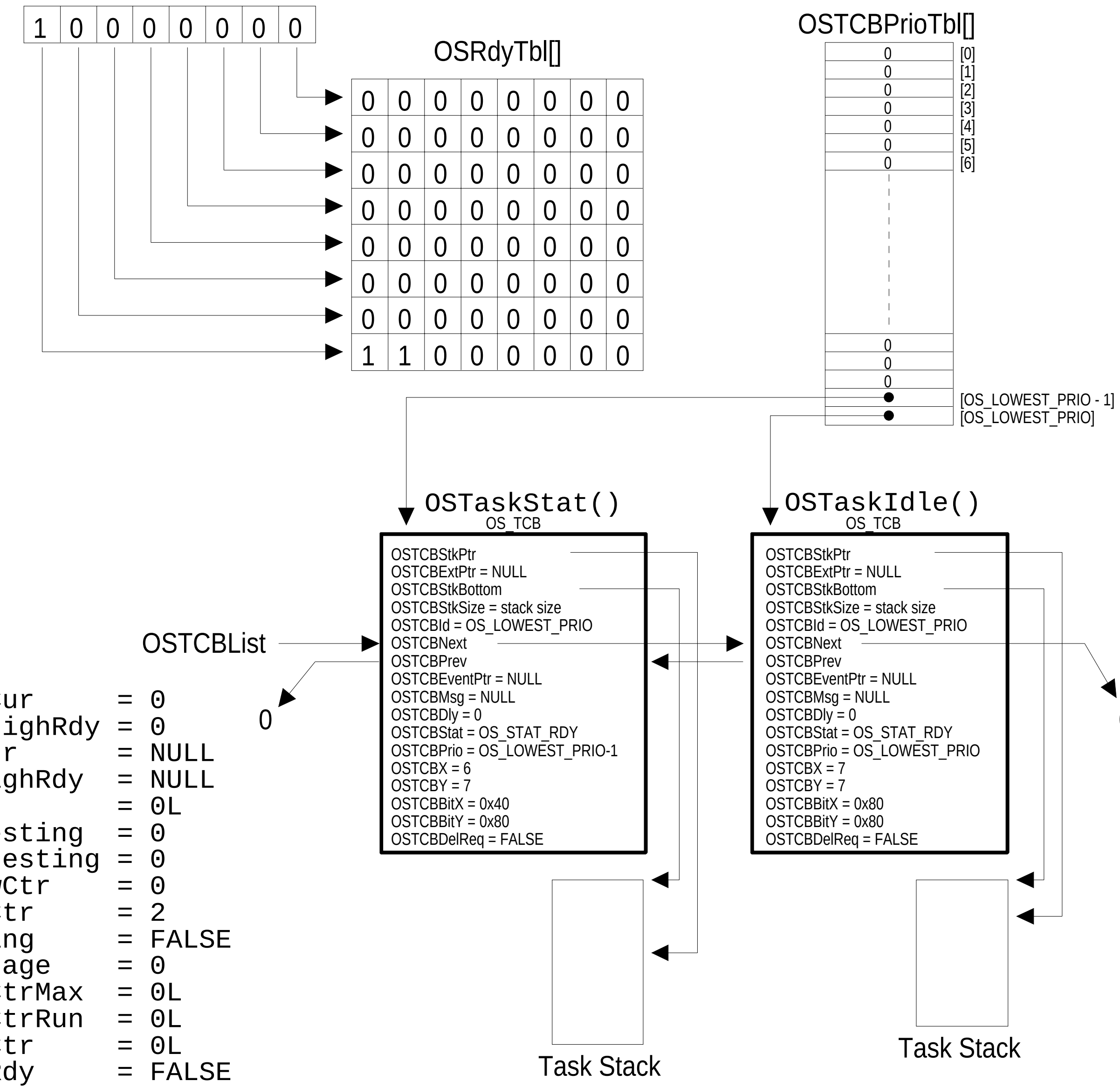
TCB list

Ready list

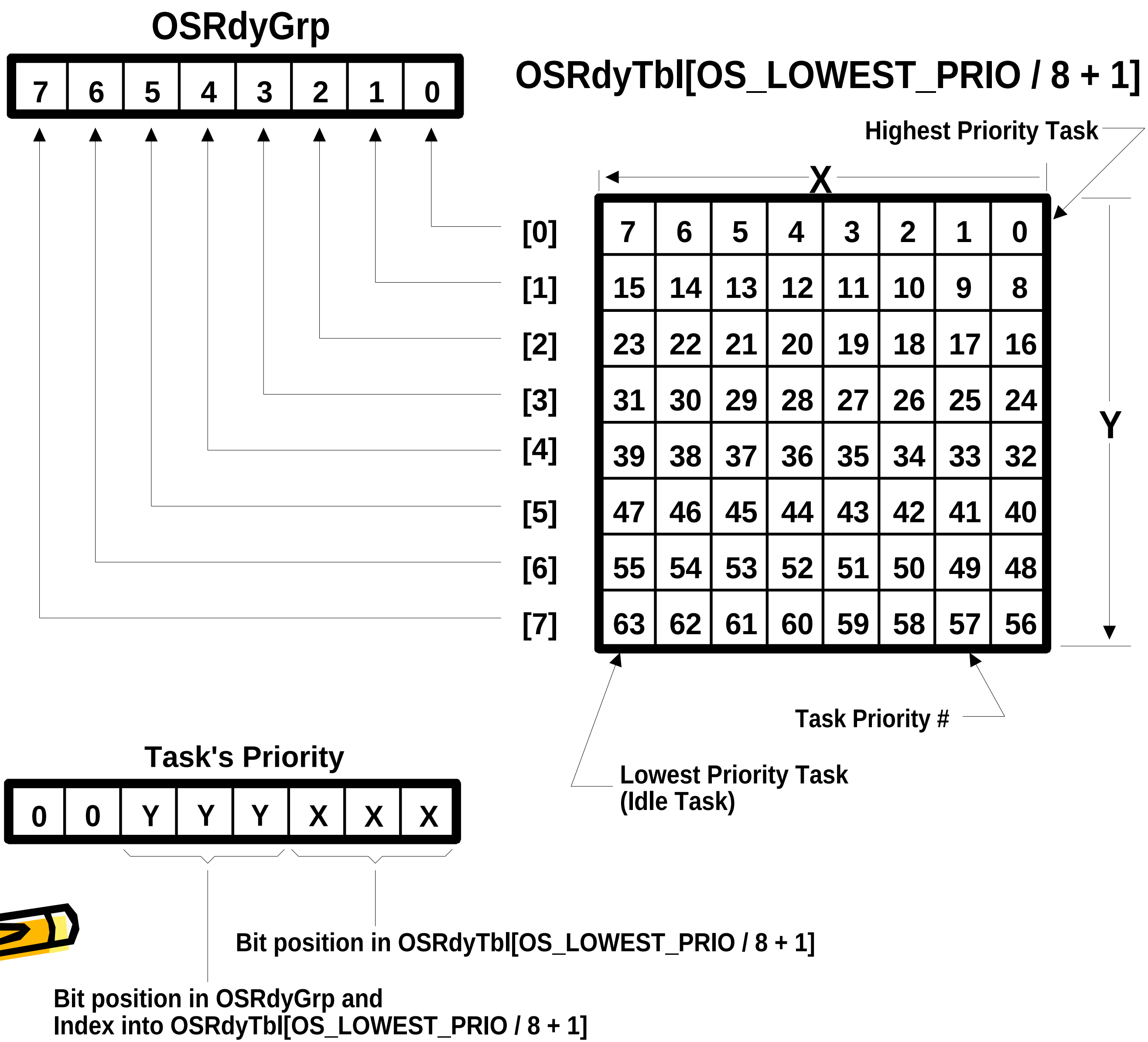
# Ready List

- Ready list is a special bitmap to reflect which task is currently in the ready state.
  - Each task is identified by its unique priority in the bitmap.
- A primary design consideration of the ready list is how to efficiently locate the highest-priority ready task.
  - The designer decides to trade some ROM space for an improved performance.
- If a linear list is adopted, it takes O(n) to locate the highest-priority ready task.
  - It takes O(log n) if a heap is adopted.
  - By the design of ready list of uC/OS-2, it takes only O(1).
    - Note that the space consumption is much more than other approaches.
    - It also depends on the bus width.

# Ready List




```
OSPrioCur      = 0
OSPrioHighRdy  = 0
OSTCBCur       = NULL
OSTCBHighRdy   = NULL
OSTime         = 0L
OSIntNesting   = 0
OSLockNesting  = 0
OSCtxSwCtr     = 0
OSTaskCtr      = 2
OSRunning      = FALSE
OSCPUUsage     = 0
OSIdleCtrMax   = 0L
OSIdleCtrRun   = 0L
OSIdleCtr      = 0L
OSStatRdy      = FALSE
```



OSRdyGrp
7 6 5 4 3 2 1 0
OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Highest Priority Task
X
Y
[0] 7 6 5 4 3 2 1 0
[1] 15 14 13 12 11 10 9 8
[2] 23 22 21 20 19 18 17 16
[3] 31 30 29 28 27 26 25 24
[4] 39 38 37 36 35 34 33 32
[5] 47 46 45 44 43 42 41 40
[6] 55 54 53 52 51 50 49 48
[7] 63 62 61 60 59 58 57 56
Task Priority #
Lowest Priority Task
(Idle Task)
Task's Priority
0 0 Y Y Y X X X
Bit position in OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Bit position in OSRdyGrp and
Index into OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]

## OSMapTbl

| Index | Bit mask (Binary) |
|---|---|
| 0 | 00000001 |
| 1 | 00000010 |
| 2 | 00000100 |
| 3 | 00001000 |
| 4 | 00010000 |
| 5 | 00100000 |
| 6 | 01000000 |
| 7 | 10000000 |

Bit 0 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[0]** is 1.
Bit 1 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[1]** is 1.
Bit 2 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[2]** is 1.
Bit 3 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[3]** is 1.
Bit 4 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[4]** is 1.
Bit 5 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[5]** is 1.
Bit 6 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[6]** is 1.
Bit 7 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[7]** is 1.

- Make a task ready to run:

```
OSRdyGrp            |= OSMapTbl[prio >> 3];
OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

- Remove a task from the ready list:

```
if ((OSRdyTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0)
    OSRdyGrp &= ~OSMapTbl[prio >> 3];
```

What does this code do?

# Coding style?

The author writes:

```
if ((OSRdyTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0)
    OSRdyGrp &= ~OSMapTbl[prio >> 3];
```

How about this:

```
char x,y,mask;

    x = prio & 0x07;
    y = prio >> 3;
    mask = ~(OSMapTbl[x]);             // a mask for bit clearing
    if((OSRdyTbl[x] &= mask) == 0)     // clear the task's bit
    {                       // the group bit should be cleared too
        mask = ~(OSMapTbl[y]);         // another bit mask…
        OSRdyGrp &= mask;              // clear the group bit
    }
```

# Coding Style?

```
mov     al,byte ptr [bp-17]
mov     ah,0
and     ax,7
lea     dx,word ptr [bp-8]
add     ax,dx
mov     bx,ax
mov     al,byte ptr ss:[bx]
not     al
mov     dl,byte ptr [bp-17]
mov     dh,0
sar     dx,3
lea     bx,word ptr [bp-16]
add     dx,bx
mov     bx,dx
and     byte ptr ss:[bx],al
mov     al,byte ptr ss:[bx]
or      al,al
jne     short @1@86
mov     al,byte ptr [bp-17]
mov     ah,0
sar     ax,3
lea     dx,word ptr [bp-8]
add     ax,dx
mov     bx,ax
mov     al,byte ptr ss:[bx]
not     al
and     byte ptr [bp-18],al
```

```
mov     al,byte ptr [bp-17]
and     al,7
mov     byte ptr [bp-19],al
mov     al,byte ptr [bp-17]
mov     ah,0
sar     ax,3
mov     byte ptr [bp-20],al
mov     al,byte ptr [bp-19]
mov     ah,0
lea     dx,word ptr [bp-8]
add     ax,dx
mov     bx,ax
mov     al,byte ptr ss:[bx]
not     al
mov     cl,al
mov     al,byte ptr [bp-19]
mov     ah,0
lea     dx,word ptr [bp-16]
add     ax,dx
mov     bx,ax
and     byte ptr ss:[bx],cl
mov     al,byte ptr ss:[bx]
or      al,al
jne     short @1@142
mov     al,byte ptr [bp-20]
mov     ah,0
lea     dx,word ptr [bp-8]
add     ax,dx
mov     bx,ax
mov     al,byte ptr ss:[bx]
not     al
mov     cl,al
```

```
INT8U  const  OSUnMapTbl[] = {
   0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0x00 to 0x0F   */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0x10 to 0x1F   */
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0x20 to 0x2F   */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0x30 to 0x3F   */
   6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0x40 to 0x4F   */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0x50 to 0x5F   */
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0x60 to 0x6F   */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0x70 to 0x7F   */
   7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0x80 to 0x8F   */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0x90 to 0x9F   */
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0xA0 to 0xAF   */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0xB0 to 0xBF   */
   6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0xC0 to 0xCF   */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0xD0 to 0xDF   */
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,      /* 0xE0 to 0xEF   */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0       /* 0xF0 to 0xFF   */
};
```

- Finding the highest-priority task ready to run:

```
y    = OSUnMapTbl[OSRdyGrp];
x    = OSUnMapTbl[OSRdyTbl[y]];
prio = (y << 3) + x;
```

This matrix is used to locate the first LSB which is ‘1’, by given a value.

For example, if 00110010 is given, then ‘1’ is returned.

# Task Scheduling

- The scheduler always schedules the highest-priority ready task to run .

- Task-level scheduling and ISR-level scheduling are done by OS_Sched() and OSIntExit(), respectively.
  - The difference is the saving/restoration of PSW (or CPU flags).

- uC/OS-2 scheduling time is a predictable amount of time, i.e., a constant time.
  - For example, the design of the ready list intends to achieve this objective.

```
void OSSched (void)
{
    INT8U y;
    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {                          (1)
        y            = OSUnMapTbl[OSRdyGrp];                            (2)
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);    (2)
        if (OSPrioHighRdy != OSPrioCur) {                               (3)
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];                 (4)
            OSCtxSwCtr++;                                               (5)
            OS_TASK_SW();                                               (6)
        }
    }
    OS_EXIT_CRITICAL();
}
```

(1) Rescheduling will not be done if the scheduler is locked or an ISR is currently serviced (why?).
(2) Find the highest-priority ready task.
(3) If it is not the current task, then
(4) ~(6) Perform a context-switch.

# Task Scheduling

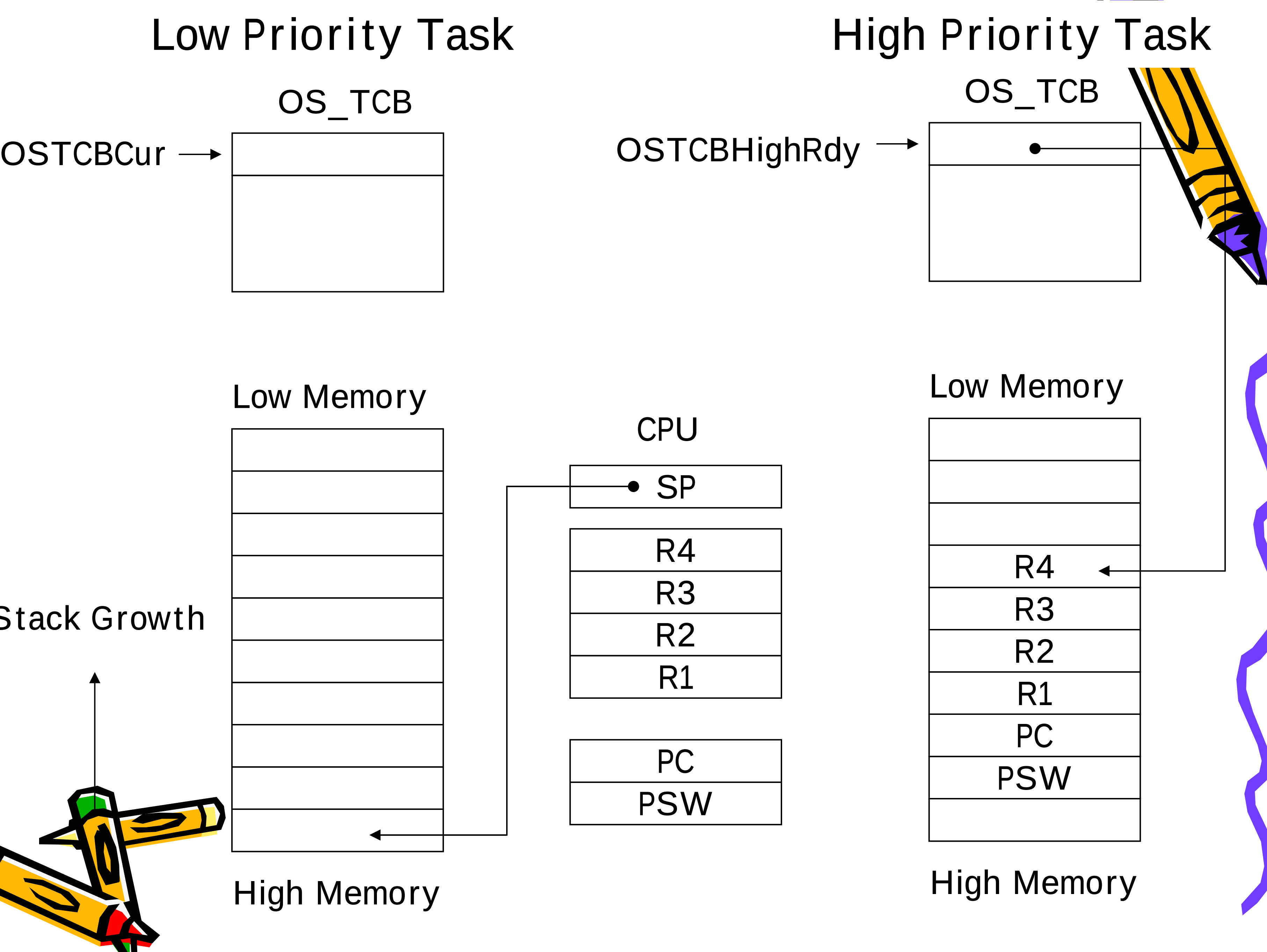
- A context switch must save all CPU registers and PSW of the preempted task onto its stack, and then restore the CPU registers and PSW of the highest-priority ready task from its stack.

- Task-level scheduling will simulate that as if preemption/scheduling is done in an ISR.
  - OS_TASK_SW() will trigger a software interrupt. (why?)
  - The interrupt is directed to the context switch handler OSCtxSw(), which is installed when uC/OS-2 is initialized.

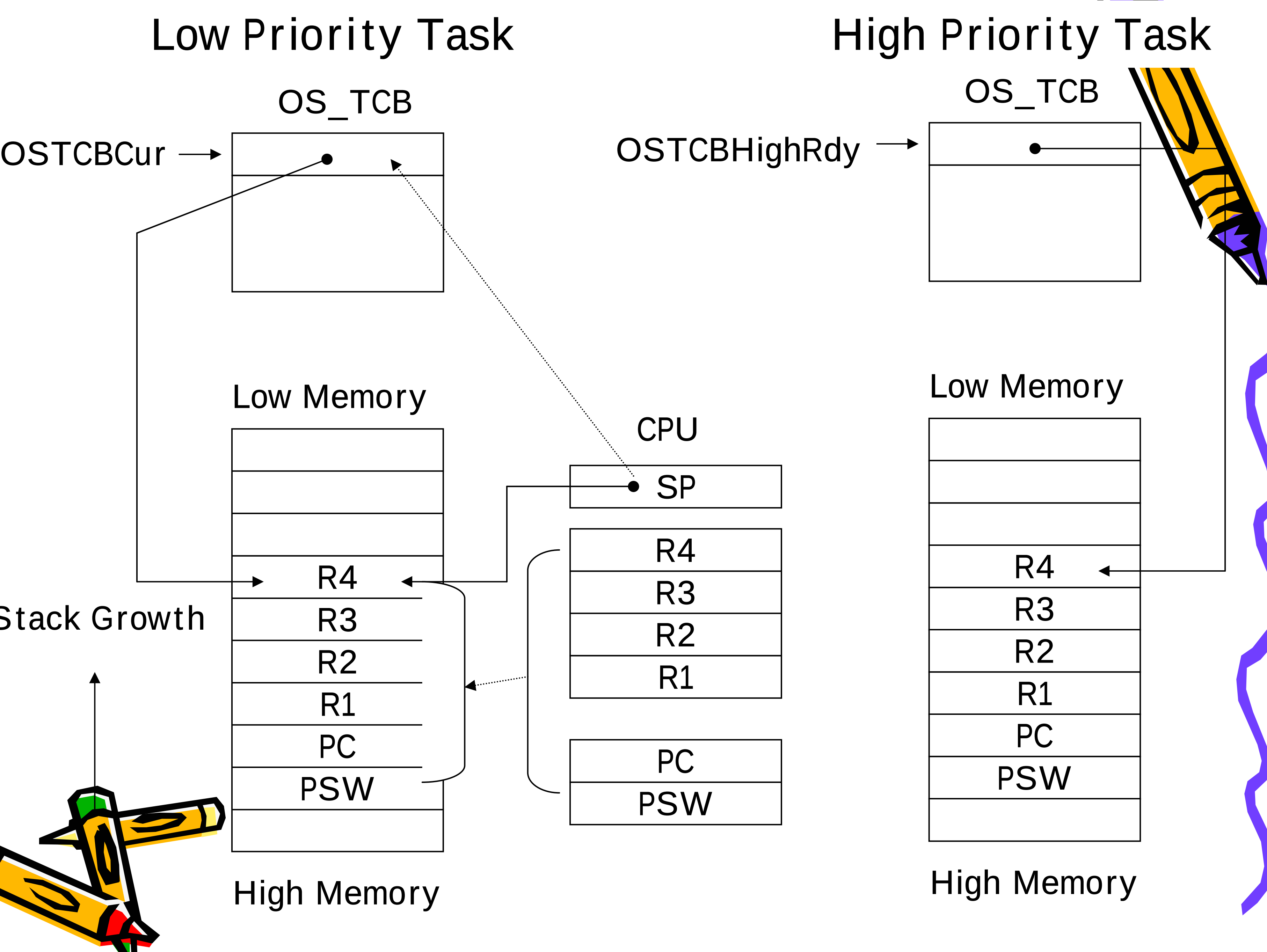
- Interrupts are disabled during the finding of the highest-priority ready task to prevent another ISR's from making some tasks ready.

# Task Level Context Switch

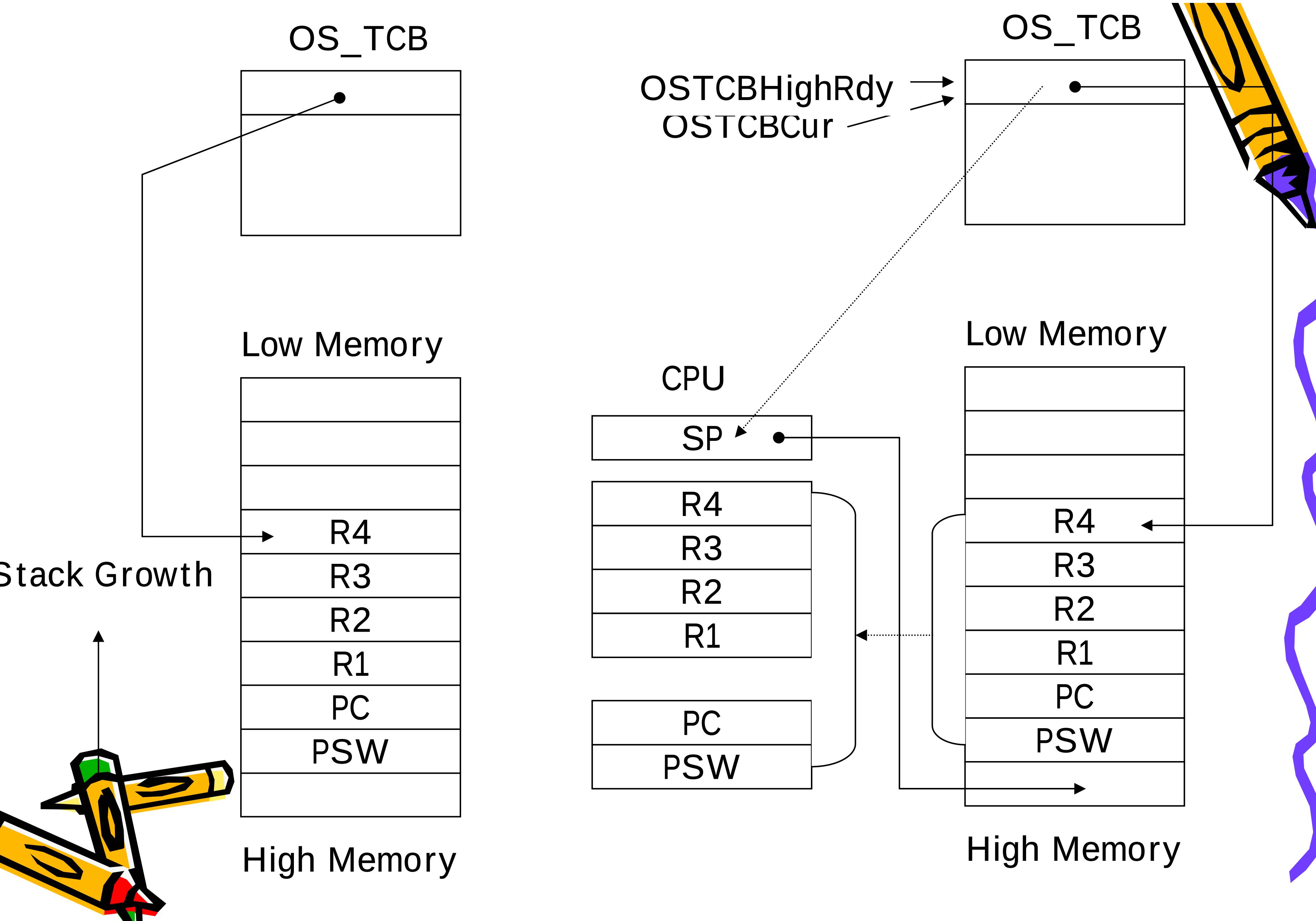
- By default, context switches are handled at interrupt-level, therefore task-level scheduling will invoke a software interrupt to simulate that.
  - Hardware dependent, porting must be done.



Low Priority Task
High Priority Task
OS_TCB
OSTCBCur
OS_TCB
OSTCBHighRdy
Low Memory
CPU
SP
R4
R3
R2
R1
PC
PSW
Stack Growth
High Memory
Low Memory
R4
R3
R2
R1
PC
PSW
High Memory


Low Priority Task
High Priority Task
OS_TCB
OSTCBCur
OS_TCB
OSTCBHighRdy
Low Memory
CPU
SP
Low Memory
R4
R3
R2
R1
PC
PSW
Stack Growth
High Memory
R4
R3
R2
R1
PC
PSW
R4
R3
R2
R1
PC
PSW
High Memory

Low Priority Task High Priority Task



# Locking and Unlocking the Scheduler

- OSSchedLock() prevent high-priority ready tasks from being scheduled to run while interrupts are still recognized.
- OSSchedLock() and OSSchedUnlock() are used in pairs.
- OSLockNesting keeps track of the number of OSSchedLock() has been called. (how? why?)
- After calling OSSchedLock(), you must not call kernel services which might cause context switch, such as OSFlagPend(), OSMboxPend(), OSMutexPend(), OSQPend(), OSSemPend(), OSTaskSuspend(), OSTimeDly, OSTimeDlyHMSM() until OSLockNesting == 0. Or the system will be locked up.
- Sometimes we disable scheduling but with interrupts are still recognized because we hope to avoid lengthy interrupt latencies without introducing race conditions.

# OSSchedLock()

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


    if (OSRunning == TRUE) {      /* Make sure multitasking is running            */
        OS_ENTER_CRITICAL();
        if (OSLockNesting < 255) {/* Prevent OSLockNesting from wrapping back to 0*/
            OSLockNesting++;      /* Increment lock nesting level                 */
        }
        OS_EXIT_CRITICAL();
    }
}
```

# OSSchedUnlock()

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


    if (OSRunning == TRUE) {         /* Make sure multitasking is running    */
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) {     /* Do not decrement if already 0        */
            OSLockNesting--;         /* Decrement lock nesting level         */
            if ((OSLockNesting == 0) &&
              (OSIntNesting == 0)) { /* See if sched. enabled and not an ISR */
                OS_EXIT_CRITICAL();
                OS_Sched();          /* See if a HPT is ready                */
            } else {
                OS_EXIT_CRITICAL();
            }
        } else {
            OS_EXIT_CRITICAL();
        }
    }
}
```

# The Idle Task

- The idle task is always the lowest-priority task and can not be deleted or suspended by user-tasks.
- To conserve power dissipation, you can issue a HALT instruction in the idle task.
- Do not call delay, suspend services in OSTaskIdleHook()!!

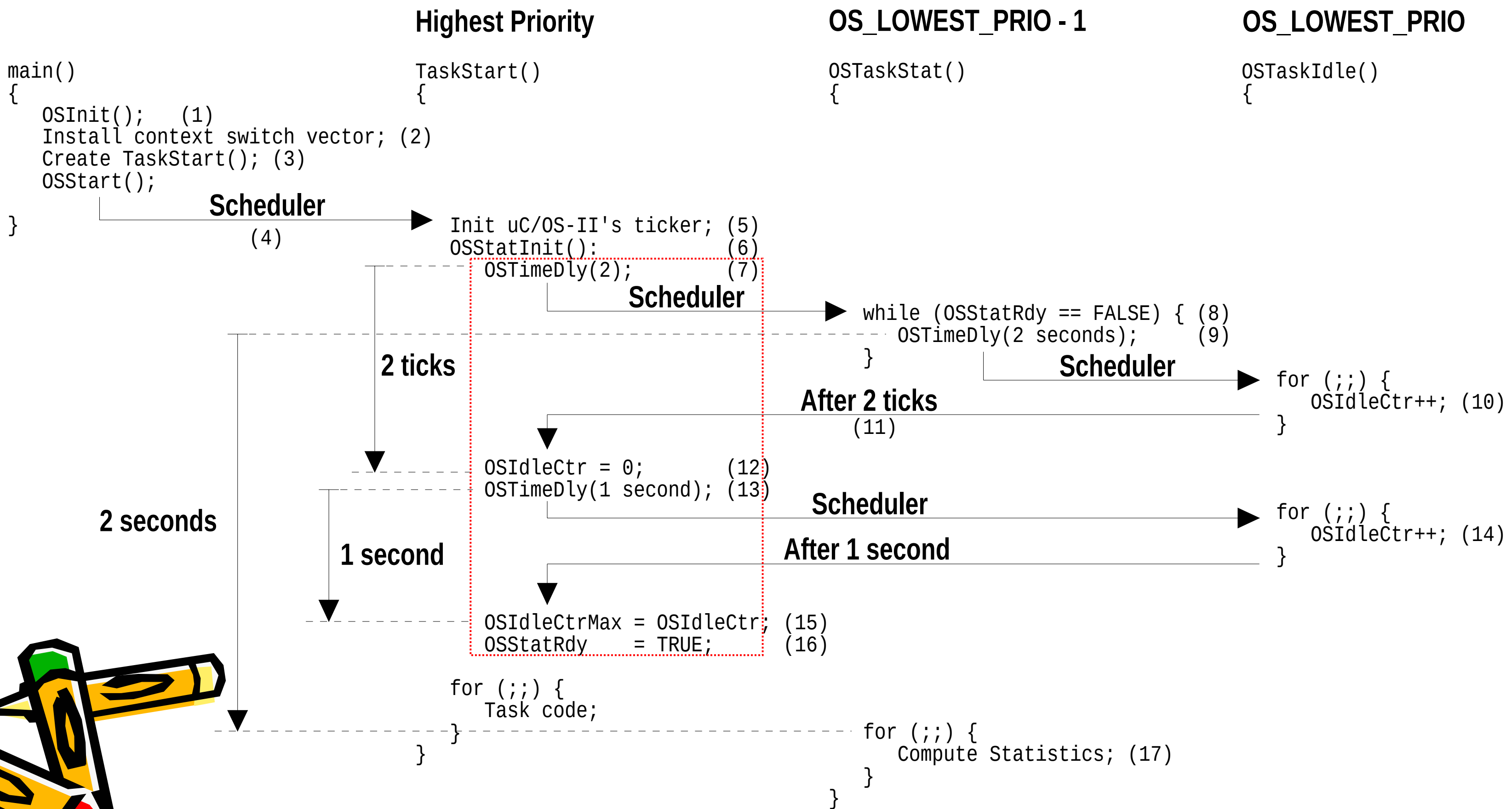
```
void  OS_TaskIdle (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


    pdata = pdata;
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtr++;
        OS_EXIT_CRITICAL();
        OSTaskIdleHook();
    }
}
```

# The Statistics Task

- It is created by uC/OS-2, and it executes every second to compute the percentage of CPU usage.
- OSStatInit() must be called before OSStart() is called.
- With a OS_LOWEST_PRIO – 1 priority.

```
void main (void)
{
    OSInit();                /* Initialize uC/OS-II                         (1)*/
    /* Install uC/OS-II's context switch vector                              */
    /* Create your startup task (for sake of discussion, TaskStart())  (2)*/
    OSStart();               /* Start multitasking                          (3)*/
}
void TaskStart (void *pdata)
{
    /* Install and initialize µC/OS-II’s ticker                         (4)*/
    OSStatInit();            /* Initialize statistics task                  (5)*/
    /* Create your application task(s)                                       */
    for (;;) {
        /* Code for TaskStart() goes here!                                   */
    }
}
```

# The Statistics Task



# The Statistics Task

(7) **TaskStart: delay 2 ticks**→ transfer CPU to the stat task to do some initializations.

(9) **OS_TaskStat: delay 2 seconds**→ yield the CPU to the task TaskStart and the idle task.

(13) **TaskStart: delay 1 second**→ let the idle task to count OSIdleCtr for 1 second. (note that the stat task is still not delayed).

(15) **TaskStart**: on the timer expiration in (13), now OSIdleCtr contains the value can be reached in 1 second.

- Notes:
  - Since OSStatinit() assume that the idle task will count the OSOdleCtr at full CPU speed, you must not install an idle hook before calling OSStatInit().
  - After the stat task is initialized, it is OK to install a CPU idle hook and perform some power-conserving operations, since the idle task entirely consumes the CPU power just for the purpose of being idle.

# The Statistics Task

- By calling OSStatInit(), we've got how high the idle counter can reach in 1 second (OSIdleCtrMax).

- The percentage of CPU usage can be calculated by the actual idle counter and the OSIdleCtrMax.

$$OSCPUUsage_{(\%)} = 100 \times \left( 1 - \boxed{\frac{OSIdleCtr}{OSIdleCtrMax}} \right)$$

This term is always 0 under integer operation

$$OSCPUUsage_{(\%)} = \left( 100 - \frac{\boxed{100 \times OSIdleCtr}}{OSIdleCtrMax} \right)$$

This term might overflow under fast processors! (42,949,672)

$$OSCPUUsage_{(\%)} = \left( 100 - \frac{OSIdleCtr}{\left( \frac{OSIdleCtrMax}{100} \right)} \right)$$

# The Statistics Task

```
#if OS_TASK_STAT_EN > 0
void  OS_TaskStat (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT32U     run;
    INT32U     max;
    INT8S      usage;


    pdata = pdata;
    while (OSStatRdy == FALSE) {
        OSTimeDly(2 * OS_TICKS_PER_SEC);
    }
    max = OSIdleCtrMax / 100L;
```

```
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtrRun = OSIdleCtr;
        run          = OSIdleCtr;
        OSIdleCtr    = 0L;
        OS_EXIT_CRITICAL();
        if (max > 0L) {
            usage = (INT8S)(100L - run / max);
            if (usage >= 0) {
                OSCPUUsage = usage;
            } else {
                OSCPUUsage = 0;
            }
        } else {
            OSCPUUsage = 0;
            max        = OSIdleCtrMax / 100L;
        }
        OSTaskStatHook();
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

# Interrupts under uC/OS-2

- uC/OS-2 requires an ISR written in assembly, if your compiler does not support in-line assembly.

```
Your ISR:
    Save all CPU registers;                                          (1)
    Call OSIntEnter() or, increment OSIntNesting directly;           (2)
    If(OSIntNesting == 1)                                            (3)
        OSTCBCur->OSTCBStkPtr = SP;                                  (4)
    Clear the interrupting device;                                   (5)
    Re-enable interrupts (optional);                                 (6)
    Execute user code to service ISR;                                (7)
    Call OSIntExit();                                                (8)
    Restore all CPU registers;                                       (9)
    Execute a return from interrupt instruction;                     (10)
```

# Interrupts under uC/OS-2

(1) In an ISR, uC/OS-2 requires that all CPU registers are saved onto the interrupted task.

- For processors like Motorola 68030_, a different stack is used for ISR.
- For such case, the stack pointer of the interrupted task can be obtained from OSTCBCur (offset 0).

(2) Increase the interrupt-nesting counter counter.

(4) If it is the first interrupt-nesting level, we immediately save the stack pointer to OSTCBCur.

- We do this because a context-switch might occur.

# Interrupts under uC/OS-2

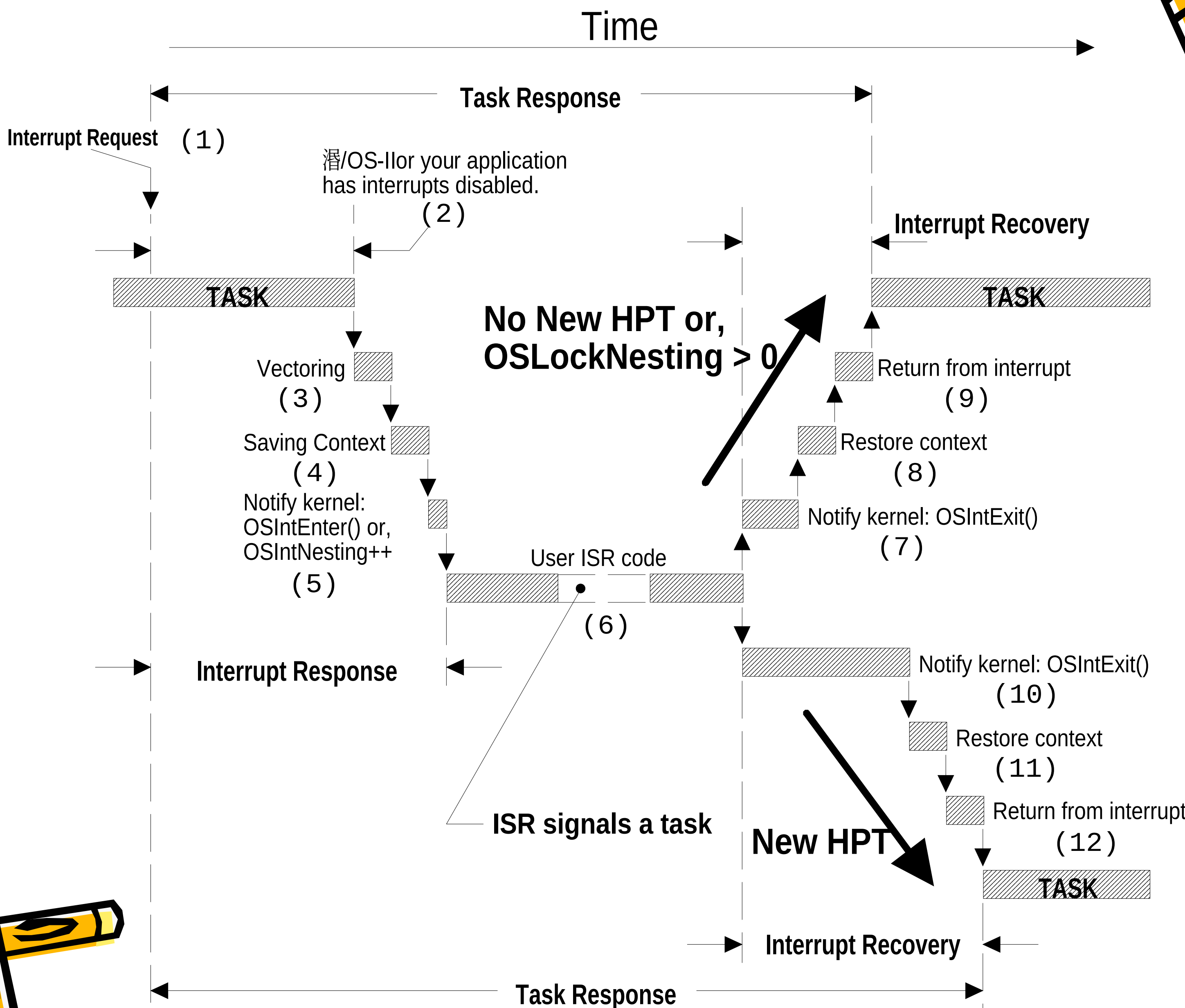
(8) Call OSIntExit(), which checks if we are in the inner-level of nested interrupts. If not, the scheduler is called.

- A potential context-switch might occur.
- Interrupt-nesting counter is decremented.

(9) On the return to this point, there might be several high-priority tasks ran by the CPU.

- Since uC/OS-2 is a preemptive kernel.

(10) The CPU registers are restored from the stack and the control is returned to the interrupted instruction.


Time
Task Response
Interrupt Request
(1)
滑/OS-IIor your application
has interrupts disabled.
(2)
Interrupt Recovery
TASK
No New HPT or,
OSLockNesting > 0
TASK
Vectoring
(3)
Return from interrupt
(9)
Saving Context
(4)
Restore context
(8)
Notify kernel:
OSIntEnter() or,
OSIntNesting++
(5)
Notify kernel: OSIntExit()
(7)
User ISR code
(6)
Interrupt Response
Notify kernel: OSIntExit()
(10)
Restore context
(11)
Return from interrupt
(12)
ISR signals a task
New HPT
TASK
Interrupt Recovery
Task Response

# Interrupts under uC/OS-2

```
void OSIntExit (void)
{
    OS_ENTER_CRITICAL();
    if ((--OSIntNesting | OSLockNesting) == 0) {
        OSIntExitY    = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((OSIntExitY << 3) +
                        OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OSIntCtxSw();
        }
    }
    OS_EXIT_CRITICAL();
}
```

If scheduler is not locked and no interrupt nesting

If there is another high-priority task ready

A context switch is performed.

Note that OSIntCtxSw() is called instead of calling OS_TASK_SW() because the ISR already saves the CPU registers onto the stack.

```
void OSIntEnter (void)
{
    OS_ENTER_CRITICAL();
    OSIntNesting++;
    OS_EXIT_CRITICAL();
}
```

# Clock Tick

- A time source is needed to keep track of time delays and timeouts.
- You must enable ticker interrupts after multitasking is started.
  - In the TaskStart() task in the examples.
  - Do not do this before OSStart().
- Clock ticks are serviced by calling OSTimeTick() from a tick ISR.
- Clock tick ISR is always a port (of uC/OS-2) of a CPU. Since we have to access CPU registers in the tick ISR.

# Clock Tick

```
void OSTickISR(void)
{
    Save processor registers;
    Call OSIntEnter() or increment OSIntNesting;
    If(OSIntNesting == 1)
            OSTCBCur->OSTCBStkPtr = SP;
    Call OSTimeTick();
    Clear interrupting device;
    Re-enable interrupts (optional);
    Call OSIntExit();
    Restore processor registers;
    Execute a return from interrupt instruction;
}
```

```
void  OSTimeTick (void)
{
    OS_TCB    *ptcb;


    OSTimeTickHook();

    if (OSRunning == TRUE) {
      ptcb = OSTCBList;
      while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {
        OS_ENTER_CRITICAL();
        if (ptcb->OSTCBDly != 0) {
          if (--ptcb->OSTCBDly == 0) {
            if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == OS_STAT_RDY) {
              OSRdyGrp               |= ptcb->OSTCBBitY;
              OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
            } else {
              ptcb->OSTCBDly = 1;
            }
          }
        }
        ptcb = ptcb->OSTCBNext;
        OS_EXIT_CRITICAL();
      }
    }
}
```

For all TCB's

Decrement delay-counter if needed

If the delay-counter reaches zero, make the task ready. Or the task remains waiting.

# Clock Tick

- OSTimeTick() is a hardware-independent routine to service the tick ISR.

- A delta-list is more efficient on the decrementing of .OSTCBDly.
  - Constant time to determine if a task should be made ready.
  - Linear time to put a task in the list.
  - Compare it with the approach of uC/OS-2?

# Clock Tick

- You can also move the bunch of code in the tick ISR to a user task:

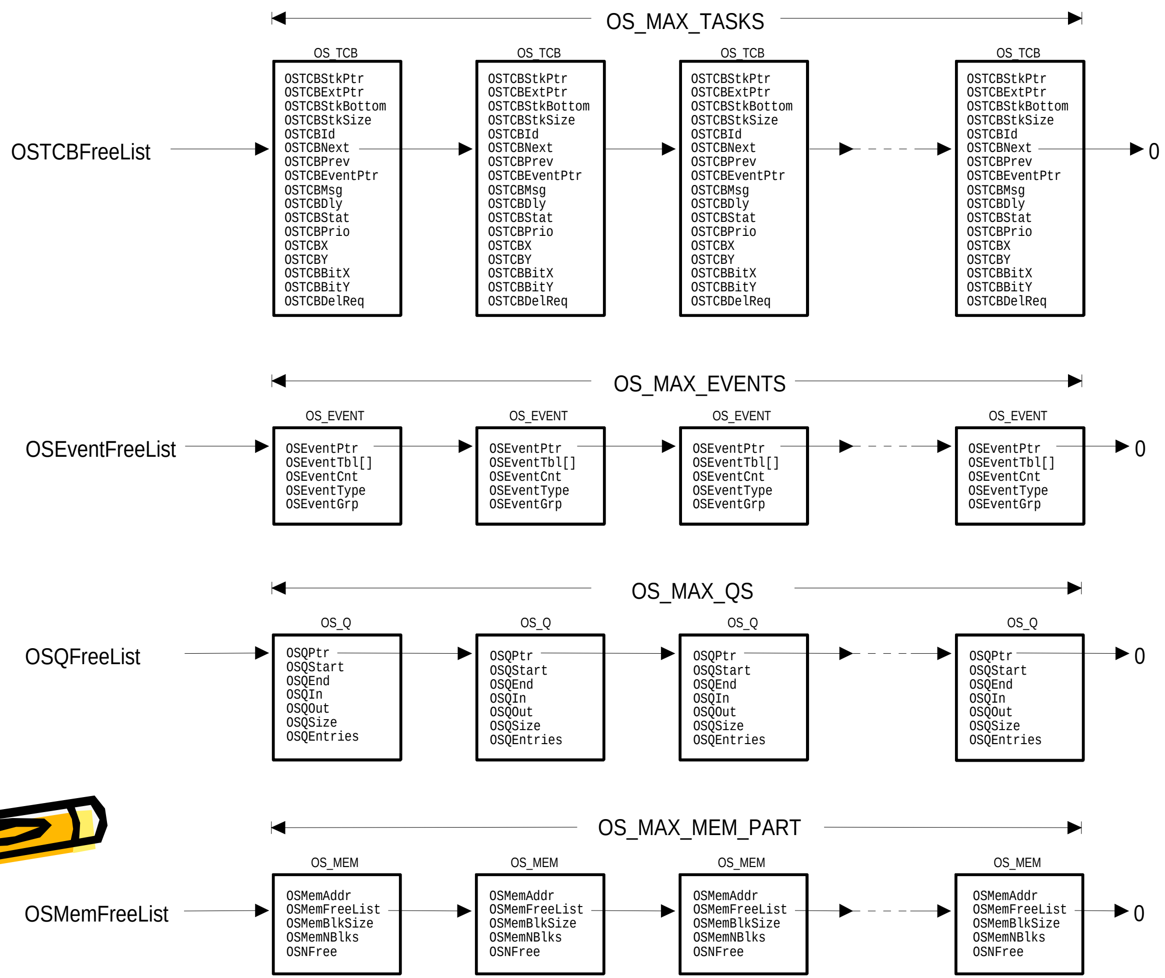
```
void OSTickISR(void)
{
    Save processor registers;
    Call OSIntEnter() or increment OSIntNesting;
    If(OSIntNesting == 1)
    OSTCBCur->OSTCBStkPtr = SP;

    Post a 'dummy' message (e.g. (void *)1)
      to the tick mailbox;

    Call OSIntExit();
    Restore processor registers;
    Execute a return from interrupt instruction;
}
```

Post a message

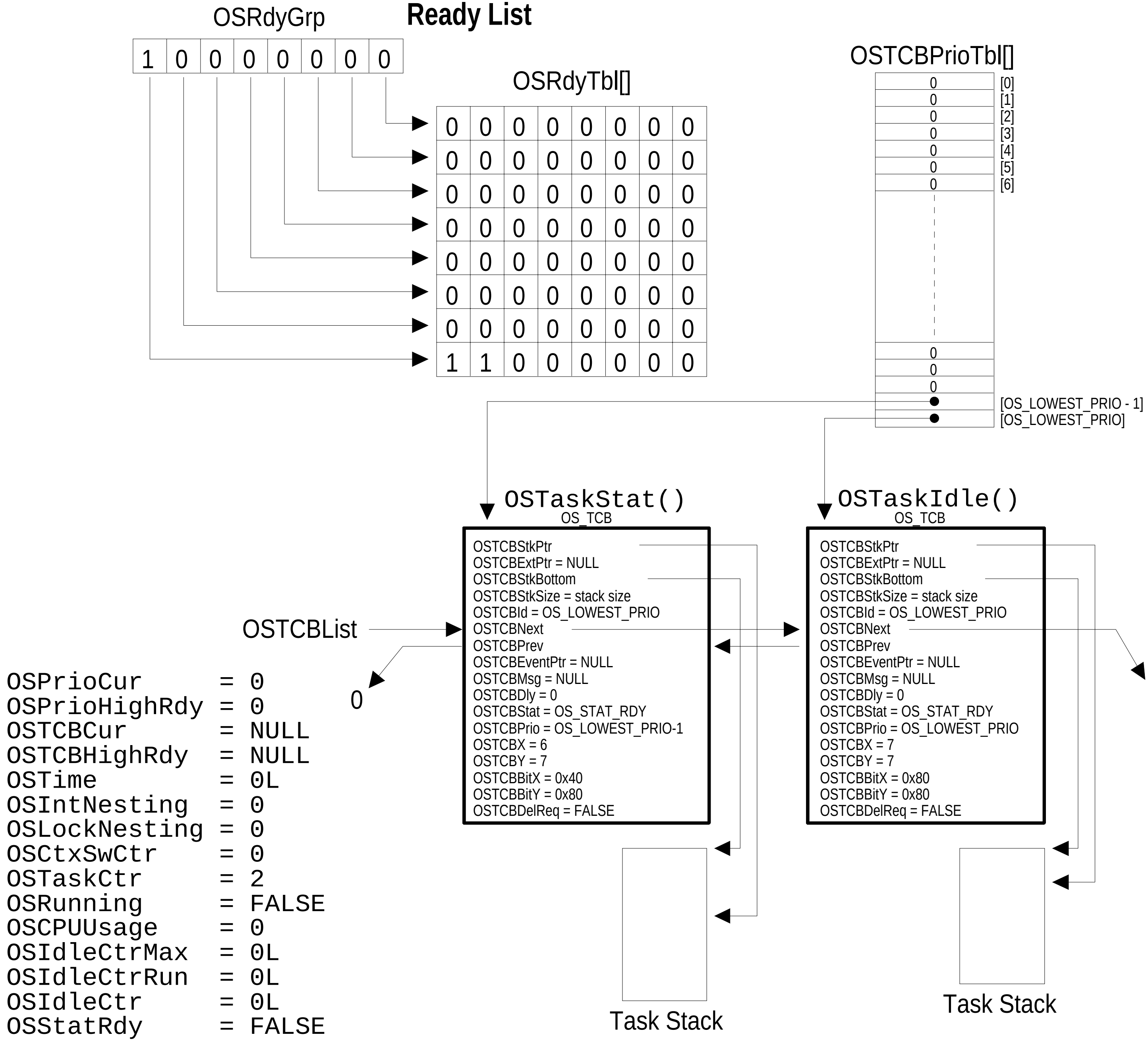
```
void TickTask (void *pdata)
{
    pdata = pdata;
    for (;;) {
        OSMboxPend(...);
        OSTimeTick();
        OS_Sched();
    }
}
```

Do the rest of the work

# uC/OS-2 Initialization



# Ready List



OSRdyGrp
1 0 0 0 0 0 0 0
OSRdyTbl[]
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
OSTCBPrioTbl[]
[0] 0
[1] 0
[2] 0
[3] 0
[4] 0
[5] 0
[6] 0
0
0
0
[OS_LOWEST_PRIO - 1]
[OS_LOWEST_PRIO]
OSTaskStat()
OS_TCB
OSTCBStkPtr
OSTCBExtPtr = NULL
OSTCBStkBottom
OSTCBStkSize = stack size
OSTCBId = OS_LOWEST_PRIO
OSTCBNext
OSTCBPrev
OSTCBEventPtr = NULL
OSTCBMsg = NULL
OSTCBDly = 0
OSTCBStat = OS_STAT_RDY
OSTCBPrio = OS_LOWEST_PRIO-1
OSTCBX = 6
OSTCBY = 7
OSTCBBitX = 0x40
OSTCBBitY = 0x80
OSTCBDelReq = FALSE
OSTaskIdle()
OS_TCB
OSTCBStkPtr
OSTCBExtPtr = NULL
OSTCBStkBottom
OSTCBStkSize = stack size
OSTCBId = OS_LOWEST_PRIO
OSTCBNext
OSTCBPrev
OSTCBEventPtr = NULL
OSTCBMsg = NULL
OSTCBDly = 0
OSTCBStat = OS_STAT_RDY
OSTCBPrio = OS_LOWEST_PRIO
OSTCBX = 7
OSTCBY = 7
OSTCBBitX = 0x80
OSTCBBitY = 0x80
OSTCBDelReq = FALSE
OSTCBList
0
0
Task Stack
Task Stack
OSPrioCur = 0
OSPrioHighRdy = 0
OSTCBCur = NULL
OSTCBHighRdy = NULL
OSTime = 0L
OSIntNesting = 0
OSLockNesting = 0
OSCtxSwCtr = 0
OSTaskCtr = 2
OSRunning = FALSE
OSCPUUsage = 0
OSIdleCtrMax = 0L
OSIdleCtrRun = 0L
OSIdleCtr = 0L
OSStatRdy = FALSE

# Starting uC/OS-2

- OSInit() initializes the data structures for uC/OS-2 and creates OS_TaskIdle().

- OSStart() pops the CPU registers of the highest-priority ready task and then executes a return from interrupt instruction.
  - It never returns to the caller of OSStart() (i.e., main()).

# Starting uC/OS-2

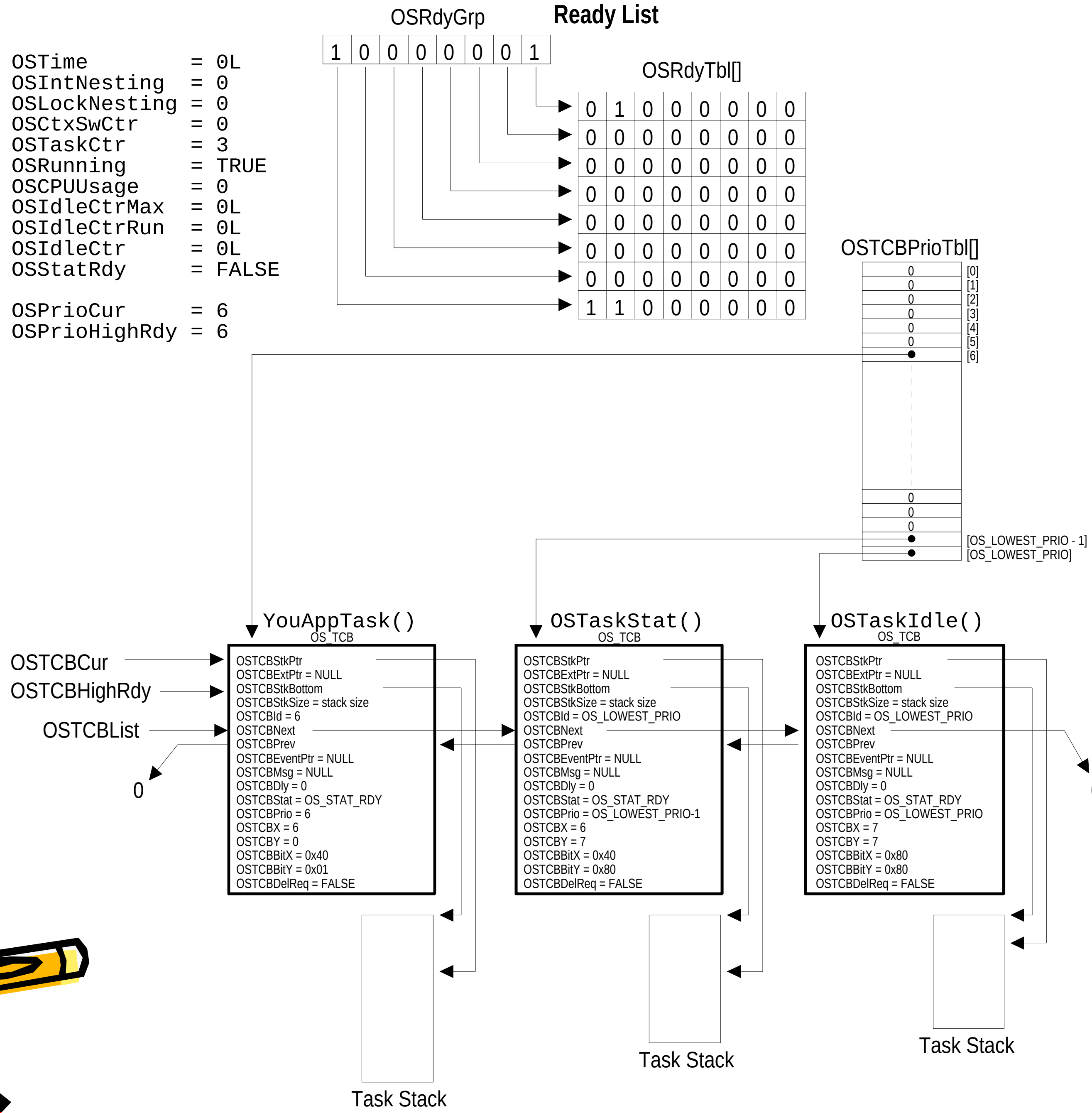
```
void main (void)
{
    OSInit();          /* Initialize uC/OS-II                      */
    .
    Create at least 1 task using either OSTaskCreate() or OSTaskCreateExt();
    .
    OSStart();         /* Start multitasking!  OSStart() will not return */
}
```

```
void OSStart (void)
{
    INT8U y;
    INT8U x;
    if (OSRunning == FALSE) {
        y            = OSUnMapTbl[OSRdyGrp];
        x            = OSUnMapTbl[OSRdyTbl[y]];
        OSPrioHighRdy = (INT8U)((y << 3) + x);
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
        OSTCBCur      = OSTCBHighRdy;
        OSStartHighRdy();
    }
}
```

Start the highest-priority ready task

# Ready List

```
OSTime         = 0L
OSIntNesting   = 0
OSLockNesting  = 0
OSCtxSwCtr     = 0
OSTaskCtr      = 3
OSRunning      = TRUE
OSCPUUsage     = 0
OSIdleCtrMax   = 0L
OSIdleCtrRun   = 0L
OSIdleCtr      = 0L
OSStatRdy      = FALSE

OSPrioCur      = 6
OSPrioHighRdy  = 6
```



# Summary

- In this chapter, you should learn that:
  - What a task is, how uC/OS-2 manages a task, and related data structures.
  - How the scheduler works, and the detailed operations done in context switches.
  - The responsibility of the idle task and the statistics task and how they works.
  - How interrupts are serviced in uC/OS-2.
  - The initialization and starting of uC/OS-2.