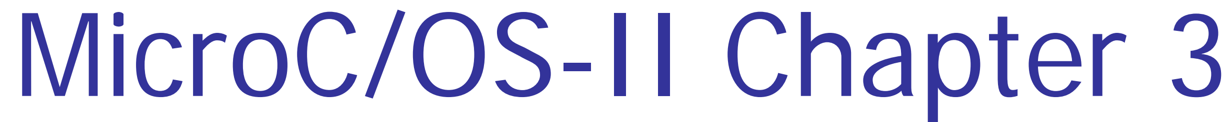


中興資科所 盧慶達
學號:79256022
指導教授 張軒彬

Chapter3 Kernel Structure

3.00 Critical Sections OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()
3.01 Tasks
3.02 Task States
3.03 Task Control Blocks(OS_TCB)
3.04 Ready List
3.05 Task Scheduling
3.06 Task Level Context Switch OS_TASK_SW()
3.07 Lock and Unlocking the Scheduler
3.08 Idle Task
3.09 Statistics Task
3.10 Interrupts Under μC/OSⅡ
3.11 Clock Tick
3.12 μC/OSⅡ Initialization
3.13 Starting μC/OSⅡ
3.14 Obtain the Current μC/OSⅡ Version

3.0 Critical Section

μC/OSⅡ use two macro(find in OS_CPU.H) to disable and enable interrupts:

OS_ENTER_CRITICAL();

/* μC/OSⅡ critical section*/

OS_EXIT_CRITICAL();

Note:

if you disable interrupt before calling a service(pend calls) such as OSTimeDly(),your application will crash.

Because you must awaked task by tick interrupt.

3.0 Critical Section

The two macros can be implemented using three different methods.

The method used is selected by the #define constant

OS_CRITICAL_METHOD.(define in OS_CPU.H)

3.0 Critical Section

OS_CRITICAL_METHOD==1

To invoke the processor instruction to disabled interrupt.

Note:

If you call a μC/OSⅡ function with interrupt disabled ,on return from a μC/OSⅡ service ,interrupt are enabled.

If you had disabled interrupt prior to calling μC/OSⅡ function ,you might want them to be disabled on return from the μC/OSⅡ function.

3.0 Critical Section

OS_CRITICAL_METHOD==2

Save the interrupt status onto the stack and then disable interrupt.

If you call a μC/OSⅡ service with interrupt disabled either Interrrupt disabled or enabled,the status is preserved across the call.

```
#define OS_ENTER_CRITICAL()
            asm(“”PUSH   PSW”)
            asm(“”DI”)
#define OS_EXITR_CRITICAL()
            asm(“”POP     PSW”)
```

3.0 Critical Section

OS_CRITICAL_METHOD==3

Some cmopliers provide you with extension that allow you to obtain the current value of the processor status word(PSW) and save it into a local variable declaried with a C function.

```
void Some_uCOS_II_Service(arguments)
{
OS_SPU_SR cpu_sr;
cpu_sr=get_processor_psw();
disabled_interrupts();
/*critical section of code*/
set_processor_psw(cpu_sr);
}
```

3.01 Tasks

A task is typically an infinite loop function.

```
Void Your (void *pdata)
{
for(;;){
/*USER CODE*/
Call one of μC/OSII'sservice:
/*USER CODE*/
    }
}
```

Task can delete itself by call OSTaskDel(OS_PRIO_SELF)

OS_PRIO_SELF is task's priority.

3.01 Tasks

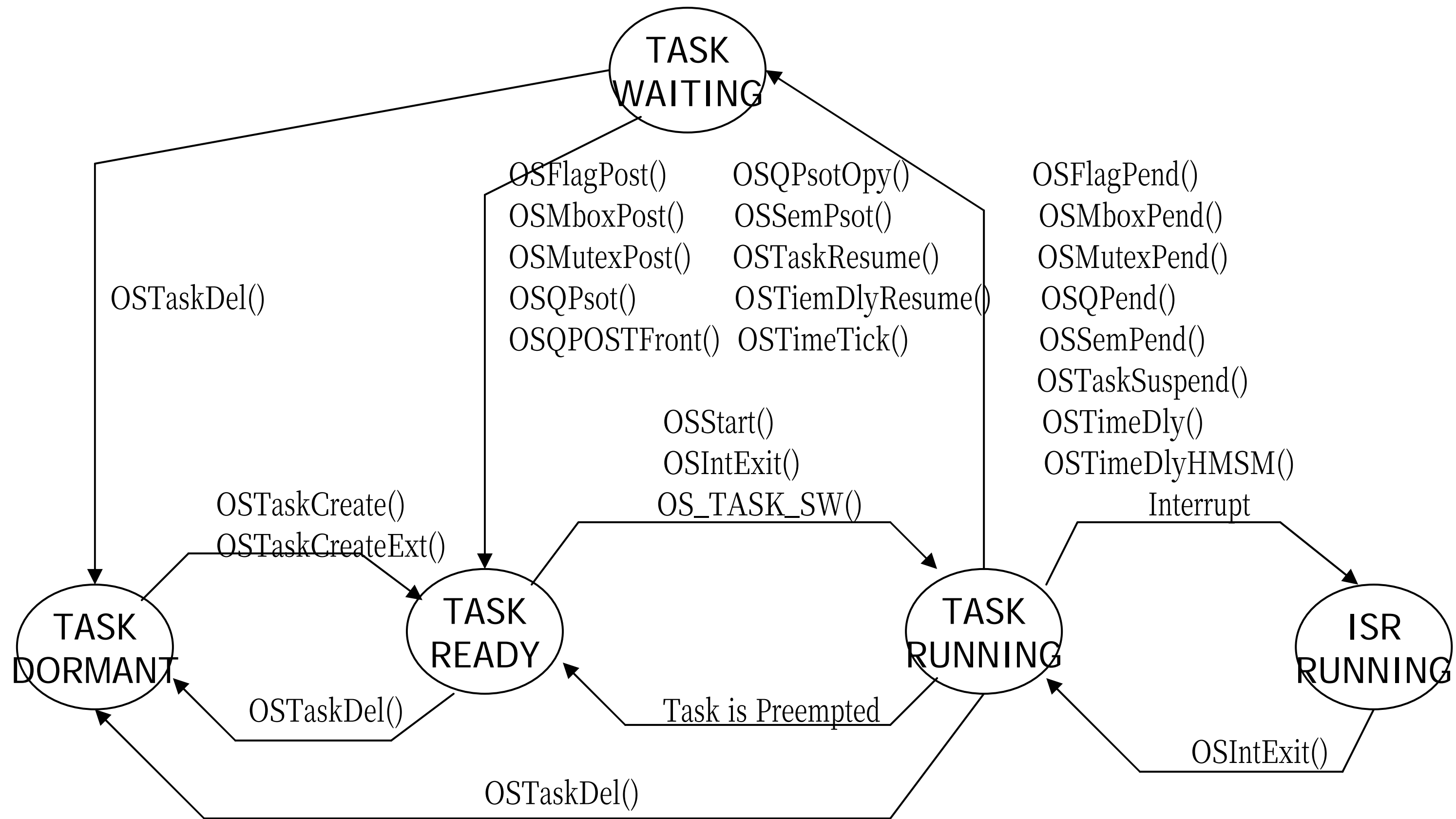
1.actually you only have up to 56 tasks for use.

2.you can change OS_LOWEST_PRIO in OS_CFG.H make your tasks up to 62 tasks.

3. μC/OSⅡ can manage your task,you must by passing its address along with other argument to OSTaskCreate() or OSTaskCreateExt().

4.in current μC/OSⅡ the task priority number also serves as task Identifier.

3.02 Task States



3.02 Task States

1.Call OSTaskCreate()OSTaskCreateExt() to tell μC/OSⅡ task's starting address ,task's priority,stack space used.

2.Multitasking is started by calling OSStart().

OSStart() must only called once during startup and start the high priority that has been created during your initialization code.

3.03 Task Control Block

1.When a task is created it is assigned a task control block,OS_TCB
2.OS_TCB is a data structure that is used to by μC/OSⅡ to maintain the state of the a task when it is preempted .
3.When task regain control of the CPU TCB allows task to resume execution exactly where it left off.

3.03 Task Control Block

```
typedef struct os_tcb{
OS_TASK *OSTCBStkPtr;
#if OS_TASK_CREATE_EXT_EN >0
void *OSTCBExtPtr;
OS_STK *OSTCBStkBottom;
INT32U      OSTCBStkSize;
INT16U      OSTCBOpt;
INT16U      OSTCBid;
#endif
struct os_tcb *OSTCBNext;
struct os_tcb *OSTCBPrev;
#if
}
```

3.03 Task Control Block

OSTCBStkPtr:

1.a pointer to the current top-of-stack for the task.

OSTCBExtPtr:

1.a pointer to a use_definable task control block extension.

2.to extend TCB without having to change the source code for μC/OSⅡ:

a.the name of task

b.keep track of the execution time of the task

c.to keep track of times a task has switch-in

3.03 Task Control Block

OSTCBStkBottom:

1.a pointer to the bottom of task's stack.

2.used by OSTaskStkChk() to check the size of a task's stack at run Time, only you create a task with TaskCreateExt().

OSTCBStkSize:

1.hold size of the stack in number of element instead of bytes.

Ex:

1000 elements 32-bits=4000 bytes

1000 elements 16-bits=2000 bytes

3.03 Task Control Block

OSTCBOpt:

1.hold option that can be passed to OSTaskCreateExt().

2.has three options

a.OS_TASK_OPT_STK_CHK

to specify to OSTaskCreateExt() that tack checking is enabled for the task being created

b.OS_TASK_OPT_STK_CLR

indicate stack to cleared when task is created.(for stack checking)

c.OS_TASK_OPT_SAVE_FP

Tells OSTaskCreateExt() that task will be doing floating point

Computation.

3.03 Task Control Block

OSTCBId:

To hold an identifier for the task.

OSTCBNext and OSTCBPrev

Are used to doubly link OS_TCBs.

1.OSTimeTick use OSTCBNext to update the OSTimeDly.

OSTCBEventPtr:

Is a pointer to an event control block and is disabled later.

OSTCBMsg:

Is a pointer to a message sent to a task.

3.03 Task Control Block

OSTCBFlagNode:

a.is a pointer to an event flag node

b.OSTaskDly() used when we delete a task wait on an event flag Group.

OSTCBFlagRdy:

Contain the event flag made the task ready to run when the task was waiting on an event flag group.

3.03 Task Control Block

OSTCBDly:

a.when a task need to be delayed for a certain number of clock ticks.

b.a task need to pend for an event to occur with a timeout.

OSTCBStat:

Contain state of task.

Ex:

OS_STAT_READY:the task is ready to run.

3.03 Task Control Block

```
#define  OS_STAT_RDY          0x00      /* Ready to run
    */
#define  OS_STAT_SEM          0x01      /* Pending on semaphore
    */
#define  OS_STAT_MBOX         0x02      /* Pending on mailbox
    */
#define  OS_STAT_Q            0x04      /* Pending on queue
    */
#define  OS_STAT_SUSPEND      0x08      /* Task is suspended
    */
#define  OS_STAT_MUTEX        0x10      /* Pending on mutual
    exclusion semaphore                 */
#define  OS_STAT_FLAG         0x20      /* Pending on event flag group
    */
```

3.03 Task Control Block

OSTCBPrio:

Contain the task priority.

OSTCBX,OSTCBY,OSTCBBitX,OSTCBBitY:

a.Used to accelerate the process of making a task ready to run

b.Make a task wait for an event.

OSTCBX = priority >>3;

OSTCBY = OSMapTbl[priority >>3];

OSTCBBitX =priority &0x07;

OSTCBBitY =OSMapTbl[priority &0x07];

3.03 Task Control Block

OSTCBDelReq:

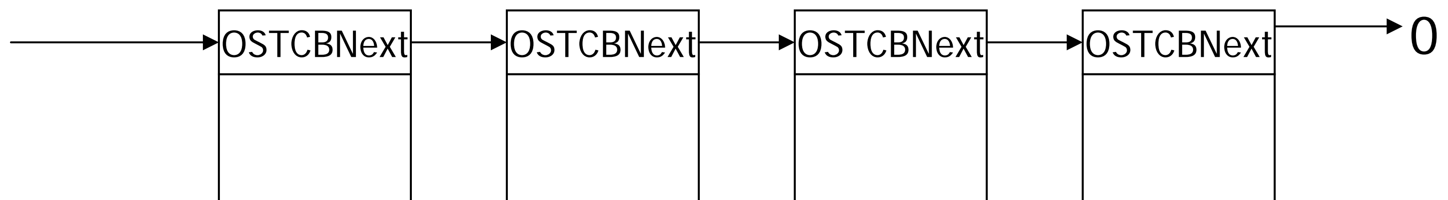
Is a boolean used to indicate whether or not a task has requested that the current task be deleted.

All OS_TCBs are placed in OSTCBTbl[].

OS_N_SYS_TASK means tasks os internal use:1.idle 2.statistic

OSTCBTbl[OS_MAX_TASKS+OS_N_SYS_TASK-1]

OSTCBFreeList



3.03 Task Control Block

OS_TCBInit() initialize OS_TCB when task is created.

OS_TCBInit() called by OSTaskCreate() or OSTaskCreateExt().

OS_TCBInit() receives seven arguments:

prio:task priority.

ptos:a pointer to the top of stack (OSTCBStkPtr)

pbos:a pointer to the stack bottom (OSTCBStkBottom)

id:the task identifier (OSTCBId)

stk_size:the total size of stack (OSTCBStkSize)

pext:the value to place in the OSTCBExtPtr.

opt:the OSTCB option (OSTCBOpt)

3.04 Ready List

Each task that is ready to run is placed in aready list consisting of two variables.

1.OSRdyGrp:

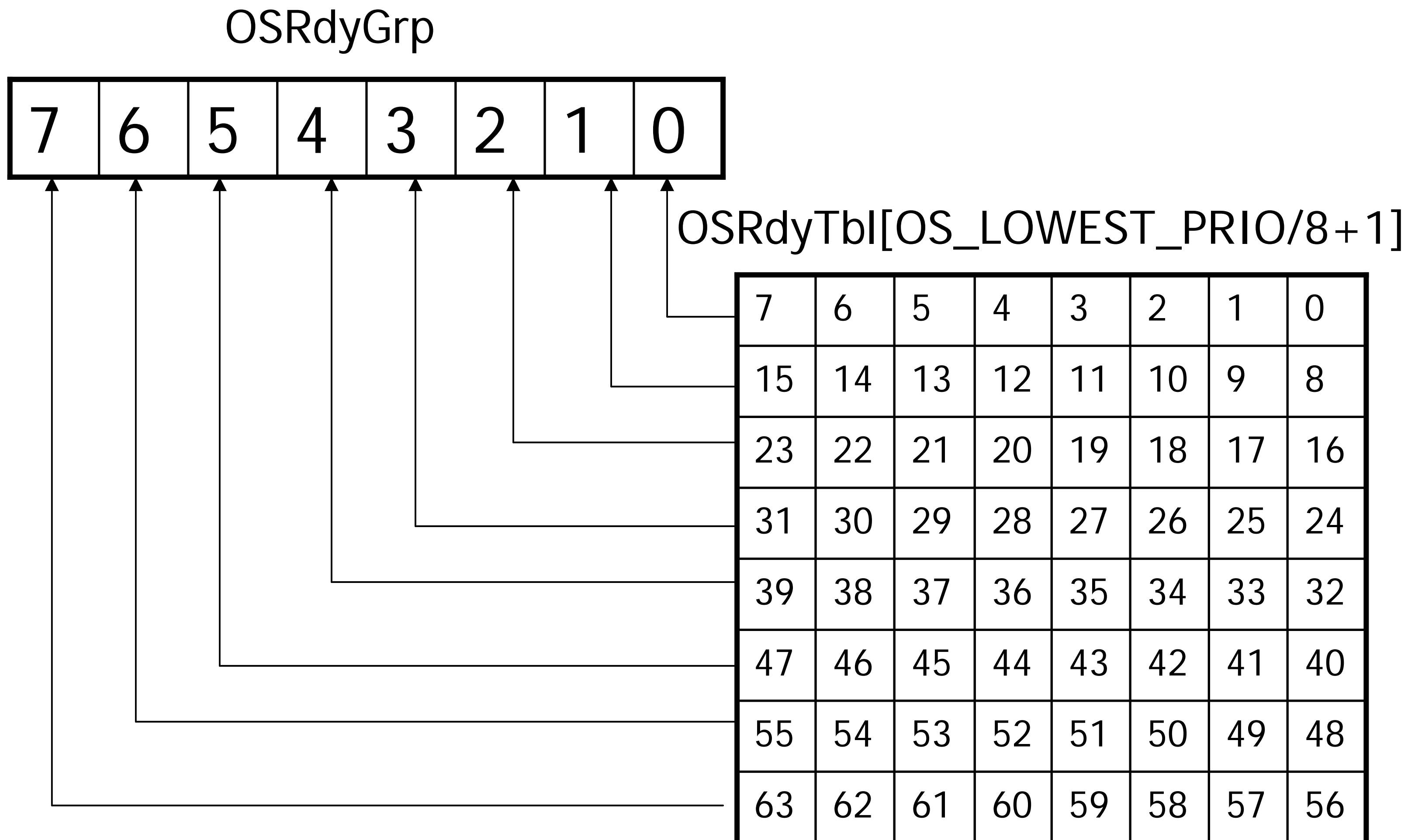
Task priority are grouped in OSRdyGrp, each bit in OSRdyGrp Indicates when a task in a group is ready to run.

2.OSRdyTbl[]

When a task is ready to run it also sets its corresponding bit in the ready table.

The size of OSRdyTbl[] depend on OS_LOWEST _PRIO.

3.04 Ready List



3.04 Ready List

OSMapTbl[]

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

3.04 Ready List

Making a task ready to run

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

Removing a task from the ready list

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

3.04 Ready List

OSUnMapTbl[256] is a priority resolution table.

Eight bits represent when tasks are ready in a group.

The least significant bit has the highest priority.

Using this byte to index OSUnMapTbl[] return the bit position of the highest priority bit set.

3.04 Ready List

```
INT*U 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 0xBF */
4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,   /* 0xF0 to 0xFF */};
```

3.04 Ready List

Ex:

OSGDdyGrp:

7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	0

The rightmost bit is group 3.

OSRdyTbl[3]=

7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0

The rightmost bit is group 2.

Finding the highest priority task ready to run

y =OSUnMapTbl[OSRdyGrp];

x =OSUnMapTbl[OSRdyTbl[y]];

use OSUnMapTbl[OSRdyGrp] to find the row in OSRdyTbl[].

use OSUnMapTbl[OSRdyTbl[y]] to find the position bit in

OSRdyTbl[y];

the highest priority in the ready list is :

Prio=(y<<3)+x;

3.05 Task Scheduling

Scheduler determine the highest priority to run from ready list.

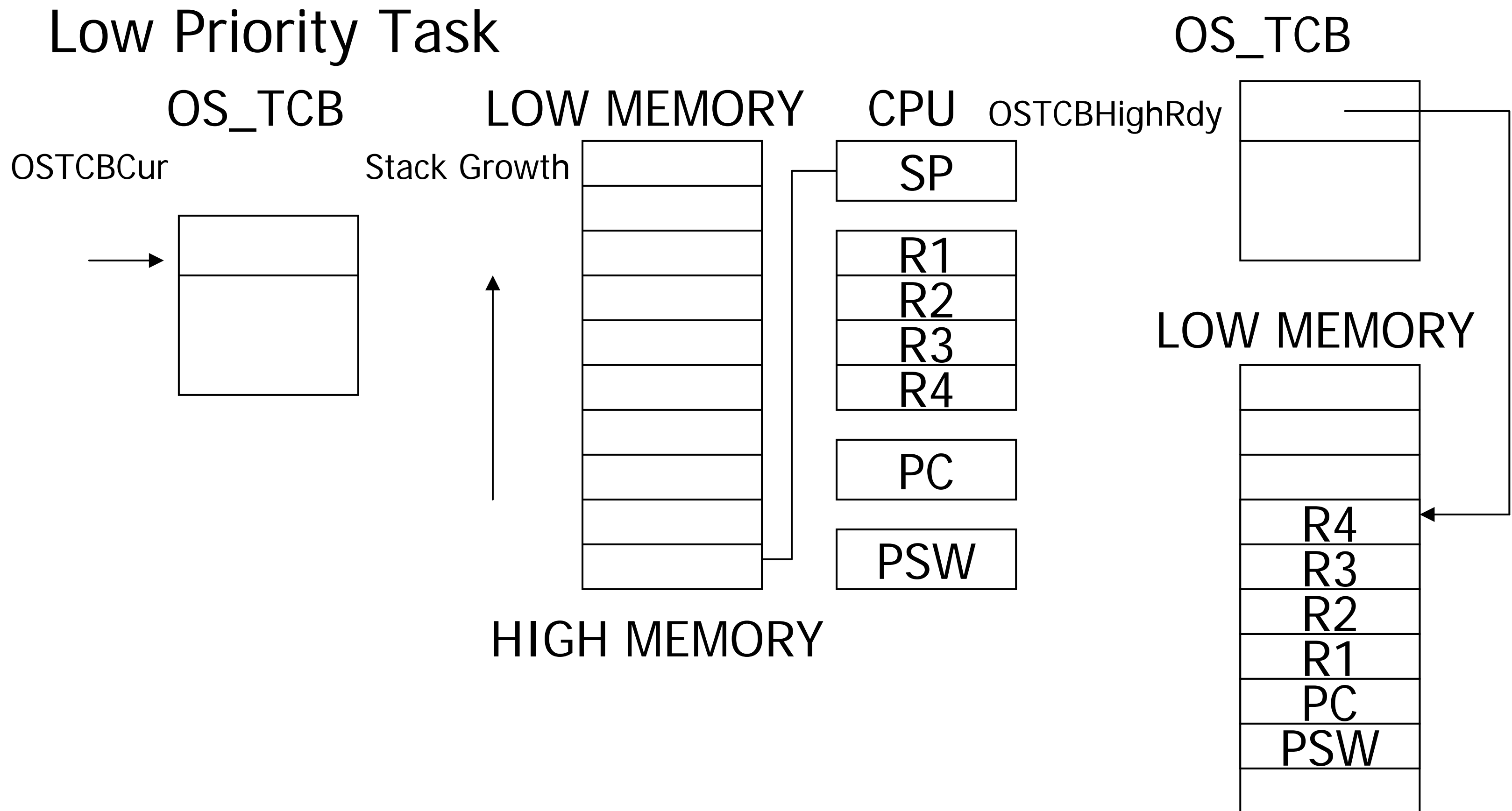
Task_level scheduling is performed by OS_Sched().

ISR-level scheduling is handled by another function OSIntExit().

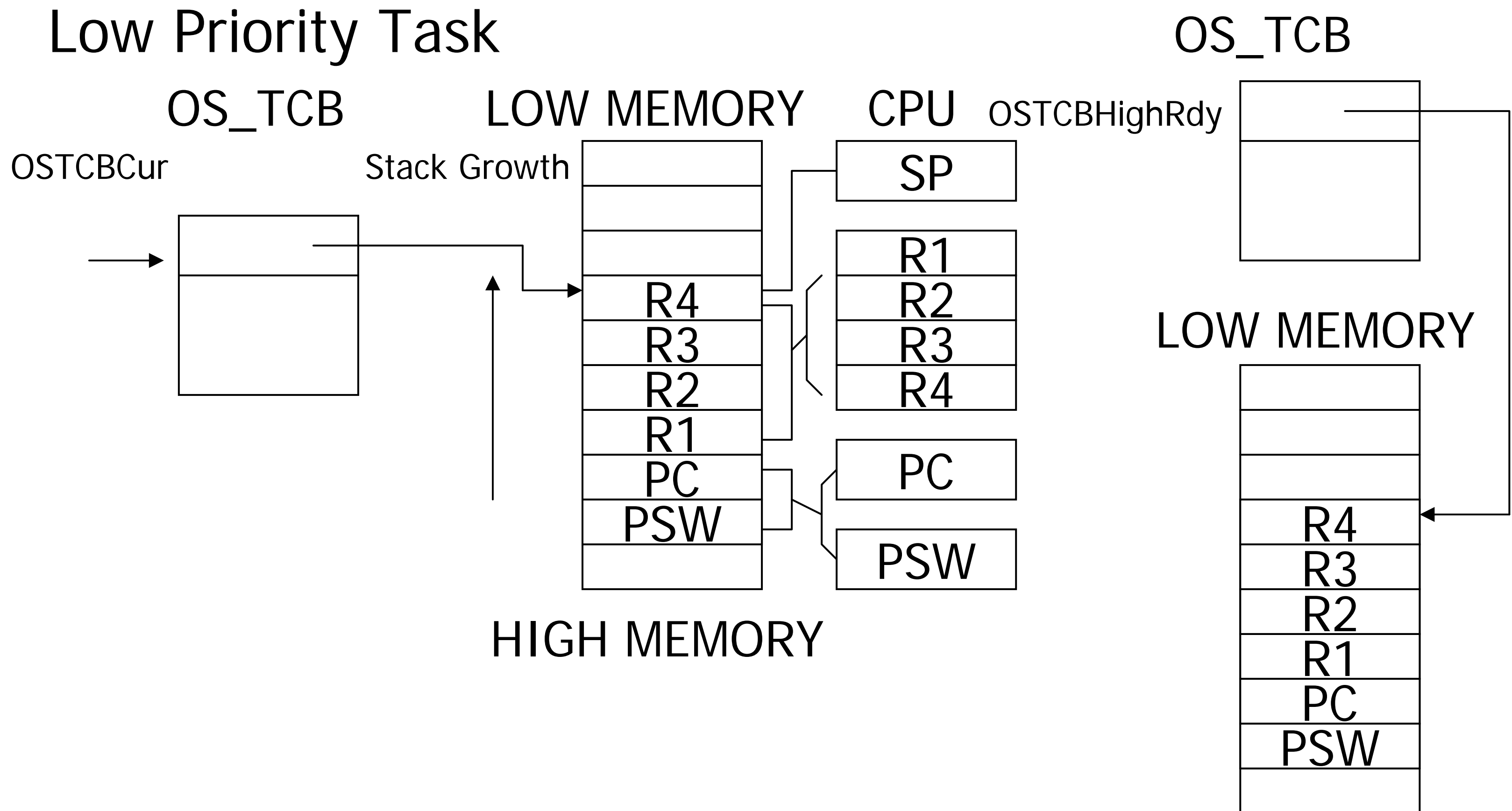
3.05 Task scheduling

```
Void OS_Sched(void)
{
#if OS_CRITICAL_METHOD==3
OS_CPU_SR cpu_sr;
#endif
INT8U        y;
OS_ENTER_CRITICAL();
if((OSIntNesting==0)&&(OSLockNesting==0)){
y              =OSUnMapTbl[OSRdyGrp];
OSPrioHighRdy=(INT8U)((y<<3)+OSUnMapTbl[OSRdyTbl[y]]);
if(OSPrioHighRdy !=OSPrioCur){
OSTCBHighRdy=OSTCBPrioTbl[OSPrioHighRdy];
OSCtxSwCtr++;
OS_TASK_SW();
}
}
OS_EXIT_CRITICAL();
}
```

3.06Task Level Context Switch



3.06Task Level Context Switch



3.06Task Level Context Switch

```
#define  uCOS                 0x80                 /* Interrupt vector # used
    for context switch    */
#define  OS_TASK_SW()         asm  INT   uCOS
```

3.07Locking and Unlocking the Scheduler

1.OSSchedLock() function is used to prevent task rescheduling until its counterpart OSSchedUnlock() is called.
2.Task call OSSchedLock() keeps control of CPU even though other higher priority task are ready to run.
3.OSLockNesting keep track of the number of times OSSchedLock() has been called.
4.To prevent other task access OSLockNesting you can it in critical Section.
If OSLockNesting is 0, Scheduling is re-enabled.

3.07Locking and Unlocking the Scheduler

```
void OSSchedLock(void)
{
#if OS_CRITICAL_Method==3
OS_CPU_SR cpu_sr;
#endif
if (OSRunning==TRUE) {
OS_ENTER_CRITICAL();
If (OSLockNesting<255){
OSLockNesting++;
}
OS_EXIT_CRITICAL();
}
}
```

3.07Locking and Unlocking the Scheduler

```
void OSSchedUnlock(void)
{
#if OS_CRITICAL_Method==3
OS_CPU_SR cpu_sr;
#endif
if(OSSRunning==TURE){
OS_ENTER_CRITICAL();
if((OSLockNesting==0)&&(OSIntNesting==0)) {
OS_EXIT_CRITICAL();
OS_Sched();
}else{
OS_EXIT_CRITICAL();
}
}
}
```

3.08 Idle Task

1. μC/OSⅡ always creates a task (idle task)is executed when none of the other task are ready to run.

2.idle task is always set to lowest priority.

```
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();
}
}
```

3.08 Idle Task

OSIdkeCtr is a 32-bits counter to determine the percentage of

CPU time actually being consumed by the application software.

OSTaskIdleHook() is a function you can write to do just about anything you want to do.

Ex

you can use OSTaskIdleHook() to stop the CPU so that it can

Enter low_power mode.

3.09 Statistic Task

1. μC/OSⅡ contain a task that provides run-time statistic

2.OS_TaskStat() if you set configuration constant

OS_TASK_STAT_EN=1

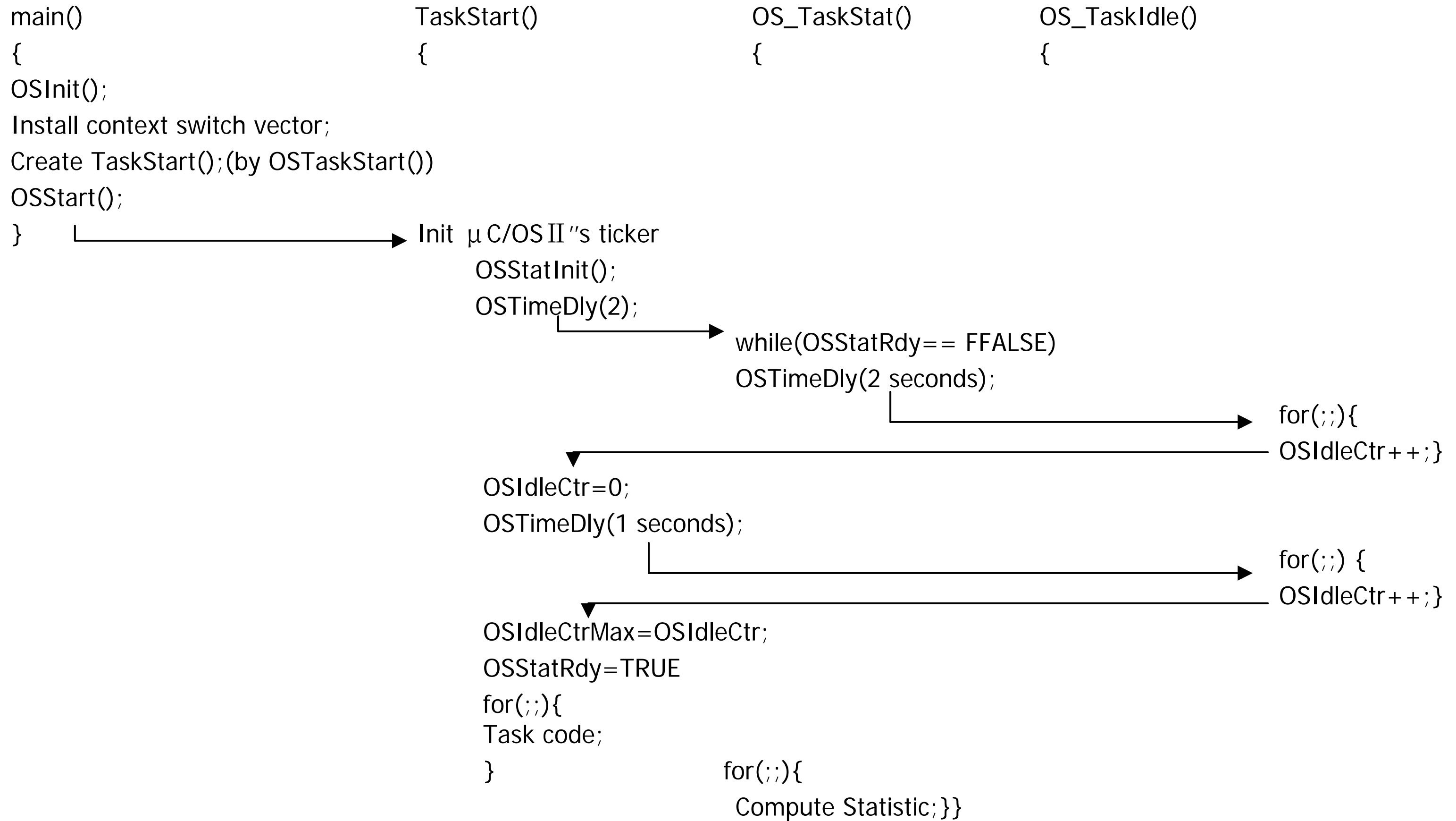
3.OS_TaskStat() execute every second and computes the percentage of CPU usage is put in 8-bits variable OSCPUUsage.

4.you must call OSStatInit() from the first and only task created in your application during initialization.

3.09 Statistic Task

```
void main (void)
{
OSInit()；
/*install μC/OSⅡ′s context vector */
/*Create your startup task */
OSStart()；
}
void TaskStart(void *pdata)
{
/*install and initialize μC/OSⅡ″s ticker*/
OSStatInit()；
/*Create your application task*/
For(；；){
/*
code for TaskStart()
*/
}
}
```

3.09 Statistic Task



3.09 Statistic Task

```
void OSStatInit(void)
{
#if OS_CRITICAL_METHOD==3
OS_CPU_SR cpu_sr;
#endif
OSTimeDly(2);
OS_ENTER_CRITICAL();
OSIdleCtr=0L;
OS_EXIT_CRITICAL();
OSTimeDly(OS_TICKS_PER_SEC);
OS_ENTER_CRITICAL();
OSIdleCtrMax=OSIdleCtr;
OS_EXIT_CRITICAL();
OSStatRdy=TRUE;
OS_EXIT_CRITICAL();
}
```

3.09 Statistic Task

```
void OS_TaskStat(void *pdata)
{
#if OS_CRITICAL_MEYHOD==3
OS_CPU_SR cpu_sr;
#endif
INT32U run,max;
INT8S usage;
pdata=pdata;
while(OSStatRdy==FLASE){
OSTimeFly(2*OS_TICKS_PER_SEC);
}
max=OSIdleCtrMax/100L;
for(;;){
OS_ENTER_CRITICAL();
OSIdleCtrRun=OSIdleCtr;
Run=OSIdleCtr;
OSIdleCtr=0;
OS_EXIT_CRITICAL();
}
if(max>0L)
{
usage=(INT8S)(100-run/max);
if(usage>=0){
OSCPUUsage=usage;
}else{
OSCPUUsage=0;
}
}else{
OSCPUUsage=0;
max=OSIdleCtrMax/100L;
}
OSTaskStatHook();
OSTimeDly(OS_TICKS_PER_SEC);
}
}
```

3.09 Statistic Task

1.OSCPUUsage(%)=100*(1-OSIdleCtr/OSIdleCtrMax)

Because have can’t use FP

2.OSCPUUsage(%)=100*(1-OSIdleCtr*100/OSIdleCtrMax)

fear of overflow

3. OSCPUUsage(%)=100*(1-OSIdleCtr/(OSIdleCtrMax/100))

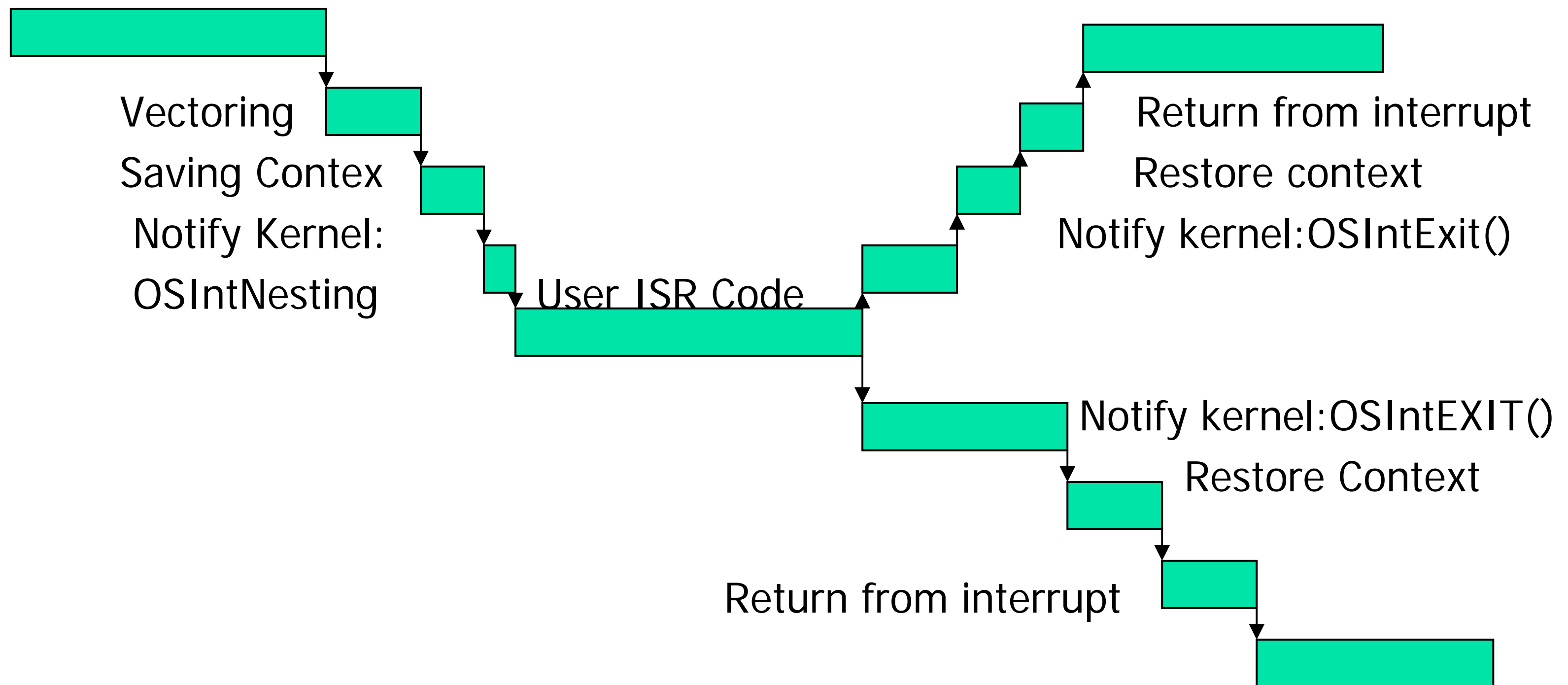
3.10Interrupt Under μC/OSⅡ

1. μC/OSⅡ requires that an ISR be written in assembly language.

2.if your Ccompiler supports in_line assembly language, you can put the ISR code directly in a C source file.

```
ISR:
Save all CPU register;
Call OSIntEnter(),or increment OSIntNesting directly;
if(OSIntNesting==1){
OSTCBCur->OS_TCBStkPtr=sp;
}
Clear interrputing device;
Re-enable interrupts(optional)
Execute user code to service ISR;
Call OSIntExit();
Restore all CPU registers;
Execute a return from interrupt instruction;
```

3.10Interrupt Under μC/OSⅡ



3.11 Clock Tick

μC/OSⅡ requires that you provide a periodic time source to keep track of time delays and timeouts.

The actual frequency of the clock tick depends on the desired tick resolution of your application.

Two way to obtain a tick source:

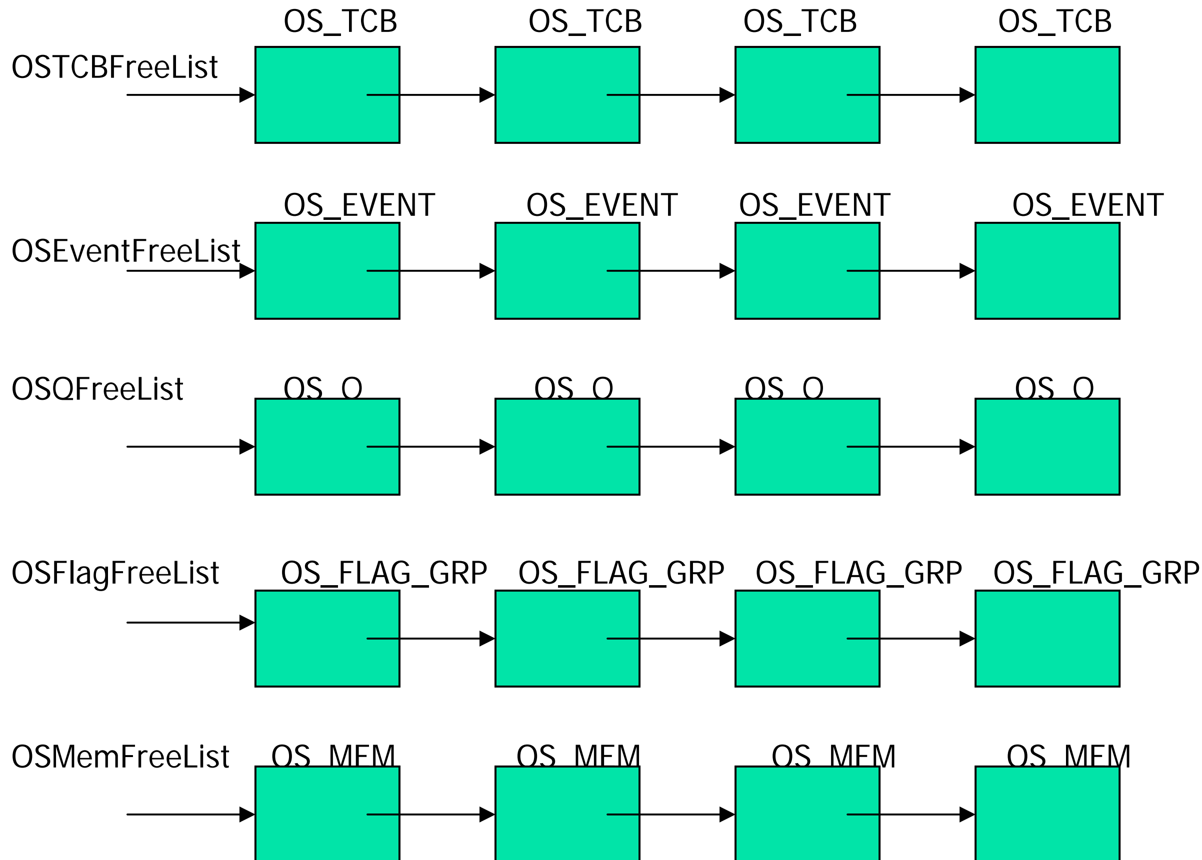
1.dedicating a hardware timer

2.generating an interrupt from an AC power line signal

3.12 μC/OSⅡ Initialization

You must call OSInit() to initializes all μC/OSⅡ variable and data structure and creates the idle task OS_TaskIdle().
if OS_TASK_STAT_En and OS_TASK_CREATE_EXT_EN are both set 1 OSInit() also creates the statistic task OS_TASKStat() .

3.12 μC/OS II Initialization



3.13 Starting μC/OSⅡ

void main(void)

{

OSInit()；

Create at least 1 task using either OSTaskCreate() or OSTaskCreateExt()；

OSStart()；

}

You start multitasking by calling OSStart().

3.13 Starting μC/OS II

```
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]; /*find OS_TCB of highest priority task*/
OSTCBCur          = OSTCBHighRdy;
OSStartHighRdy()
}
```

3.14 Obtaining the Current μC/OSⅡ Version

Calling OSVersion() to obtain current version of μC/OSⅡ

```
INT16U OSVersion (void)
{
return(OS_VERSION)
}
```

Returns version number, multiplied by 100

Ex version2.52 is returned as 252