The μC/OS-II Real-Time Operating System

Dr. Sarwan Singh

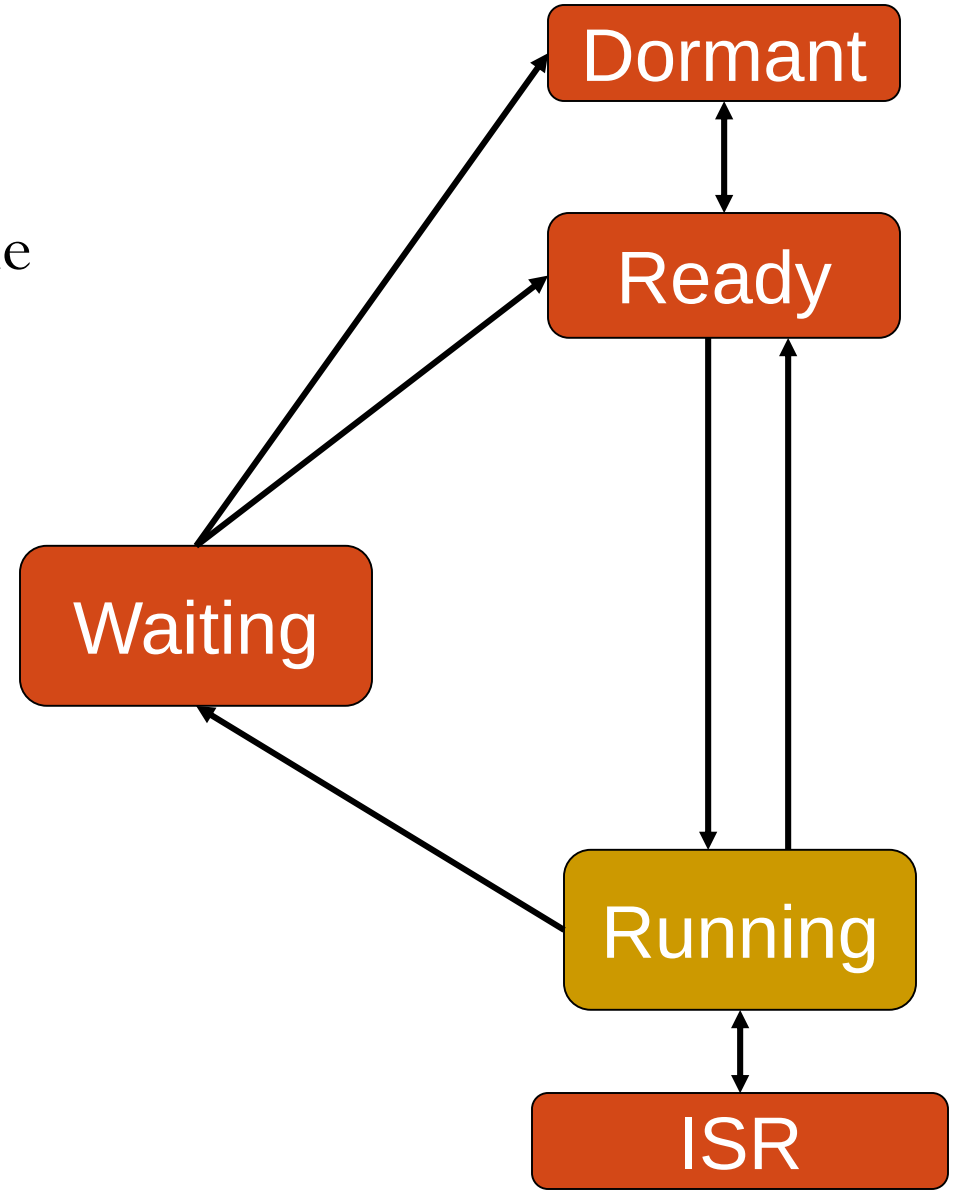
NIELIT Chandigarh

µC/OS-II

- Real-time kernel
  - Portable, scalable, preemptive RTOS
  - Ported to over 90 processors
- Pronounced "microC OS two"
- Written by Jean J. Labrosse of Micrium, http://ucos-ii.com
- Extensive information in **MicroC/OS-II:The Real-Time Kernel (A complete portable, ROMable scalable preemptive RTOS),** Jean J. LaBrosse, CMP Books

Task States

- Five possible states for a task to be in
  - Dormant – not yet visible to OS (use OSTaskCreate(), etc.)
  - Ready
  - Running
  - Waiting
  - ISR – preempted by an ISR
- See manual for details



Task Scheduling

- Scheduler runs highest-priority task using OSSched()
  - OSRdyTbl has a set bit for each ready task
  - Checks to see if context switch is needed
  - Macro OS_TASK_SW performs context switch
    - Implemented as software interrupt which points to OSCtxSw
    - Save registers of task being switched out
    - Restore registers of task being switched in
- Scheduler locking
  - Can lock scheduler to prevent other tasks from running (ISRs can still run)
    - OSSchedLock()
    - OSSchedUnlock()
  - Nesting of OSSchedLock possible
  - Don't lock the scheduler and then perform a system call which could put your task into the WAITING state!
- Idle task
  - Runs when nothing else is ready
  - Automatically has prioirty OS_LOWEST_PRIO
  - Only increments a counter for use in estimating processor idle time

Where Is The Code Which Makes It Work?

- Selecting a thread to run
  - OSSched() in os_core2.c
- Context switching
  - OS_TASK_SW in os_cpu.h
  - OSCtxSw in os_cpu_a.a30
- What runs if no tasks are ready?
  - OSTaskIdle() in os_core2.c

TCB for µC/OS-II

```
typedef struct os_tcb {  /* Extended TCB code is italicized */
   OS_STK *OSTCBStkPtr;          /* Pointer to current top of stack
   */
   void    *OSTCBExtPtr;         /* Pointer to user definable data
   for TCB
                                        extension */
    OS_STK *OSTCBStkBottom;      /* Pointer to bottom of stack - last
                                        valid address */
    INT32U OSTCBStkSize;         /* Size of task stack (in bytes) */
    INT16U OSTCBOpt;             /* Task options as passed by
                                        OSTaskCreateExt() */
    INT16U OSTCBId;              /* Task ID (0..65535) */
   struct os_tcb *OSTCBNext;     /* Pointer to next TCB in the TCB
   list */
   struct os_tcb *OSTCBPrev;     /* Pointer to previous TCB in list
   */
   OS_EVENT *OSTCBEventPtr;      /* Pointer to event control block */
   void *OSTCBMsg;               /* Message received from
   OSMboxPost() or
                                        OSQPost() */
   INT16U OSTCBDly;              /* Nbr ticks to delay task or,
   timeout
                                        waiting for event */
    INT8U OSTCBStat;             /* Task status */
    INT8U OSTCBPrio;             /* Task priority (0 == highest,
                                        63 == lowest) */
   … ready table position information …
   BOOLEAN OSTCBDelReq;          /* Indicates whether a task needs to
                                        delete itself */
} OS_TCB;
```

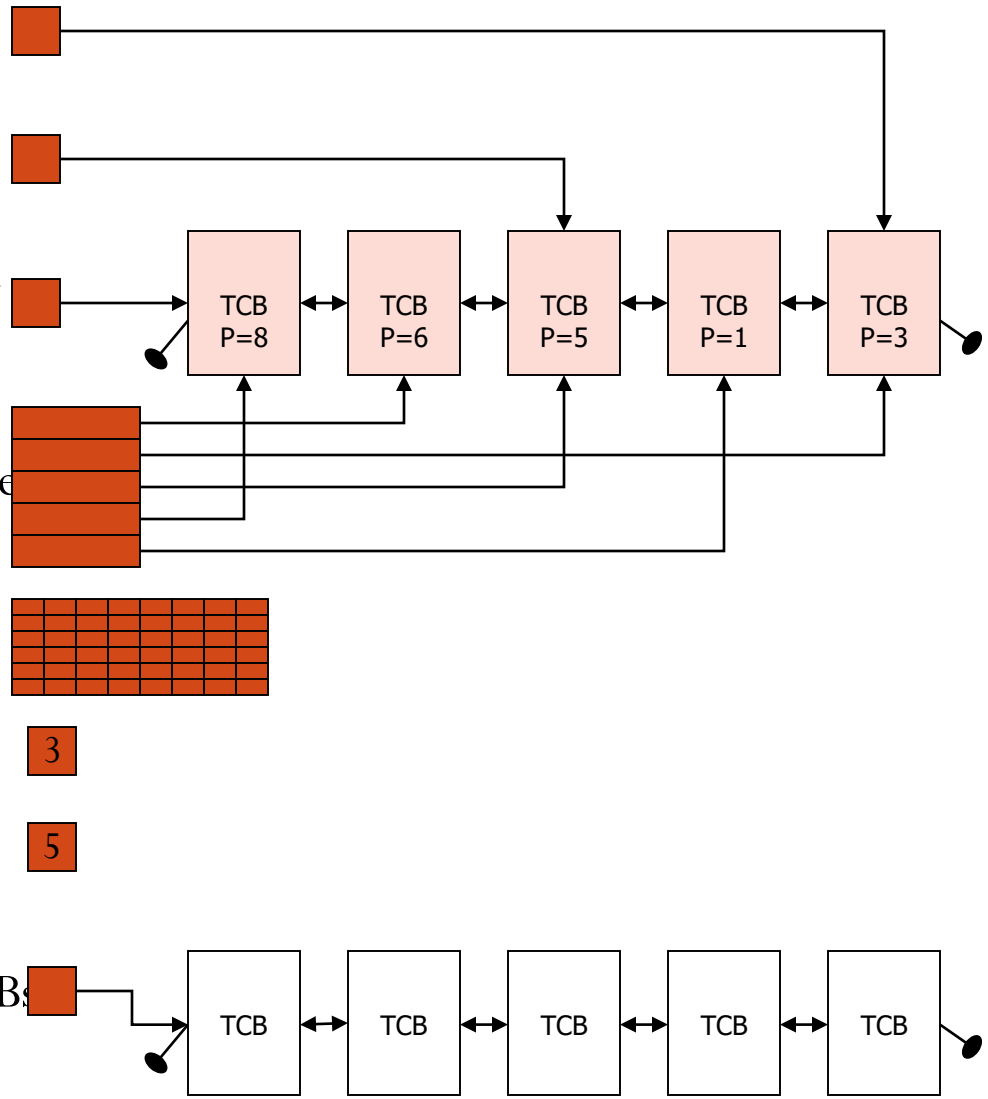
Task States

- Task status OSTCBStat

```
/*TASK STATUS (Bit definition for OSTCBStat)          */
#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 */
```

Data Structures for µC/OS-II

- OSTCBCur - Pointer to TCB of currently running task
- OSTCBHighRdy - Pointer to highest priority TCB ready to run
- OSTCBList - Pointer to doubly linked list of TCBs
- OSTCBPrioTbl[OS_LOWEST_PRIO + 1] - Table of pointers to create TCBs, ordered by priority
- OSReadyTbl - Encoded table of tasks ready to run
- OSPrioCur – Current task priority
- OSPrioHighRdy – Priority of highest ready task
- OSTCBFreeList - List of free OS_TCBs use for creating new tasks



Enabling Interrupts

- Macros OS_ENTER_CRITICAL, OS_EXIT_CRITICAL
- Note: three methods are provided in os_cpu.h
  - #1 doesn't restore interrupt state, just renables interrupts
  - #2 saves and restores state, but stack pointer must be same at enter/exit points – ***use this one!***
  - #3 uses a variable to hold state
    - Is not reentrant
    - Should be a global variable, not declared in function StartSystemTick()

System Clock Tick

- OS needs periodic timer for time delays and timeouts
- Recommended frequency 10-200 Hz (trade off overhead vs. response time (and accuracy of delays)
- Must enable these interrupts after calling OSStart()
- Student exercise
  - Which timer is used for this purpose on the QSK62P?
  - What is the frequency?
- OSTick() ISR
  - Calls OSTimeTick()
    - Calls hook to a function of your choosing
    - Decrements non-zero delay fields (OSTCBDly) for all task control blocks
    - If a delay field reaches zero, make task ready to run (unless it was suspended)
  - Increments counter variable OSTime (32-bit counter)
  - Then returns from interrupt
- Interface
  - OSTimeGet(): Ticks (OSTime value) since OSStart was called
  - OSTimeSet(): Set value of this counter

Overview of Writing an Application

- Scale the OS resources to match the application
  - See os_cfg.h
- Define a stack for each task
- Write tasks
- Write ISRs
- Write main() to Initialize and start up the OS (main.c)
  - Initialize MCU, display, OS
  - Start timer to generate system tick
  - Create semaphores, etc.
  - Create tasks
  - Call OSStart()

Configuration and Scaling

- For efficiency and code size, default version of OS supports limited functionality and resources
- When developing an application, must verify these are sufficient (or may have to track down strange bugs)
  - Can't just blindly develop program without considering what's available
- Edit ***os_cfg.h*** to configure the OS to meet your application's needs
  - # events, # tasks, whether mailboxes are supported, etc.

Writing ISRs

- Structure needed
  - Save CPU registers – *NC30 compiler adds this automatically*
  - Call OSIntEnter() or increment OSIntNesting (faster, so preferred)
    - OSIntEnter uses OS_ENTER_CRITICAL and OS_EXIT_CRITICAL, so make sure these use method 2 (save on stack)
  - Execute code to service interrupt – *body of ISR*
  - Call OSIntExit()
    - Has OS find the highest priority task to run after this ISR finishes (like OSSched())
  - Restore CPU registers – *compiler adds this automatically*
  - Execute return from interrupt instruction – *compiler adds this automatically*
- Good practices
  - Make ISR as quick as possible. Only do time-critical work here, and defer remaining work to task code.
  - Have ISR notify task of event, possibly send data
    - OSSemPost – raise flag indicating event happened
    - OSMboxPost – put message with data in mailbox (1)
    - OSQPost – put message with data in queue (n)
      - Example: Unload data from UART receive buffer (overflows with 2 characters), put into a longer queue (e.g. overflows after 128 characters) which is serviced by task

Writing Tasks

- Define a stack for each task
  - Must be a global (static) array of base type OS_STK
- Task structure: two options
  - Function with infinite loop (e.g. for periodic task)
    - Each time the loop is executed, it must call an OS function which can yield the processor (e.g. OSSemPend(), OSMboxPend(), OSQPend(), OSTaskSuspend(), OSTimeDly(), OSTimeDlyHMSM())
  - Function which runs once and then deletes itself from scheduler
    - Task ends in OSTaskDel()

Task Creation

- OSTaskCreate() in os_task.c
  - Create a task
  - Arguments: pointer to task code (function), pointer to argument, pointer to top of stack (use TOS macro), desired priority (unique)
- OSTaskCreateExt() in os_task.c
  - Create a task
  - Arguments: same as for OSTaskCreate(), plus
    - id: user-specified unique task identifier number
    - pbos: pointer to bottom of stack. Used for stack checking (if enabled).
    - stk_size: number of elements in stack. Used for stack checking (if enabled).
    - pext: pointer to user-supplied task-specific data area (e.g. string with task name)
    - opt: options to control how task is created.

More Task Management

- OSTaskSuspend()
  - Task will not run again until after it is resumed
  - Sets OS_STAT_SUSPEND flag, removes task from ready list if there
  - Argument: Task priority (used to identify task)
- OSTaskResume()
  - Task will run again once any time delay expires and task is in ready queue
  - Clears OS_STAT_SUSPEND flag
  - Argument: Task priority (used to identify task)
- OSTaskDel()
  - Sets task to DORMANT state, so no longer scheduled by OS
  - Removed from OS data structures: ready list, wait lists for semaphores/mailboxes/queues, etc.
- OSTaskChangePrio()
  - Identify task by (current) priority
  - Changes task's priority
- OSTaskQuery()
  - Identify task by priority
  - Copies that task's TCB into a user-supplied structure
  - Useful for debugging

Time Management

- Application-requested delays
  - Task A calls OSTimeDly or OSTimeDlyHMSM() in os_time.c
  - TCB->OSTCBDly set to indicate number of ticks to wait
  - Rembember that OSTickISR() in os_cpu_a.a30, OSTimeTick() in os_core2.c decrement this field and determine when it expires
  - Task B can resume Task A by calling OSTimeDlyResume()

Example: uC/OSII Demo

- Tasks
  - Task 1
    - Flashes red LED
    - Displays count of loop iterations on LCD top line
  - Task 2
    - Flashes green LED
  - Task 3
    - Flashes yellow LED

Debugging with an RTOS

- Did you scale the RTOS to your application?
  - Number of tasks, semaphores, queues, mailboxes, etc.
- Always check result/error codes for system calls
  - Light an LED if there's an error
- Why doesn't my thread run?
  - Look at scheduler's data structures via debugger
    - OSReadyTbl: Table of tasks ready to run
      - Bitfield array
    - TCB: Task control block
      - OSTCBStat: status field
  - If the error LED goes on, set a breakpoint there and see what happened
- Does your thread have enough stack space?
  - sprintf takes a lot. Floating point math does too.
- Did you remember to call OSTaskCreate for your thread?
- Is your thread structured as an infinite loop with an OS call on which to block?
- Are interrupts working properly? Try substituting a polling function to isolate the problem.
- Is there enough memory for your program? Check the .map file
  - RAM: 0400h to 0137Eh
  - Flash ROM: 0F0000h to 0FF8FFh

Summary

- Basics of using uC/OS-II
  - Task states and scheduling
  - Time tick
  - How to structure an application
  - How to create and manage tasks
  - How to delay a task
- How to debug when using an RTOS

Introduction to µC/OS-II

- µC/OS-II is a portable, ROMable, scalable, deterministic, pre-emptive, real-time multitasking kernel
- Simple to use and simple to implement, but very effective compared to the price/performance ratio.
- Can manage up to 64 tasks
- Supports all types of processors from 8-bit to 64-bit, ported to more than 100 microprocessors and microcontrollers
- Extensible with many additional modules (TCP/IP stack, USB host and client, FAT file system etc.)

Additional modules

Real-Time Operating System (RTOS)

TCP-IP protocol stack

Host USB stack

Device USB stack

CAN protocol stack

Runtime data monitor

µC/Shell and µC/LCD driver

Microsoft compatible FAT File System

Basic concepts

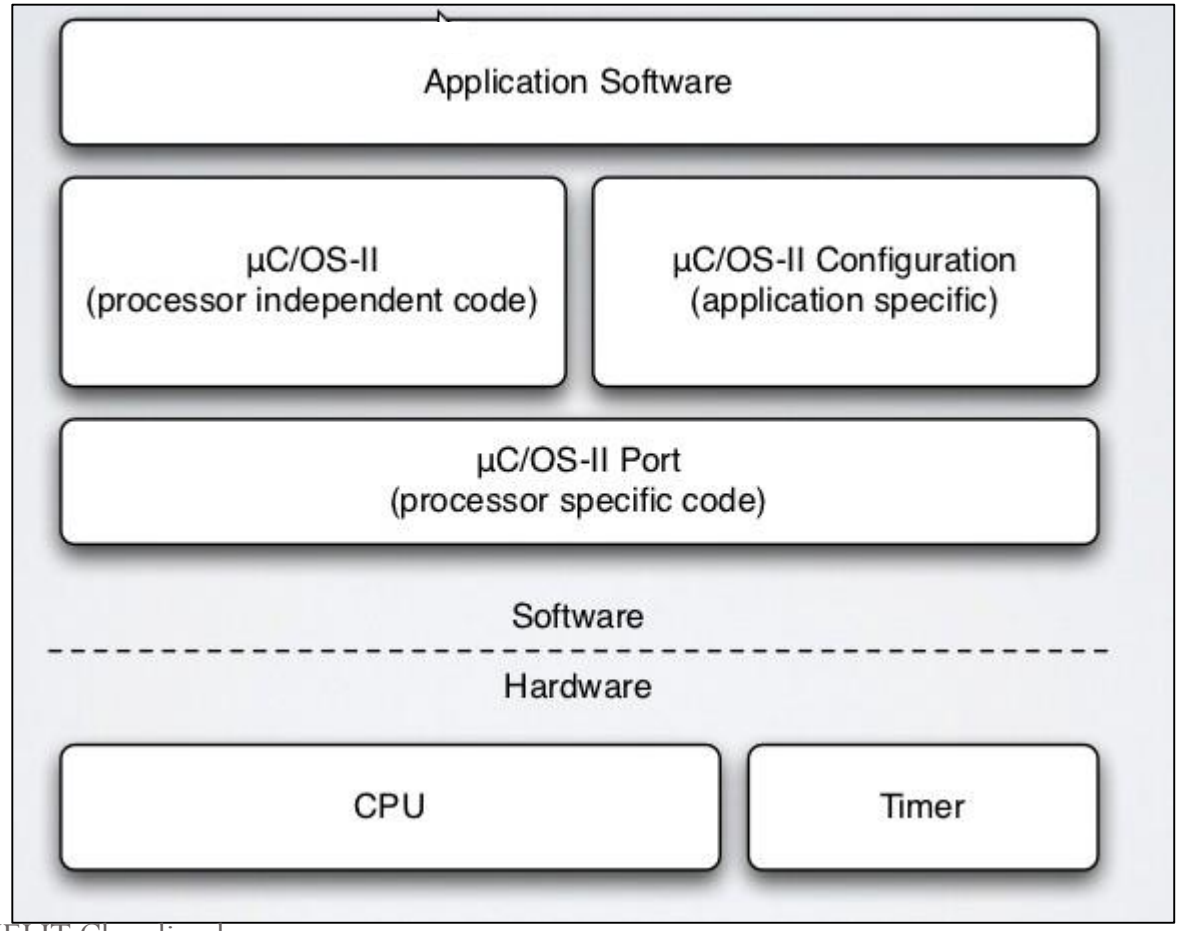
Processor independent code:

Common code which is available in the form of source code. This code is independent of the hardware or processor.

Processor specific code:

According to the specification of the processor. Partially C code, partially assembly.

µC/OS-II software architecture



Processor independent code

ucos_ii.h	Master header file for µC/OS-II
ucos_ii.c	Master program file for µC/OS-II
os_core.c	Core functions
os_flag.c	Eventflag management
os_mbox.c	Mailbox management
os_mem.c	Memory management
os_mutex.c	Mutex management
os_q.c	Queue management
os_sem.c	Semaphore management
os_task.c	Task management
os_time.c	Time management
os_tmr.c	Timer management
os_cfg_r.h	Configuration file for µC/OS-II (template file)
os_dbg_r.c	Debugger constants (template file)

Processor dependent code

app_cfg.h	General application specific header file
os_cfg.h	µC/OS-II configuration header, application dependent
os_cpu.h	680x0 specific header file
os_cpu_c.c	680x0 specific C code
os_fcpu_c.c	680x0 specific C code with 68881 FPU support
os_cpu_a.asm	680x0 specific assembly code
os_fcpu_a.asm	680x0 specific assembly code w. 68881 FPU support
os_boot.asm	IDE68K specific assembly code (vectors, basic IO)

Software license

The µC/OS-II operating system software is neither freeware nor Open Source software. A licensed copy of version 2.91 of the operating system should be obtained directly from Micriµm

LICENSING TERMS:
µC/OS-II is provided in source form for FREE evaluation, for educational use or for peaceful research. If you plan on using µC/OS-II in a commercial product you need to contact Micriµm to properly license its use in your product. We provide ALL the source code for your convenience and to help you experience µC/OS-II. The fact that the source is provided does NOT mean that you can use it without paying a licensing fee.

µC/OS-II API

The Application Program Interface of µC/OS-II consists of a set of C-functions. These functions are used to write the application programs (tasks) with µC/OS-II

API functions are used to handle

- Tasks
- Time management
- Mailboxes
- Semaphores and Mutexes
- Memory Management
- Timers

etc.

µC/OS-II Initialization

Call OSInit() before you call any of its other services.

OSInit() creates the idle task (OSTaskIdle()) which is always

ready to run.

The priority of OSTaskIdle() is always set to OS_LOWEST_PRIO

If OS_TASK_STAT_EN and OS_TASK_CREATE_EXT_EN in file os_cfg.h are both set to '1', OSInit() also creates the statistics task, OSTaskStat() and makes it ready to run.

The priority of OSTaskStat() is always OS_LOWEST_PRIO – 1.

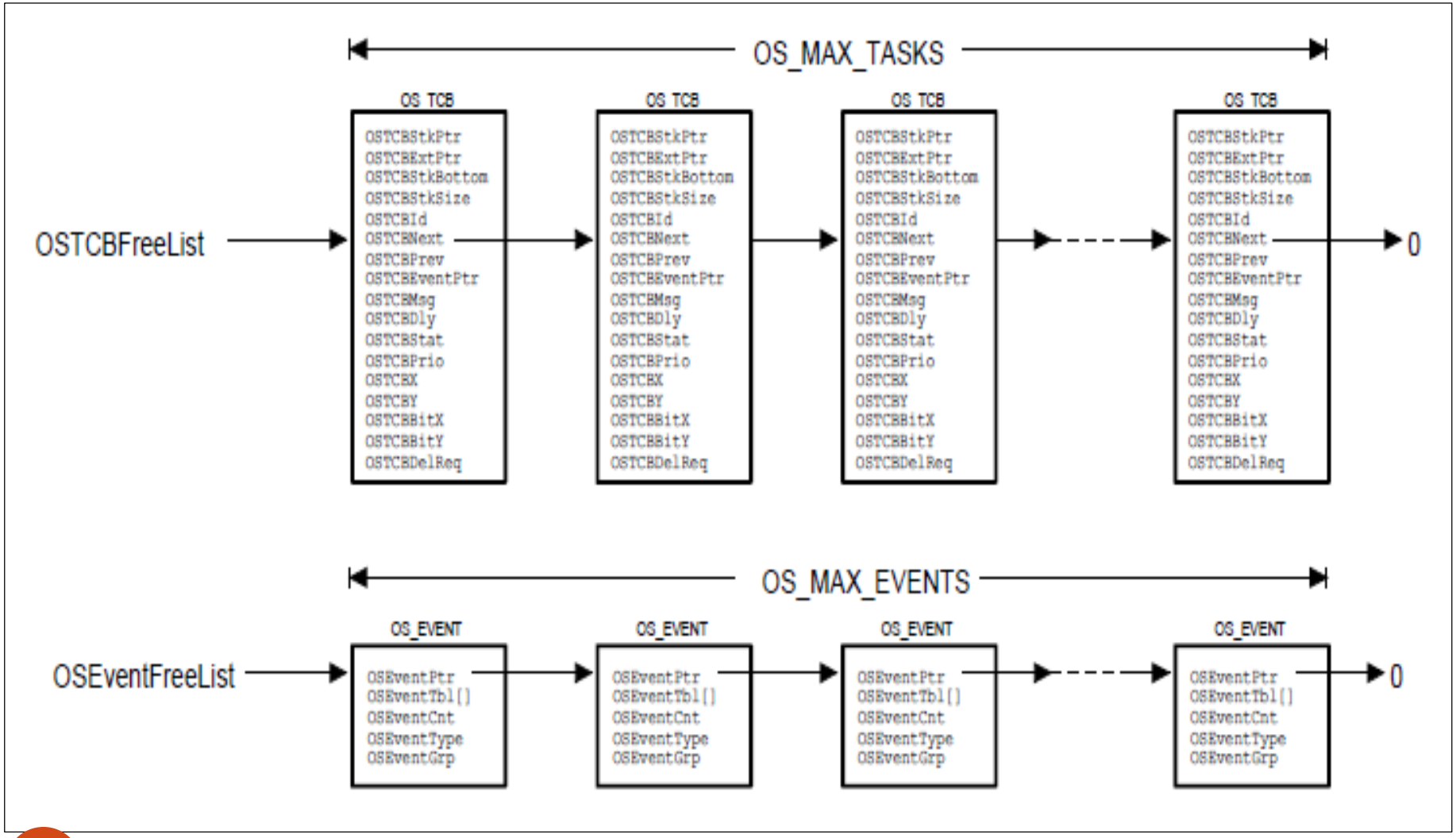
OSInit()

```
void OSInit(void)
{
    OSInitHookBegin();
    OS_InitMisc();
    OS_InitRdyList();
    OS_InitTCBList();
    OS_InitEventList();
    OS_InitTaskIdle();

#if (OS_FLAG_EN > 0u) && (OS_MAX_FLAGS > 0u)
    OS_FlagInit();
#endif

#if (OS_MEM_EN > 0u) && (OS_MAX_MEM_PART > 0u)
    OS_MemInit();
#endif
```

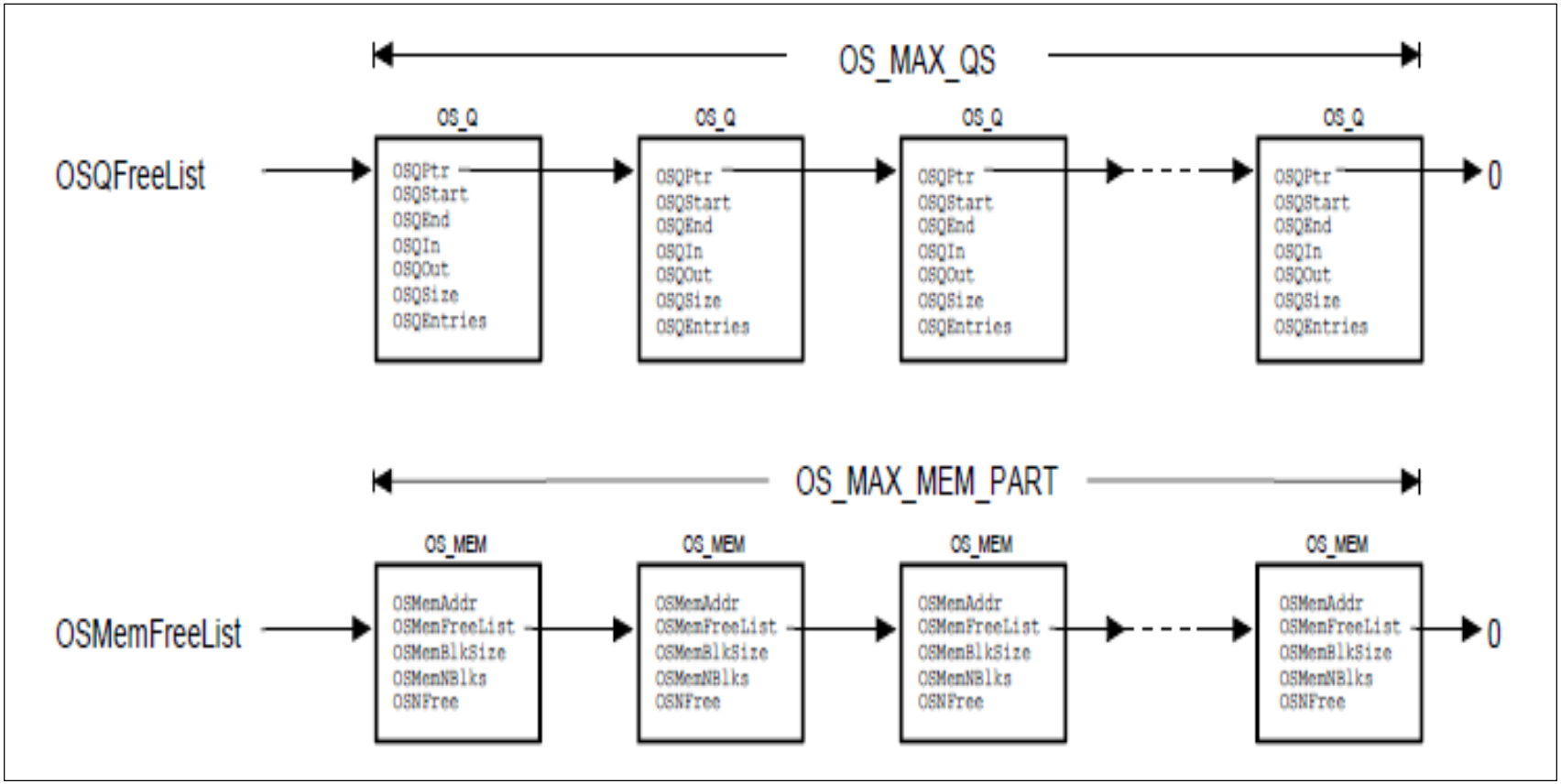
OSInit() (continued)

```
#if (OS_Q_EN > 0u) && (OS_MAX_QS > 0u)
    OS_QInit();
#endif

#if OS_TASK_STAT_EN > 0u
    OS_InitTaskStat();
#endif

#if OS_TMR_EN > 0u
    OSTmr_Init();
#endif

#if OS_DEBUG_EN > 0u
    OSDebugInit();
#endif

    OSInitHookEnd();
}
```

Free pools


OS_MAX_TASKS
OS TCB
OSTCBStkPtr
OSTCBExtPtr
OSTCBStkBottom
OSTCBStkSize
OSTCBId
OSTCBNext
OSTCBPrev
OSTCBEventPtr
OSTCBMsg
OSTCBDly
OSTCBStat
OSTCBPrio
OSTCBX
OSTCBY
OSTCBBitX
OSTCBBitY
OSTCBDelReq
OSTCBFreeList
0
OS_MAX_EVENTS
OS_EVENT
OSEventPtr
OSEventTbl[]
OSEventCnt
OSEventType
OSEventGrp
OSEventFreeList
0

Free pools (continued)



OSStart()

```
void OSStart(void)
{
    INT8U x, y;

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

The main() function

```
void main(void)
{
    OSInit();                  /* Initialize uC/OS-II    */
    /* Install uC/OS-II's context switch vector    */
    /* Create your startup task (e.g. TaskStart()) */
    OSStart();                 /* Start multitasking     */
}

void TaskStart(void *pdata)
{
/* Install and initalize uC/OS-II's ticker    */
    OSStatInit();   /* Initialize statistics task  */
    /* Create your application task(s) */
    for (;;) }
        /* Code for TaskStart() goes here! */
    }
}
```

Critical Sections

Two macro's are used together to wrap around the critical section

- OS_ENTER_CRITICAL() : Disable interrupts
- OS_EXIT_CRITICAL() : Enable Interrupts

Defined in OS_CPU.H

Three implementations:

- OS_CRITICAL_METHOD = 1, 2, 3

Critical sections (continued)

OS_CRITICAL_METHOD = 1

The status register (or PSW) is not saved at the calls to **OS_ENTER_CRITICAL()**. The status of interrupts may change on calling µC/OS-II functions

OS_CRITICAL_METHOD = 2

The status register is saved on the stack at **OS_ENTER_CRITICAL()** and restored at **OS_EXIT_CRITICAL()**. The compiler must use in-line assembly instructions.

OS_CRITICAL_METHOD = 3

The status register is saved in a local variable, declared in the task function, at **OS_ENTER_CRITICAL()**. This variable is used to restore the status register at **OS_EXIT_CRITICAL()**.

Tasks

µC/OS-II can manage up to 64 tasks. Each task has a unique priority. Two tasks with the lowest priority are reserved for the system.

This means, in theory, we can create up to 62 tasks

OS_LOWEST_PRIO (= 63), the priority of the idle task, is the lowest priority task in the system.

Idle task means that if there is no task in the ready queue, the CPU will execute the idle task.

Task Management

- Creating a task
- Stack checking
- Changing the task priority
- Getting Information about a task
- Suspending a task
- Resuming a task
- Request to delete a task
- Deleting a task

Task creation

There are two functions to create a task:

1. OSTaskCreate()
2. OSTaskCreateExt()

OSTaskCreate( ) requires 4 arguments:

1. ‘task’ is pointer to the task function
2. ‘pdata’ is a pointer to an argument that is passed to the task function when it starts executing
3. ‘ptos’ is a pointer to the top of the stack
4. ‘prio’ is the task priority

OSTaskCreateExt()

OSTaskCreateExt() offers more functionality but at the expense of additional overhead. This function requires 9 arguments. The first four arguments are same as with OSTaskCreate()

Extra arguments:

5. ‘id’ a unique identifier for the task being created. This argument has been added for future expansion and is not used by µC/OS-II, usually set to task priority.
6. ‘pbos’ is a pointer to the task's bottom of stack, used to perform stack checking
7. ‘stk_size’ is the size of the stack in number of elements. This argument is also used for stack checking

OSTaskCreateExt() (continued)

8. ‘pext’ is a pointer to a user supplied data area that can be used to extend the size of the task control block
9. ‘opt’ specifies options to OSTaskCreateExt(). This argument specifies whether stack checking is allowed, whether the stack will be cleared, and whether floating point operations are performed by the task.

The file os_cfg.h contains a list of available options

1. OS_TASK_OPT_STK_CHK
2. OS_TASK_OPT_STK_CLR
3. OS_TASK_OPT_SAVE_FP

Task initialization

OSTaskCreate() calls OSTaskStkInit () to initialize the stack for the task. OSTaskInit() is defined in the file os_cpu.c

µC/OS-II supports processors that have stacks that grow from either high memory to low memory or from low memory to high memory (defined by OS_STK_GROWTH in the file os_cpu.h)

OSTCBInit() initializes the TCB for this task. Defined in os_core.c

OSTaskCreateHook() is a user specified function that extends the functionality of OSTaskCreate() but can increase interrupt latency

Finally, OSTaskCreate() calls OSSched() to determine whether the new task has a higher priority than its creator.

If the task was created before multitasking has started (i.e. OSStart() was not called yet), then the scheduler is not called.

Task function

A task is typically an infinite loop function

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

Task deletion

A task can delete itself upon completion.

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

Task Stack

Each task must have its own stack. A stack must be declared as being of type OS_STK and must consist of contiguous memory locations.

We pass the top of stack pointer in the task create function

Memory can be allocated statically or dynamically

Static allocation

```
OS_STK MyTaskStack[STACKSIZE];
```

Dynamic allocation

```
OS_STK *pstk;

pstk = (OS_STK *) malloc(STACKSIZE);
if (pstk != (OS_STK *) 0) {
    /* Create the task */
}
```

Task Stack (continued)

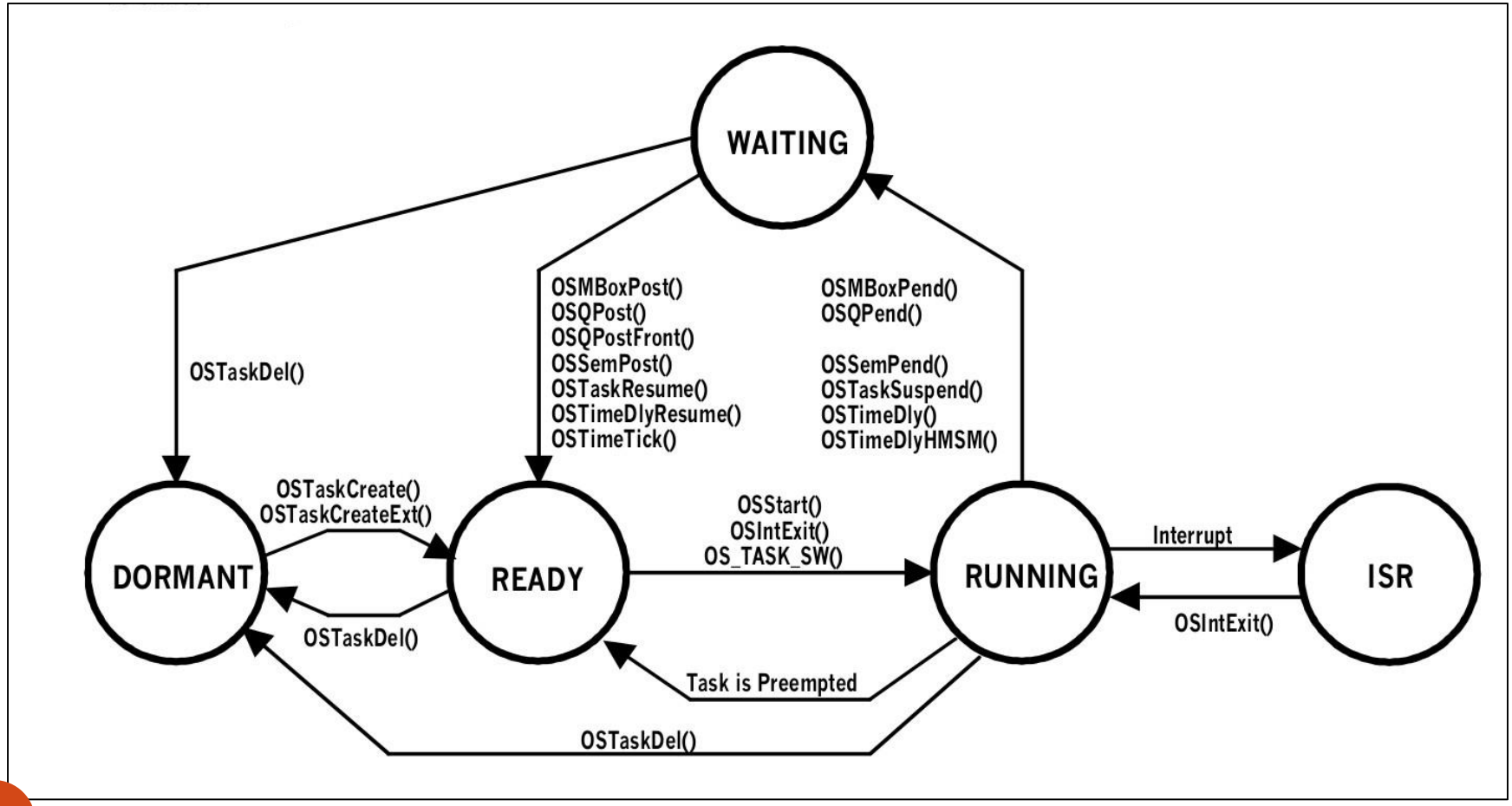
If in file os_cpu.h the constant OS_STK_GROWH is set to 1 you must pass the **highest** memory location of the stack to the task-create function because the stack grows from high to low memory

```
OS_STK TaskStack[STACKSIZE];
OSTaskCreate(task, pdata, &TaskStack[STACKSIZE - 1], prio);
```

If the constant OS_STK_GROWTH is set to 0, you must pass the **lowest** memory location of the stack to the task-create function because the stack grows from low to high memory

```
OS_STK TaskStack[STACKSIZE];
OSTaskCreate(task, pdata, &TaskStack[0], prio);
```

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
OSIntExit()
OSTaskDel()
Task is Preempted
OSTaskDel()

Task Control Block

A task control block is a data structure called OS_TCB, that is used by µC/OS-II to store the state of a task when it is preempted.

When the task regains control of the CPU, data in the task control block allows the task to resume execution exactly where it left off.

All OS_TCB's reside in RAM.

The OS_TCB is initialized when a task is created.

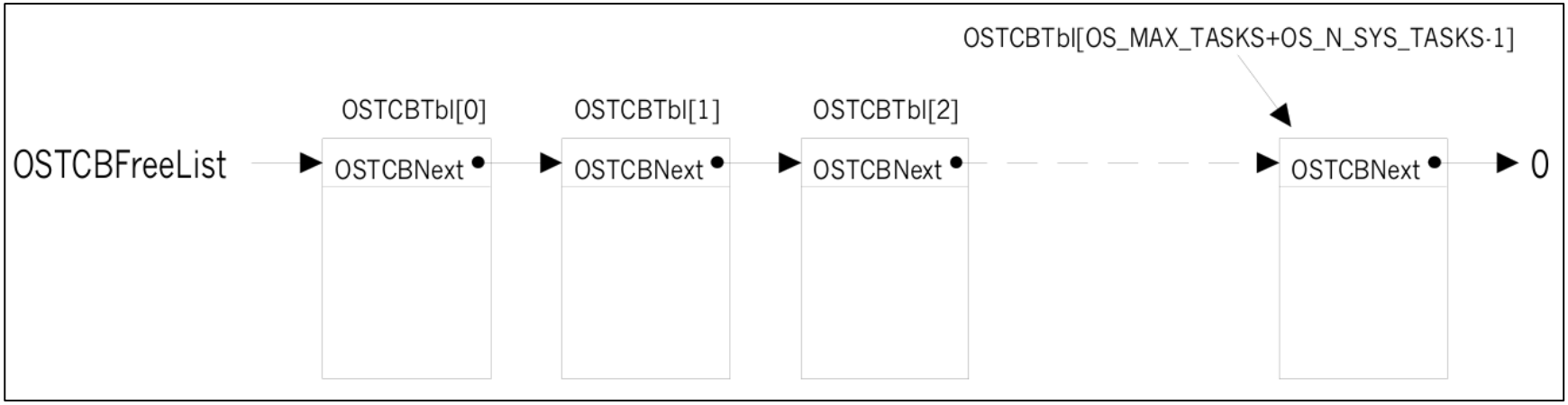
OS_TCB structure

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;

#if OS_TASK_CREATE_EX_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
```

OS_TCB structure (continued)

```
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    OS_EVENT       *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;
```

Free Task Control Blocks



Global constants and variables:

- **`OS_MAX_TASKS`**
- **`OS_N_SYS_TASKS`**
- **`OSTCBTbl[]`**
- **`OSTCBFreeList`**

Task priorities

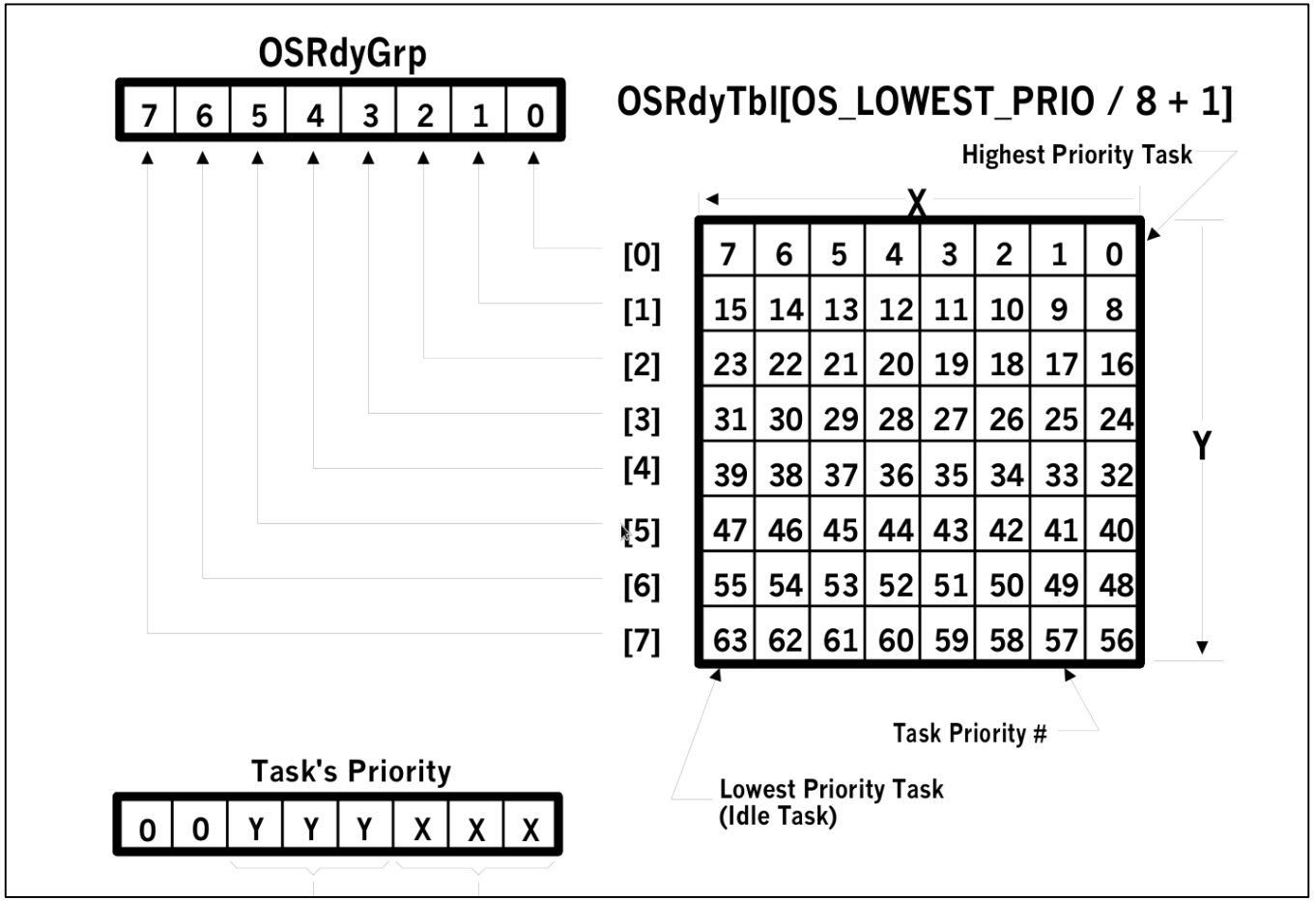
Each task is assigned a unique priority level between 0 (highest) and OS_LOWEST_PRIO inclusively.

Task priority OS_LOWEST_PRIO is always assigned to the idle task when µC/OS-II is initialized.

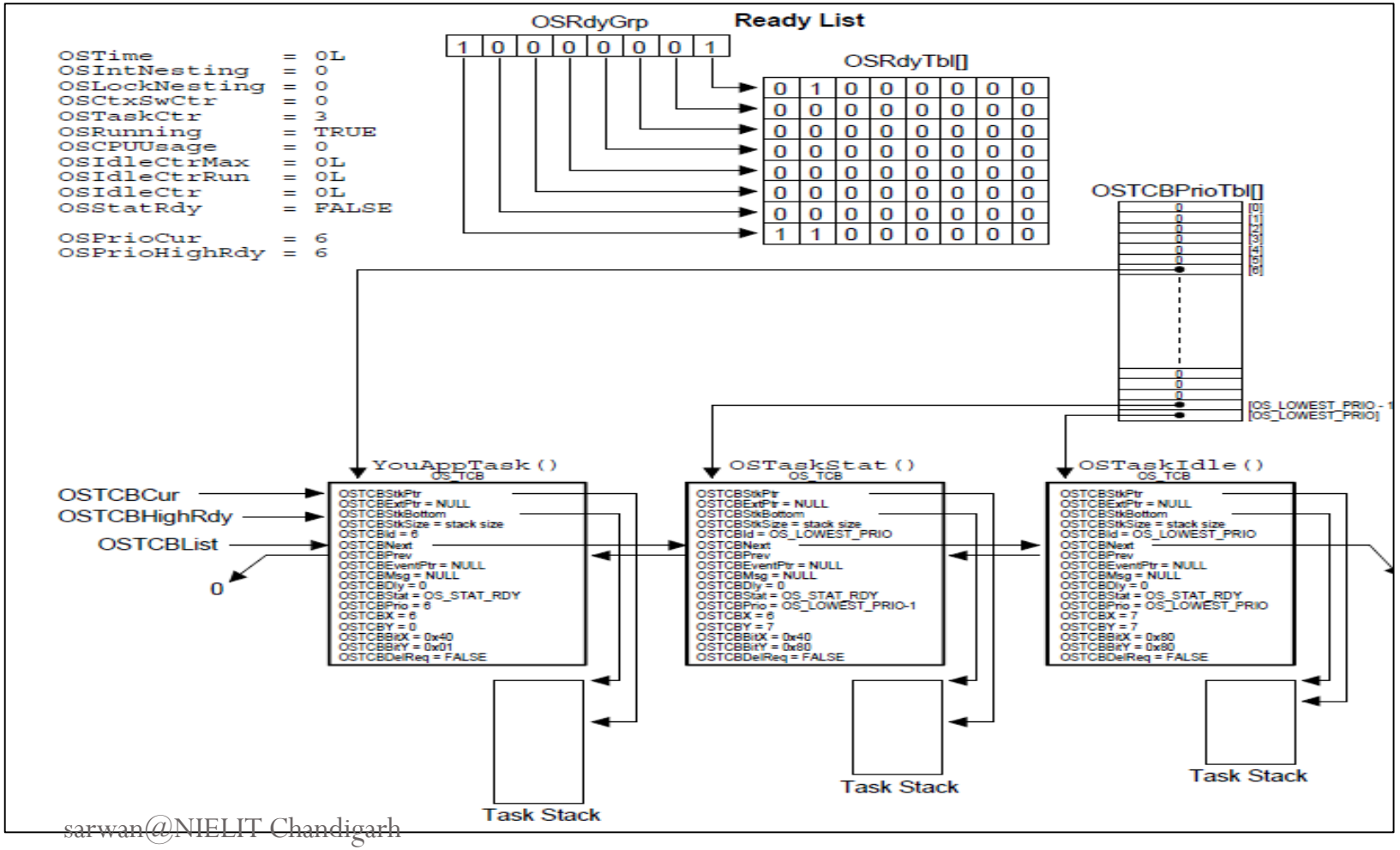
A task that is ready to run is represented by a '1' in the Ready List bitmap at a location according to its priority.

The task with the highest priority is found by searching the Ready List from position 0 upwards to position OS_LOWEST_PRIO (= 63)

The Ready List



Ready List with 3 tasks



Task insertion and deletion

To insert a task in the ready list:

```
OSRdyGrp |= ptcb->OSTCBBitY;
OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
```

To delete a task from ready list

```
if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0)
     OSRdyGrp &= ~ptcb->OSTCBBitY;
```

ptcb is a pointer to the task's TCB structure

OSTCBBitX and OSTCBBitY

OSTCBBitX and OSTCBBitY are pre-computed fields in the TCB structure, initialized when the task is created and the task's TCB is initialized

```
ptcb->OSTCBX = prio & 0x07;
ptcb->OSTCBY = prio >> 3;
ptcb->OSTCBBitX = 1 << ptcb->OSTCBX;
ptcb->OSTCBBitY = 1 << ptcb->OSTCBY;
```

ptcb is a pointer to the task's TCB structure, **prio** is the task's priority

Finding the highest priority task

To find the highest priority task ready to run

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

The highest priority task which is ready to run is found with three statements. The time to find the highest priority task does not depend on the priority of the task or on the number of tasks in the ready list.

OSUnMapTbl[]

`OSUnMapTbl[]` (in the file os_core.c) is a table in ROM, used to equate an 8-bit bitmask to an index from 0 to 7 as shown in the table below.

Bitmask (binary)	Index
0 0 0 0 0 0 0 0	-
0 0 0 0 0 0 0 1	0
0 0 0 0 0 0 1 0	1
0 0 0 0 0 0 1 1	0
0 0 0 0 0 1 0 0	2
0 0 0 0 0 1 0 1	0
0 0 0 0 0 1 1 0	1
0 0 0 0 0 1 1 1	0
0 0 0 0 1 0 0 0	3
etc.	etc.
(256 entries)	

Index is position of least-significant '1' in the bitmask

Task Scheduling

µC/OS-II always executes the highest priority task that is ready to run.

Which task has highest priority is determined by the scheduler.

Task level scheduling is performed by **`OSSched()`**

ISR level scheduling is performed by **`OSIntExit()`**

µC/OS-II task scheduling time is constant irrespective of the number of tasks created by the application.

OS_Sched()

```
void OS_Sched(void)
{
    INT8U y;

    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {
        y = OSUnMapTbl[OSRdyGrp];
        X = OSUnMapTbl[OSRdyTbl[y]];
        OSPrioHighRdy = (y << 3) + x;
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OS_TASK_SW();
        }
    }
    OS_EXIT_CRITICAL();
}
```

Context Switch

`OSTCBHighRdy` points to the OS_TCB of the highest priority task by indexing into **`OSTCBPrioTbl[]`**.

`OSCtxSwCtr` keeps track of number of context switches.

`OS_TASK_SW()` is the actual context switch function (a macro in the file os_cpu.h).

All of the code in **`OSSched()`** is considered a critical section.

Locking & Unlocking the Scheduler

The function **`OSSchedLock()`** is used to prevent task rescheduling until its counterpart, **`OSSchedUnlock()`**, is called.

The task that calls **`OSSchedLock()`** keeps control of the CPU even though other higher priority tasks are ready to run.

Interrupts, however, are still recognized and serviced (assuming interrupts are enabled).

The variable **`OSLockNesting`** keeps track of the number of times **`OSSchedLock()`** has been called to allow for nesting.

OS_SchedLock()

```
void OS_SchedLock(void)
{
    if (OSRunning == TRUE) {
        OS_ENTER_CRITICAL();
        OSLockNesting++;
        OS_EXIT_CRITICAL();
    }
}
```

OS_SchedUnlock()

```
void OS_SchedUnLock(void)
{
    if (OSRunning == TRUE) {
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) {
            OSLockNesting--;
            if ((OSLockNesting | OSIntNesting) == 0) {
                OS_EXIT_CRITICAL();
                OSSched();
            } else {
                OS_EXIT_CRITICAL();
            }
        } else {
            OS_EXIT_CRITICAL();
        }
    }
}
```

What is the Idle Task?

If there is no task, then the CPU will execute the idle task.

The idle task cannot be deleted because it should always be present in case there is no other task to run.

The idle task function **`OSTaskIdle()`** has the lowest priority, **`OS_LOWEST_PRIO`**.

`OSTaskIdle()` does nothing but increments a 32-bit counter called **`OSIdleCtr`**.

`OSIdleCtr` is used by the statistics task to determine how much CPU time (in percentage) is actually being used by the application task software

Idle Task

```
void OS_TaskIdle(void *pdata)
{
    pdata = pdata;
    for (;;) {
       OS_ENTER_CRITICAL();
       OSIdleCtr++;
       OS_EXIT_CRITICAL();
    }
}
```

Stack Checking

The OSTaskStkChk() function tells how much free stack space is available for your application.

To enable the µC/OS-II stack checking feature you must set OS_TASK_CREATE_EXT to '1' in file os_cfg.h

Create the task using OSTaskCreateExt() and give the task more space than you think it really needs.

Set 'opt' argument in OSTaskCreateExt() to OS_TASK_OPT_STK_CHK

Call OSTaskStkChk() from a task by specifying the priority of the task you want to check. You can inquire about any task stack, not just the running task.

Stack Checking (continued)

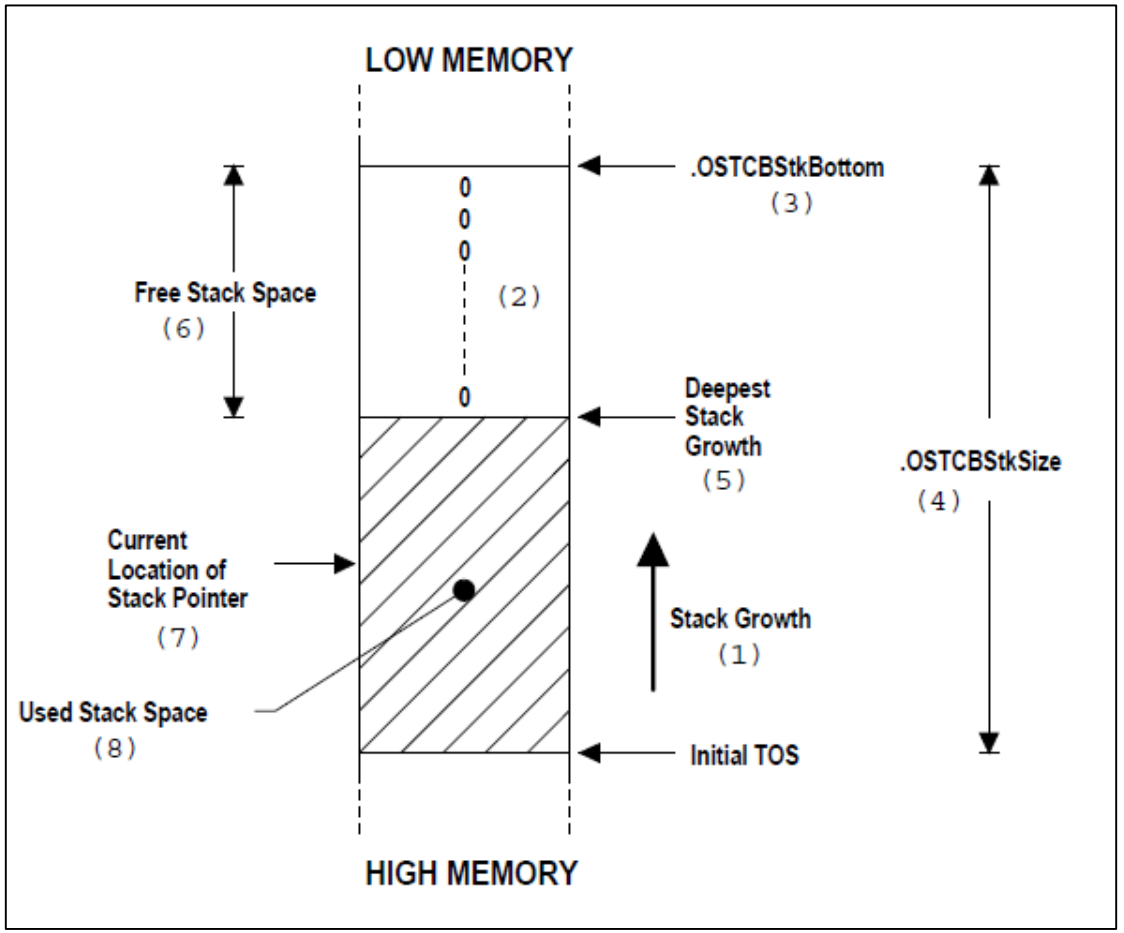
Assume `OS_STK_GROWTH` is set to 1 (Stack grows to lower memory addresses).

To perform stack checking, µC/OS-II requires that the stack is filled with zeros when the task is created

µC/OS-II needs to know the address of the Bottom-Of-Stack (BOS) and the size of the stack that is assigned to the task. These two values are stored in the task's OS_TCB when the task is created.

µC/OS-II determines the free stack space by counting the zero-elements from the Bottom-Of-Stack to the first no-zero element.

Stack Checking (continued)



Suspending a Task

Sometimes we need to explicitly suspend the execution of a task.

Suspension is done by calling OSTaskSuspend()

The suspended task can only be resumed by calling OSTaskResume()

Task suspension is cumulative.

A task can suspend itself or another task.

Resuming a Task

Sometimes we need to resume a task that we have suspended some time before.

A suspended task can only be resumed by calling OSTaskResume()

A task is resumed only if the number of calls to OSTaskResume() matches the calls to OSTaskSuspend()

A task cannot resume itself, only another task can do this.

Changing the task priority

Call the OSTaskChangePrio() function if you want to change priority of task at runtime.

Getting Information about a task

A task can retrieve information about itself or another application tasks by calling OSTaskQuery()

The function OSTaskQuery() gets a copy of specified task's OS_TCB.

To call OSTaskQuery(), your application must first allocate storage for OS_TCB.

After calling OSTaskQuery() this OS_TCB contains a snapshot of the OS_TCB for the desired task.

Requesting to delete a task

When a task deletes another task, there is a risk that there is no systematically de-allocation of the resources created by the other task.

This would lead to memory leaks which are not acceptable.

In this case it is required that a task tells the other task to delete itself when it is done with it's resources.

We can do this by using OSTaskDelReq() function.

Both the requesting task and task to be deleted must call OSTaskDelReq()

Deleting a Task

The OSTaskDel() function deletes a task, but that does not mean that we are deleting the code of the task from memory.

The task is returned to the dormant state. The task is simply no longer scheduled by µC/OS-II.

The OSTaskDel() function ensures that a task cannot not be deleted within an ISR.

Make sure you should never delete the idle task.

A task can delete itself by specifying OS_PRIO_SELF as the argument.

Task Statistics

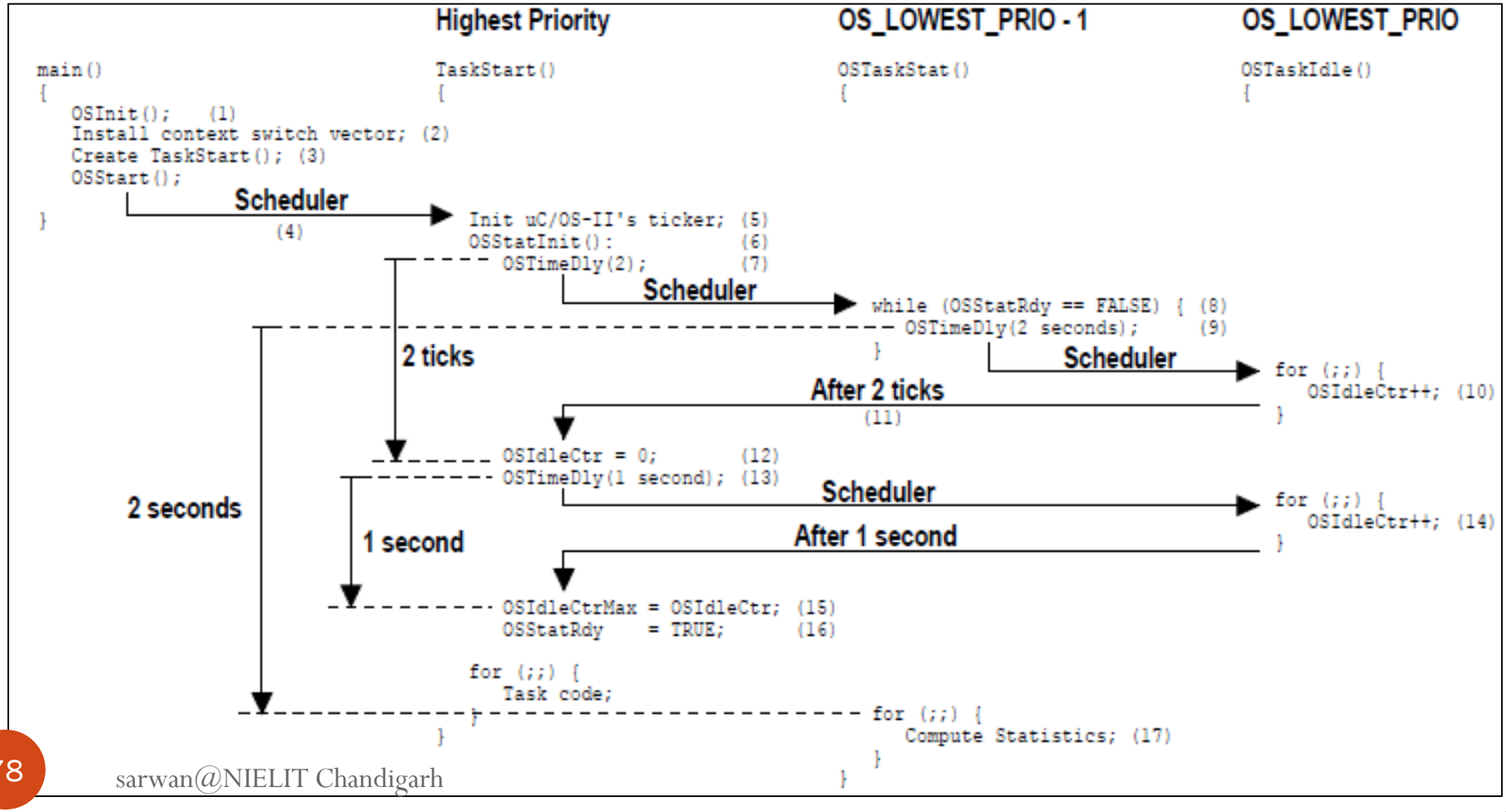
µC/OS-II contains a task, **`OS_TaskStat()`**, that provides run-time statistics and is created by µC/OS-II if the configuration constant **`OS_TASK_STAT_EN`** in the file os_cfg.h is set to 1.

When enabled, **`OS_TaskStat()`** runs once every second and reads the value of OSIdleCtr accumulated while running the idle task in the preceding second.

From this, and **`OS_IdleCtrMax`**, the value accumulated when only the idle process was running, **`OS_TaskStat()`** computes the percentage of CPU time used while running the application tasks.

This value is placed in an 8 bit signed integer variable **`OSCPUUsage`**. The resolution of **`OSCPUUsage`** is 1%.

Computing CPU usage



Statistics Task initialisation

```
void OSStatInit(void)
{
    OSTimeDly(2);
    OS_ENTER_CRITICAL();
    OSIdleCtr = 0;
    OS_EXIT_CRITICAL();
    OSTimeDly(OS_TICKS_PER_SEC);
    OS_ENTER_CRITICAL();
    OSIdleCtrMax = OSIdleCtr;
    OSStatRdy = TRUE;
    OS_EXIT_CRITICAL();
}
```

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

OSTaskStat()

```
void OSTaskStat(void *pdata)
{
    INT32U run;
    INT8S  usage;

    pdata = pdata
    while (OSStatRdy == FALSE) {
        OSTimeDly(2 * OS_TICKS_PER_SEC);
    }
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtrRun = OSIdlCtr;
        run = OSIdleCtr;
        OSIdleCtr = 0;
        OS_EXIT_CRITICAL();
```

OSTASKSTAT() (continued)

```
        if (OSIdleCtrMax > 0) {
            usage = 100 - run / (OSIdleCtrMax / 100);
            if (usage > 100) {
                OSCPUUsage = 100;
            } else if (usage < 0) {
                OSCPUUsage = 0;
            } else {
                OSCPUUsage = usage;
            }
        } else {
            OSCPUUsage = 0;
        }
        OSTaskStatHook();
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

Interrupts with µC/OS-II

µC/OS-II requires that an Interrupt Service Routine (ISR) be written in assembly language.

µC/OS-II needs to know that you are servicing an ISR and thus you need to either call **`OSIntEnter()`** or increment the global variable **`OSIntNesting`**.

µC/OS-II allows you to nest interrupts because it keeps track of nesting in **`OSIntNesting`**.

The ISR terminates by calling **`OSIntExit()`** which decrements the interrupt nesting counter.

```
YourISR:
    Save all CPU registers;
    Call OSIntEnter() or increment OSIntNesting directly;
    Execute user code to service ISR;
    Call OSIntExit() or decrement OSIntNesting directly;
    Restore all CPU registers;
    Execute <Return from Interrupt> instruction;
```

OSIntEnter()

The function **`OSIntEnter()`** notifies µC/OS-II about the beginning of an interrupt service routine.

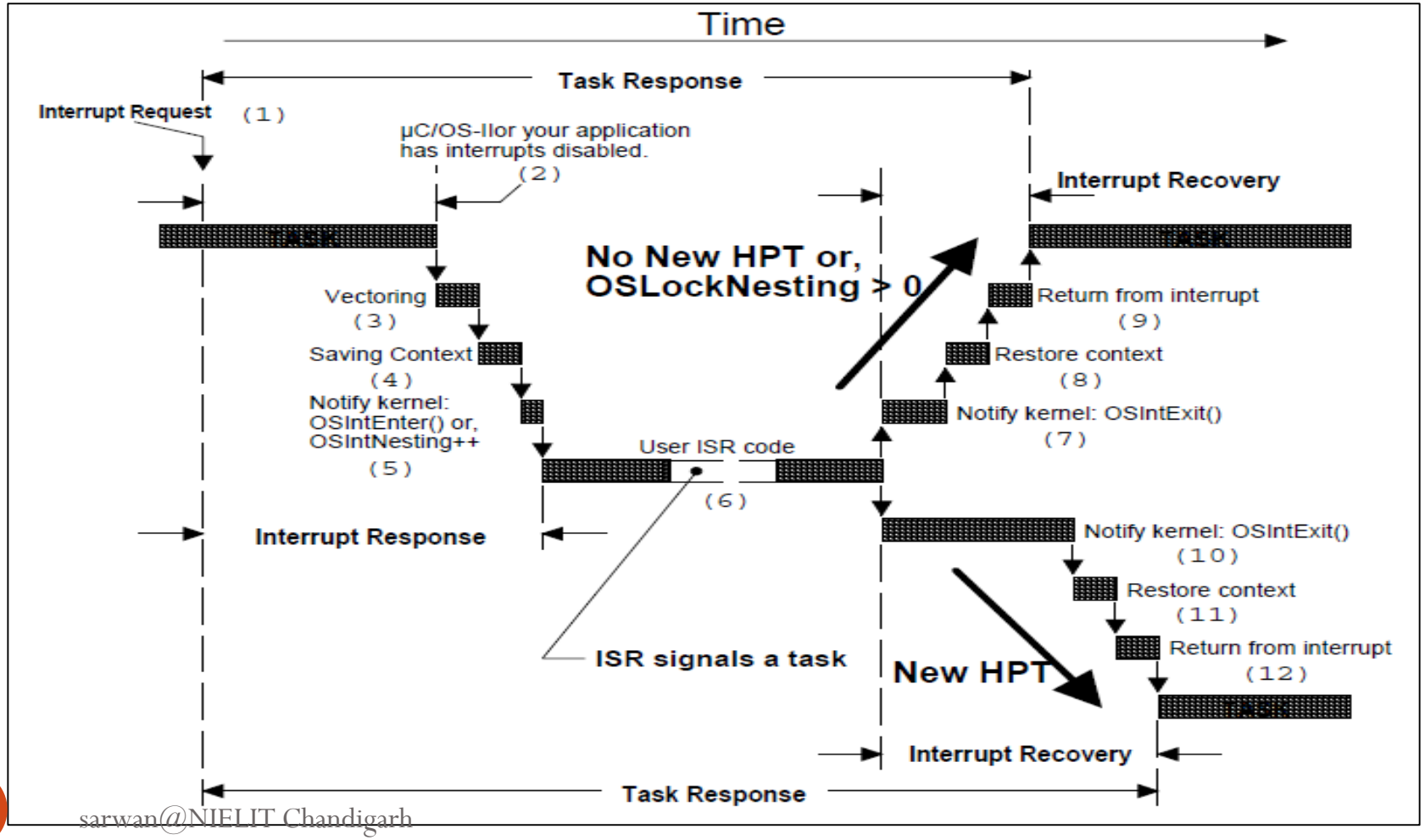
The function **`OSIntExit()`** notifies µC/OS-II about the ending of an interrupt service routine.

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

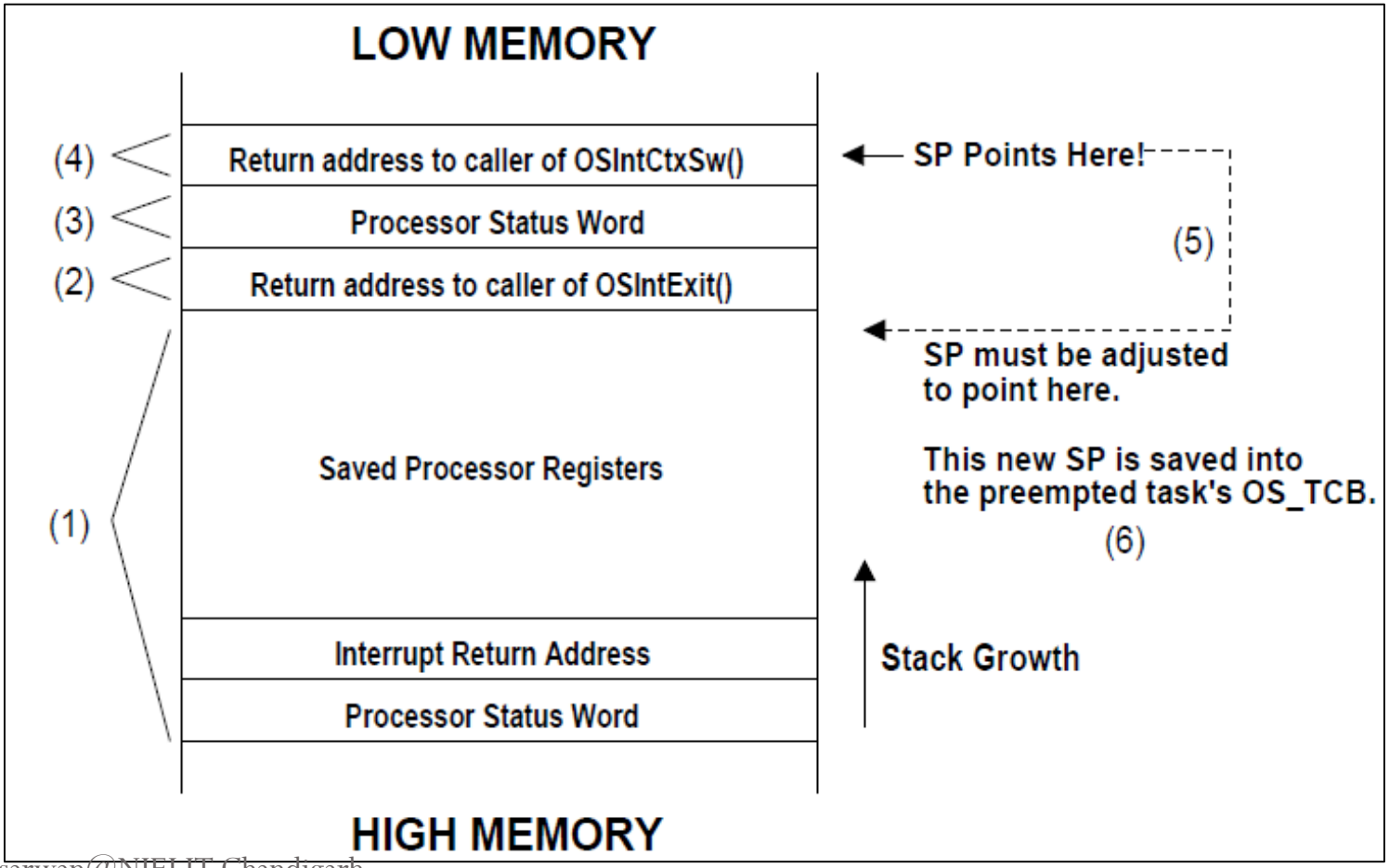
OSIntExit()

```
void OSIntExit(void)
{
    INT8U x, y;

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

Servicing an Interrupt



Stack cleanup by OSIntCtxSw()



Get µC/OS-II version

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

The version number returned by OSVersion() contains a major and a minor version number.

For instance, when OSVersion() returns 291, the real version number of µC/OS-II is 2.91

Semaphores

A semaphore consists of an integer counter and a wait list

OSSemPend():

- counter--;
- If the value of the semaphore < 0, then the task is blocked and moved to the wait list.
- A time-out value can be specified.

OSSemPost():

- counter++;
- If the value of the semaphore >= 0, then a task in the wait list is removed from the wait list.
- Reschedule if necessary.

Mailbox

A mailbox is for data exchange between tasks and consists of a data pointer and a wait list.

OSMboxPost():

- If there is already a message in the mailbox, then an error is returned (message not overwritten).
- The message (a pointer) is posted in the mailbox.
- If tasks are waiting for a message from the mailbox, then the task with the highest priority is removed from the wait-list and scheduled to run.

OSMboxPend():

- If the mailbox is empty, the task is blocked and moved to the wait-list.
- The message in the mailbox is retrieved.
- A time-out value can be specified.

Message Queue

A message queue can hold many data elements (on a FIFO basis) and consists of an array of elements and a wait list.

OSQPost():

- The highest-priority pending task (if there is one in the wait-list) is scheduled to run and receives the message.
- If there is no task in the wait-list, the message is appended to the queue.

OSQPend():

- If there is a message in the queue, the message is retrieved from the queue.
- If there no message, the task is moved to the wait-list and becomes blocked.

Hooks

A hook function will be called by µC/OS-II when the corresponding event occurs.

For example, OSTaskSwHook () is called every time a context switch occurs.

The constant OS_CPU_HOOK_EN must be set to 1 in file os_cfg.h to enable hooks.

The hook functions are defined in the file os_cpu_c.h

Hook functions

```
void OSInitHookBegin(void);
void OSInitHookEnd(void);
void OSTaskCreateHook(OS_TCB *ptcb);
void OSTaskDelHook(OS_TCB *ptcb);
void OSTaskIdleHook(void);
void OSTaskReturnHook(OS_TCB *ptcb);
void OSTaskStatHook(void);
void OSTaskSwHook(void);
void OSTCBInitHook(OS_TCB *ptcb);
void OSTimeTickHook(void);
```

Example Program

```
#include <ucos_ii.h>
#include <stdio.h>

#define STACKSIZE  256

/* Stacks */
OS_STK Task1Stk[STACKSIZE];
OS_STK Task2Stk[STACKSIZE];

/* Prototypes */
void Task1(void *);
void Task2(void *);

void main(void)
{
    OSInit();
    OSTaskCreate(Task1, OS_NULL, &Task1Stk[STACKSIZE], 10);
    OSTaskCreate(Task2, OS_NULL, &Task2Stk[STACKSIZE], 11);
    OSStart();
}
```

Example Program (continued)

```
void Task1(void *pdata)
{
    for (;;) {
       printf("  This is Task #1\n");
       OSTimeDlyHMSM(0, 0, 1, 0);
    }
}

void Task2(void *pdata)
{
    for (;;) {
       printf("    This is Task #2\n");
       OSTimeDlyHMSM(0, 0, 3, 0);
    }
}
```

How does this program work?

1. Declare stacks and define prototypes for the task functions Task1() and Task2()
2. In main(), call OSInit() to initialize µC/OS-II.
3. Create two tasks, Task1 with priority 10 and Task2 with priority 11.
4. Call OSStart() to start multitasking.

The functions for task #1 and task #2 are almost identical, both print “This is task #1”, resp. “This is task #2” and wait for 1 resp. 3 seconds to continue.

Running the program

We can run (simulate) this program with IDE68K.

In IDE68K set the Default directory to C:¥IDE68K¥OS Examples and the Include directory to C:Ide68k¥Include;C:¥Ide68k¥uCOSII.

Open existing project Example1(.prj).

Compile the project and run it on either the Command Line Simulator or the Visual Simulator.

See also the pdf-document *Programming with uCOS-II and Ide68k.*

This and other documents are accessible through the Help menu of IDE68K and can be downloaded from the IDE68K homepage, http://home.kpn.nl/pj.fondse/ide68k