Describing μC/OS-II RTOS

Guillaume Kremer

November 2009

Contents

1 Kernel **3**
- 1.1 Characteristics . . . 3
- 1.2 File Structure . . . 3
- 1.3 Critical sections . . . 5
- 1.4 Task . . . 6
- 1.5 Special tasks . . . 9
- 1.6 Managing tasks . . . 9
- 1.7 Tasks scheduler and tasks context switch . . . 11
- 1.8 Interrupts . . . 12
- 1.9 Initialization and start . . . 12
- 1.10 Time Management . . . 14
- 1.11 Event Control Block . . . 16

2 Programming model **18**
- 2.1 Tasks . . . 18
- 2.2 Semaphores . . . 20
- 2.3 Mutexes . . . 21
- 2.4 Event flag groups . . . 22
- 2.5 Mail Box . . . 23
- 2.6 Message queues . . . 25
- 2.7 Memory partitions . . . 26

3 OS Components and drivers **28**
- 3.1 μC/FS . . . 28
- 3.2 μC/TCP-IP . . . 28
- 3.3 Driver example . . . 29

4 Demo **30**
- 4.1 Demo1: Philosophers dinner on x86 . . . 30
- 4.2 Demo2: Philosophers dinner solution . . . 31
- 4.3 Compiler . . . 32
- 4.4 Configuration . . . 32

A Appendices **35**
- A.1 Dining philosophers implementation . . . 35
- A.2 Dining philosophers implementation, waiter solution . . . 39
- A.3 Driver example: serial driver . . . 46

Introduction

μC/OS-II is a hard real time kernel for embedded uses. It provides mechanisms for scheduling, synchronization and communication between processes and time management. It shares general characteristics with other OS but also exhibits specific schemes that need to be explained in order to be well used. A general overview is given here, with the goal to provide a fast programming start to every reader.

In this report, I will first describe general characteristics of μC/OS-II, then I will explain the inner working of the kernel. The third part will tackle interface provided for accessing kernel mechanisms such as semaphores or mail boxes. Four part will be about OS components and drivers.

Eventually I describe two demos made about the philosophers dinners which use threads, semaphores, mailboxes and messages queues, the compiler used and the configuration file used.

Chapter 1

Kernel

All the information and figures in this chapter have mainly been gathered from [6].

1.1 Characteristics

μC/OS-II is a multitask real time kernel. Up to 64 tasks can run at the same time. It is preemptive as it always runs the ready task with the higher priority.

The kernel is written in ANSI C so is portable. If one has the right cross compiler it is also romable. μC/OS-II is scalable since its footprint size is tunable according to application request. Eventually each task has its own stack that size can be reduced precisely.

Every routines have a known running time thus μC/OS-II is deterministic. The kernel provides services as semaphores, mutex, event flags groups, mailbox, messages queues. It also permits time management such as delays. μC/OS-II manages interrupts.

One can find the source code quite easily, anyway it is not completely open source and ports are often written for proprietary software.

Finally since July 2000, the kernel complies with Federal Aviation Administration and its RTCA DO-178B rules and is certified for avionic product.

1.2 File Structure

Still the sources can be seen as implementation details, file structure is important to understand platform specific and independent code. The figure 1.1 describes four types of class belonging to the software layer. Three lower level layers pertain to kernel sources, namely processor-independent code layer, application-specific code layer and processor-specific code layer.

Processor-Specific code contains functions used to interact with specific hardware and necessary for the execution of upper layer code. File describing context switching method and interrupt enabling methods belongs here.

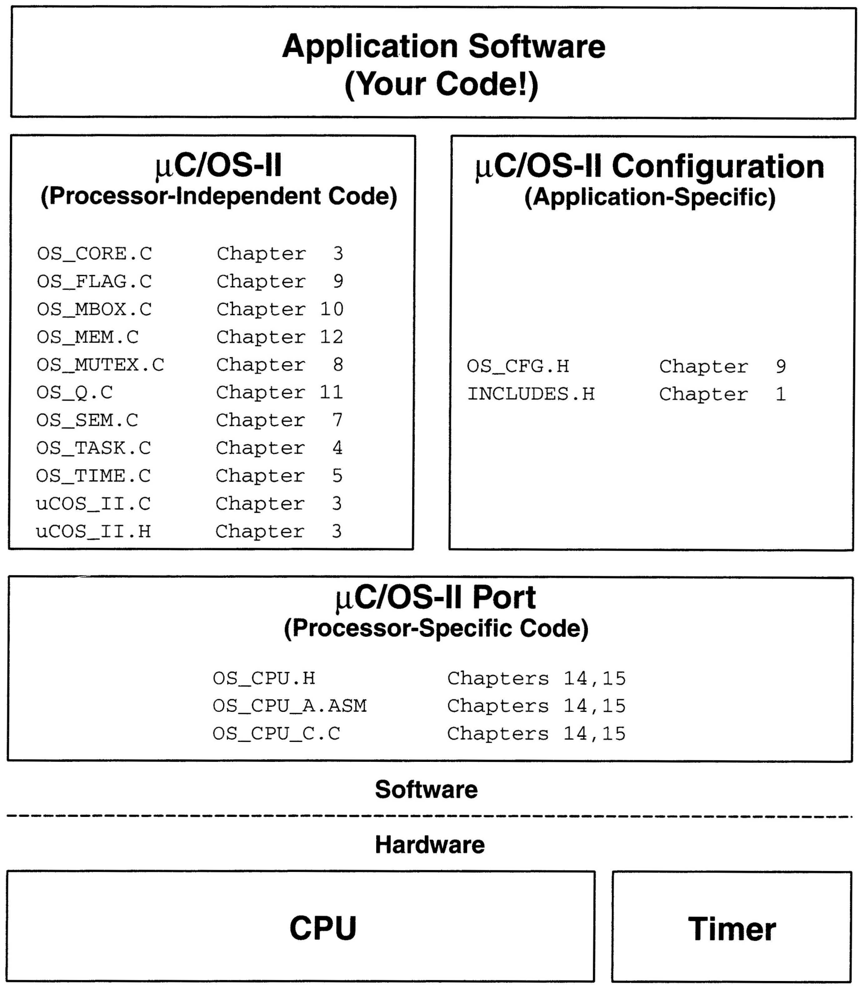


Figure 1.1: Code layers. Among μC/OS-II layers, processor-specific code is written in assembly language. Application-specific and processor-Independent code are written in C language. Porting μC/OS-II is writing assembly code for the new platform.

Processor-Independent code corresponds to the coding of high level mechanism such as memory management or semaphores. They rely on processor-specific code when needed.

Application-Specific methods correspond to configuration methods.
On top of the kernel, user software is running.

In the following sections, I describe the mechanisms the kernel provides and their internal.

1.3 Critical sections

Critical sections are code areas that need to run uninterruptedly and indivisibly. This is the case when multiple code areas need to access a shared resource as in the context of multitasking.

To overcome this, the kernel disable all interruptions at the beginning of a critical function , meaning that tasks cannot be scheduled and therefore current running task cannot be preempted. When the kernel leaves critical section, the kernel re-enable interruptions.

Processors can provide low-level instructions to disable interruptions. Disabling interrupts can be done either by use of inline assembly either by call of high-level language extensions. μC/OS-II offers an unified way to disable interruptions by providing macros OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL() that take care of low-level details.

Three methods exists to implements these macros. One has to choose the right method depending of processor and compiler capabilities. Since this choice depends of implementation details definitions of these macros are located in the file OS_CPU.H, see Fig.1.1 page 4.

The x86 port of μC/OS-II described in [6] and compiled with Borland C/C++ Compiler v4.5 uses the second method[5]:

```
#define  OS_CRITICAL_METHOD    2
```

First Method

The first method is to use processors instructions to enable or disable interrupts. This method is the simplest however when returning from a μC/OS-II service call with interrupts disabled, these will eventually be enabled. Therefore this is not the best implementation. However with some processors or compilers this is the only one possible.

Second Method

The second way is to save the interrupt disable status and then disable interrupts when entering critical sections. Implementation of OS_EXIT_CRITICAL() is done by restoring interrupt disable status. Therefore after a μC/OS-II routine call, interrupts status will be unchanged compared to before. Pseudocode for this macro is given in [6]:

```
#define OS_ENTER_CRITICAL() \
asm(" PUSH PSW") \
asm(" DI") \

#define OS_EXIT_CRITICAL() \
asm(" POP PSW")
```

PSW stands for processor status register whereas the DI instruction disables interruptions. This pseudocode assume that the compiler provide inline assembly and be able to figure that the stack pointer has been changed when pushing something on the stack. Otherwise if your compiler is using relative addressing mode, then all stack offset will be wrong after a call to OS_ENTER_CRITICAL.

The x86 port is:

```
#define  OS_ENTER_CRITICAL()  asm {PUSHF; CLI}
#define  OS_EXIT_CRITICAL()   asm  POPF
```

Third Method

The third method is relying on the compiler for allowing access to the processor interrupt status word and save it into a local variable. This is dependent of your compiler and thus the code is said to be not portable.

1.4 Task

A task is a program that thinks it has the CPU for itself[6]. A task is also referred to something that takes time instead as something that takes space as opposed to memory [3], [10] gives use a typical task:

```
void Task (void *p_arg)
{
    Do something with argument p_arg;
    Task initialization;
    for (;;) {
        /* Processing (Your Code)                       */
        Wait for event;     /* Time to expire ...     */
                            /* Signal from ISR ...   */
                            /* Signal from task ...  */
        /* Processing (Your Code)                         */
    }
}
```

A task runs as an infinite loop where all task behavior happened.

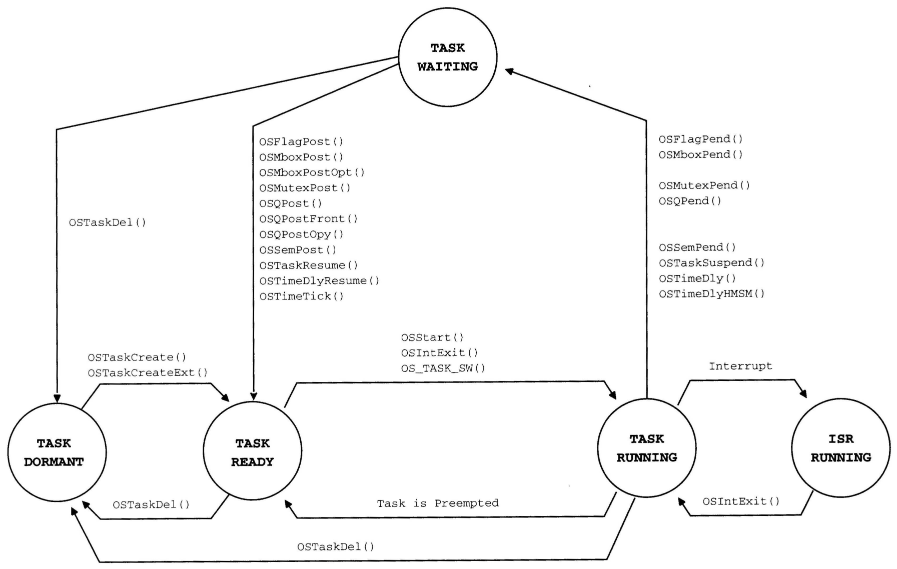


Figure 1.2: The five states a task can take. Task dormant state is set to tasks no currently running. Ready tasks are running tasks not waiting for an event and which have been preempted. Waiting tasks are tasks blocking for an event such as tasks waiting for semaphores or delayed tasks. Running tasks are started tasks not yet preempted. ISR running state is set to interrupted tasks waiting for the return from interrupt. Changing from one state to another is made by calling the functions displayed on the edges.

In μC/OS-II, a task has a state among the five described in Fig. 1.2.

Only one task can run at a time. However 255 tasks can cohabit within the same system. The order of execution is determined by task priority of each task. A task is represented in the kernel by the structure OS_TCB:

```
typedef struct os_tcb {
    OS_STK        *OSTCBStkPtr; /* Pointer to current top of stack */

#if OS_TASK_CREATE_EXT_EN > 0
    void          *OSTCBExtPtr; /* Pointer to user definable data for TCB extension */
    OS_STK        *OSTCBStkBottom; /* Pointer to bottom of stack */
    INT32U         OSTCBStkSize; /* Size of task stack (in number of stack elements) */
    INT16U         OSTCBOpt; /* Task options as passed by OSTaskCreateExt() */
    INT16U         OSTCBId; /* Task ID (0..65535)    */
#endif

    struct os_tcb *OSTCBNext; /* Pointer to next     TCB in the TCB list */
    struct os_tcb *OSTCBPrev; /* Pointer to previous TCB in the TCB list */

#if ((OS_Q_EN > 0) && (OS_MAX_BS > 0)) || (OS_MBOX_EN > 0)
|| (OS_SEM_EN > 0) || (OS_MUTEX_EN > 0)
    OS_EVENT      *OSTCBEventPtr; /* Pointer to event control block */
#endif

#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    void          *OSTCBMsg; /* Message received from OSMboxPost() or OSQPost() */
#endif

#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
#if OS_TASK_DEL_EN > 0
    OS_FLAG_NODE  *OSTCBFlagNode; /* Pointer to event flag node */
#endif
    OS_FLAGS       OSTCBFlagsRdy; /* Event flags that made task ready to run */
#endif

    INT16U         OSTCBDly;           /* Nbr ticks to delay task or, timeout waiting for event
    INT8U          OSTCBStat;          /* Task status                                  */
    INT8U          OSTCBPrio;          /* Task priority (0 == highest, 63 == lowest) */

    INT8U          OSTCBX;             /* Bit position in group  corresponding to task priority
    INT8U          OSTCBY;             /* Index into ready table corresponding to task priority
    INT8U          OSTCBBitX;          /* Bit mask to access bit position in ready table */
    INT8U          OSTCBBitY;          /* Bit mask to access bit position in ready group */

#if OS_TASK_DEL_EN > 0
    BOOLEAN        OSTCBDelReq;        /* Indicates whether a task needs to delete itself  */
#endif
} OS_TCB;
```

OSTCB structure contains information about tasks individual values. Each task has its own stack therefore one could find the top and base stack pointers along with its size.
OSTCBPrio is the priority value which identify a task. OSTCB? and OSTCBBit?

are precomputed values used for placing the task in the ready list. OSTCBDelReq flag keeps track about deletion request made from another task about it.

1.5 Special tasks

Idle Task

The idle task is running when no other task can run. It is the task with the lowest priority, i.e. the task with priority OS_LOWEST_PRIO. The idle task can never be deleted. The idle task continuously increments a statistic counter and makes a call to OSTaskIdleHook(). This function can be specialized by user to accomplish any actions.

Statistic task

μC/OS-II provides a task that computes CPU times statistics. This task is optional and is only created if the flag OS_TASK_STAT_EN is set to 1 in OSCfg.H. The statistic task always has a priority of OS_LOWEST_PRIO-1. The idle task computes the time percentage of CPU usage given by:

OSCPUSAGE (%)=(100 - OSIdleCtr/((OSIdleCtrMax)/100))

OSIdleCtr is a 32 bits counter incremented by the idle task when running
OSIdleCtrMax is the maximum value of OSIdleCtr

It then calls the function OSTaskStatHook(), that user can specialize.

1.6 Managing tasks

Free list

Before its main execution begins, the kernel allocates OS_MAX_TASKS OS_TCB structures placed in OSTCBFreeList.
When a new task is created, the kernel assignees it a free OS_TCB structure. To keep track with free and assigned OS_TCBs, the free control blocks are linked in a list pointed to by OSTCBFreeList. To assign a TCB to a task the kernel removes the first structure of the linked list and initializes it with a call to OS_TCBInit(), see the listing 1.4 page 8 for details. When a task is deleted, the corresponding OS_TCB is returned to the list.
The tasks linked in the free list are in a dormant states.

Ready list

The kernel keeps track of runnable tasks with ready list. A task in the ready state is in the ready list. In μC/OS-II a priority can only have one task associated. In the following discussion, priority can refer to the associated task and reciprocally.

The ready list is composed of OSRdyGrp and the table OSRdyTbl.
OSRdyGrp is a 8 bits structure where each bit corresponds to a different group of priorities.
OSRdyTbl is a table composed of $\frac{NB\_PRIO}{8}$ 8 bits values corresponding to priorities group. Each bit in the table corresponding to a different priority. Eight priorities

in OSRdyTBL compose a group and each priority belongs to one group. The relationship between OSRdyGrp and OSRdyTbl is shown in the Fig.1.3.

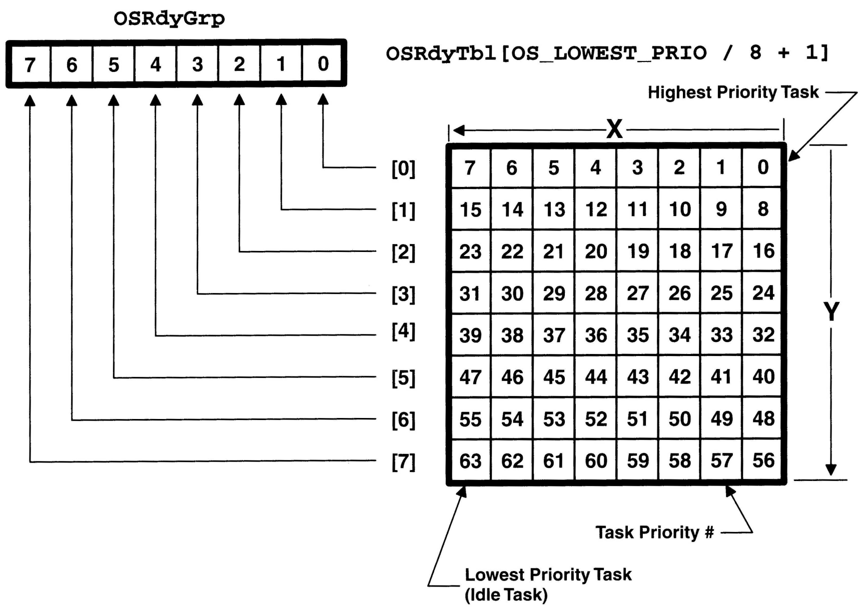


Figure 1.3: Each lines in OSRdyTbl corresponds to a bit in OSRdyGrp. Priorities increase toward the right and the up with zero being the highest priority task.

[6] gives us the rules for assigning bits in OSRdyGrp to priorities group:

```
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.
```

The group of a task is given by the priority field in its task control block. Bits 3,4,5 corresponds to the index in the OSRdyTbl and the bit position in OSRdyTbl. Bits 0,1,2 corresponds to the bit position in OSRdyTbl at index. When the task corresponding to a priority is ready, it switches on the bit of OSRdyGroup corresponding to its group of priorities. Similarly When a task switches to a state other than ready, it zeroes the corresponding bit.

To determine the next task to run, the kernel first has to check the lowest ready bit with value 1 corresponding to the highest priority group with at least one runnable task and then find out the highest priority task within. The maximal number of task scanned corresponds to the size of OSRdyGrp and so is 8.

As an example, if OSRdyGrp is 01100100 then the index in OSRdyTbl is 2. If OSRdyTBL contains 01110010 the task to run is the task with priority 3*8+1 = 25.

1.7 Tasks scheduler and tasks context switch

Scheduling

The task with the highest priority always run. Choosing the right priority is done by the scheduler in OS_SCHED(). Note that the scheduler only does task scheduling, interrupt service routines are scheduled by another function.

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


    OS_ENTER_CRITICAL();
    if ((OSIntNesting == 0) && (OSLockNesting == 0)) { /* Sched. only if all ISRs done &
not locked    */
        y             = OSUnMapTbl[OSRdyGrp];          /* Get pointer to HPT ready to run */
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {              /* No Ctx Sw if current task is highest
 rdy     */
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;                              /* Increment context switch counter */
            OS_TASK_SW();                              /* Perform a context switch */
}
    }
    OS_EXIT_CRITICAL();
}
```

Listing of the OS_SCHED() function from file OS_SCHED.C.

Interrupts are disabled during scheduling, so that ready lists are protected. If interrupts were enabled, a race condition could appear, making it possible for a thread to modify a ready bit during scheduling and thus to enable a lower priority task to run.

The scheduling consists of five steps:
1-The scheduler checks if it has not been called from an ISR or if the scheduling lock has been activated
2-The scheduler gets the highest priority task in the ready list.
3-The scheduler checks that the current task and the task found in the previous step are different. This is done to avoid useless commutation.
4-The commuting function called in step 5 uses the variable OSTCBHighRdy which is set here to the TCB corresponding to the highest priority task.
5-The scheduler commutes with OS_TASK_SW() for commutation

Task context switch

The context of a task is the contents of all the CPU registers. A context switch suspend the current task pointed to by OSTCBCUR and resumes the highest priority task. It saves the current content on the stack of the running stack and restores the saved context of the new task.

Task Context switch is implemented by OS_TASK_SW() that calls the architecture specific function OSCtxSW(). OSCtxSW copies and restores all the registers and takes care of changing the pointers to TCB structures. It then executes a return from interrupt.

1.8 Interrupts

In μC/OS-II, ISR are written in assembly language. As for the task, Interrupt Service Routine follow a general pattern. It first saves all the register onto the current stack frame or on a different stack, depending of the processor. The ISR notifies the kernel about itself to avoid nested interrupts. This is done by calling OSIntEnter() or by incrementing OSIntNesting. Note that some processors allow nested interrupts and thus the kernel doesn't have to be notified.

The stack pointer is saved in OsTCBCur$\rightarrow$OSTCBStkPtr. The ISR clears the interrupt source before re-enabling them to be sure not to call the same interrupt again.

The interrupt is serviced, this is the application specific part of the ISR. The interrupt nesting counter is decremented by calling OSIntExit(). This function checks that the counter is greater than 0. In the other case, a higher priority task has been spawn by any ISR. If so μC/OS-II returns to the higher priority task instead of returning to the interrupted one.
If no higher priority task exists, the ISR continues to execute and restores the registers. Eventually the ISR returns from interrupts.

Fig. 1.4 taken from [6] shows a situation case after an interruption.

1.9 Initialization and start

Prior to run tasks, the kernel requires OSInit() to do initializations. OSInit() initializes data structures and variables and start the idle and statistics tasks discussed before in 1.5. OSINIT() has to be called before creating any tasks.
OSInit() creates five free pools implemented as singly linked list. OSTCBFreeList links OS_MAX_TASKS OSTcbs used to create new task, OSEventFreeList which is a list of OS_MAX_EVENTS events structure, OS_Q contains OS_MAX_QS queues, OS_FLAG_GROUP links OS_FLAG_GRP structures and OS_MEM has OS_MAX_MEM_PART mailboxes. Variables are defined in OS_CFG.H.

At least one task needs to be created before starting multitasking. The kernel is started with OSStart(). It finds the highest priority task created, see scheduling

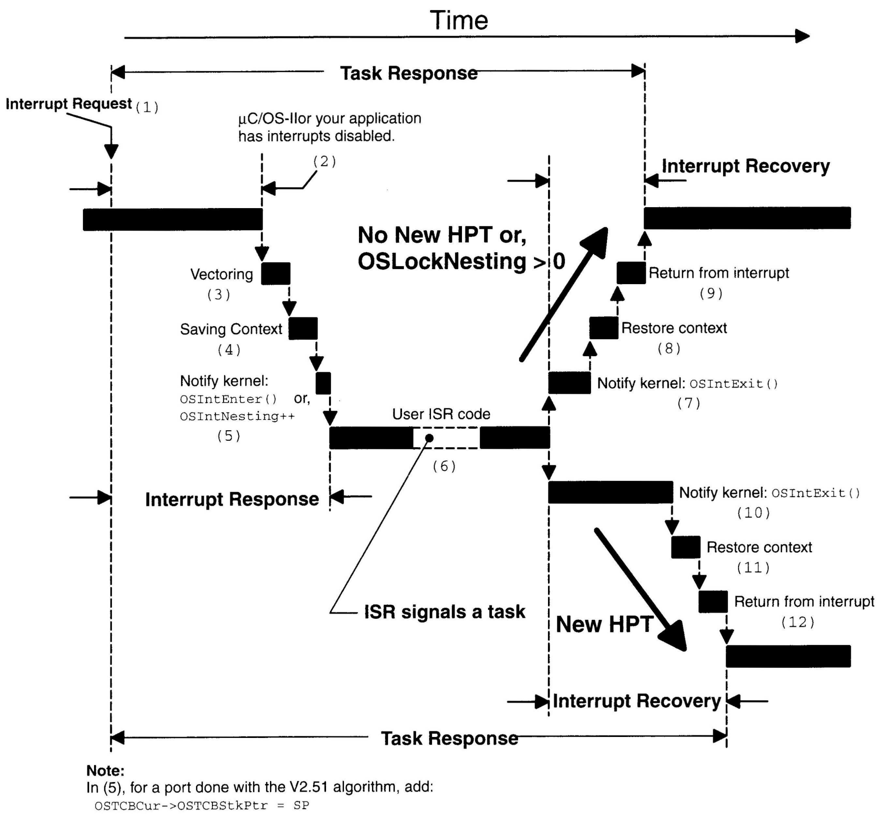

Time
Task Response
Interrupt Request (1)
µC/OS-IIor your application has interrupts disabled.
(2)
Interrupt Recovery
No New HPT or, OSLockNesting > 0
Vectoring (3)
Saving Context (4)
Notify kernel: OSIntEnter() or, OSIntNesting++ (5)
User ISR code
(6)
Return from interrupt (9)
Restore context (8)
Notify kernel: OSIntExit() (7)
Interrupt Response
Notify kernel: OSIntExit() (10)
Restore context (11)
Return from interrupt (12)
ISR signals a task
New HPT
Interrupt Recovery
Task Response
Note:
In (5), for a port done with the V2.51 algorithm, add:
OSTCBCur->OSTCBStkPtr = SP

Figure 1.4: Event flow after when servicing an interrupt. Actions 1, 2 and 3 are made by the kernel to handle the interrupt and vector it to the right ISR. Actions 4 and 5 are taken by the ISR to save all registers and signal itself. Action 6 is the basic ISR code. Actions 7, 8, 9 and 10, 11, 12 belong to two different cases. First case is no higher priority task(HPT) has been triggered, OSIntExit() takes less time to return from interrupt. In the second case, an HPT has been triggered, and so OSIntExit() takes longer time because a context switch is done.

for details. It then executes OSStartHighReady() which restores registers from the task space and executes a return from interrupt. This triggers the execution of the higher priority task. OSStartHighReady() never returns to OSStart().

The figure 1.5 presents the variables and data structure after multitasking has started.

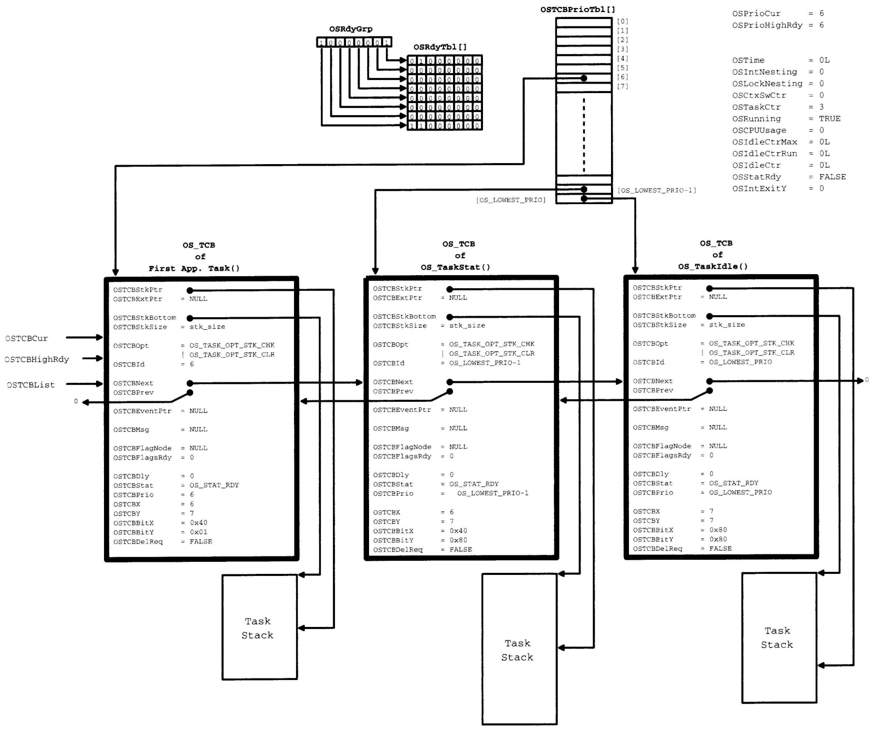


Figure 1.5: The ready list is detailed along with different values. The current system has 3 tasks running, among them the idle and the statistics tasks. The current priority of the running task is 6.

1.10 Time Management

An interrupt occurs periodically at a rate given by the system. The rate must be superior to 10Hz and inferior or equal to 100Hz and is fixed in OS_CFG.H. OSTimeClick() is called when a clock tick occurs.

μC/OS-II provides five functions to interact with time in tasks.

OSTimeDly()

A task can delay itself for a specified amount of clock ticks with OSTimeDly(). It is called in a task and causes a context switch. The high priority task ready to run is scheduled. The suspended task is resumed after the time occurs or when a task calls OSTimeDlyResume() if the task is the highest priority ready.

Calling OSTimeDly() with a value of 0 has no effect. The maximum value is 65535.

An important precision has to be brought. Given the case where a task calls OSTimeDly() with a value of 1 and the processor is not very loaded, the task is likely to be waiting for a time inferior of a time cycle. This has to be with delay resolution inferior to 1.

OSTimeDlyHMSM()

This function is similar to OSTimeDly() in that it permits to delay a task. However it has to be called with hour, minutes, seconds and milliseconds arguments. Again the suspended task can be wake if the time expires or if another task calls OSTimeDlyResume() given that the task is the highest priority ready task.

An important limitation of OSTimeDlyHMSM is that a task delayed with it shouldn't be resumed by OSTimeDlyResume(). Especially when the corresponding number of ticks to wait is greater than 65535.

This is given by the following excerpt from OSTimeDlyHMSM() in the file OS_TIME.C.

```
loops = (INT16U)(ticks / 65536L); /* Compute the integral number of 65536 tick delays*/
ticks = ticks % 65536L; /* Obtain  the fractional number of ticks*/
OSTimeDly((INT16U)ticks);
while (loops > 0) {
    OSTimeDly(32768);
    OSTimeDly(32768);
    loops--;
}
```

where ticks is the corresponding number of clock cycles to wait. If loops is greater than zero and so if ticks is greater than 65536, it calls OSTimeDly() several times. Therefore a simple call to OSTimeResume() is useless.

OSTimeResume()

A suspended task can be wake before its delay expires by another task calling OSTimeResume() with the priority of the suspended task.

The OSTimeDlyResume() can also resume a task waiting for an event, this is described in 1.11. A task is waiting for an event for a limited amount of time. Calling OSTimeDlyResume() on this task causes a timeout to occurs.

A task can suspend itself for a limited amount of time without calling OSTimeDly() neither OSTimeDlyHMSM(). It simply creates a semaphore, mutex, mailbox or queue with a specified timeout. Resuming such a task is done by incrementing the mutex for example. This techniques needs the creation of an event control block and which waste ram.

OSTime value

The kernel keeps the number of clock ticks that occurred since multitasking has been started with OSStart(). The value resets to zero after 4,294,967,295 ticks occurs.
One can get the value of the OSTime counter with OSTimeGet(). Setting OSTime is done with OSTimeSet and a 32 bits unsigned integer value.

1.11 Event Control Block

Tasks and ISRs can interact by means of signals sent to event control block. ECB can be used to synchronize tasks also semaphores, mutex or mailbox are specialized ECBs.

An ECB is :

```
typedef struct {
    INT8U   OSEventType; /* Type of event control block (see OS_EVENT_TYPE_???)      */
    INT8U   OSEventGrp; /* Group corresponding to tasks waiting for event to occur  */
    INT16U  OSEventCnt; /* Semaphore Count (not used if other EVENT type)           */
    void   *OSEventPtr; /* Pointer to message or queue structure                    */
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur
} OS_EVENT;
```

An ECB is an event type which sets how the block is used for. Possible values are OS_EVENT_TYPE_SEM, OS_EVENT_TYPE_MUTEX, OS_EVENT_TYPE_MBOX and OS_EVENT_TYPE_Q. When used as a mailbox or a message queue, OSEventPtr is the mailbox or queue containing the event. OSEventTbl and OSEventGrp are the same as OSRdyTbl and OSRdyGrp in a ready list, they refer to tasks waiting for an event. OSEventTbl and OSEventGrp constitute a wait list.
OSEventCnt is a count when the type is a semaphore and the priority value when the type is a mutex.

Waiting list

Each task waiting for an event are placed in the wait list according to their priority. When the event occurs, the task waiting with the higher priority is set ready to run.
OSEventTbl is a 8-bits value array of size i$\frac{OS_LOWEST_PRIO}{8}$+1. OSRdyGrp is a 8 bits-long value where each bit is mapped to a specific index in OSEventTbl. When a task is waiting for an event, it's bit in OSEventTbl is set to 1. Similarly to the ready list system, the bit in OSEventGrp corresponding to the index in OSEventTbl is set to 1 if at least one of the bit of the 8-bits value at OSEventTbl at index is set.

Since ready lists and waiting list are similar, Fig. **??** is a good approach to understand wait list.

Adding a task in the waiting list

Adding a task is done by:

```
pevent->OSEventGrp |= OSMapTbl[prio >> 3];
pevent->OSEventTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

Since it only uses shift and OR operations, the time required to insert a task is constant. The index in OSEventTbl and OSEventGrp are given by the prio field of the task, the Figure 1.6 ref gives further details about it. As for the ready group OSMapTbl is used. Finding the highest task to run and removing a task in the

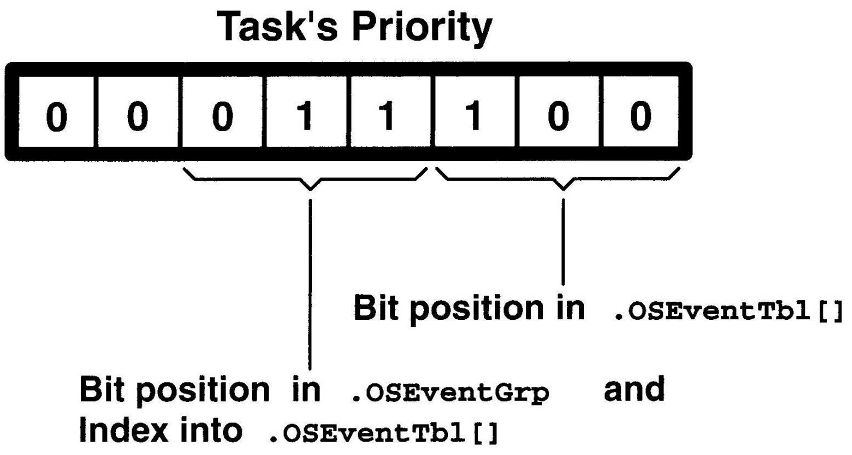

Task's Priority
0 0 0 1 1 1 0 0
Bit position in .OSEventTbl[]
Bit position in .OSEventGrp and
Index into .OSEventTbl[]

Figure 1.6: The bit position in the OSEventTbl is computed with the 3 lower bits whereas the bit position in OSEventGrp and the index in OSEventTbl are computed with bits 3, 4, 5

waiting list are similar to operations on ReadyGrp and are not detailed here.

ECB Free list

The kernel maintains a list of free ECBs. This list is a linked list initialized when OSInit() is called, initializations has been described in 1.9 . When a new semaphore, mutex, mailbox or queue is created, the first free event control block in the free list is removed and initialized. When deleting a semaphore, mutex, mailbox or queue, the ECB is returned to the free list.

Programming model for ECB

A user can call four functions to act on ECBs. OS_EventWaitListInit() initializes an ECB by signaling that no task is waiting. OS_Event_TaskRdy() is responsible for finding the highest priority task in the wait list, changes its state and removes it from the wait list.
OS_EventTaskWait() is called to remove a task from a ready list and put it in a wait list. OSEventTo() makes a task ready to run because a timeout occurs before the event was triggered.

Chapter 2

Programming model

2.1 Tasks

Task as been tackled before in **??**. As said before, a task is represented by OSTCB in the kernel. The kernel keeps a list of free OSTCBs and so no OSTCB is allocated at run time. Each task as a unique priority and each tasks has its own stack. Before multitasking start, at least one task must be running.

In the following discussion, stack checking is introduced, it consists to check if the space specified for the stack is sufficient.

Creating a task

OSTaskCreate() and OSTaskCreateExt() create new tasks. OSTaskCreateExt() is an extended version.

The first function takes four arguments which are a pointer to the function executed by the task, some data for the task, a stack and the priority for the new task. OSTaskCreate() initializes the stack by calling OSTaskStkInit() and get an initialized OSTCB with OSTCBInit(). The priority given for the task must not be used yet by another task.

OSTaskCreateExt() takes five more arguments that OSTaskCreate(). These are an id, a pointer to the task's bottom-of-stack, a stack-size, a pointer to a data area and options. The id arg is given for extension of μC/OS and is not used. The second argument is used to perform stack-checking. The stack-size specifies the number of elements of the stack. Options specifies details such are cleaning the stack before starting the task, or if the task uses floating points arithmetic.

Deleting a task

OSTaskDel deletes a task by is priority. It first checks if the task exists and if it is not the idle task. It removes the task from the event waiting list it belongs to if so. The state of the task is set to OS_STAT_RDY and the task counter is decremented. The OSTCB is returned to OSTCBFreeList. The function calls the scheduler and return OS_NO_ERR.

Requesting to delete a task

A task can request another task to delete itself and so gives time to it to delete all the memory it allocates or any semaphores it acquired. This is done with OSTaskDelReq(). The function takes a priority as argument. If the priority is different from the one of the current task, it asks the task with that priority to delete by setting the OSTCBDelReq in the OSTCB of the target task to OS_TASK_DEL_REQ. If the priority is equal to the one of the current task, it returns the value of OSTCBDelReq in OSTCBCur. A task requested for deletion can do various action such as returning memory or closing buffers to delete itself properly.

Stack allocation

It is better to allocate the stack statically and not dynamically. Static means at compile time and dynamically means by calling malloc() C function for example. This is to avoid memory fragmentation which is the result of dynamic memory allocation and freeing that data.

Stack checking

Stack checking detects if the stack size is sufficient for its task. OSTaskStkChk() is responsible for doing the stack checking. It checks for zero value on the stack and fills the OS_STK_DATA structure given in argument. OS_STK_DATA contains information about the stack.

[6] gives advices about running this function. It must be run during a long time and exhibit the behavior of the function in the worst case. It is also recommended to allocate between 10 and 100 percent more size for the stack than measured with OSTaskStkChk().

Changing the priority of a task

OSTaskChangePrio() changes the priority of a task. The old and new priority are given in arguments. The priority of idle task cannot be changed. The function changes the priority in the OSTCB of the task with old priority. The position of the task in the ready list is changed to the new one if the task is ready. OSTCBY, OSTCBX and all the values corresponding to position in the ready list are also recomputed . The function checks that the task doesn't wait for any event. If so it changes its position in the wait list of the ECB. Eventually before returning with the code OS_NO_ERR the function calls the scheduler.

Suspending and resuming a task

A task can be suspended with OSTaskSuspend() and resumed with OSTaskResume(). Suspend a task means deleting it from the ready list and set its state to OS_STAT_SUSPEND. Resuming a task changes its state to OS_STATE_READY and puts it in the ready list. It then calls the scheduler.

Getting information on a task

OSTaskQuery() gives a snapshot from a task. It fills an allocated OSTCB given as argument along with the task priority. The OSTCB returned is a copy of the task with the priority specified.

2.2 Semaphores

A semaphore is used when tasks want to concurrently access shared limited resources. A semaphore is made from a 16 bit counter and of a list of task waiting for the semaphore to be available. The maximum number of resources is 65535.
Semaphore are specialized event control blocks with the OSEventType field set to OS_EVENT_TYPE_SEM. μC/OS-II provides functions to create, delete, wait on or signal a semaphore. Waiting on a semaphore can be blocking or no.

Creating and deleting a Semaphore

Semaphores are created by calling the function OSSemCreate with the count value. OS_Semcreate takes the first empty ECB from the ECB free list and initializes it by setting the count value and the correct type. The function returns a pointer to the ECB.

Deleting a semaphore is made with the OSSemDel function that takes the event control block, an option for deleting and an integer pointer for error output. The function first checks that the ecb is a semaphore. If opt is set to OS_DEL_NO_PEND it then checks if it exists tasks waiting for the semaphore. If so it simply fill the error output with OS_ERR_TASK_WAITING and returns. If no other tasks are waiting or if opt is OS_EXIT_CRITICAL it then sets the type of the ECB to OS_EVENT_TYPE_UNUSED and returns it to the ecb free list.

Waiting for a semaphore

Waiting on a semaphore can be blocking or not. Waiting with blocking is done with the function OSSemPend, that takes the event block of the semaphore, a timeout and an error address.
After ISR checks have bee done the function checks the count of the semaphore. If the count is greater than zero, it decrements it and return. If no resources are available, it changes the state of the current task to OS_STAT_SEM and sets its timeout. The scheduler is then called.

When the scheduler returns, if the task is still waiting on the semaphore and the timeout has expired, it is removed from the wait list of the semaphore and the error is set to OS_TIMEOUT.
Waiting without blocking is called with OSSemAccept which decreases and returns the cnt field of the semaphore in argument.

Signaling a semaphore

One can signal a semaphore with OSSemPost. It checks if the ecb given in argument is not empty and that it is the type OS_EVENT_TYPE.
If tasks are waiting for the semaphore, it wakes the take with the highest priority in the wait list and calls the scheduler. When the scheduler returns or if there was no tasks waiting, if the count is inferior to 65535, it increments. It then returns.

Querying

One can query information about a semaphore with OSSemQuery. It takes as argument an event and a data structure. The data structure is filled and returned by the function with the characteristics of the semaphore.

2.3 Mutexes

Mutexes provide exclusive access to resources. In μC/OS-II mutexes are made to avoid the priority inversion problem described in **??**. Mutexes share a lot of code with semaphores, therefore a fast study of its structure is done here.

Avoiding priority inversion

Mutexes have a special priority value hold in OSEventCnt field. It is used when a task wanting to acquire a mutex is blocked because another task with lower priority has already acquired it. As μC/OS-II is a hard real time kernel, the lower priority task won't run and the tasks will be waiting forever. To solve that problem, the task with lower priority gets the priority hold by the event control block. This increases the chances to unblock the mutex. When the lower priority task frees the mutex, it gets back its initial priority.

Implementation

OSEventCnt value contains both the priority value called pip and the mutex value. The pip value corresponds to the upper 8 bits and the state to the others bits:

```
pevent->OSEventCnt = (prio << 8) | OS_MUTEX_AVAILABLE;
```

Creating a mutex is done with OSMutexCreate(). A major part of it is the same as OSSemCreate(). OSMutexCreate() is called with priority value.

Waiting on a mutex in a blocking way is made with OSMutexPend(). It checks if the ecb has the right type and if the function has not been called from an isr. If the mutex is available, the current task write its priority in the lower 8 bits of the OSEventCnt field of the ecb. The current task then acquires the semaphore and returns.
If the mutex is unavailable and the pip has not been set yet, the priority of the task who has the mutex is compared with the priority of the current task wanting the semaphore. If the task wanting the semaphore has a upper priority, the priority of the task which has the mutex is set to pip and the waiting list for the mutex is updated. The function then calls the scheduler.

Signaling a mutex is done with OSMutexPost(). OSMutexPost() is the same as OSSemPost() excepts it checks that the tasks which got the semaphore had its priority increased. If so it set it recovers its original value. The function then checks if tasks are waiting on the mutex. If so, the highest priority task acquires the mutex.

OSMutexDel(), OSMutexQuery(), OSMutexAccept() behaves the same has respectively OSSemDel(), OSSemQuery() and OSSemAccept() since priority inversion problem doesn't appear here.

2.4 Event flag groups

Event flag can be used by task to wait for events. An Event flag is a bit-field which are states of the events and a list of tasks waiting for one or several events of that group. Only tasks can wait for event. ISRs can only signal.

Internals

The wait list is different from the one find in the semaphore and mutex. It consists of a doubly linked list of OS_FLAG_NODE which defines how a specific task is waiting for events:

```
typedef struct {                    /* Event Flag Wait List Node */
    void         *OSFlagNodeNext; /* Pointer to next     NODE in wait list */
    void         *OSFlagNodePrev; /* Pointer to previous NODE in wait list */
    void         *OSFlagNodeTCB; /* Pointer to TCB of waiting task */
    void         *OSFlagNodeFlagGrp; /* Pointer to Event Flag Group */
    OS_FLAGS      OSFlagNodeFlags; /* Event flag to wait on */
    INT8U         OSFlagNodeWaitType; /* Type of wait: */
                        /*      OS_FLAG_WAIT_AND */
                        /*      OS_FLAG_WAIT_ALL */
                        /*      OS_FLAG_WAIT_OR */
                        /*      OS_FLAG_WAIT_ANY */
} OS_FLAG_NODE;
```

A task can wait for all or any events to be set, or cleared. The listing above gives all the possibilities:

```
#define  OS_FLAG_WAIT_CLR_ALL 0  /* Wait for ALL the bits specified to be CLR (i.e. 0) */
#define  OS_FLAG_WAIT_CLR_AND 0

#define  OS_FLAG_WAIT_CLR_ANY 1 /* Wait for ANY of the bits specified to be CLR (i.e. 0)*/
#define  OS_FLAG_WAIT_CLR_OR  1

#define  OS_FLAG_WAIT_SET_ALL 2 /* Wait for ALL the bits specified to be SET (i.e. 1)*/
#define  OS_FLAG_WAIT_SET_AND 2

#define  OS_FLAG_WAIT_SET_ANY 3 /* Wait for ANY of the bits specified to be SET (i.e. 1) */
#define  OS_FLAG_WAIT_SET_OR  3
```

Note that values ending with AND and ALL are similar as well as values ending with OR and ANY. One can use either the one or the other.

Creating an Event flag groups

Creating an event flag group is made with OSFlagCreate(). It consists to take the first event flag groups structure from OSFlagFreeList and to adjust it. OSFlagFreeList is a pool of empty OS_FLAG_GRP structure created in OSInit.
OSFlagCreate takes as argument the flags used to initialize the structure.

Deleting an event flag group

This is made with OSFlagDel(). Similarly to what happens with OSSemDel(), the function takes as argument an option for deletion. If opt is OS_DEL_NO_PEND then the kernel has to wait for all the tasks waiting for the group of event to unblock or to stop waiting for the event group and simply returns. If no tasks are waiting or opt is OS_DEL_ALWAYS the event flag group is always deleted and the OS_FLAG_GRP struct is returned to the pool of free structures.

Waiting for one or several events

A task running can sets itself waiting for events in an event flag group by calling OSFlagPend(). It takes the event flag group, the flags to be waiting for, the wait type, a timeout and a pointer to be filled for error in arguments. The wait type is one of the type listed above and describes the policy of the task. As in the semaphore and mutex, a timeout is set.

A task can choose to clear after reading, this is done by ADDing the value OS_FLAG_CONSUME to the wait type. OSFlagPend() checks that the waiting events is not currently true, if so it simply returns. Otherwise it calls OS_FlagBlock() which adds a task to an event flag group wait list.

OS_FlagBlock simply adds a OS_FLAG_NODE structure corresponding to the current task at the beginning of the waiting list of the event flag group given in argument.

Setting or clearing an event in an event flag group

This is done by OSFlagPost(). It can be called from within an ISR.

It computes the new flags in the event group block and check all the OS_FLAG_NODE structure in the wait list if their waiting condition is fulfilled. If so the corresponding task is set ready with OS_FlagTaskReady(). Eventually the scheduler is called.

The OS_FlagTaskReady() function simply changes the status of the task to OS_STAT_RDY. At the end it calls OS_FlagUnlink() which removes the OS_FLAG_NODE corresponding to the task from the waiting list.

Checking if an event has happened

This function is similar to OSFlagPend() excepts that it only checks if the condition is fulfilled. If not it returns an error. One can get the flag status with OSFlagQuery().

2.5 Mail Box

Mailbox are structures used by the tasks and ISR to share a pointer variables which is set to a structure containing data. As for semaphores and mutex, a mailbox is a specialized ECB structure with OSEventType equal to OS_EVENT_TYPE_MBOX and OSEventPtr pointing to the message. Tasks waiting for the mail box are placed in the wait list.
The kernel provides routines to create, delete, post, wait with or without blocking.

Create a mailbox

OSMBoxCreate() creates a mailbox. It takes a pointer as argument and initializes a message box. The new message box is currently an empty ecb from OSEventFreeList.

Deleting a mailbox

OSMboxDel() deletes a mailbox according to the options given in argument. As with semaphores, and others, the kernel can either wait for all the task to be ready and returns with OS_DEL_NO_PEND or delete the mail box anyway with OS_DEL_ALWAYS. Once deleted, the ECB is then returned to OSEventFreeList.

Getting a message

OSSemPend() waits for a message to be available. If OSEventPtr from the ecb is different from zero, the function returns the message. If no message is present, it sets the state of the current task to OS_STAT_MBOX and the timeout. It then calls the scheduler. When returning from the scheduler, the function checks that no task has posted a message directly to the task, if so it returns the current message. Otherwise a timeout has occurred and the task is removed from the waiting list.

OSMboxAccept() checks if a message is present and gets it. If no message is present it returns a null pointer.

Posting a message

OSMboxPost permits to post a message to a specified message box. The function checks that OSEventPtr is empty, if not it returns OS_MBOX_FULL. If tasks are waiting it sends the message to the higher priority task with OS_EventTaskRdy(). If no tasks are in the wait list, it fills OSEventPtr with the message.
Another function called OSMboxPostOpt() permits posting to a mailbox but provides increased options. Broadcasting a message to all the tasks waiting in a wait list is possible.

Getting the status of a mailbox

One can query information from a mailbox with OSMboxQuery().

Using mailbox

Mailbox can be used for mutual exclusion. Task wanting exclusive access to resource calls OSSemPend() and releases it with OSSemPost(). Note that, using mutex is better since it provides a protection against priority inversion.

Mailboxes can be used to add delay. A task calls OSMboxPend() with a timeout. If no task posts, the timeout expires and this is similar to OSTimeDly(). Another task can resume a delayed task by posting.

2.6 Message queues

As mailboxes, message queues allow tasks or ISRs to communicate through pointers to structures. However message queues are circular buffers that can be used as FIFOs or LIFOs.

The OSEventPtr of an event control block specialized as a queue points to a queue control block. A queue control block is described by the following code:

```
typedef struct os_q { /* QUEUE CONTROL BLOCK */
    struct os_q   *OSQPtr;/* Link to next queue control block in list of free blocks*/
    void         **OSQStart;/* Pointer to start of queue data   */
    void         **OSQEnd; /* Pointer to end   of queue data   */
    void         **OSQIn; /* Pointer to where next message will be inserted  in   the Q  */
    void         **OSQOut;  /* Pointer to where next message will be extracted from the Q  */
    INT16U         OSQSize; /* Size of queue (maximum number of entries)  */
    INT16U         OSQEntries; /* Current number of entries in the queue*/
} OS_Q;
```

The QCB is pointing to the start of the buffer with OSQStart and to the end with OSQEnd. It also has a pointer to the next place to insert and a place to next place to output. Fig. **??** gives a better understanding of a circular buffer. Nine routines permits to manage message queues.

Creating a message queue

OSQCreate() gets and initializes an event control block and a queue control block. It takes as argument a pointer to the buffer and its size. The ECB and QCB are taken from OSEventFreeList and OSQFreeList.

It then initializes all the pointer to the buffer. The type of the event control block is set to OS_EVENT_TYPE_Q. The function returns the event.

Deleting a message queue

As with other structures an option is given along with the message queue given in argument. It specifies whether the queue must be deleted even if tasks are waiting. If option is OS_DEL_NO_PEND the function fills the error with OS_ERR_TASK_WAITING and return. If option is OS_DEL_ALWAYS, the function returns the ECB and QCB to the free lists and calls the scheduler even if tasks were waiting.

Getting a message with blocking

A task can wait at a queue until a message is coming if no messages exists yet with OSQPend().

It checks that a message is present in the queue, if so it takes the message pointed by OSQOut in the queue control block and decrements the number of elements in the queue. If no message are present in the queue, the current task is put in the wait list of the event and its timeout is set. The scheduler is called. When returning from it, OSQPend() checks if a message has been sent to the queue. If so it gives the message to the current task. Otherwise the timeout error happened.

Getting a message without blocking

One can get a message without blocking with OSQAccept(). It only checks the queue for messages. If message exists it return the value pointed by OSQOut. If no message are present it simply returns a null pointer.

Sending a message in a queue

This section describes OSQPostOpt() which replaces both OSQPost() and OSQPostFront().

OSQPost() sends a message in back of the queue as a FIFO whereas OSQPostFront() sends it in front of the queue as a LIFO. OSQPostOpt implements the two techniques and permits broadcast to all the tasks waiting.

Arguments given are a pointer to the event control block, a message to post and an option mask. The option can be a combination of OS_Q_POST_FRONT and OS_Q_POST_BROADCAST. No using OS_Q_POST_FRONT implicitly specifies to use the queue as a FIFO.

Flushing a queue

One can flush a queue with OSQFlush(). This is very fast since it only sets OSQIn and OSQOut to OSQStart and resets OSQEntries.

OSQQuery

The status of a message queue is obtained with OSQQuery and returns information in an OS_Q_DATA given in arguments.

2.7 Memory partitions

μC/OS-II provides memory partitions for dynamic memory allocations.

A partition is a contiguous area split in memory blocks. Free blocks are kept in a free list. Therefore acquiring and releasing a block is made in constant time.

Partitions avoid fragmentation caused by malloc() and free() usage since memory is allocated and frees one time for all the blocks.

An OS_MEM structure describes a memory partition:

```
typedef struct {/* MEMORY CONTROL BLOCK */
    void   *OSMemAddr;/* Pointer to beginning of memory partition */
    void   *OSMemFreeList; /* Pointer to list of free memory blocks */
    INT32U  OSMemBlkSize; /* Size (in bytes) of each block of memory */
    INT32U  OSMemNBlks; /* Total number of blocks in this partition */
    INT32U  OSMemNFree; /* Number of memory blocks remaining in this partition */
} OS_MEM;
```

OSMemAddr is a pointer to the beginning of the overall data area. The first free block is pointed to by OSMemFreeList. As the block size is different from one partition to another, it is hold in the structure. OS_MEM holds the number of blocks and free blocks.

Creating a partition

OSMemCreate() gets and initializes an OS_MEM structure from OSMemFreeList. The function takes as argument a pointer to the memory area, the number of blocks, the block size size and an error pointer. It returns the initialized OS_MEM structure.

Getting a block

OSMemGet() returns a block from an existing partition. The block is the first free block from the OSMemFreeList field of the OS_MEM structure. If no blocks are available, it sets the error to OS_MEM_NO_FREE_BLOCKS and returns a null pointer.

Returning a block

OSMemPut() returns a block to a partition. The block size must be the same as in the partition. Free slot must be available in the partition, otherwise the OS_MEM_FULL error is returned.

Querying a partition

One can get a snapshot from a partition with the OSMemQuery() function. The function fills a OS_MEM_DATA structure which contains information about it.

Chapter 3

OS Components and drivers

Although any filesystems or drivers can be ported to μC/OS-II, Micrium provides building blocks to ease writing new applications. I describe two of them, the filesystem block and the TCP/IP stack. Eventually I will give an example driver.

I only give a short description here.

3.1 μC/FS

[7] describes the filesystem block.

μC/FS is a FAT based filesystem for embedded uses. It is configurable to minimize RAM footprint. Optional components are available such as the use of journaling. It provides drivers to be used with.

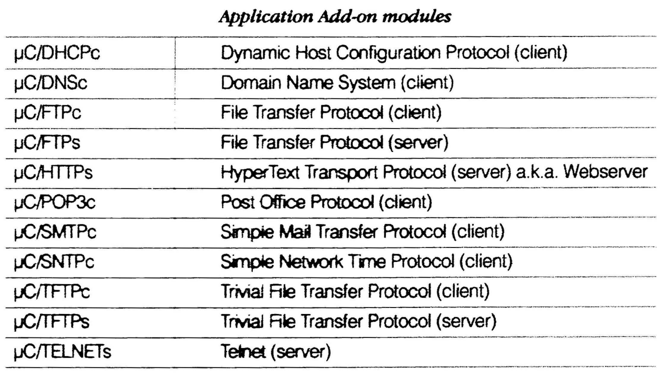
The filesystem provides facilities to run on flash disk and improve its reliability. Therefore it exhibits atomic writes for data integrity, wear-leveling and bad block management. Since flash devices have a reduced life time in certain conditions [4], it is better to uniform access on the overall device. Bad block management is required on NAND flash.

μC/FS can be used with or without an OS. It is processor independent and provides a POSIX interface.

3.2 μC/TCP-IP

A TCP/IP protocol stack is proposed to be used with μC/OS-II and μC/OS-III. It is furnished by Micrium to be used in embedded context. It can run on 16, 32 and 64 bits processors. Again memory footprint can be tuned.

μC/TCP-IP is written in ANSI-C and can be ported to any other RTOS. The figure 3.1 from [9] presents application modules available.



Application Add-on modules

µC/DHCPc	Dynamic Host Configuration Protocol (client)
µC/DNSc	Domain Name System (client)
µC/FTPc	File Transfer Protocol (client)
µC/FTPs	File Transfer Protocol (server)
µC/HTTPs	HyperText Transport Protocol (server) a.k.a. Webserver
µC/POP3c	Post Office Protocol (client)
µC/SMTPc	Simple Mail Transfer Protocol (client)
µC/SNTPc	Simple Network Time Protocol (client)
µC/TFTPc	Trivial File Transfer Protocol (client)
µC/TFTPs	Trivial File Transfer Protocol (server)
µC/TELNETs	Telnet (server)

Figure 3.1: Application Add-on modules for μC/TCP-IP stack.

3.3 Driver example

The example is a serial driver given in annexe A.3.

The driver is written in C, also it doesn't differ a lot from any other RTOS driver.

The driver defines interrupts. Critical sections needs to be implemented in μC/OS way by using OS_ENTER_CRITICAL and OS_EXIT_CRITICAL. Other parts of code are classical serial driver code.

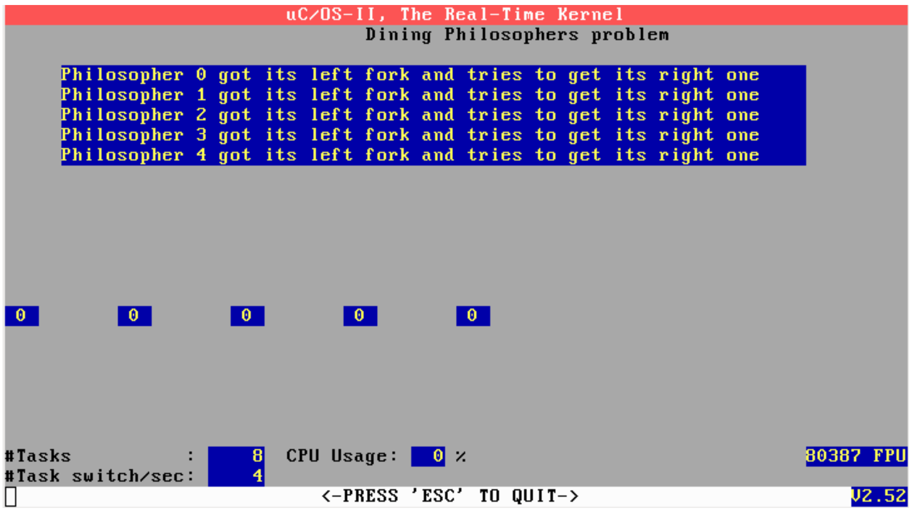
Chapter 4

Demo

4.1 Demo1: Philosophers dinner on x86

Demo1 is an implementation of the philosopher problem described in [1]. It uses the x86 port from Jean J. Labrosse described in [6]. The demo has been compiled with Borland C/C++ 5.02. The application is given in annexe A.1.

The philosophers can take two states. They can be eating for a specified amount of time or they can be thinking. The demo implements five philosophers as threads. The forks are semaphores with initial count set to one. A philosopher always takes and releases its left fork first.
We simulate the worst case by inserting delay after the philosopher took its first fork. The figure 4.1 shows an application run. A dead lock happened.



```
                        uC/OS-II, The Real-Time Kernel
                           Dining Philosophers problem

   Philosopher 0 got its left fork and tries to get its right one
   Philosopher 1 got its left fork and tries to get its right one
   Philosopher 2 got its left fork and tries to get its right one
   Philosopher 3 got its left fork and tries to get its right one
   Philosopher 4 got its left fork and tries to get its right one




 0      0      0      0      0




#Tasks          :    8  CPU Usage:  0 %                     80387 FPU
#Task switch/sec:    4
                   <-PRESS 'ESC' TO QUIT->                       V2.52
```

Figure 4.1: Each philosopher is waiting for its neighbor to release a fork. The situation cannot evolve. The line after philosophers output is the count of the semaphore used to control the state of the fork. The semaphores are taken meaning that no fork is available.

4.2 Demo2: Philosophers dinner solution

We propose a solution known as the waiter solution, given in annex **??**. A philosopher that wants a fork first has to ask a waiter. If the two forks are available and no clearance has been given yet for that forks, the waiter agree and the forks are given to the philosopher.
The request is made by sending philosopher's id to waiter queue. A philosopher is allowed to take forks when the content of its associated mailbox is set to WAITER_SAYS_YES. Figure 4.2 presents the scheme used.

The result is shown in 4.3 as we can see no deadlock appears.

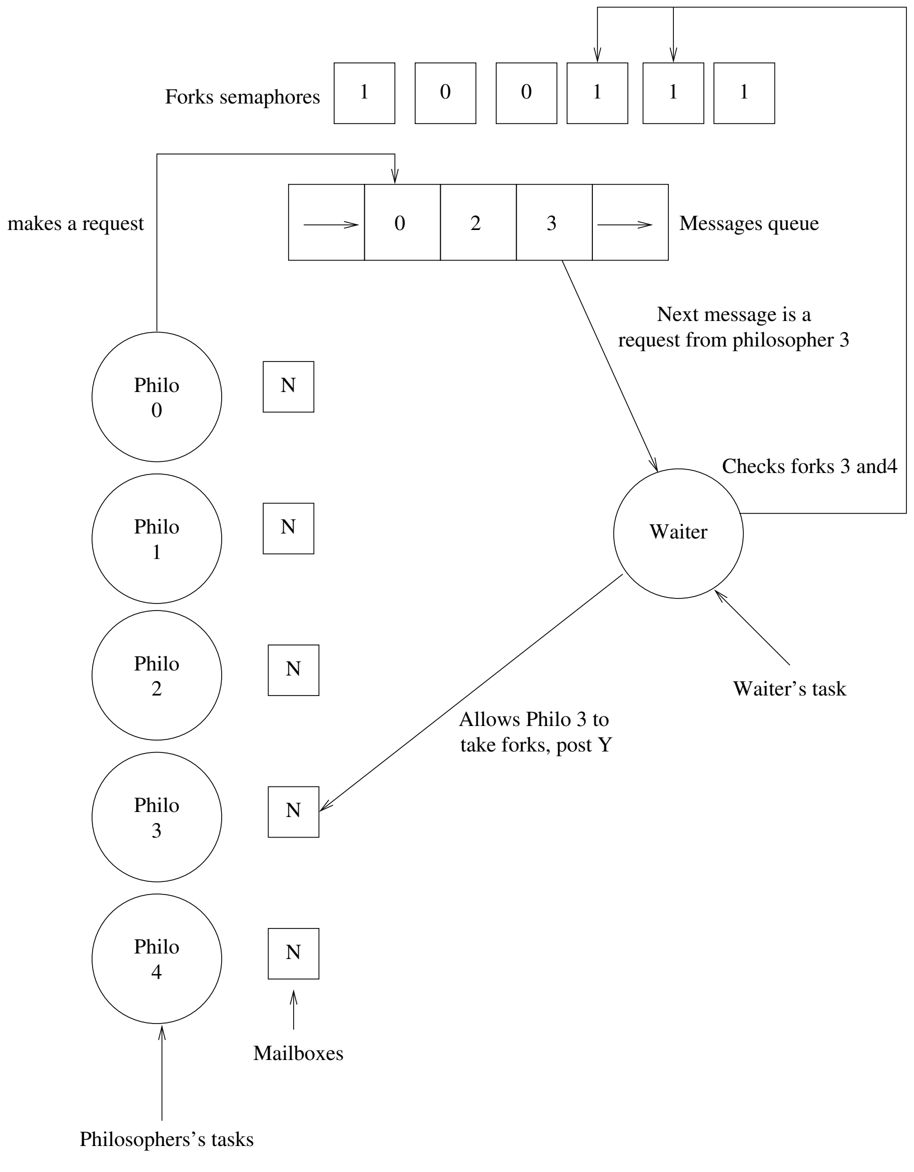
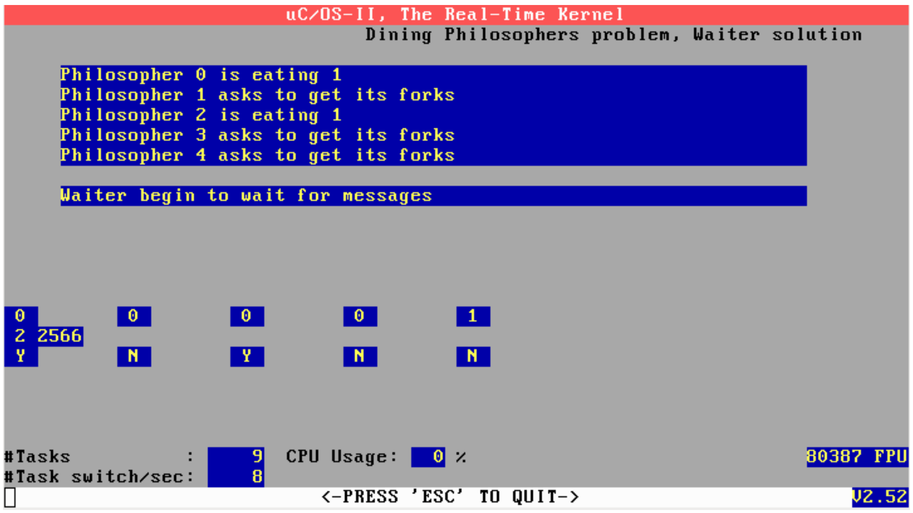


Figure 4.2: The waiter solution use semaphores, mailboxes and message queue.



```
                          uC/OS-II, The Real-Time Kernel
                                 Dining Philosophers problem, Waiter solution

     Philosopher 0 is eating 1
     Philosopher 1 asks to get its forks
     Philosopher 2 is eating 1
     Philosopher 3 asks to get its forks
     Philosopher 4 asks to get its forks

     Waiter begin to wait for messages




0          0          0          0          1
2 2566
Y          N          Y          N          N



#Tasks          :      9  CPU Usage:   0 %                         80387 FPU
#Task switch/sec:      8
                              <-PRESS 'ESC' TO QUIT->                   V2.52
```

Figure 4.3: Philosophers zero and two are eating. Others have made a request to the waiter and wait a positive response. The first line after philosophers output is the state of the semaphores, second one is the number of messages and size of the queue. Third line is the content of mailboxes, philosophers zero and two have been allowed to take forks.

4.3 Compiler

The demo has to be compiled with Borland C/C++ compiler with version prior or equal to 5.02. The x86 port uses real mode instructions and so requires a compiler running in it. This can cause problems since that version is no longer sale.

4.4 Configuration

One had to specify OS_MAX_EVENTS in OS_CFG.H to a sufficient number. For that demo, it has been set to ten.

Conclusion

μC/OS-II has been widely described here. Along with the inner working of the kernel, I described the services. I shown a practical case study which is an implementation of the dinning philosophers problem and one of its solution. More precisions can be found in [6].

Since 1992, numerous applications use μC/OS-II. As it begin to get old Micrium provides a new version of its os, namely μC/OS-III. μC/OS-III has been released in 2009 and keeps a lot of features from its predecessors[8]. The number of tasks on a system is unlimited and tasks can share priority. The kernel uses a round-robin scheduler and is written in C ANSI.

Bibliography

[1] Dining philosophers problem. *http://en.wikipedia.org/wiki/Dining_philosophers_problem*.

[2] Priority inversion. *http://en.wikipedia.org/wiki/Priority_inversion*.

[3] Process. *http://en.wikipedia.org/wiki/Process_(computing)*.

[4] Wear levelling. *http://en.wikipedia.org/wiki/Wear_levelling*.

[5] Jean J. Labrosse. microc/os-ii port for x86. Published with the book Microc/OS-II The real time kernel.

[6] Jean J. Labrosse. *Microc/OS-II*. R & D Books, 1998.

[7] Christian Legare. μc/fs. *http://micrium.com/download/fswhitepaper.pdf*.

[8] Micrium. μc/os-iii. *http://micrium.com/newmicrium/uploads/file/datasheets/ucos-iii_datasheet.pdf*.

[9] Micrium. μc/tpc-ip. *http://micrium.com/download/tcp-ip_dsheet.pdf*.

[10] NXP. *training.micrium.ucos-ii.rtos.ppt*. http://www.standardics.nxp.com/support/boards/ird/ppt/training.micrium.ucos-ii.rtos.ppt.

Appendix A

Appendices

A.1 Dining philosophers implementation

```
#include "includes.h"

/*
Dining Philosophers demo
date: november 2009
author: Guillaume Kremer
some portions of code are taken from Jean J.Labrosse x86 port
TO be compiled with Borland v55 C/C++ compiler
*/

#define NB_PHILOSOPHERS 5 /*Number of Philosophers*/
#define MAX_BUFFER 100 /*Size for printing buffer*/
/*
The task size has to been specified carefully
each task has 3 buffer of MAX_BUFFER size so me allocate
at least 3*MAX_BUFFER
others=32+sizeOf(OS
*/
#define TASK_STAKE_SIZE        (3*MAX_BUFFER)+32+sizeof(OS_SEM_DATA)
/*Tasks size*/
#define NB_FORKS NB_PHILOSOPHERS /*Number of forks*/
#define TIME_TO_EAT 20 /*Time taken by a philosopher to eat*/
#define TIMEOUT_FORK 0 /*Timeout to take a fork*/

void TaskStart(void *pdata);
void Task(void *pdata);
static void TaskStartCreatePHILOSOPHERS(void);
static void TaskStartDispInit(void);
static void TaskStartDisp(void);
int Print_with_id(int index, int state, int cpt);
int Print_Sem_Status();

INT8U ids[NB_PHILOSOPHERS]; /*ids for the philosophers*/
OS_STK TaskStartStk[TASK_STAKE_SIZE]; /*Stacks for the start task*/
OS_STK TaskStk[NB_PHILOSOPHERS][TASK_STAKE_SIZE]; /*Stacks for the philoso-
phers tasks*/

OS_EVENT *FORKS[NB_FORKS]; /*FORKS for the Philosophers*/

int
```

```
main(argc, argv, argenv)
        int argc;
        char **argv, **argenv;
{
        INT8U i; /*loop value*/
        INT8U err; /*error value*/

            PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK); /*Clear the
screen*/

        OSInit(); /*Initialise microC/OS-II*/

        PC_DOSSaveReturn();

        PC_VectSet(uCOS, OSCtxSw);

        for(i = 0; i < NB_PHILOSOPHERS; i++)
        {
                FORKS[i] = OSSemCreate((INT16U)1); /*Semaphores initialization with a
cnt of 1 <> mutual exclusion*/
        }

        OSTaskCreate(TaskStart, (void *)0, &TaskStartStk[TASK_STAKE_SIZE - 1], 0);
/*Creation of the starting task*/
        OSStart(); /*Start the OS*/

        return 0;

}

/*Some x86 initialisation, starting task*/
void
TaskStart(pdata)
        void *pdata;
{
#if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr;
#endif

        INT8U i;
        INT16S key;

        pdata = pdata;

        TaskStartDispInit(); /*Init the display*/
        OS_ENTER_CRITICAL(); /*Critical section*/

        PC_VectSet(0x08, OSTickISR); /*Enable ISR*/

        PC_SetTickRate(OS_TICKS_PER_SEC); /*Change the clock frequency*/
        OS_EXIT_CRITICAL(); /*End of critical section*/

        OSStatInit(); /*Initialisation of the Statistic task*/

        TaskStartCreatePHILOSOPHERS(); /*Task which create philosophers task*/

        for(;;)
        {
                TaskStartDisp(); /* Update the display*/

                if (PC_GetKey(&key) == TRUE) { /* See if key has been pressed*/
                    if (key == 0x1B) { /* Yes, see if it's the ESCAPE key*/
                        PC_DOSReturn(); /* Return to DOS*/
                    }
                }

                OSCtxSwCtr = 0;
                OSTimeDlyHMSM(0, 0, 1, 0); /*Sleeping for one second*/
        }

}

/*Creating the philosophers tasks*/
static void
TaskStartCreatePHILOSOPHERS()
{
        INT8U i; /*Loop value*/

        for(i = 0; i < NB_PHILOSOPHERS; i++)
        {
                ids[i] = i;
                OSTaskCreate(Task, (void*)&(ids[i]), &((TaskStk[i])[TASK_STAKE_SIZE -
1]), i + 1);
        }

}

/*Philosopher task*/
void
Task(pdata)
        void *pdata;
{
        INT8U index; /*Index of the task*/
        INT8U err; /*Error value*/
        INT16U cpt; /*Number of time the philosopher has eaten*/

        index = *(INT8U*)pdata; /*Cast of the argument*/

        Print_with_id(index, 1, 0); /*print Hello world*/

        Print_Sem_Status(); /*Print the sem status*/

        OSTimeDlyHMSM(0,0,2,0); /*2 seconds delay*/

        cpt = 0;

        for(;;)
        {

                Print_Sem_Status();

                Print_with_id(index, 2, cpt); /*print trying to get left fork*/

              OSSemPend(FORKS[(index + 1) % NB_FORKS], TIMEOUT_FORK, &err);
/*bloking wait for the left fork*/
                while(err == OS_TIMEOUT) /*while timeout*/
                {
                        Print_with_id(index, 3, cpt); /*print timeout happend*/

                        Print_Sem_Status();

                        OSTimeDlyHMSM(0, 0, 1, 0); /*1 second delay*/
                }

                Print_Sem_Status();

                OSTimeDlyHMSM(0, 0, 1, 0);

                Print_with_id(index, 4, cpt); /*print trying to get right fork*/

                OSSemPend(FORKS[index], TIMEOUT_FORK, &err); /*blocking wait for
the right fork*/
                while(err == OS_TIMEOUT)
                {
                        Print_with_id(index, 5, cpt); /*print timeout happend*/

                        Print_Sem_Status();

                        OSTimeDlyHMSM(0, 0, 1, 0);
                }

                cpt++;

                Print_with_id(index, 7, cpt); /*print philosopher is eating*/

                OSTimeDlyHMSM(0, 0, TIME_TO_EAT, 0); /*eating delay time*/

                  OSSemPost(FORKS[(index + 1) % NB_FORKS]); /*releasing right fork
semaphore*/
                OSSemPost(FORKS[index]); /*releasing left fork semaphore*/

        }


}
```

A.2 Dining philosophers implementation, waiter solution

```
#include "includes.h"

/*
Dining Philosophers demo
date: november 2009
author: Guillaume Kremer
some portions of code are taken from Jean J.Labrosse x86 port
TO be compiled with Borland v55 C/C++ compiler
*/

#define NB_PHILOSOPHERS 5 /*Number of Philosophers*/
#define MAX_BUFFER 100 /*Size for printing buffer*/
#define SIZE_QUEUE 256
/*
The task size has to been specified carefully
each task has 3 buffer of MAX_BUFFER size so me allocate
at least 3*MAX_BUFFER
others=32+sizeOf(OS
*/
#define TASK_STAKE_SIZE      (4*MAX_BUFFER)+64+sizeof(OS_SEM_DATA)+2*sizeof(OS_MBOX_DATA)+2*si
/*Tasks size*/
#define TASK_WAITER_STAKE_SIZE (16+(NB_PHILOSOPHERS*8)+2*sizeof(OS_SEM_DATA)+3*sizeof(OS_MB
#define NB_FORKS NB_PHILOSOPHERS /*Number of forks*/
#define TIME_TO_EAT 20 /*Time taken by a philosopher to eat*/
#define TIMEOUT_FORK 0 /*Timeout to take a fork*/

#define WAITER_UNDEFINED 0
#define WAITER_SAYS_NO 1
#define WAITER_SAYS_YES 2

void TaskStart(void *pdata);
void Task(void *pdata);
void Task_Waiter(void *pdata);
static void TaskStartCreatePHILOSOPHERSAndWAITER(void);
static void TaskStartDispInit(void);
static void TaskStartDisp(void);
int Print_with_id(int index, int state, int cpt);
int Print_Sem_Status();
int Print_Q_Status();
int Print_Mbox_Status();

INT8U ids[NB_PHILOSOPHERS]; /*ids for the philosophers*/
OS_STK TaskStartStk[TASK_STAKE_SIZE]; /*Stacks for the start task*/
OS_STK TaskStk[NB_PHILOSOPHERS][TASK_STAKE_SIZE]; /*Stacks for the philoso-
phers tasks*/
OS_STK TaskWaiterStk[TASK_WAITER_STAKE_SIZE];

void *Array_Queue[SIZE_QUEUE];

OS_EVENT *FORKS[NB_FORKS]; /*FORKS for the Philosophers*/
OS_EVENT *SEM_FORKS;
OS_EVENT *MAILBOXES[NB_PHILOSOPHERS];
OS_EVENT *QUEUE;

int
```

```
main(argc, argv, argenv)
        int argc;
        char **argv, **argenv;
{
        INT8U i; /*loop value*/
        int i_;
        INT8U err; /*error value*/
        INT8U initial_messages;
        INT8U initial_queue;

            PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK); /*Clear the
screen*/

        OSInit(); /*Initialise microC/OS-II*/

        PC_DOSSaveReturn();

        PC_VectSet(uCOS, OSCtxSw);

        SEM_FORKS = OSSemCreate((INT16U)1);
        initial_queue = 0;

        for(i_ = 0; i_ < SIZE_QUEUE; i_++)
        {
              Array_Queue[i_] = (void *) &initial_queue;/*The queue is initialized to a known
value*/
        }

        QUEUE = OSQCreate(&Array_Queue[0], SIZE_QUEUE);

        initial_messages = WAITER_SAYS_NO;/*Mailboxes are initialized to no*/
        for(i = 0; i < NB_PHILOSOPHERS; i++)
        {
                FORKS[i] = OSSemCreate((INT16U)1); /*Semaphores initialization with a
cnt of 1 <> mutual exclusion*/
                 MAILBOXES[i] = OSMboxCreate(&initial_messages);
        }


        OSTaskCreate(TaskStart, (void *)0, &TaskStartStk[TASK_STAKE_SIZE - 1], 0);
/*Creation of the starting task*/
        OSStart(); /*Start the OS*/

        return 0;

}

/*Some x86 initialisation, starting task*/
void
TaskStart(pdata)
        void *pdata;
{
#if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr;
#endif

        INT8U i;
        INT16S key;
        pdata = pdata;

        TaskStartDispInit(); /*Init the display*/
        OS_ENTER_CRITICAL(); /*Critical section*/

        PC_VectSet(0x08, OSTickISR); /*Enable ISR*/

        PC_SetTickRate(OS_TICKS_PER_SEC); /*Change the clock frequency*/
        OS_EXIT_CRITICAL(); /*End of critical section*/

        OSStatInit(); /*Initialisation of the Statistic task*/

            TaskStartCreatePHILOSOPHERSAndWAITER(); /*Task which create
philosophers task*/

        for(;;)
        {
                TaskStartDisp(); /* Update the display*/

                if (PC_GetKey(&key) == TRUE) { /* See if key has been pressed*/
                    if (key == 0x1B) { /* Yes, see if it's the ESCAPE key*/
                        PC_DOSReturn(); /* Return to DOS*/
                    }
                }

                OSCtxSwCtr = 0;
                OSTimeDlyHMSM(0, 0, 1, 0); /*Sleeping for one second*/
        }

}

/*Creating the philosophers tasks and the waiter*/
static void
TaskStartCreatePHILOSOPHERSAndWAITER()
{
        INT8U i; /*Loop value*/

     OSTaskCreate(Task_Waiter, NULL, &TaskWaiterStk[TASK_WAITER_STAKE_SIZE
- 1], 10);

        for(i = 0; i < NB_PHILOSOPHERS; i++)
        {
                ids[i] = i;
               OSTaskCreate(Task, (void*)&(ids[i]), &((TaskStk[i])[TASK_STAKE_SIZE -
1]), i + 4);
        }

}
/*Creating the philosophers tasks*/
static void
TaskStartCreatePHILOSOPHERS()
{
        INT8U i; /*Loop value*/

        for(i = 0; i < NB_PHILOSOPHERS; i++)
        {
                ids[i] = i;
                OSTaskCreate(Task, (void*)&(ids[i]), &((TaskStk[i])[TASK_STAKE_SIZE -
1]), i + 1);
        }

}

/*Waiter task*/
void
Task_Waiter(pdata)
        void *pdata;
{
        INT8U mess;
        INT8U err;
        OS_SEM_DATA sem_data1;
        OS_SEM_DATA sem_data2;
        OS_MBOX_DATA m_data1;
        OS_MBOX_DATA m_data2;
        OS_Q_DATA qdata;
        void *ret;
        INT8U messages[NB_PHILOSOPHERS];

        pdata = pdata;

        Print_Mbox_Status();
        Print_with_id(NB_PHILOSOPHERS + 1, 10, 0); /*printing waiter is coming*/

        for(;;)
        {
                Print_Mbox_Status();
                Print_with_id(NB_PHILOSOPHERS + 1, 11, 0);

                 mess = *(INT8U*) OSQPend(QUEUE, TIMEOUT_FORK, &err);/*Get a
request in the queue*/
                while(err == OS_TIMEOUT)
                {
                        Print_with_id(NB_PHILOSOPHERS + 1, 15,0);/*printing timeout*/
                        OSTimeDlyHMSM(0, 0, 1, 0); /*1 second delay*/
                        mess = *(INT8U*) OSQPend(&QUEUE, TIMEOUT_FORK, &err);
                }

                if(mess < NB_PHILOSOPHERS)/*if the id request is coherent*/
                {
                        Print_Mbox_Status();
                        Print_with_id(NB_PHILOSOPHERS + 1, 12, mess);
                        Print_Mbox_Status();

                        OSSemPend(SEM_FORKS, TIMEOUT_FORK, &err);
                        Print_with_id(NB_PHILOSOPHERS + 1, 13, 0);
                        Print_Mbox_Status();

                        OSSemQuery(FORKS[mess], &sem_data1);
                        OSSemQuery(FORKS[(mess + 1) % NB_FORKS], &sem_data2);

                        if(mess != 0)/*if id is not 0*/
                        {
                                OSMboxQuery(MAILBOXES[mess - 1], &m_data1);
                        }
                        else
```

```
                    {
                            OSMboxQuery(MAILBOXES[(NB_PHILOSOPHERS - 1) %
NB_FORKS], &m_data1);
                    }

                OSMboxQuery(MAILBOXES[(mess + 1) % NB_FORKS], &m_data2);/*Is
right philosopher eating*/
                    Print_Mbox_Status();

                    if(sem_data1.OSCnt && sem_data2.OSCnt)/*If left and right fork are
free*/
                    {
                        Print_Mbox_Status();
                        Print_with_id(NB_PHILOSOPHERS + 1, 16, mess);

                        if(((*((INT8U*)m_data1.OSMsg)) == WAITER_SAYS_NO) &&
((*((INT8U*)m_data2.OSMsg)) == WAITER_SAYS_NO))/*Is left and right philosophers
not eating*/
                        {
                            Print_with_id(NB_PHILOSOPHERS + 1, 14, mess);
                            messages[mess] = WAITER_SAYS_YES;
                                OSMboxAccept(MAILBOXES[mess]);/*empty the
mailboxw*/

                                    if((ret = OSMboxPostOpt(MAILBOXES[mess],
&messages[mess], 0))!=OS_NO_ERR)/*send yes to the mailbox*/
                            {
                                        Print_with_id(NB_PHILOSOPHERS + 1, 18,
*((INT8U*)ret));
                            }

                            Print_Mbox_Status();
                        }
                    }

                    OSSemPost(SEM_FORKS);
                }

        }
}

/*Philosopher task*/
void
Task(pdata)
        void *pdata;
{
        INT8U index; /*Index of the task*/
        INT8U err; /*Error value*/
        INT16U cpt; /*Number of time the philosopher has eaten*/
        INT8U mess;
        OS_MBOX_DATA mdata;

        index = *(INT8U*)pdata; /*Cast of the argument*/

        Print_with_id(index, 1, 0); /*print Hello world*/

        Print_Sem_Status(); /*Print the sem status*/
```

```
    OSTimeDlyHMSM(0,0,2,0); /*2 seconds delay*/
    mess = WAITER_SAYS_NO;
    cpt = 0;

    for(;;)
    {
        Print_Mbox_Status();
        while(mess != WAITER_SAYS_YES)/*While the waiter says no*/
        {
            Print_Mbox_Status();

            OSQPost(QUEUE, (void*)&index); /*asks a request*/
            Print_Q_Status();
            Print_Mbox_Status();
            Print_with_id(index, 8, cpt); /*print trying to get forks*/

            OSMboxQuery(MAILBOXES[index], &mdata);
            mess = *(INT8U*) mdata.OSMsg;

            while(err == OS_TIMEOUT) /*while timeout*/
            {
                Print_with_id(index, 9, cpt); /*print timeout happend*/

                Print_Sem_Status();

                OSTimeDlyHMSM(0, 0, 1, 0); /*1 second delay*/
                    mess = *(INT8U*) OSMboxPend(&MAILBOXES[index],
TIMEOUT_FORK, &err); /*bloking wait for message*/
            }
            Print_Sem_Status();
            OSTimeDlyHMSM(0, 0, 1, 0); /*1 second delay*/

        }
            Print_with_id(index, 17, cpt); /*print trying to get forks*/
            Print_Mbox_Status();
            OSTimeDlyHMSM(0, 0, 1, 0); /*1 second delay*/

        OSSemPend(SEM_FORKS, TIMEOUT_FORK, &err);

        Print_with_id(index, 2, cpt); /*print trying to get left fork*/
        OSTimeDlyHMSM(0, 0, 1, 0); /*1 second delay*/

       OSSemPend(FORKS[(index + 1) % NB_FORKS], TIMEOUT_FORK, &err);
/*bloking wait for the left fork*/
        while(err == OS_TIMEOUT) /*while timeout*/
        {
            Print_with_id(index, 3, cpt); /*print timeout happend*/

            Print_Sem_Status();

            OSTimeDlyHMSM(0, 0, 1, 0); /*1 second delay*/
           OSSemPend(FORKS[(index + 1) % NB_FORKS], TIMEOUT_FORK,
&err); /*bloking wait for the left fork*/
        }

        Print_Sem_Status();

        Print_with_id(index, 4, cpt); /*print trying to get right fork*/
        OSTimeDlyHMSM(0, 0, 1, 0); /*1 second delay*/

        OSSemPend(FORKS[index], TIMEOUT_FORK, &err); /*blocking wait for
the right fork*/
        while(err == OS_TIMEOUT)
        {
            Print_with_id(index, 5, cpt); /*print timeout happend*/

            Print_Sem_Status();

            OSTimeDlyHMSM(0, 0, 1, 0);
              OSSemPend(FORKS[index], TIMEOUT_FORK, &err); /*blocking
wait for the right fork*/
        }


        Print_Mbox_Status();
        OSSemPost(SEM_FORKS);

        cpt++;

        Print_with_id(index, 7, cpt); /*print philosopher is eating*/

        OSTimeDlyHMSM(0, 0, TIME_TO_EAT, 0); /*eating delay time*/

        mess = WAITER_SAYS_NO;

        OSMboxAccept(MAILBOXES[index]);
        Print_Mbox_Status();
        OSMboxPostOpt(MAILBOXES[index], &mess, 0);
        Print_Mbox_Status();

          OSSemPost(FORKS[(index + 1) % NB_FORKS]); /*releasing right fork
semaphore*/
        OSSemPost(FORKS[index]); /*releasing left fork semaphore*/

        OSTimeDlyHMSM(0, 0, 1, 0);
    }


}
```

A.3 Driver example: serial driver

Thanks to Jean-Philippe Babau.

```
#include <msp430x14x.h>
#include <io.h>
#include <signal.h>
#include <iomacros.h>
#include  <ucos_ii.h>
#define MAXBUFFER  64

typedef struct tagComBuffer{
   unsigned char Buffer[MAXBUFFER];
   unsigned Head,Tail;
   unsigned Count;
}ComBuffer;

ComBuffer ComRXBuffer,ComTXBuffer;
int PutBuf(ComBuffer  *Buf,unsigned char Data);
unsigned char GetBuf(ComBuffer  *Buf);
unsigned GetBufCount(ComBuffer  *Buf);
unsigned ComOpen;
unsigned ComError;
unsigned ComBusy;



void initUART(int BaudRate)
{
  volatile unsigned int i;
  long BaudRateDivisor;
  ComRXBuffer.Head = ComRXBuffer.Tail = ComRXBuffer.Count = 0;
  ComTXBuffer.Head = ComTXBuffer.Tail = ComTXBuffer.Count = 0;

  P3SEL |= 0x30;                          // P3.4,5 = USART0 TXD/RXD
  BCSCTL1 |= XTS;                           // ACLK = LFXT1 = HF XTAL

  do
  {
    IFG1 &= ~OFIFG;                         // Clear OSCFault flag
    for (i = 0xFF; i > 0; i--);          // Time for flag to set
  }
  while ((IFG1 & OFIFG));                   // OSCFault flag still set?

  BCSCTL2 |= SELM_3;                        // MCLK = LFXT1 (safe)

UCTL0 &=0x01;

  ME1 |= UTXE0 + URXE0;                       // Enable USART0 TXD/RXD
  UCTL0 |= CHAR;                          // 8-bit character
  UTCTL0 |= SSEL0;                        // UCLK = ACLK
  BaudRateDivisor = 8000000;               // assuming 8MHz clock as on EasyWeb2
  BaudRateDivisor = BaudRateDivisor / (long) BaudRate;
  UBR00 = BaudRateDivisor & 0xff;
  BaudRateDivisor = BaudRateDivisor >> 8;
  UBR10 = BaudRateDivisor & 0xff;
  UMCTL0 = 0x00;                         // no modulation
  UCTL0 &= ~SWRST;                          // Initialize USART state machine
  IE1 |= URXIE0 + UTXIE0;                   // Enable USART0 RX and TX interrupts
  ComOpen = 1;
  ComError = 0;
  //TXBUF0 = 0xff;            // send a break out at start
}

//#pragma vector=UART0RX_VECTOR
interrupt (UART0RX_VECTOR) usart0_rx (void)
{ OSIntEnter();
  if (PutBuf(&ComRXBuffer,RXBUF0) )
    ComError = 1; // if PutBuf returns a non-zero value then there is an error
 OSIntExit();
}

//#pragma vector=UART0TX_VECTOR
interrupt (UART0TX_VECTOR) usart0_tx (void)
{  OSIntEnter();
//  while (!(IFG1 & UTXIFG0));                // USART0 TX buffer ready?
//  TXBUF0 = RXBUF0;                           // RXBUF0 to TXBUF0
  if (GetBufCount(&ComTXBuffer))
        TXBUF0=GetBuf(&ComTXBuffer);
 OSIntExit();
}


//lit le contenue du buffer
extern int ReadCom(int Max,unsigned char *Buffer)
{

  unsigned i;
  if (!ComOpen)
    return (-1);
  i=0;
  while ((i < Max) && (GetBufCount(&ComRXBuffer)))
    Buffer[i++] = GetBuf(&ComRXBuffer);
  if (i>0)
  {
    Buffer[i]=0;
    return(i);
  }
  else {
    return(0);
  }
};


//ecrit dans le buffer
extern int WriteCom(int Count,unsigned char *Buffer)
{
  unsigned i,BufLen;
  if (!ComOpen)
    return (-1);
  if (!Buffer[0]) // blank string?
    return(0);
  BufLen = GetBufCount(&ComTXBuffer);
  for(i=0;i<Count;i++)
    PutBuf(&ComTXBuffer,Buffer[i]);
  // might have to kick-start interrupt driven comms if the UART is idle
  if ( (!BufLen) && !(IFG1 & UTXIFG0) )
```

```
        TXBUF0 = GetBuf(&ComTXBuffer);
  return 0;
};


//ajoute dans le buffer
int PutBuf(ComBuffer *Buf,unsigned char Data)
{
#if OS_CRITICAL_METHOD == 3                       /* Allocate storage for CPU status
register             */
    OS_CPU_SR  cpu_sr;
    cpu_sr = 0;                                   /* Prevent compiler warning                            */
#endif
    if ( (Buf->Head==Buf->Tail) && (Buf->Count!=0))
        return(1);  /* OverFlow */
    OS_ENTER_CRITICAL();
    Buf->Buffer[Buf->Head++] = Data;
    Buf->Count++;
    if (Buf->Head==MAXBUFFER)
        Buf->Head=0;
   OS_EXIT_CRITICAL();
    return(0);
};


unsigned char GetBuf(ComBuffer *Buf)
{
    unsigned char Data;
#if OS_CRITICAL_METHOD == 3                       /* Allocate storage for CPU status
register             */
    OS_CPU_SR  cpu_sr;
    cpu_sr = 0;                                   /* Prevent compiler warning                            */
#endif
    if ( Buf->Count==0 )
        return (0);
    OS_ENTER_CRITICAL();
    Data = Buf->Buffer[Buf->Tail++];
    if (Buf->Tail == MAXBUFFER)
        Buf->Tail = 0;
    Buf->Count--;
    OS_EXIT_CRITICAL();
    return (Data);
}

unsigned int GetBufCount(ComBuffer *Buf)
{
    return Buf->Count;
};
```