RTOS: µC/OS

Chen Weixiang @ Jun 20, 2016 | TAG: RTOS

This article introduces the Real-Time Operating System **µC/OS**.

µC/OS Releases

µC/OS

The Micro-Controller Operating Systems (**MicroC/OS**, stylized as **µC/OS**) is a real-time operating system (RTOS) designed by embedded software developer, *Jean J. Labrosse* in 1991.

µC/OS-II

Refer to µC/OS-II Release Notes (https://doc.micrium.com/pages/viewpage.action?pageId=12851586).

Releases	Date	Notes
V2.86	2007-09-12	Major release as it contains a new feature called **Multi-Pend** (implemented in *OSEventPendMulti()*) which allows a task to pend on multiple objects (semaphores, mailboxes or queues).
V2.87	2009-01-07	Major release as it contains changes to some of the elements of data structures as well as **#define** configuration constants.
V2.88	2009-01-21	Minor release as it contains only a few items that were not caught prior to releasing V2.87.
V2.89	2009-06-09	Minor release as it contains only a few items that were not caught prior to releasing V2.88.
V2.90	2010-05-18	Minor release as it contains only a few items that were not caught prior to releasing V2.89.

Releases	Date	Notes
V2.92	2010-09-20	Minor release as it contains only a few items that were not caught prior to releasing V2.90.
V2.92.07	2010-09-20	Minor release as it contains only a few items that were not caught prior to releasing V2.92.
V2.92.08	2013-03-31	Minor release and only contains the addition of one feature: Thread Local Storage (TLS)
V2.92.09	2013-08-19	Minor release as it contains only a few items that were not caught prior to releasing V2.92.08.
V2.92.10	2013-12-27	Minor release as it contains only a few items that were not caught prior to releasing V2.92.09.
V2.92.11	2014-04-16	Minor release as it contains only a few items that were not caught prior to releasing V2.92.10.

µC/OS-III

Refer to µC/OS-III Release Notes (https://doc.micrium.com/pages/viewpage.action?pageId=12851580).

Releases	Date	Notes
V3.01.00	2009-12-07	
V3.01.01	2010-01-11	
V3.01.02	2010-05-14	
V3.02.00	2011-08-01	
V3.03.00	2012-02-14	
V3.03.01	2012-05-17	
V3.04.00	2013-09-19	
V3.04.01	2013-10-17	
V3.04.02	2013-12-20	
V3.04.03	2014-02-21	
V3.04.04	2014-03-12	
V3.04.05	2015-02-16	

Releases	Date	Notes
V3.05.00	2015-05-29	
V3.05.01	2015-07-06	

Features Comparison

Refer to µC/OS, µC/OS-II and µC/OS-III Features Comparison (https://doc.micrium.com/display/osiiidoc/uC-OS+uC-OS-II+and+uC-OS-III+Features+Comparison).

Feature	µC/OS	µC/OS-II	> µC/OS-III
Year introduced	1992	1998	2009
Book	Yes	Yes	Yes
Source code available	Yes	Yes	Yes
Preemptive Multitasking	Yes	Yes	Yes
Maximum number of tasks	64	255	Unlimited
Number of tasks at each priority level	1	1	Unlimited
Round Robin Scheduling	No	No	Yes
Semaphores	Yes	Yes	Yes
Mutual Exclusion Semaphores	No	Yes	Yes (Nestable)
Event Flags	No	Yes	Yes
Message Mailboxes	Yes	Yes	No (not needed)
Message Queues	Yes	Yes	Yes
Fixed Sized Memory Management	No	Yes	Yes
Signal a task without requiring a semaphore	No	No	Yes
Option to Post without scheduling	No	No	Yes

Feature	µC/OS	µC/OS-II	> µC/OS-III
Send messages to a task without requiring a message queue	No	No	Yes
Software Timers	No	Yes	Yes
Task suspend/resume	No	Yes	Yes (Nestable)
Deadlock prevention	Yes	Yes	Yes
Scalable	Yes	Yes	Yes
Code Footprint	3K to 8K	6K to 26K	6K to 24K
Data Footprint	1K+	1K+	1K+
ROMable	Yes	Yes	Yes
Run-time configurable	No	No	Yes
Compile-time configurable	Yes	Yes	Yes
ASCII names for each kernel object	No	Yes	Yes
Pend on multiple objects	No	Yes	Yes
Task registers	No	Yes	Yes
Built-in performance measurements	No	Limited	Extensive
User definable hook functions	No	Yes	Yes
Time stamps on posts	No	No	Yes
Built-in Kernel Awareness support	No	Yes	Yes
Optimizable Scheduler in assembly language	No	No	Yes
Catch a task that returns	No	No	Yes
Tick handling at task level	No	No	Yes
Source code available	Yes	Yes	Yes
Number of services	~20	~90	~70
MISRA-C:1998	No	Yes (except 10 rules)	N/A

Feature	µC/OS	µC/OS-II	> µC/OS-III
MISRA-C:2004	No	No	Yes (except 7 rules)
DO178B Level A and EUROCAE ED-12B	No	Yes	Yes
Medical FDA pre-market notification (510(k)) and pre-market approval (PMA)	No	Yes	Yes
SIL3/SIL4 IEC for transportation and nuclear systems	No	Yes	Yes
IEC-61508	No	Yes	Yes

µC/OS-II

µC/OS-II Features

µC/OS-II is a completely portable, ROMable, scalable, preemptive, real-time, multitasking kernel. µC/OS-II is written in ANSI C and contains a small portion of assembly language code to adapt it to different processor architectures.

- **Portable**: Most of µC/OS-II is written in highly portable ANSI C, with target microprocessor-specific code written in assembly language. Assembly language is kept to a minimum to make µC/OS-II easy to port to other processors.
- **ROMable**: µC/OS-II was designed for embedded applications, which means that if you have the proper tool chain (i.e., C compiler, assembler, and linker/locator), you can actually embed µC/OS-II as part of a product.
- **Scalable**: You can use only the services you need in your application, which means that a product can use just a few µC/OS-II services, while another product can benefit from the full set of features. Scalability allows you to reduce the amount of memory (both RAM and ROM) needed by µC/OS-II on a per-product basis. Scalability is accomplished with the use of conditional compilation. Simply specify (through #define constants) which features you need for your application or product.
- **Preemptive**: µC/OS-II is a fully preemptive real-time kernel, which means that µC/OS-II always runs the highest priority task that is ready. Most commercial kernels are preemptive, and µC/OS-II is comparable in

performance with many of them.

- **Multitasking**: µC/OS-II can manage up to 64 tasks; however, you reserve eight of these tasks for µC/OS-II, leaving your application up to 56 tasks. Each task has a unique priority assigned to it, which means that µC/OS-II cannot do round-robin scheduling. There are thus 64 priority levels.

- **Deterministic**: Execution times for most of µC/OS-II functions and services are deterministic, which means that you can always know how much time µC/OS-II will take to execute a function or a service. Except for **OSTimeTick()** and some of the event flag services, execution times of µC/OS-II services do not depend on the number of tasks running in your application.

- **Task Stacks**: Each task requires its own stack; however, µC/OS-II allows each task to have a different stack size, which allows you to reduce the amount of RAM needed in your application. With µC/OS-II's stack-checking feature, you can determine exactly how much stack space each task actually requires.

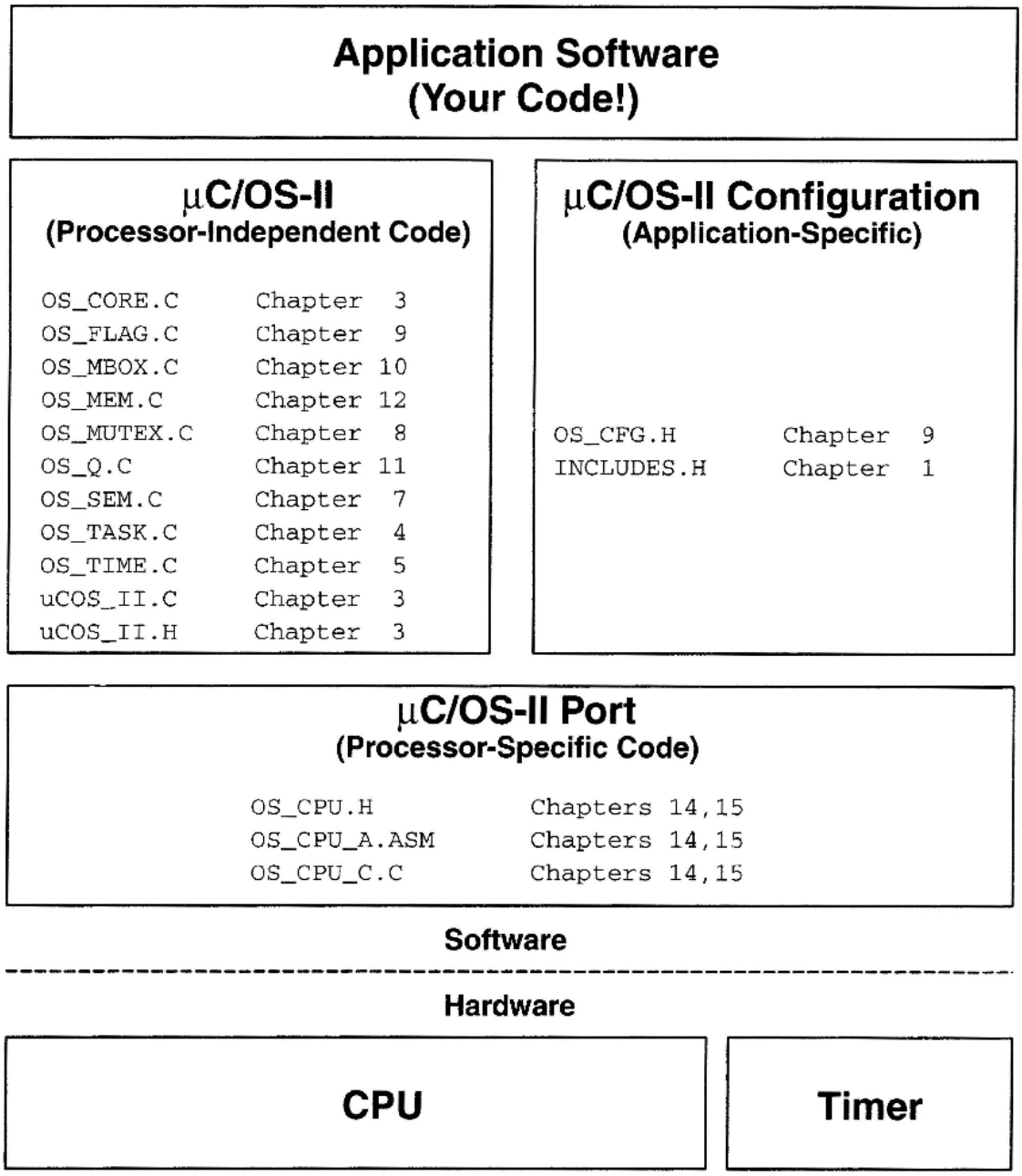
- **Services**: µC/OS-II provides a number of system services, such as semaphores, mutual exclusion semaphores, event flags, message mailboxes, message queues, fixed-sized memory partitions, task management, time management functions, and more.

- **Interrupt Management**: Interrupts can suspend the execution of a task. If a higher priority task is awakened as a result of the interrupt, the highest priority task runs as soon as all nested interrupts complete. Interrupts can be nested up to 255 levels deep.

- **Robust and Reliable**: µC/OS-II is based on µC/OS, which has been used in hundreds of commercial applications since 1992. µC/OS-II uses the same core and most of the same functions as µC/OS, yet offers many more features. *Also, in July of 2000, µC/OS-II was certified in an avionics product by the Federal Aviation Administration (FAA) for use in commercial aircraft by meeting the demanding requirements of the RTCA DO-178B standard for software used in avionics equipment*. In order to meet the requirements of this standard, it must be possible to demonstrate through documentation and testing that the software is both robust and safe. This issue is particularly important for an operating system as it demonstates that it has the proven quality to be usable in any application. Every feature, function, and line of code of µC/OS-II has been examined and tested to demonstrate that it is safe and robust enough to be used in safety-critical systems where human life is on the line.

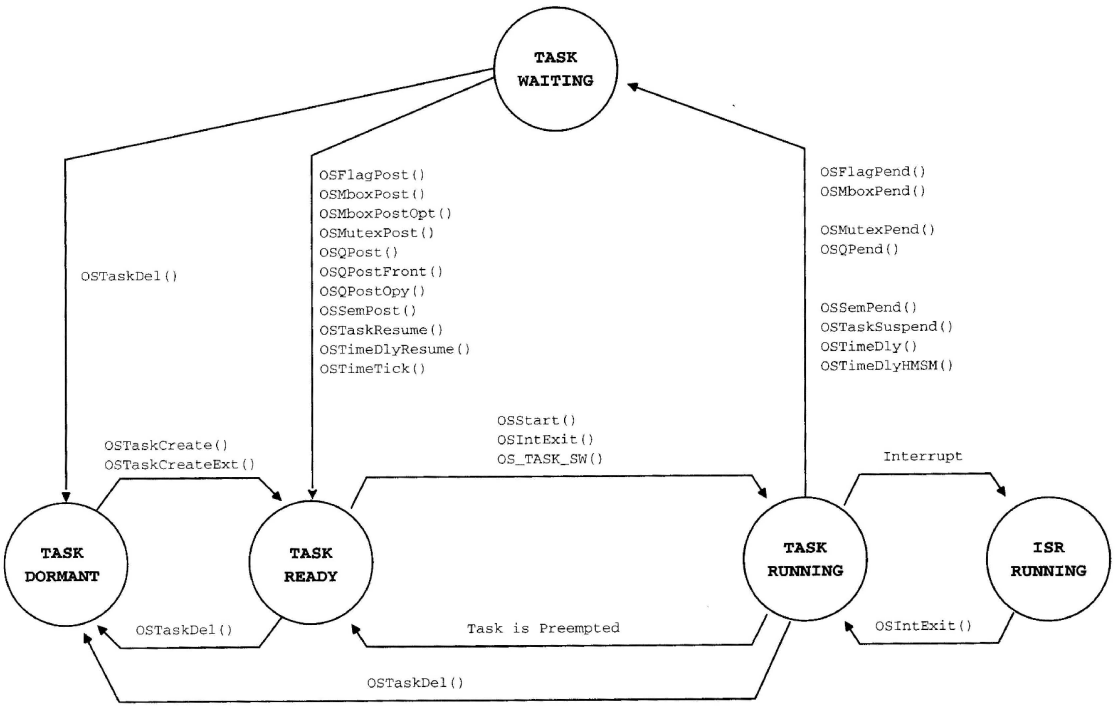
Code Layers

The following figure shows the µC/OS-II code layers, which is got from *Describing uCOS-II RTOS*, by *Guillaume Kremer*.



Tasks

The following figure shows the task states, which is got from *Describing uCOS-II RTOS*, by *Guillaume Kremer*.



- 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.

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 (**OS_TCB**.*OSTCBPrio*). A task is represented in the kernel by the structure **OS_TCB**.

- **Creating a task**

  OSTaskCreate() and **OSTaskCreateExt()** create new tasks. The priority given for the task must not be used yet by another task.

- **Deleting a task**

  OSTaskDel() deletes a task by is priority.

  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()**.

- **Changing the priority of a task**

  OSTaskChangePrio() changes the priority of a task. The priority of idle task cannot be changed.

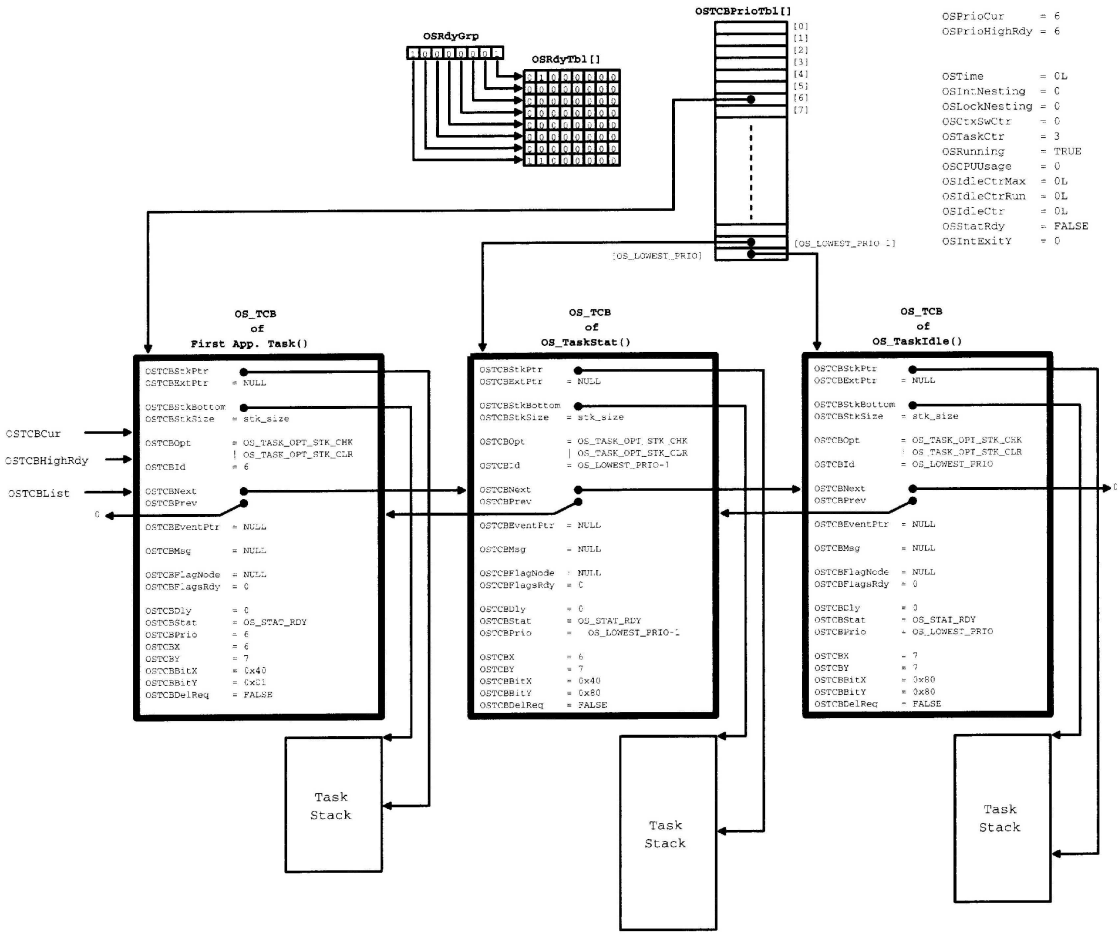
- **Suspending and resuming a task**

  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_STAT_RDY** 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 **OS_TCB** given as argument along with the task priority. The OS_TCB returned is a copy of the task with the priority specified.

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



Stack

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.

Event Control Block (ECB)

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:

```
#if (OS_EVENT_EN) && (OS_MAX_EVENTS > 0)
typedef struct os_event {
    /* Type of event control block (see OS_EVENT_TYPE_xxxx)    */
    INT8U    OSEventType;

    /* Pointer to message or queue structure                   */
    void    *OSEventPtr;

    /* Semaphore Count (not used if other EVENT type)          */
    INT16U   OSEventCnt;

#if OS_LOWEST_PRIO <= 63
    /* Group corresponding to tasks waiting for event to occur */
    INT8U    OSEventGrp;

    /* List of tasks waiting for event to occur                */
    INT8U    OSEventTbl[OS_EVENT_TBL_SIZE];
#else
    /* Group corresponding to tasks waiting for event to occur */
    INT16U   OSEventGrp;

    /* List of tasks waiting for event to occur                */
    INT16U   OSEventTbl[OS_EVENT_TBL_SIZE];
#endif

#if OS_EVENT_NAME_SIZE > 1
    INT8U    OSEventName[OS_EVENT_NAME_SIZE];
#endif
} OS_EVENT;
#endif
```

An ECB is an event type which sets how the block is used for. That's the possible values of *OSEventType* are:

- OS_EVENT_TYPE_MBOX
- OS_EVENT_TYPE_Q
- OS_EVENT_TYPE_SEM
- OS_EVENT_TYPE_MUTEX

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.

A user can call four functions to act on ECBs:

- **OS_EventWaitListInit()** initializes an ECB by signaling that no task is waiting.
- **OS_EventTaskRdy()** 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.

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.

- **Create a mailbox**

  OSMBoxCreate() creates a mailbox.

- **Deleting a mailbox**

  OSMboxDel() deletes a mailbox according to the options given in argument.

- **Getting a message**

  OSSemPend() waits for a message to be available. **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.

  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.

Message Queues

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 (ECB) specialized as a queue points to a queue control block. A queue control block is described by the following code:

```
#if OS_Q_EN > 0
typedef struct os_q {   /* QUEUE CONTROL BLOCK                          */
    /* Link to next queue control block in list of free blocks     */
    struct os_q   *OSQPtr;

    /* Pointer to start of queue data                              */
    void         **OSQStart;

    /* Pointer to end   of queue data                              */
    void         **OSQEnd;

    /* Pointer to where next message will be inserted  in   the Q  */
    void         **OSQIn;

    /* Pointer to where next message will be extracted from the Q  */
    void         **OSQOut;

    /* Size of queue (maximum number of entries)                   */
    INT16U         OSQSize;

    /* Current number of entries in the queue                      */
    INT16U         OSQEntries;
} OS_Q;


typedef struct os_q_data {
    /* Pointer to next message to be extracted from queue          */
    void          *OSMsg;

    /* Number of messages in message queue                         */
    INT16U         OSNMsgs;

    /* Size of message queue                                       */
    INT16U         OSQSize;

#if OS_LOWEST_PRIO <= 63
    /* List of tasks waiting for event to occur         */
    INT8U          OSEventTbl[OS_EVENT_TBL_SIZE];
    /* Group corresponding to tasks waiting for event to occur     */
    INT8U          OSEventGrp;
#else
    /* List of tasks waiting for event to occur         */
    INT16U         OSEventTbl[OS_EVENT_TBL_SIZE];
    /* Group corresponding to tasks waiting for event to occur     */
    INT16U         OSEventGrp;
#endif
} OS_Q_DATA;
#endif
```

- **Creating a message queue**

OSQCreate() gets and initializes an event control block and a queue control block. It then initializes all the pointer to the buffer. The type of the event control block (ECB) is set to **OS_EVENT_TYPE_Q**.

- **Deleting a message queue**

  OSQDel() delete a message queue.

- **Getting a message with blocking**

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

- **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**

  OSQPostOpt() replaces both **OSQPost()** and **OSQPostFront()**.

- **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.

- **Query a message queue**

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

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 (ECBs) 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 a Semaphore**

  Semaphores are created by calling the function **OSSemCreate()** with the

count value.

- **Deleting a Semaphore**

  Deleting a semaphore is made with the **OSSemDel()** function that takes the event control block (ECB), an option for deleting and an integer pointer for error output.

- **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.

  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_SEM.

- **Querying a semaphore**

  One can query information about a semaphore with **OSSemQuery()**.

Mutex

Mutexes provide exclusive access to resources. In µC/OS-II mutexes are made to avoid the priority inversion problem.

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 (ECB). This increases the chances to unblock the mutex. When the lower priority task frees the mutex, it gets back its initial priority.

The *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**

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

- **Deleting a mutex**

  Deleting a mutex is made with **OSMutexDel()**.

- **Waiting for a mutex**

  Waiting on a mutex in a blocking way is made with **OSMutexPend()**.

  Waiting without blocking is called with **OSMutexAccept()** which decreases and returns the *cnt* field of the mutex in argument.

- **Signaling a mutex**

  Signaling a mutex is done with **OSMutexPost()**.

- **Querying a mutex**

  One can query information about a mutex with **OSMutexQuery()**.

Event Flag Group (EFG)

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.**

```
typedef struct os_flag_grp {    /* Event Flag Group            */
    /* Should be set to OS_EVENT_TYPE_FLAG                      */
    INT8U         OSFlagType;

    /* Pointer to first NODE of task waiting on event flag     */
    void         *OSFlagWaitList;

    /* 8, 16 or 32 bit flags                                    */
    OS_FLAGS      OSFlagFlags;

#if OS_FLAG_NAME_SIZE > 1
    INT8U         OSFlagName[OS_FLAG_NAME_SIZE];
#endif
} OS_FLAG_GRP;

typedef struct os_flag_node {   /* Event Flag Wait List Node   */
    /* Pointer to next     NODE in wait list                    */
    void         *OSFlagNodeNext;

    /* Pointer to previous NODE in wait list                    */
    void         *OSFlagNodePrev;

    /* Pointer to TCB of waiting task                           */
    void         *OSFlagNodeTCB;

    /* Pointer to Event Flag Group                              */
    void         *OSFlagNodeFlagGrp;

    /* Event flag to wait on                                   */
    OS_FLAGS      OSFlagNodeFlags;

    INT8U         OSFlagNodeWaitType;   /* Type of wait:       */
                                        /* OS_FLAG_WAIT_AND    */
                                        /* OS_FLAG_WAIT_ALL    */
                                        /* OS_FLAG_WAIT_OR     */
                                        /* OS_FLAG_WAIT_ANY    */
} OS_FLAG_NODE;
```

- **Creating an Event flag group**

  Creating an event flag group is made with **OSFlagCreate()**.

- **Deleting an event flag group**

  This is made with **OSFlagDel()**.

- **Waiting for one or several events**

  A task running can sets itself waiting for events in an event flag group by calling **OSFlagPend()**.

- **Setting or clearing an event in an event flag group**

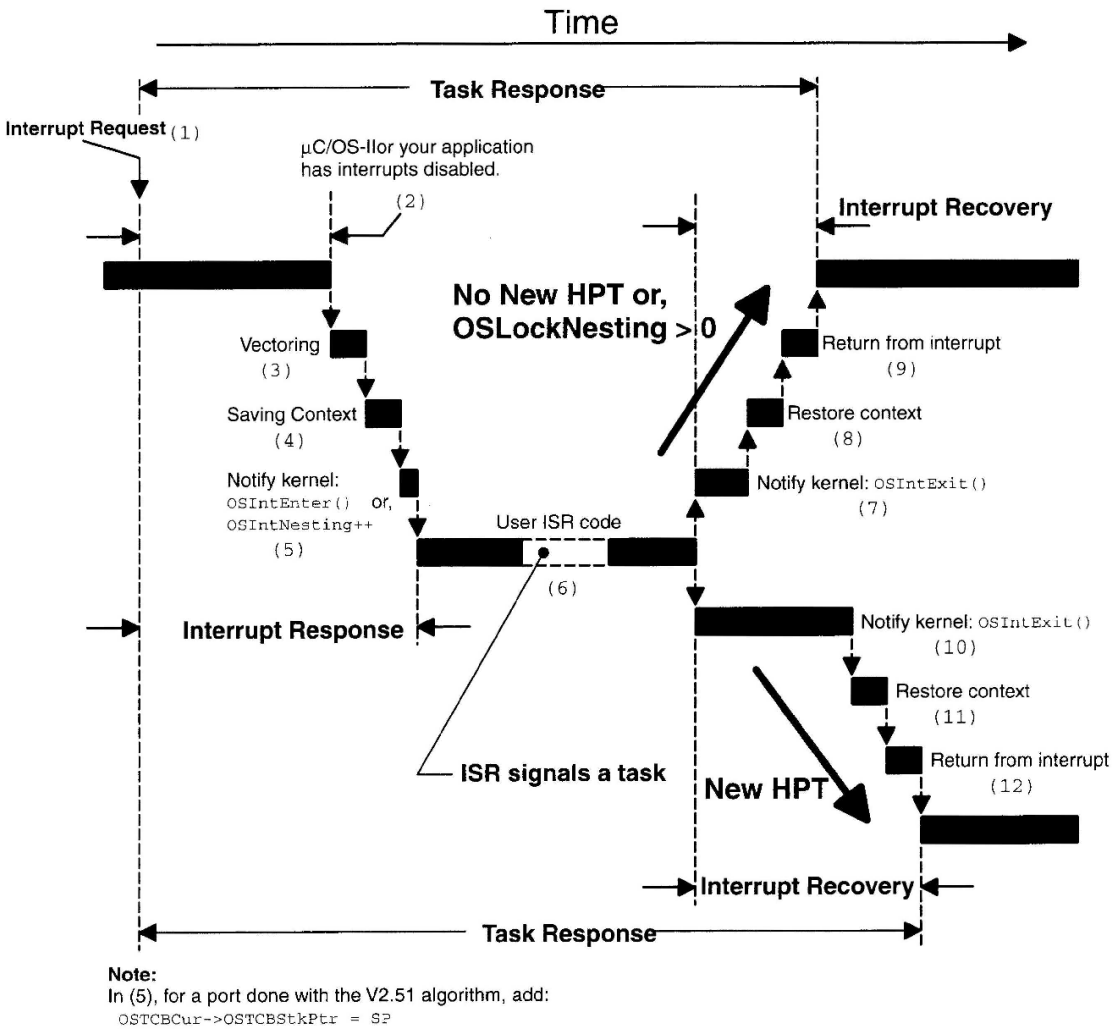
  This is done by **OSFlagPost()**. It can be called from within an ISR.

- **Checking if an event has happened**

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

Interrupt Service Routine (ISR)

The following figure shows a situation case after an interruption:



- 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.

Time Management

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.

- **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.

- **OSTimeResume()**

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

- **OSTimeGet()**

  Get the value of the OSTime counter.

- **OSTimeSet()**

  Set the value of the OSTime counter.

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:

```
#if (OS_MEM_EN > 0) && (OS_MAX_MEM_PART > 0)
typedef struct os_mem {     /* MEMORY CONTROL BLOCK        */
    /* Pointer to beginning of memory partition            */
    void   *OSMemAddr;

    /* Pointer to list of free memory blocks               */
    void   *OSMemFreeList;

    /* Size (in bytes) of each block of memory             */
    INT32U  OSMemBlkSize;

    /* Total number of blocks in this partition            */
    INT32U  OSMemNBlks;

    /* Number of memory blocks remaining in this partition */
    INT32U  OSMemNFree;

#if OS_MEM_NAME_SIZE > 1
    /* Memory partition name                               */
    INT8U   OSMemName[OS_MEM_NAME_SIZE];
#endif
} OS_MEM;


typedef struct os_mem_data {
    /* Pointer to the beginning address of the memory partition   */
    void   *OSAddr;

    /* Pointer to the beginning of the free list of memory blocks */
    void   *OSFreeList;

    /* Size (in bytes) of each memory block                       */
    INT32U  OSBlkSize;

    /* Total number of blocks in the partition                    */
    INT32U  OSNBlks;

    /* Number of memory blocks free                               */
    INT32U  OSNFree;

    /* Number of memory blocks used                               */
    INT32U  OSNUsed;
} OS_MEM_DATA;
#endif
```

- **Creating a partition**

  OSMemCreate() gets and initializes an OS_MEM structure from *OSMemFreeList*.

- **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.

- **Returning a block**

  OSMemPut() returns a block to a partition. The block size must be the same as in the partition.

- **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.

OS Components and Drivers

- **µC/FS**

  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, 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.

- **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.

Examples

```
int main()
{
    /* (1) OSInit() initialises microC/OS-II, which has to be called
     *     before creating any tasks. */
    OSInit();

    /* (2) Create a task by methods:
     *     OSTaskCreate(), or OSTaskCreateExt() */
    OSTaskCreate(..);

    /* (3) Start the kernel. */
    OSStart();

    /* (4) Other issues */
    return 0;
}
```

References

- Micrium Official Site (https://www.micrium.com)
- µC/OS-II Documentation Home (https://doc.micrium.com/display/osiidoc/home)
- µC/OS-III Documentation Home (https://doc.micrium.com/display/osiiidoc/uC-OS-III+Documentation+Home)

- *Describing uCOS-II RTOS*, published on November 2009, by *Guillaume Kremer*
- *MicroC/OS-II: The Real-Time Kernel, Second Edition*