Micriμm

μC/OS-II

Kernel Awareness

www.micrium.com

Table of Contents

1.0 Introduction

This document describes the different variables that need to be displayed in order to implement 'Kernel Awareness' for µC/OS-II. It is assumed that you have access to the source code for µC/OS-II (downloadable from www.micrium.com) as well as the µC/OS-II book (ISBN 978-1-57820-103-7) which is available from Amazon.com or other fine book stores.

In this document we will present 'screen shots'. These are only provided as examples of what has previously been done. However, you are free to implement the Kernel Awareness that best suits your debug environment.

µC/OS-II performs substantial run-time statistics that can be displayed by kernel-aware debuggers and/or µC/Probe. µC/OS-II also provides information about the configuration of the system. Specifically, the amount of RAM used by µC/OS-II, including all internal variables and task stacks.

We will be using a `Courier New` font to represent variable or function names found in µC/OS-II. Kernel Awareness actually only requires the reading and possibly writing of variables. Functions are only mentioned as a reference.

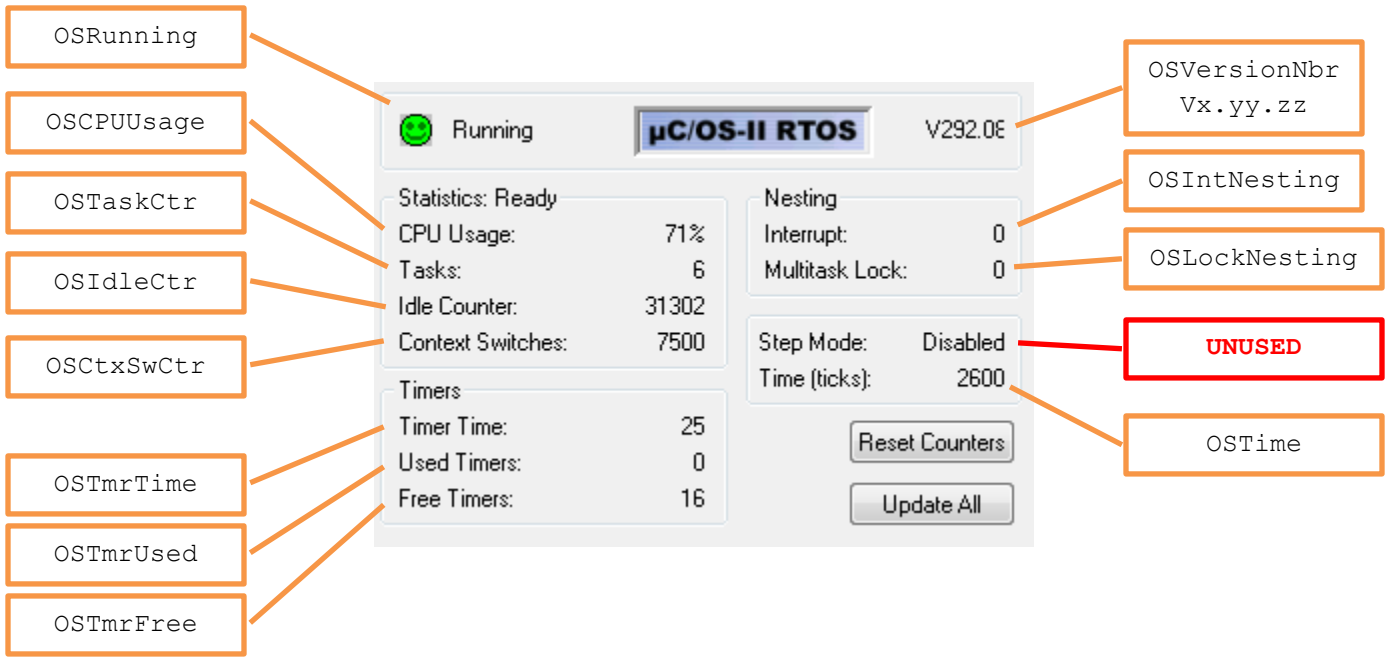
Throughout the text in this document, you will see references in angular brackets about data types. These are 'compiler specific' and, for ARM, are defined as follows:

```
/*
*********************************************************************************************
*                                          DATA TYPES
*                                      (Compiler Specific)
*********************************************************************************************
*/

typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;                    /* Unsigned  8 bit quantity                 */
typedef signed   char  INT8S;                    /* Signed    8 bit quantity                 */
typedef unsigned short INT16U;                   /* Unsigned 16 bit quantity                 */
typedef signed   short INT16S;                   /* Signed   16 bit quantity                 */
typedef unsigned int   INT32U;                   /* Unsigned 32 bit quantity                 */
typedef signed   int   INT32S;                   /* Signed   32 bit quantity                 */
typedef float          FP32;                     /* Single precision floating point          */
typedef double         FP64;                     /* Double precision floating point          */

typedef unsigned int   OS_STK;                   /* Each stack entry is 32-bit wide          */
typedef unsigned int   OS_CPU_SR;                /* Define size of CPU status register       */
```

'ASCII string' means that the value is a pointer to an ASCII string that is NUL terminated.

The information found in this document is assumed to be correct but is not guaranteed to be without mistakes or omissions.

2.0 General Status



2.1 OSRunning <BOOLEAN>

`OSRunning` is set to 1 when the kernel is running (multitasking has started) and 0 when not. You can replace the value by a symbol and/or a text string as shown above.

2.2 OSCPUUsage <INT8U>

`OSCPUUsage` contains CPU usage in 1% increment (0 to 100). The '%' is added in the formatting. This variable ONLY exists if the variable `OSTaskStatEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

2.3 OSTaskCtr <INT8U>

`OSTaskCtr` indicates the total number of tasks created in the application. This includes the idle task and statistic task (if enabled).

2.4 OSIdleCtr <INT32U>

`OSIdleCtr` increments whenever µC/OS-II is in the idle task. OSIdleCtr is reset every 100 ms by `OS_TaskStat()`.

2.5 OSCtxSwCtr <INT32U>

`OSCtxSwCtr` increments whenever µC/OS-II performs a context switch. When `OSCtxSwCtr` overflows, it goes back to 0.

2.6 OSTmrTime <INT32U>

`OSTmrTime is` incremented by `OSTmr_Task()` which typically executes every 100 ms. When `OSTmrTime` overflows, it goes back to 0. This variable ONLY exists if the variable `OSTmrEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

2.7 OSTmrUsed <INT16U>

`OSTmrUsed` indicates how many timers are actually in use (created). This variable ONLY exists if the variable `OSTmrEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

2.8 OSTmrFree <INT16U>

`OSTmrFree` indicates how many timers are available from the timer pool. This variable ONLY exists if the variable `OSTmrEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

2.9 OSVersionNumber <INT16U>

`OSVersionNbr` is a 16-bit value and represents the version number as '`xyyzz`'. When displaying, you should separate the fields using decimal points, i.e. `x.yy.zz`.

2.10 OSIntNesting <INT8U>

`OSIntNesting` indicates the level of interrupt nesting. If this value is 2, for example, this indicates that an interrupt was interrupted by another interrupt.

2.11 OSLockNesting <INT8U>

`OSLockNesting` indicates the number of times `OSSchedLock()` has been called. When `OSLockNesting` is greater than 0 then the scheduler is locked and rescheduling will not occur until `OSLockNesting` is decremented back to 0 by calling `OSSchedUnlock()`. Note that `OSTaskDel()` also increments `OSLockNesting` for a brief instant to prevent rescheduling while deleting a task.

2.12 Step Mode

This functionality has been removed.

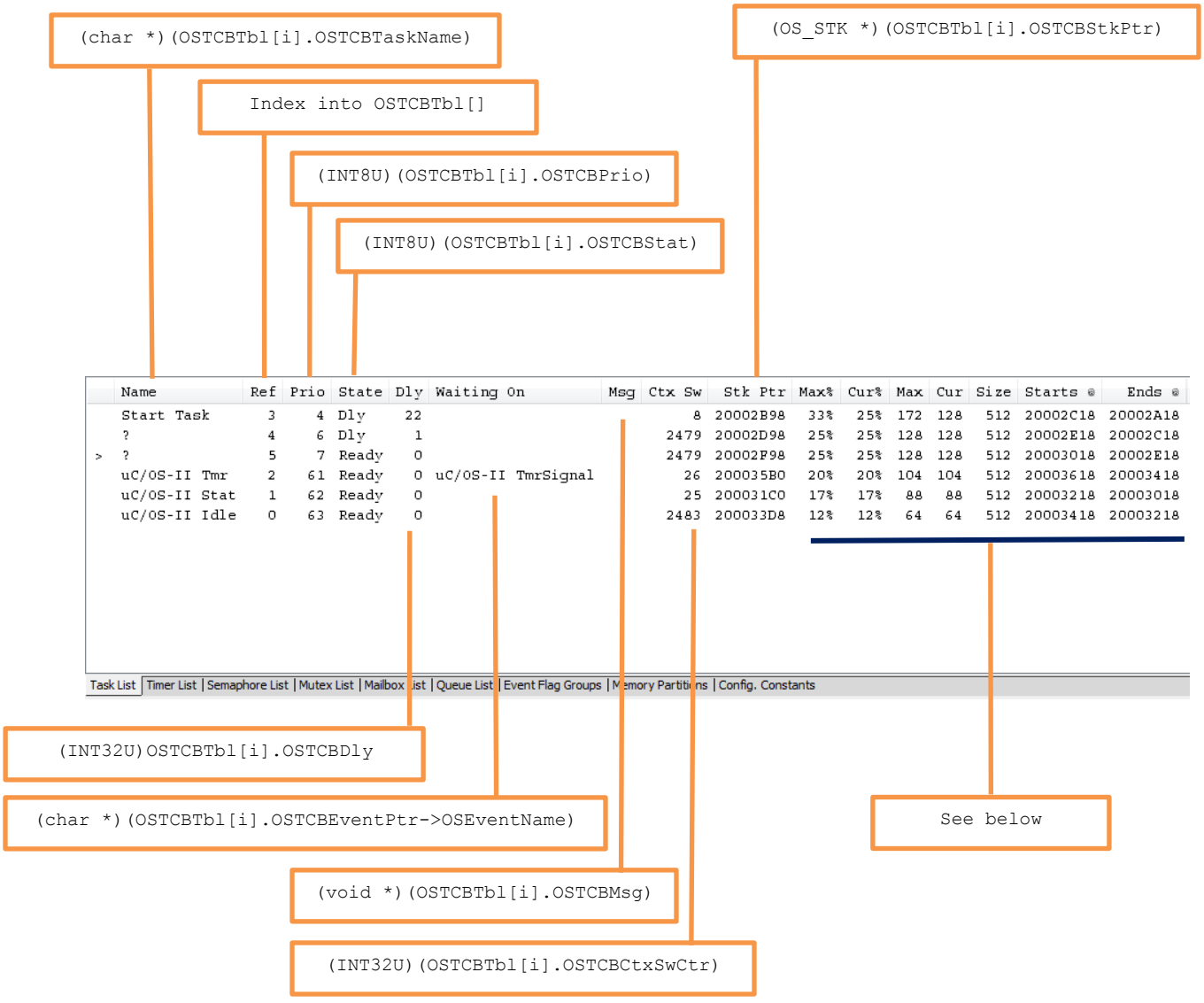
2.13 OSTime <INT32U>

`OSTime` is the tick counter and is incremented by `OSTimeTick()`. When `OSTime` overflows, it goes back to 0.

3.0 Task List

The task list shows the contents of each task's TCB (Task Control Block). Each task is assigned a TCB when the task is created. TCBs are assigned by the kernel from a pool of TCBs. There are `OSTaskMax` entries in the TCB pool. When a task is deleted, the TCB of that task is returned to the TCB pool. The pool consist of a linked list created from the entries in `OSTCBTbl[]`.

You don't actually need to access the linked list. Instead, you index into `OSTCBTbl[]` from 0 to `OSTaskCtr-1`.



3.01 .OSTCBTaskName <ASCII string>

`OSTCBTbl[i].OSTCBTaskName` is a pointer to a `NUL` terminated ASCII string. This variable ONLY exists if the variable `OSTaskNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

3.02 Index into OSTCBTbl[]

The value of '`i`' above references an entry in `OSTCBTbl[]`. The task list will contain at most `OSTaskCtr` entries numbered `0` to `OSTaskCtr-1`.

3.03 .OSTCBPrio <INT8U>

`OSTCBTbl[i].OSTCBPrio` indicates the priority of the task. A low number indicates a high priority.

3.04 .OSTCBStat <INT8U>

`OSTCBTbl[i].OSTCBStat` is the current state of the task:

0x00	Ready
0x01	Pending on Semaphore
0x02	Pending on Mailbox
0x04	Pending on Message Queue
0x08	Suspended
0x10	Pending on Mutex
0x20	Pending on Event Flags
0x40	
0x80	Pending on multiple events
0x09	Pending on Semaphore and Suspended
0x0A	Pending on Mailbox and Suspended
0x0C	Pending on Message Queue and Suspended
0x18	Pending on Mutex and Suspended
0x28	Pending on Event Flags and Suspended

3.05 .OSTCBDly <INT32U>

`OSTCBTbl[i].OSTCBDly` indicates whether a task is waiting for a timeout or is simply suspended waiting for time to expire. If any of the bits in `OSTCBTbl[i].OSTCBStat` (except 0x08) is set then the task is waiting for an event and `OSTCBTbl[i].OSTCBDly` specifies the timeout.

If OSTCBTbl[i].OSTCBStat is 0x00 (i.e. Ready) but OSTCBTbl[i].OSTCBDly is non-zero then the task is actually 'sleeping' and will wake up when the time delay reaches 0 or, the time delay has been cancelled by calling OSTimeDlyResume().

3.06 .OSTCBEventPtr->OSEventName <ASCII string>

OSTCBTbl[i].OSTCBEventPtr->OSEventName is a pointer to an ASCII string that contains the name of the event the task is waiting on, assuming the task is waiting for a Semaphore, a Mailbox, a Message Queue, an Event Flag Group or a Mutex. The first byte where OSTCBTbl[i]->OSTCBEventPtr is pointing at tells you whether the data structure being pointed to is an OS_EVENT (Semaphore, a Mailbox, a Message Queue or a Mutex) or an OS_FLAG_GRP (Event Flags).

This variable ONLY exists if the variable OSEventNameEn is set to 1 which is a ROM variable declared in OS_DBG.C.
So:

Value of: ***(INT8U *)(OSTCBTbl[i]->OSTCBEventPtr)**	**Event Type**	**Use** **Data Type**
0	Unused	N/A
1	Mailbox	OS_EVENT
2	Message Queue	OS_EVENT
3	Semaphore	OS_EVENT
4	Mutex	OS_EVENT
5	Event Flag	OS_FLAG_GRP

From the value above, you are able to access the ASCII name as follows:

Value is 1, 2, 3 or 4:

```
OSTCBTbl[i]->OSTCBEventPtr->OSEventName
```

Value is 5:

A) This is a bit more complex. First, you need to obtain the address (i.e. a pointer to) of the OS_FLAG_NODE from the TCB:

```
(OS_FLAG_NODE *)OSTCBTbl[i]->OSTCBFlagNode
```

B) Then, from this pointer, get the address of the event flag group:

```
(OS_FLAG_GRP *)FlagNodePtr->OSFlagNodeFlagGrp
```

C) Then, from this pointer, get the address of the ASCII string:

```
(char *)FlagGrpPtr->OSFlagName
```

3.07 . OSTCBMsg <void *>

`OSTCBTbl[i].OSTCBMsg` contains a pointer to the last message received. This field contains a NULL pointer whenever the message is delivered to the waiting task. This variable ONLY exists if the at least ONE of the following variables is set to 1: `OSQEn` or OSMboxEn. These are declared in `OS_DBG.C`.

3.08 . OSTCBCtxSwCtr <INT32U>

`OSTCBTbl[i].OSTCBCtxSwCtr` indicates the number of times a task has been switched in. When 0, it indicates that a task has not executed yet. This variable ONLY exists if the variable `OSTaskProfileEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

3.09 . OSTCBStkPtr <OS_STK *>

`OSTCBTbl[i].OSTCBStkPtr` is a pointer to the top-of-stack of each task. Recall that with µC/OS-II, the stack frame pointed to by `OSTCBTbl[i].OSTCBStkPtr` looks as if an interrupt just occurred and the CPU registers were pushed onto the task's stack. This is true for all tasks except the current task that is executing. In this case, the CPU's stack pointer points to the current stack frame and the `OSTCBTbl[i].OSTCBStkPtr` entry points to the stack frame when the task was last suspended. However, there is little value to the programmer for this information.

3.10 Other Stack Data

The next seven columns are provided to display stack usage data.

```
Max%      = OSTCBTbl[i].OSTCBStkUsed * 100
          / OSTCBTbl[i].OSTCBStkSize;

Cur%      = No longer used

Max       = OSTCBTbl[i].OSTCBStkUsed;

Size      = OSTCBTbl[i].OSTCBStkSize;

Starts @  = OSTCBTbl[i].OSTCBStkBottom
          + OSTCBTbl[i].OSTCBStkSize;

Ends   @  = OSTCBTbl[i].OSTCBStkBottom;
```

4.0 Constants

The file `OS_DBG.C` provides information about the RAM usage of µC/OS-II as shown below. All values are `INT16U` except for `OSEndiannessTest` which is a 32 bit value.

`OSEndiannessTest` is a variable that can be examined to determine whether the CPU is a big or little endian machine. If the byte found at the base address of `OSEndiannessTest` is `0x12` then the CPU is a little endian machine. If the value is `0x78` then it's a big endian machine.

`OSDataSize` indicates the total RAM size used by µC/OS-II which includes the RAM needed for the idle and statistics task stacks.

Name	Value
Endianness	Little
OSDataSize	8506
OSDebugEn	1
OSEventEn	1
OSEventMax	175
OSEventNameEn	1
OSEventSize	24
OSEventTblSize	4200
OSFlagEn	1
OSFlagGrpSize	16
OSFlagMax	5
OSFlagNameEn	1
OSFlagNodeSize	20
OSFlagWidth	2
OSLowestPrio	63
OSMboxEn	1
OSMemEn	1
OSMemMax	5
OSMemNameEn	1
OSMemSize	24
OSMemTblSize	120
OSMutexEn	1
OSPtrSize	4
OSQEn	1
OSQMax	4
OSQSize	24
OSRdyTblSize	8
OSSemEn	1
OSStkWidth	4
OSTaskCreateEn	1
OSTaskCreateExtEn	1
OSTaskDelEn	1
OSTaskIdleStkSize	128
OSTaskMax	22
OSTaskNameEn	1
OSTaskProfileEn	1
OSTaskStatEn	1
OSTaskStatStkChkEn	1
OSTaskStatStkSize	128
OSTaskSwHookEn	1
OSTCBPrioTblMax	64
OSTCBSize	88
OSTicksPerSec	1000
OSTimeTickHookEn	1
OSTmrCfgMax	16
OSTmrCfgNameEn	1
OSTmrCfgTicksPerSec	10
OSTmrCfgWheelSize	8
OSTmrEn	1
OSTmrSize	40
OSTmrTblSize	640
OSTmrWheelSize	8
OSTmrWheelTblSize	64
OSVersionNbr	29208

Task List | **Config. Constants**

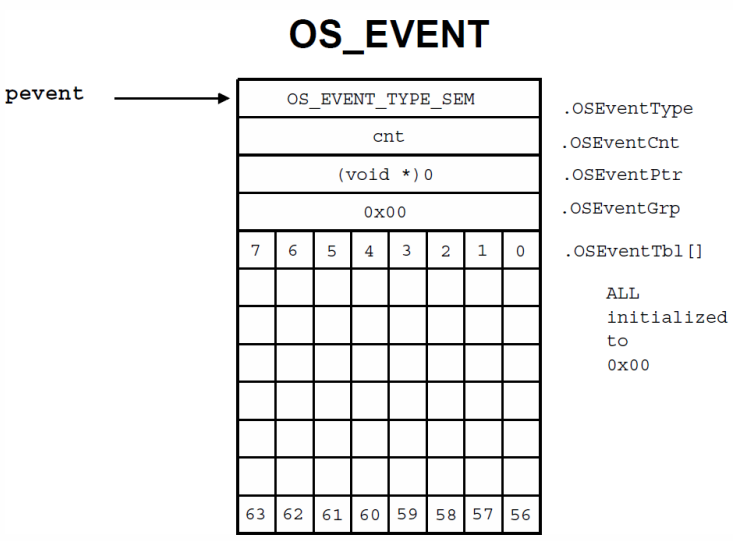
5.0 Semaphore List

The kernel awareness semaphore list ONLY exists if `OSSemEn` is set to 1 (see `OS_DBG.C`).

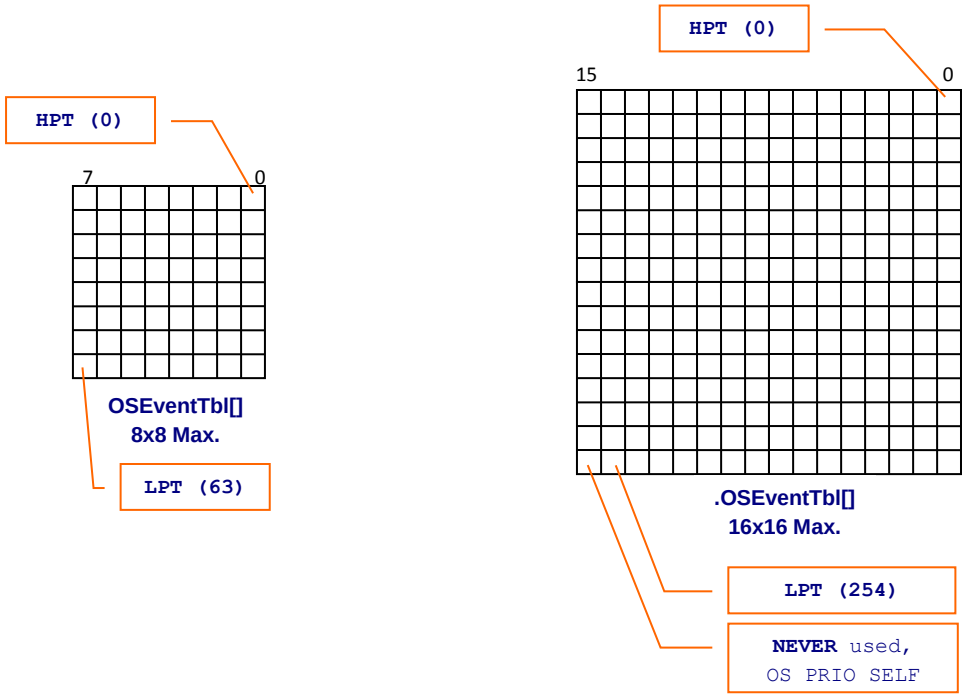
Semaphores are created by calling `OSSemCreate()` and when a semaphore is created, an `OS_EVENT` structure is assigned from a pool of `OS_EVENT`s. You should note that an `OS_EVENT` is also used to store a mailbox, queue or mutex. To distinguish between these different events, the first byte of an `OS_EVENT` specifies the event type:

Event Type	Value of `.OSEventType`
Unused	0
Mailbox	1
Queue	2
Semaphore	**3**
Mutex	4
Event Flag	5

The `OS_EVENT` structure as used by semaphores is shown below:



The bitmap represented by `.OSEventTbl[]` indicates which task is waiting for the semaphore. A 1 in the bitmap indicates that a task (at the priority corresponding to the bit position) is waiting for the semaphore. For example, if bit 3 in `.OSEventTbl[0]` is set to 1 then, task at priority 3 is waiting for the semaphore. As of V2.80, µC/OS-II supports up to 255 tasks and thus the above 8x8 table can actually be a 16x16 table as shown below. The variable '`OSLowestPrio`' can be read by the debugger to determine whether `.OSEventTbl[]` contains `INT8U` or `INT16U` entries.



OS_LOWEST_PRIO <= 63 **OS_LOWEST_PRIO > 63**

To display the semaphore list, you will need to scan the `OSEventTbl[]` and display the contents of the `OSEventTbl[]` for those entries that have `OSEventTbl[i].OSEventType` equal to `3`. Note that `.OSEventType` is an `INT8U`.

Example display:

Name	Ref	Count	Tasks Waiting	OS_EVENT @
Create Sem	5	0	-	0x20000078
Serial Lock	4	1	-	0x20000060
Serial Rx Wait	3	0	-	0x20000048
Serial Tx Wait	2	0	-	0x20000030
uC/OS-II TmrLock	0	1	-	0x20000000
uC/OS-II Tmr Signal	1	0	61-uC/OS-II Tmr	0x20000018

5.01 Name <ASCII string>

`OSEventTbl[i].OSEventName` is a pointer to a `NUL` terminated ASCII string. Note that this field is not shown in the illustration above because it was added after the illustration was done. This variable ONLY exists if the variable `OSEventNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

5.02 Ref.

This corresponds to the index into `OSEventTbl[]` for the semaphore.

5.03 Count <INT16U>

This entry corresponds to the semaphore count value (a 16-bit value) and is found in `OSEventTbl[i].OSEventCnt`.

5.04 Tasks Waiting <ASCII string>

There are two methods you can use to determine which tasks are waiting for the semaphore.

1) You can scan the `OSTCBTbl[]` and find which `OSTCBTbl[j].OSTCBEventPtr` points to the `OSEventTbl[]` entry you are displaying. In other words, assuming that '`i`' corresponds to the current `OSEventTbl[]` entry and '`j`' to the scanned TCB table:

```
for (j = 0; j < OSTaskCtr; j++) {
    if (OSTCBTbl[j].OSTCBEventPtr == &OSEventTbl[i]) {
        Print the name of the task (i.e. OSTCBTbl[j].OSTCBTaskName)
    }
}
```

2) You can also scan the .`OSEventTbl[]` and find which bit are set in the table. The bit position corresponds to the task priority that is waiting for the semaphore. From this, you can use the priority number to index into `OSTCBTbl[]` and determine the name of the task waiting for the semaphore (i.e. `OSTCBTbl[prio].OSTCBTaskName`).

You should note that in the 'Task Waiting' column, you could add the task priority in front of the task name as shown: "`prio-task name`". The task priority is obtained by `OSTCBTbl[j].OSTCBPrio`.

This variable ONLY exists if the variable `OSTaskNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

5.05 OS_EVENT @ <Hex address>

This corresponds to the address of the `OSEventTbl[]` entry, i.e. `&OSEventTbl[i]`.

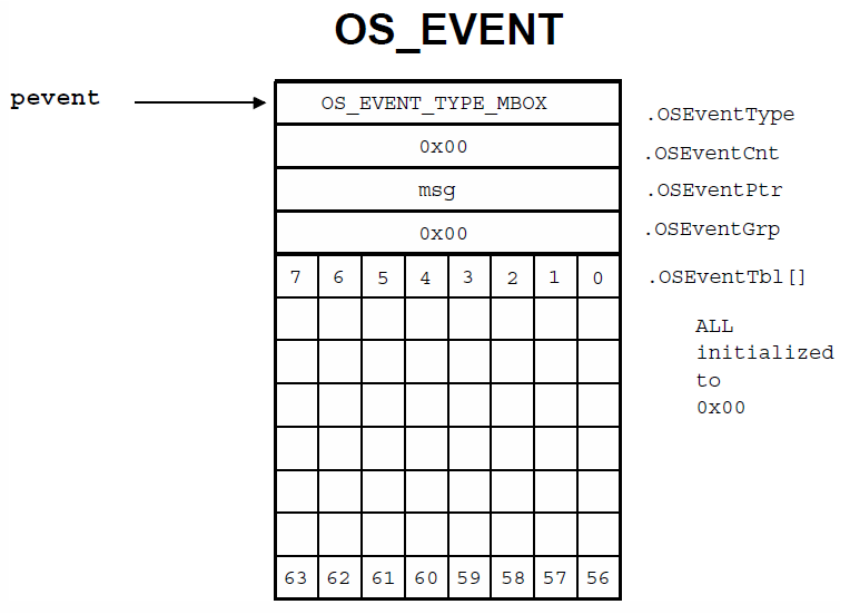
6.0 Mailbox List

The kernel awareness mailbox list ONLY exists if `OSMboxEn` is set to 1 (see `OS_DBG.C`).

Mailboxes are created by calling `OSMboxCreate()` and when a mailbox is created, an `OS_EVENT` structure is assigned from a pool of `OS_EVENT`s. You should note that an `OS_EVENT` is also used to store a semaphore, queue or mutex. To distinguish between these different events, the first byte of an `OS_EVENT` specifies the event type:

Event Type	Value of `.OSEventType`
Unused	0
Mailbox	**1**
Queue	2
Semaphore	3
Mutex	4
Event Flag	5

The `OS_EVENT` structure as used to hold a mailbox object is shown below.



To display the mailbox, you will need to scan the `OSEventTbl[]` and display the contents of the `OSEventTbl[]` for those entries that have `OSEventTbl[i].OSEventType` equal to `1`. Note that `.OSEventType` is an `INT8U`.

Example display:

Name	Ref	Msg	Tasks Waiting	OS_EVENT @
UART Rx Mbx	6	0	16-UART Rx Task	0x20000090
UART Tx Mbx	7	0x200010F0	-	0x200000A8

6.01 Name <ASCII string>

`OSEventTbl[i].OSEventName` is a pointer to a `NUL` terminated ASCII string. Note that this field is not shown in the illustration above because it was added after the illustration was done. This variable ONLY exists if the variable `OSEventNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

6.02 Ref.

This corresponds to the index into `OSEventTbl[]` for the mailbox.

6.03 Msg <Hex address>

This entry corresponds to the content of the mailbox. The mailbox is empty when this field contains a NULL pointer. Any non-NULL pointer corresponds to a message that was posted to the mailbox. The message is placed in the `OSEventTbl[i].OSEventPtr` field.

6.04 Tasks Waiting <ASCII string>

There are two methods you can use to determine which tasks are waiting for the mailbox.

1) You can scan the `OSTCBTbl[]` and find which `OSTCBTbl[j].OSTCBEventPtr` points to the `OSEventTbl[]` entry you are displaying. In other words, assuming that '`i`' corresponds to the current `OSEventTbl[]` entry and '`j`' to the scanned TCB table:

```
for (j = 0; j < OSTaskCtr; j++) {
    if (OSTCBTbl[j].OSTCBEventPtr == &OSEventTbl[i]) {
        Print the name of the task (i.e. OSTCBTbl[j].OSTCBTaskName)
    }
}
```

2) You can also scan the .`OSEventTbl[]` and find which bit are set in the table. The bit position corresponds to the task priority that is waiting for the mailbox. From this, you can use the priority number to index into `OSTCBTbl[]` and determine the name of the task waiting for the mailbox (i.e. `OSTCBTbl[prio].OSTCBTaskName`).

You should note that in the 'Task Waiting' column, you could add the task priority in front of the task name as shown: "`prio-task name`". The task priority is obtained by `OSTCBTbl[j].OSTCBPrio`.

This variable ONLY exists if the variable `OSTaskNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

6.05 OS_EVENT @ <Hex address>

This corresponds to the address of the `OSEventTbl[]` entry, i.e. `&OSEventTbl[i]`.

7.0 Message Queue List

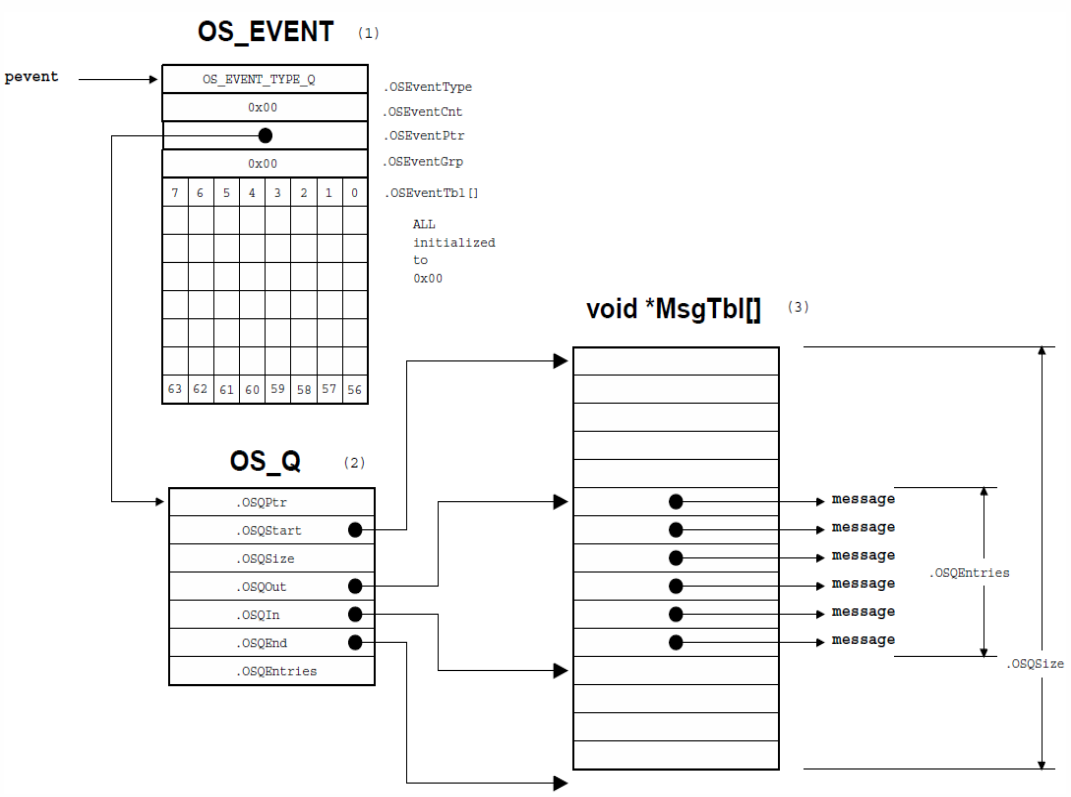
The kernel awareness message queue list ONLY exists if `OSQEn` is set to 1 (see `OS_DBG.C`).

Message queues are created by calling `OSQCreate()` and when a queue is created, an `OS_EVENT` structure is assigned from a pool of `OS_EVENT`s. You should note that an `OS_EVENT` is also used to store a semaphore, mailbox or mutex. To distinguish between these different events, the first byte of an `OS_EVENT` specifies the event type:

Event Type	Value of `.OSEventType`
Unused	0
Mailbox	1
Queue	**2**
Semaphore	3
Mutex	4
Event Flag	5

A message queue actually uses another data structure (`OS_Q`) which is allocated at the same time as the `OS_EVENT` structure. This is because an `OS_EVENT` doesn't contain all the fields needed to implement a message queue. The `OS_Q` data structure is pointed to by `.OSEventPtr` when the queue is created.

The `OS_EVENT` and `OS_Q` structures are shown below.



To display the mailbox, you will need to scan the `OSEventTbl[]` and display the contents of the `OSEventTbl[]` for those entries that have `OSEventTbl[i].OSEventType` equal to `2`. Note that `.OSEventType` is an `INT8U`.

Example display:

Name	Ref	Entries	Size	Next Msg	Tasks Waiting	OS_EVENT @	OS_Q @
UART Rx Q	8	5	10	0x20001000	-	0x200000C0	0x20001000
UART Tx Q	9	0	20	0x200010F0	15-UART Tx	0x200000D8	0x20001020
PID Ctrl	10	0	8	-	10-PID 1	0x200000E0	0x20001040
					11-PID 2		
					13-PID 4		

7.01 Name <ASCII string>

`OSEventTbl[i].OSEventName` is a pointer to a `NUL` terminated ASCII string. Note that this field is not shown in the illustration above because it was added after the illustration was done. This variable ONLY exists if the variable `OSEventNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

7.02 Ref.

This corresponds to the index into `OSEventTbl[]` for the message queue.

7.03 Entries <INT16U>

This entry corresponds to the number of messages currently placed in the message queue. To access this field you need to obtain the address of the OS_Q structure as follows:

```
OS_Q  *p_q;

p_q     = (OS_Q *)OSEventTbl[i].OSEventPtr;
Entries = p_q->OSQEntries;
```

7.04 Size <INT16U>

This entry corresponds to the maximum number of messages that can be placed in the message queue. To access this field you need to obtain the address of the OS_Q structure as follows:

```
OS_Q  *p_q;

p_q  = (OS_Q *)OSEventTbl[i].OSEventPtr;
Size = p_q->OSQSize;
```

7.05 Next Msg <void *>

This entry corresponds to the next message that will be extracted from the message queue.

```
OS_Q  *p_q;
void  *p_msg

p_q     = (OS_Q *)OSEventTbl[i].OSEventPtr;
NextMsg = *p_q->OSQOut;
```

7.06 Tasks Waiting <ASCII string>

There are two methods you can use to determine which tasks are waiting for the message queue.

1) You can scan the `OSTCBTbl[]` and find which `OSTCBTbl[j].OSTCBEventPtr` points to the `OSEventTbl[]` entry you are displaying. In other words, assuming that '`i`' corresponds to the current `OSEventTbl[]` entry and '`j`' to the scanned TCB table:

```
for (j = 0; j < OSTaskCtr; j++) {
    if (OSTCBTbl[j].OSTCBEventPtr == &OSEventTbl[i]) {
        Print the name of the task (i.e. OSTCBTbl[j].OSTCBTaskName)
    }
}
```

2) You can also scan the .`OSEventTbl[]` and find which bit are set in the table. The bit position corresponds to the task priority that is waiting for the message queue. From this, you can use the priority number to index into `OSTCBTbl[]` and determine the name of the task waiting for the message queue (i.e. `OSTCBTbl[prio].OSTCBTaskName`).

You should note that in the 'Task Waiting' column, you could add the task priority in front of the task name as shown: "`prio-task name`". The task priority is obtained by `OSTCBTbl[j].OSTCBPrio`.

This variable ONLY exists if the variable `OSTaskNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

7.07 OS_EVENT @ <Hex address>

This corresponds to the address of the `OSEventTbl[]` entry, i.e. `&OSEventTbl[i]`.

7.08 OS_Q @ <Hex address>

This corresponds to the address of the `OS_Q` used by the message queue and corresponds to: `OSEventTbl[i].OSEventPtr`.

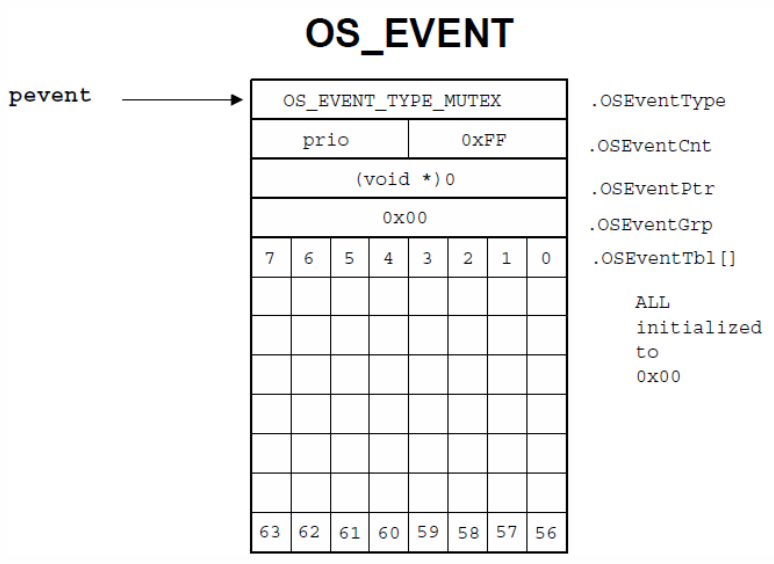
8.0 Mutex List

The kernel awareness mutex list ONLY exists if `OSMutexEn` is set to 1 (see `OS_DBG.C`).

Mutexes are created by calling `OSMutexCreate()` and when a mutex is created, an `OS_EVENT` structure is assigned from a pool of `OS_EVENT`s. You should note that an `OS_EVENT` is also used to store a mailbox, queue or semaphore. To distinguish between these different events, the first byte of an `OS_EVENT` specifies the event type:

Event Type	Value of `.OSEventType`
Unused	0
Mailbox	1
Queue	2
Semaphore	3
Mutex	**4**
Event Flag	5

The `OS_EVENT` structure (as used for Mutexes) is shown below:



To display the mutex list, you will need to scan the `OSEventTbl[]` and display the contents of the `OSEventTbl[]` for those entries that have `OSEventTbl[i].OSEventType` equal to `4`. Note that `.OSEventType` is an `INT8U`.

Example display:

Name	Ref	PIP:Owner	Tasks Waiting	OS_EVENT @
Display Mutex	11	20:25	26-User I/F Task	0x200000F8
			27-Line Draw Task	
			28-Temperature Update Task	
			33-Pressure Update Task	
SPI Mutex	12	10:Avail	-	0x20000110

8.01 Name <ASCII string>

`OSEventTbl[i].OSEventName` is a pointer to a `NUL` terminated ASCII string. Note that this field is not shown in the illustration above because it was added after the illustration was done. This variable ONLY exists if the variable `OSEventNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

8.02 Ref.

This corresponds to the index into `OSEventTbl[]` for the mutex.

8.03 PIP:Owner <INT16U>

This entry contains two fields. The lower 8 bits contains either the priority of the owner task or 0xFF. The mutex is available when 0xFF and, the lower 8 bits contains the priority of the owner when a task acquires the mutex.

The upper 8 bits contains the 'priority ceiling' priority. In other words, if a low priority task owns the mutex and a higher priority task needs to access the shared resource then the low priority task will get its priority raised to the 'priority ceiling' priority in order to reduce priority inversions.

This column should indicate:

```
PIP = OSEventTbl[i].OSEventCnt >> 8;
If ((OSEventTbl[i].OSEventCnt & 0xFF) == 0xFF)
    Display "PIP:Avail"
Else
    Display "PIP:prio"   // prio is the value of (in decimal)
                         //    OSEventTbl[i].OSEventCnt & 0xFF
```

8.04 Tasks Waiting <ASCII string>

There are two methods you can use to determine which tasks are waiting for the mutex.

1) You can scan the `OSTCBTbl[]` and find which `OSTCBTbl[j].OSTCBEventPtr` points to the `OSEventTbl[]` entry you are displaying. In other words, assuming that '`i`' corresponds to the current `OSEventTbl[]` entry and '`j`' to the scanned TCB table:

```
for (j = 0; j < OSTaskCtr; j++) {
    if (OSTCBTbl[j].OSTCBEventPtr == &OSEventTbl[i]) {
        Print the name of the task (i.e. OSTCBTbl[j].OSTCBTaskName)
    }
}
```

2) You can also scan the .`OSEventTbl[]` and find which bit are set in the table. The bit position corresponds to the task priority that is waiting for the mutex. From this, you can use the priority number to index into `OSTCBTbl[]` and determine the name of the task waiting for the mutex (i.e. `OSTCBTbl[prio].OSTCBTaskName`).

This variable ONLY exists if the variable `OSTaskNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

8.05 OS_EVENT @ <Hex address>

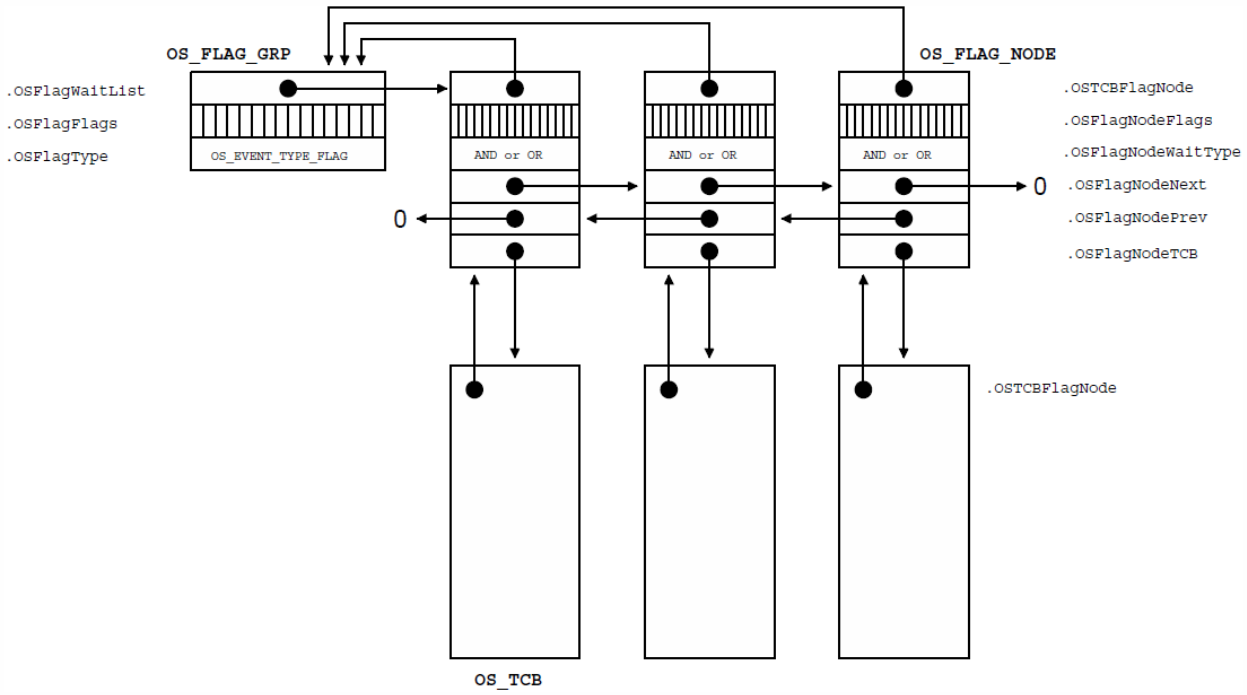
This corresponds to the address of the `OSEventTbl[]` entry, i.e. `&OSEventTbl[i]`.

9.0 Event Flag List

The kernel awareness event flag list ONLY exists if `OSFlagEn` is set to 1 (see `OS_DBG.C`).

Event flags are created by calling `OSFlagCreate()` and when an event flag is created, an `OS_FLAG_GRP` structure is assigned from a pool of `OS_FLAG_GRP`s.

To display the event flag list, you will need to scan a linked list of items (see Figure 9.2 of the µC/OS-II book, replicated here) and display multiple 'lines' based on the length of the list.



The following sub-section explains what each of the columns in the display above should contain.

Example display:

Name	Ref	Flags	OS_FLAG_GRP @	Tasks Waiting	Wait Type	Waiting for Flags
Engine Flags	0	1100 1001	0x20000200	39-RPM Calc Task	AND	0000 0010
				40-Timing Angle Task	OR	0010 0100
				47-Air Manifold Pres	OR	0001 0100

9.01 Name <ASCII string>

`OSEventTbl[i].OSEventName` is a pointer to a `NUL` terminated ASCII string. Note that this field is not shown in the illustration above because it was added after the illustration was done. This variable ONLY exists if the variable `OSFlagNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

9.02 Ref.

This corresponds to the index into `OSFlagTbl[]` for the event flag.

9.03 Flags <INT8U, INT16U or INT32U>

This entry contains the current state of each of the event flags in the event flag group. This filed is either 8, 16 or 32 bit wide depending on the value of a configuration constant. You can examine the variable '`OSFlagWidth`' which will indicate the number of bytes (i.e. 1, 2 or 4).

The value to display in this column is obtained by: `OSFlagTbl[i].OSFlagFlags`. This field should be displayed in binary format. Because of the difficulty in reading long bit strings, it's preferable to separate nibbles:

8 Bits:	xxxx xxxx
16 Bits:	xxxx xxxx : xxxx xxxx
32 Bits:	xxxx xxxx : xxxx xxxx : xxxx xxxx : xxxx xxxx

Note how colons are used to separate 8 bit portions for readability.

9.04 OS_FLAG_GRP @ <Hex address>

This corresponds to the address of the `OSFlagTbl[]` entry, i.e. `&OSFlagTbl[i]`.

9.05 Tasks Waiting <ASCII string>

In order to display the list of tasks waiting for the event flag group, you will need to go through the linked list of `OS_FLAG_NODE`. You proceed as follows:

1) You read the pointer to the beginning of the list: `OSFlagTbl[i].OSFlagWaitList` (we'll call this `p_node`). If this is NOT a `NULL` pointer then you follow the linked list of `OS_FLAG_NODE` until you encounter a `NULL` pointer.

2) For each entry of `OS_FLAG_NODE` you can obtain the name of the waiting task by getting the address of the TCB from using `p_node->OSFlagNodeTCB` (we'll call this `p_tcb`).

3) The name of the task waiting is then obtained by accessing `p_tcb->OSTCBName`.

This variable ONLY exists if the variable `OSTaskNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

9.06 Wait Type <value -> ASCII string>

This field indicates whether the task is waiting for any of the bits to be set or cleared or, for all the bits to be set or cleared.

In order to display the list of tasks waiting for the event flag group, you will need to go through the linked list of `OS_FLAG_NODE`. You proceed as follows:

1) You read the pointer to the beginning of the list: `OSFlagTbl[i].OSFlagWaitList` (we'll call this `p_node`). If this is NOT a `NULL` pointer then you follow the linked list of `OS_FLAG_NODE` until you encounter a `NULL` pointer.

2) For each entry of `OS_FLAG_NODE` you can obtain the wait type of the waiting task by accessing the `p_node->OSFlagNodeWaitType` and display this as follows:

Value of .OSFlagNodeWaitType	Display value
0	"Wait for ALL 0"
1	"Wait for ANY 0"
2	"Wait for ALL 1"
3	"Wait for ANY 1"

9.07 Waiting for Flags <binary>

This field indicates which event flags the task is waiting for.

In order to display the list of tasks waiting for the event flag group, you will need to go through the linked list of `OS_FLAG_NODE`. You proceed as follows:

1) You read the pointer to the beginning of the list: `OSFlagTbl[i].OSFlagWaitList` (we'll call this `p_node`). If this is NOT a `NULL` pointer then you follow the linked list of `OS_FLAG_NODE` until you encounter a `NULL` pointer.

2) For each entry of `OS_FLAG_NODE` you can obtain the flags that the task is waiting for by accessing the `p_node->OSFlagNodeFlags` and display this as follows, preferably in binary format. Because of the difficulty in reading long bit strings, it's preferable to separate nibbles with spaces and 8 bit quantities with ':':

8 Bits:	xxxx xxxx
16 Bits:	xxxx xxxx : xxxx xxxx
32 Bits:	xxxx xxxx : xxxx xxxx : xxxx xxxx : xxxx xxxx

10.0 Timer List

The kernel awareness timer list ONLY exists if `OSTmrEn` is set to 1 (see `OS_DBG.C`).

µC/OS-II implements software based timers allowing periodic or one-shot events to occur. Timers were added in V2.83. The application can have any number of timers (limited by RAM). The resolution of timers is typically set to 1/10 sec. Timers are managed by a 'timer task' (`OSTmr_Task()` which is found in `OS_TMR.C`).

An optional callback function can be executed when a timer expires.

In order to distribute the work done by the timer task, the management of timers is implemented using a 'timer wheel'. Each 'spoke' of the timer wheel contains timers to be updated. The actual spoke where a timer is inserted depends on the expected expiration time of the timer. The size of the timer wheel is set at configuration time (`OS_CFG.H`) and typically, the size is set to a prime number.

To display the timer list, you will need to scan a linked list of items at each spoke.

Example display:

Spoke#	#Timers	TimerName	Match	Option	Delay	Period	Callback	Callback Arg
0	2	Disc Valve Open	1000	One-Shot	100	0	DiscClose()	0
		Suction Valve Close	940	One-Shot	60	0	SuctionClose()	0
1	0							
2	0							
3	0							

10.01 Spoke# <INT8U>

This column contains the spoke number which can vary between `0` and `OS_TMR_CFG_WHEEL_SIZE-1`. You should note that the value of `OS_TMR_CFG_WHEEL_SIZE` is stored in the ROM variable `OSTmrCfgWheelSize`.

This column also corresponds to the index 'i' which is referenced in the next sub-sections.

10.02 #Timers <INT16U>

This column indicates the number of timers currently placed in the corresponding spoke. This column is given by the following value:

```
#Timers = OSTmrWheelTbl[i].OSTmrEntries;
```

10.03 Timer Name <ASCII string>

This column provides the name that was assigned to each timer when the timer was created (see `OSTmrCreate()`). Because there might be more than one timer entry per timer wheel spoke, you need to scan the list of timers in each spoke and list the value of each timer on its own line as follows:

```
OS_TMR  *p_tmr;

p_tmr = OSTmrWheelTbl[i].OSTmrFirst;
while (p_tmr != (OS_TMR *)0) {
    Display the name of the timer: p_tmr->OSTmrName;
    :
    :
    p_tmr = p_tmr->OSTmrNext;
}
```

Note that the timer name is a pointer to a `NUL` terminated ASCII string.

This variable ONLY exists if the variable `OSTmrNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

10.04 Match <INT32U>

The timer task increments the value of `OSTmrTime` every time it's executed. Each timer expires when `OSTmrTime` matches the value of the `.OSTmrMatch`.

This column provides the name that was assigned to each timer when the timer was created (see `OSTmrCreate()`). Because there might be more than one timer entry per timer wheel spoke, you need to scan the list of timers in each spoke and list the value of each timer on its own line as follows:

```
OS_TMR  *p_tmr;

p_tmr = OSTmrWheelTbl[i].OSTmrFirst;
while (p_tmr != (OS_TMR *)0) {
    :
    :
    Display the value of Match: p_tmr->OSTmrMatch;
    :
    :
    p_tmr = p_tmr->OSTmrNext;
}
```

The value should be displayed in decimal format.

10.05 Option <INT8U>

A timer can be configured for periodic or one-shot. In periodic mode, the timer restarts automatically after timing out. In one-shot the timer stops when the timer expires.

This column shows the mode for the timer and can be obtained as follows:

```
OS_TMR  *p_tmr;

p_tmr = OSTmrWheelTbl[i].OSTmrFirst;
while (p_tmr != (OS_TMR *)0) {
    :
    :
    if (p_tmr->OSTmrOpt == 1) {
        Display "ONE-SHOT";
    } else {
        Display "PERIODIC"
    }
    :
    :
    p_tmr = p_tmr->OSTmrNext;
}
```

10.06 Delay <INT32U>

Periodic mode can be started after waiting for a certain delay and is the value displayed in this column.

```
OS_TMR  *p_tmr;

p_tmr = OSTmrWheelTbl[i].OSTmrFirst;
while (p_tmr != (OS_TMR *)0) {
    :
    :
    Display the value of p_tmr->OSTmrDly;
    :
    :
    p_tmr = p_tmr->OSTmrNext;
}
```

10.07 Period <INT32U>

The amount of time between reloads of the timer value is the period and is represented by this value.

```
OS_TMR  *p_tmr;

p_tmr = OSTmrWheelTbl[i].OSTmrFirst;
while (p_tmr != (OS_TMR *)0) {
    :
    :
    Display the value of p_tmr->OSTmrPeriod;
    :
    :
    p_tmr = p_tmr->OSTmrNext;
}
```

10.08 Callback <void (*OS_TMR_CALLBACK)(void *ptmr, void *parg)>

This column is used to display either the address of the function that the timer task will execute when the timer expires or, better yet, the 'name' of that function.

```
OS_TMR  *p_tmr;

p_tmr = OSTmrWheelTbl[i].OSTmrFirst;
while (p_tmr != (OS_TMR *)0) {
    :
    :
    Display the value of p_tmr->OSTmrCallback;
    :
    :
    p_tmr = p_tmr->OSTmrNext;
}
```

10.09 CallbackArg <Hex address>

This column is used to display the argument that is passed to the callback function. It's probably best to display that value in hexadecimal.

```
OS_TMR  *p_tmr;

p_tmr = OSTmrWheelTbl[i].OSTmrFirst;
while (p_tmr != (OS_TMR *)0) {
    :
    :
    Display the value of p_tmr->OSTmrCallbackArg;
    :
    :
    p_tmr = p_tmr->OSTmrNext;
}
```

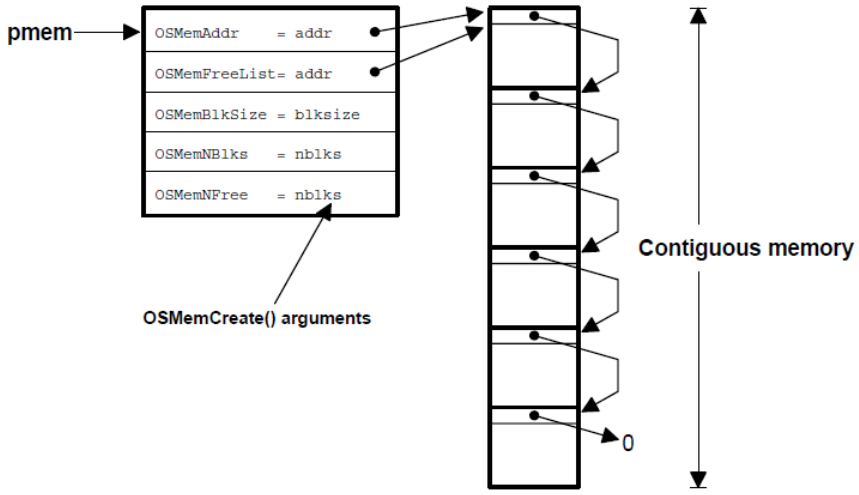
11.0 Memory Partitions

The kernel awareness semaphore list ONLY exists if `OSMemEn` is set to 1 (see `OS_DBG.C`).

Memory partitions are created by calling `OSMemCreate()` and when a memory partition is created, an `OS_MEM` structure is assigned from a pool of `OS_MEM`s.

To display the memory partition list, you will display `OSMemTbl[]` from `0` to `OSMemTblSize / OSMemSize - 1`.

Each entry of the `OSMemTbl[]` look as shown below.



Example display:

Name	Ref	Avail %	Used %	#Blks Avail	#Blks Used	#Blks Max	Blk Size (Bytes)	OS_MEM @	Starts @
Rx Buf	0	90%	10%	90	10	100	32	0x20000300	0x2001000
Tx Buf	1	12%	88%	24	176	200	16	0x20000800	0x2001100

11.01 Name <ASCII string>

`OSMemTbl[i].OSMemName` is a pointer to a `NUL` terminated ASCII string. Note that this field is not shown in the illustration above because it was added after the illustration was done. This variable ONLY exists if the variable `OSMemNameEn` is set to 1 which is a ROM variable declared in `OS_DBG.C`.

11.02 Ref.

This corresponds to the index into `OSMemTbl[]` for the memory partition.

11.03 Avail% <INT8U>

This corresponds to the following value:

```
Avail% = 100 * OSMemTbl[i].OSMemNFree
       / OSMemTbl[i].OSMemNBlks;
```

11.04 Used% <INT8U>

This corresponds to the following value:

```
Used%  = 100 * (OSMemTbl[i].OSMemNBlks - OSMemTbl[i].OSMemNFree)
       / OSMemTbl[i].OSMemNBlks;
```

11.05 #Blks Avail <INT16U>

This corresponds to the following value:

```
#Blks Avail = OSMemTbl[i].OSMemNFree;
```

11.06 #Blks Used <INT16U>

This corresponds to the following value:

```
#Blks Used = OSMemTbl[i].OSMemNBlks - OSMemTbl[i].OSMemNFree;
```

11.07 #Blks Max <INT16U>

This corresponds to the following value:

```
#Blks Max = OSMemTbl[i].OSMemNBlks;
```

11.08 Blk Size <INT16U>

This corresponds to the following value:

```
#Blk Size = OSMemTbl[i].OSMemBlkSize;
```

11.09 OS_MEM @ <Hex address>

This corresponds to the following value:

```
OS_MEM@ = &OSMemTbl[i];
```

11.10 Starts @ <Hex address>

This corresponds to the following value:

```
Starts @ = OSMemTbl[i].OSMemAddr;
```