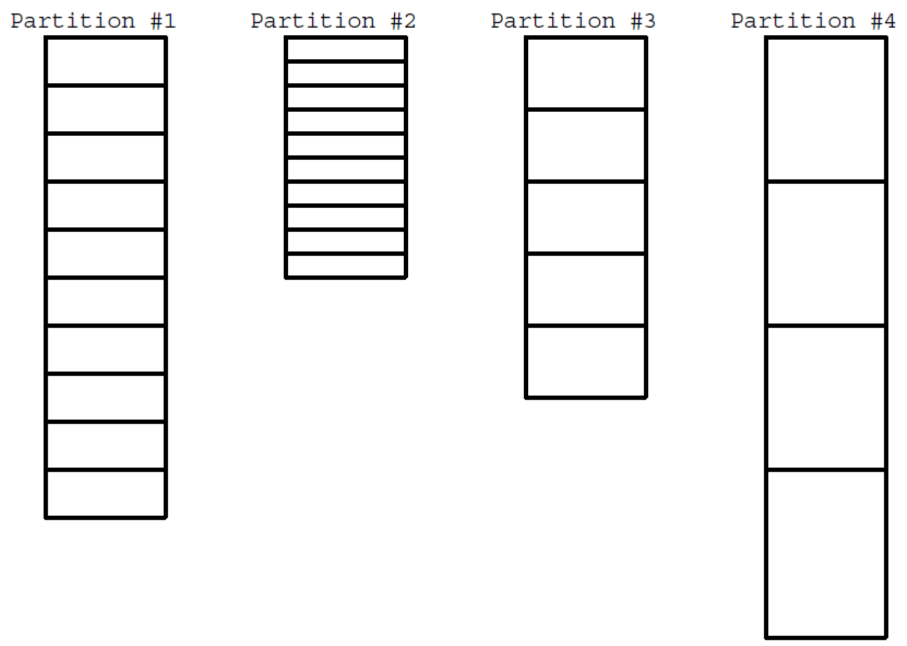
~~~Memory Management in MicroC/OS-II~~~

Using dynamic memory allocators such as malloc() and free() in embedded real-time systems can be dangerous. It might not be possible to obtain a single contiguous memory area due to fragmentation. Fragmentation means the development of a large number of separate free areas. In MicroC/OS-II, the memory allocators used obtain fixed sized memory blocks from a partition made of a contiguous memory area.

The partition has a finite integral number of same-sized memory blocks. The allocation/deallocation operations performed on these blocks are done in constant time and deterministic. Meaning, when an application using MicroC/OS-II requests a memory block from a partition, the allocation process takes a constant amount of time, regardless of the size of the partition or the number of allocated blocks. Similarly, when a memory block is deallocated back to the partition, the deallocation process also takes a constant amount of time. By providing constant-time allocation and deallocation, MicroC/OS-II allows developers to accurately plan and control the execution flow of their application. When freeing a particular block, it needs to return back from the partition it came from. This would not result in fragmentation.

Figure 12.2 Multiple memory partitions.



The following functions are listed for the uC/OS-II memory management:

```
OSMemCreate()- Used to Create a memory partition which is a contiguous block of memory divided
into fixed sized blocks.

OSMemGet()- Used to obtain memory blocks from the memory partition created using OSMemCreate().

OSMemNameGet()- Used to retrieve the name assigned to a memory partition.

OSMemNameSet()- Used to set or change the name of a memory partition.
```

```
OSMemPut()- Used to release a memory block back to the memory partition.

OSMemQuery()- Used to retrieve information about the status if a memory partition.

OS_MemClr() – Used to clear a contiguous block of RAM.
```

Similar to TCB's mentioned in the Process management article, we have Memory Partition Data structures for Memory Management.

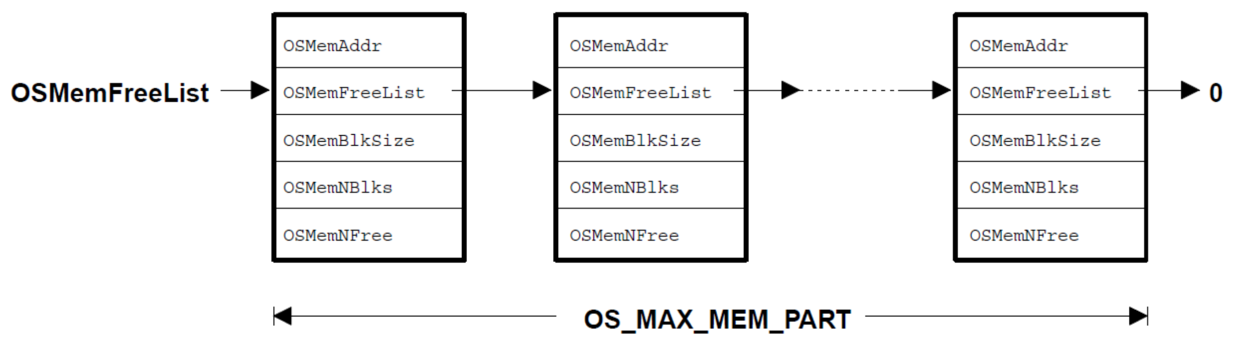
```
typedef struct os_mem {                     /* MEMORY CONTROL BLOCK                              */
    void    *OSMemAddr;                     /* Pointer to beginning of memory partition          */
    void    *OSMemFreeList;                 /* Pointer to list of free memory blocks             */
    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     */
#if OS_MEM_NAME_EN > 0u
    INT8U   *OSMemName;                     /* Memory partition name                             */
#endif
} OS_MEM;
```

```
typedef struct os_mem_data {
    void    *OSAddr;                        /* Ptr to the beginning address of the memory partition    */
    void    *OSFreeList;                    /* Ptr to the beginning of the free list of memory blocks  */
    INT32U   OSBlkSize;                     /* Size (in bytes) of each memory block              */
    INT32U   OSNBlks;                       /* Total number of blocks in the partition           */
    INT32U   OSNFree;                       /* Number of memory blocks free                      */
    INT32U   OSNUsed;                       /* Number of memory blocks used                      */
} OS_MEM_DATA;
#endif

OS_EXT  OS_MEM            OSMemTbl[OS_MAX_MEM_PART];/* Storage for memory partition manager            */
// OS_MAX_MEM_PART defined to 5 indicating that a maximum number of 5 memory partitions can be created.
```

As discussed in the first article that describes about the Initialization process of MicroC/OS-II. The OS_MemInit() is called to initialize the memory partition manager.

Figure 12.3 List of free memory control blocks.



OSMemCreate():

```
OS_MEM  *OSMemCreate (void   *addr,
                      INT32U  nblks,
                      INT32U  blksize,
                      INT8U  *perr)
{
    OS_MEM    *pmem;
    INT8U     *pblk;
    void     **plink;
    INT32U     loops;
    INT32U     i;
#if OS_CRITICAL_METHOD == 3u                          /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr = 0u;
#endif



#ifdef OS_SAFETY_CRITICAL
    if (perr == (INT8U *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return ((OS_MEM *)0);
    }
#endif

#ifdef OS_SAFETY_CRITICAL_IEC61508
    if (OSSafetyCriticalStartFlag == OS_TRUE) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        *perr = OS_ERR_ILLEGAL_CREATE_RUN_TIME;
        return ((OS_MEM *)0);
    }
#endif

#if OS_ARG_CHK_EN > 0u
    if (addr == (void *)0) {                          /* Must pass a valid address for the memory part.*/
        *perr = OS_ERR_MEM_INVALID_ADDR;
        return ((OS_MEM *)0);
    }
    if (((INT32U)addr & (sizeof(void *) - 1u)) != 0u){  /* Must be pointer size aligned                */
        *perr = OS_ERR_MEM_INVALID_ADDR;
        return ((OS_MEM *)0);
    }
    if (nblks < 2u) {                                 /* Must have at least 2 blocks per partition     */
        *perr = OS_ERR_MEM_INVALID_BLKS;
        return ((OS_MEM *)0);
    }
    if (blksize < sizeof(void *)) {                   /* Must contain space for at least a pointer     */
        *perr = OS_ERR_MEM_INVALID_SIZE;
        return ((OS_MEM *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    pmem = OSMemFreeList;                             /* Get next free memory partition              */
    if (OSMemFreeList != (OS_MEM *)0) {               /* See if pool of free partitions was empty    */
        OSMemFreeList = (OS_MEM *)OSMemFreeList->OSMemFreeList;
    }
    OS_EXIT_CRITICAL();
    if (pmem == (OS_MEM *)0) {                        /* See if we have a memory partition           */
        *perr = OS_ERR_MEM_INVALID_PART;
        return ((OS_MEM *)0);
    }
    plink = (void **)addr;                            /* Create linked list of free memory blocks    */
    pblk  = (INT8U *)addr;
    loops  = nblks - 1u;
    for (i = 0u; i < loops; i++) {
        pblk +=  blksize;                             /* Point to the FOLLOWING block                */
       *plink = (void  *)pblk;                        /* Save pointer to NEXT block in CURRENT block */
        plink = (void **)pblk;                        /* Position to  NEXT      block                */
    }
    *plink              = (void *)0;                  /* Last memory block points to NULL            */
    pmem->OSMemAddr     = addr;                       /* Store start address of memory partition     */
    pmem->OSMemFreeList = addr;                       /* Initialize pointer to pool of free blocks   */
    pmem->OSMemNFree    = nblks;                      /* Store number of free blocks in MCB          */
    pmem->OSMemNBlks    = nblks;
    pmem->OSMemBlkSize  = blksize;                    /* Store block size of each memory blocks      */

    OS_TRACE_MEM_CREATE(pmem);

    *perr               = OS_ERR_NONE;
    return (pmem);
}
```

Breakdown of OSMemCreate():

```
OS_MEM  *OSMemCreate (void   *addr, INT32U  nblks, INT32U  blksize, INT8U  *perr)
```

The OSMemCreate() function creates a fixed sized memory partition that will be managed by uC/OS-II. It takes in 4 parameters; **addr** is the starting address of the memory partition, **nblks** is the number of memory blocks to be created from the memory partition, **blksize** is the size of each block in the memory partition, **perr** is the pointer to variable containing an error message which will be set by this function.

Following are the error messages as mentioned in uC/OS-II:

OS_ERR_NONE: if the memory partition has been created correctly.
OS_ERR_ILLEGAL_CREATE_RUN_TIME: if you tried to create a memory partition after safety critical operation started.
OS_ERR_MEM_INVALID_ADDR: if you are specifying an invalid address for the memory storage of the partition or, the block does not align on a pointer boundary.
OS_ERR_MEM_INVALID_PART: no free partitions available
OS_ERR_MEM_INVALID_BLKS: user specified an invalid number of blocks (must be >= 2)

OS_ERR_MEM_INVALID_SIZE:	user specified an invalid block size - must be greater than the size of a pointer - must be able to hold an integral number of pointers

```
  OS_MEM    *pmem;
    INT8U     *pblk;
    void     **plink;
    INT32U     loops;
    INT32U     i;
#if OS_CRITICAL_METHOD == 3u                          /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr = 0u;
#endif
```

pmem is a pointer to memory structure and used to hold the address of free memory partition obtained from the OSMemFreeList.

pblk is a pointer to memory block within a memory partition. It is used to keep track of the current memory block during the creation of a linked list of free memory blocks.

plink is a pointer to pointer used to establish links between memory blocks in the linked list. It points to the address where the next memory block's pointer will be stored.

loops: It represents the number of iterations needed to create the linked list of free memory blocks.

cpu_sr: It is used to store the CPU status register. It allocates storage for the CPU status register based on the value of OS_CRITICAL_METHOD. The value 3 indicated that the code is allocating storage for the CPU status register.

```
#if OS_ARG_CHK_EN > 0u
    if (addr == (void *)0) {                          /* Must pass a valid address for the memory part.*/
        *perr = OS_ERR_MEM_INVALID_ADDR;
        return ((OS_MEM *)0);
    }
    if (((INT32U)addr & (sizeof(void *) - 1u)) != 0u){  /* Must be pointer size aligned                */
        *perr = OS_ERR_MEM_INVALID_ADDR;
        return ((OS_MEM *)0);
    }
    if (nblks < 2u) {                                 /* Must have at least 2 blocks per partition     */
        *perr = OS_ERR_MEM_INVALID_BLKS;
        return ((OS_MEM *)0);
    }
    if (blksize < sizeof(void *)) {                   /* Must contain space for at least a pointer     */
        *perr = OS_ERR_MEM_INVALID_SIZE;
        return ((OS_MEM *)0);
    }
```

The snippet performs the argument checks for **addr**, **nblks** and **blsize** parameters.
When the argument check is enabled by defining the value of OS_ARG_CHK_EN to greater than 0 the following operations are performed.

Address check ‘**addr**’ is performed by checking is **addr**, the starting address of the memory partition is a null pointer. If the **addr** is **NUL**L, it means that a valid starting address of the memory partition was not provided. In this case, the **perrr** is set to **OS_ERR_MEM_INVALID_ADDR** and returns a NULL pointer indicating a failure.
A check is performed in the next stage on ‘**addr**’ to see it is properly aligned according to the pointer size. If the result is not 0, then it is not aligned and the **perr** is set to **OS_ERR_MEM_INVALID_ADDR** and returns a null pointer.

‘**nblks**’ that indicates the number of memory blocks in a partition is check if it is less than 2. It ensures that atleast 2 memory blocks are present per partition. If it is false, **perr** is set to **OS_ERR_MEM_INVALID_BLKS** and a null pointer is returned.
To ensure that each block has enough space to hold atleast a pointer, the ‘**blksize**’ is checked with the size of pointer. If the **blksize** is smaller, **perr** is set to **OS_ERR_MEM_INVALID_SIZE** and returns a null pointer.

```
    OS_ENTER_CRITICAL();
    pmem = OSMemFreeList;                              /* Get next free memory partition                */
    if (OSMemFreeList != (OS_MEM *)0) {                /* See if pool of free partitions was empty      */
        OSMemFreeList = (OS_MEM *)OSMemFreeList->OSMemFreeList;
    }
    OS_EXIT_CRITICAL();
    if (pmem == (OS_MEM *)0) {                         /* See if we have a memory partition             */
        *perr = OS_ERR_MEM_INVALID_PART;
        return ((OS_MEM *)0);
    }
```

The section of code enters into the critical section region so that concurrent access to the **OSMemFreeList** data structure is prevented. This is used to manage the free memory partition.

The next free memory partition is stored in ‘**pmem’** once the code enters into the critical region. A check is performed on OSMemFreeList to make sure there we have free memory partitions available. Once it is true, update the **OSMemFreeList** so that it points to next free list. Once the free list is updated, the code exits the critical section.

A memory check is performed on **pmem** that should result in false when there are available memory partitions. If not, return null pointer.

```
 plink = (void **)addr;                             /* Create linked list of free memory blocks      */
    pblk  = (INT8U *)addr;
    loops  = nblks - 1u;
    for (i = 0u; i < loops; i++) {
        pblk +=  blksize;                              /* Point to the FOLLOWING block                  */
       *plink = (void  *)pblk;                         /* Save pointer to NEXT block in CURRENT block   */
        plink = (void **)pblk;                         /* Position to  NEXT      block                  */
    }
    *plink              = (void *)0;                   /* Last memory block points to NULL              */
    pmem->OSMemAddr     = addr;                        /* Store start address of memory partition       */
    pmem->OSMemFreeList = addr;                        /* Initialize pointer to pool of free blocks     */
    pmem->OSMemNFree    = nblks;                       /* Store number of free blocks in MCB            */
    pmem->OSMemNBlks    = nblks;
    pmem->OSMemBlkSize  = blksize;                        /* Store block size of each memory blocks        */

    OS_TRACE_MEM_CREATE(pmem);

    *perr               = OS_ERR_NONE;
    return (pmem);
```

Once the necessary checks are performed and the free partitions are obtained from the free list the memory is created.
A linked list of free memory blocks (‘**plink’**) is created with the starting address of ‘**addr**’ that represents the starting address of the memory partition. The ‘**pblk**’ is assigned with the ‘**addr**’ so that byte level pointer arithmetic can be performed/to access the blocks directly. Then, a loop is executed **nblks-1** times to link the free memory blocks.

In the loop: the **pblk** is incremented with the block size ‘**blksize**’ so that it points to the next/following block. A link is created from the current block to next block by assigning ‘***plink**’ with **pblk**. This saves the pointer to next block in current block. The ‘**plink**’ is updated with the ‘**pblk**’ so that in further iterations, the ‘**plink**’ points to next block/position to next block. Once the loop is run and the linked list is assigned, initialize the last memory block with NULL indicating the end of the list.

The ‘**pmem**’ structure is updated with the current values after the assignment. The start address of the memory partition is store in **OSMemAddr**, **OSMemFreeList** is assigned with ‘**addr**’ indicating initialization of pointer to pool of free blocks, **OSMemNFree** is assigned with **nblks** indicating the storage of total number of free blocks in the Memory Control Block, the same is stored in **OSMemNBlks** indicating total number of blocks in memory partition and the size of each block ‘**blksize**’ is stored in **OSMemBlkSize** member.

A trace or logging function **OS_TRACE_MEM_CREATE()** is called to record the creation of the memory partition. The ‘pmem’ structure is passed as an argument to the trace function. It is defined in **os_trace.h** and the value of ‘perr’ is updated to OS_ERR_NONE indicating no error has occurred during the memory creation.

OSMemGet():

```
void  *OSMemGet (OS_MEM   *pmem,
                 INT8U    *perr)
{
    void      *pblk;
#if OS_CRITICAL_METHOD == 3u                              /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr = 0u;
#endif


#ifdef OS_SAFETY_CRITICAL
    if (perr == (INT8U *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return ((void *)0);
    }
#endif
#if OS_ARG_CHK_EN > 0u
    if (pmem == (OS_MEM *)0) {                              /* Must point to a valid memory partition        */
        *perr = OS_ERR_MEM_INVALID_PMEM;
        return ((void *)0);
    }
#endif

    OS_TRACE_MEM_GET_ENTER(pmem);

    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree > 0u) {                            /* See if there are any free memory blocks       */
        pblk                = pmem->OSMemFreeList;          /* Yes, point to next free memory block          */
        pmem->OSMemFreeList = *(void **)pblk;               /*      Adjust pointer to new free list          */
        pmem->OSMemNFree--;                                 /*      One less memory block in this partition  */
        OS_EXIT_CRITICAL();
        *perr = OS_ERR_NONE;                                /*      No error                                 */
        OS_TRACE_MEM_GET_EXIT(*perr);
        return (pblk);                                      /*      Return memory block to caller            */
    }
    OS_EXIT_CRITICAL();
    *perr = OS_ERR_MEM_NO_FREE_BLKS;                        /* No,  Notify caller of empty memory partition  */
    OS_TRACE_MEM_GET_EXIT(*perr);
    return ((void *)0);                                     /*      Return NULL pointer to caller            */
}
```

Now that the memory is created using the OSMemCreate() function, our application can use that memory by using OSMemGet() function. The pointer that is returned from the OSMemCreate() is used by OSMemGet() to get the memory block from the partition. It is important to not that when the application uses the required block, it needs to return it back to the partition from which the memory block was taken.

The OSMemGet() takes in 2 arguments, ***pmem**, a pointer to the memory partition control block and ***perr** a pointer to a variable containing.

After allocating the CPU status register storage by using cpu_sr and performing a check if the **pmem** is pointing to a valid memory partition, the code enters into the critical section region where the further operations performed needs to be protected from concurrent access to the memory.

Initially, it is checked if any free memory blocks are available by checking if **OSMemNFree** is greater than 0. If no blocks are available, set the **perr** value to **OS_ERR_MEM_NO_FREE_BLKS** and return null pointer to the caller. When true indicates that atleast 1 memory block is available. Upon true, the **pblk** is made to point to next free memory block by assigning it with **pmem→OSMemFreeList**. The free list of **pmem** is then assigned with new free list **pblk**. Once the block is assigned to the application, the total blocks in the current partition is decremented by 1. Once the blocks are assigned, the code exits the critical section. Upon successful allocation, the value of **perr** is set to **OS_ERR_NONE** indicating no error has occurred while getting a block from the created blocks and the memory block is returned back to the caller.

OSMemPut():

As discussed above, when the application has used the memory and is done performing operations upon it, it needs to return the memory block to that specific partition from which it was taken. That operation an be done by using the OSMemPut() function.

```
INT8U  OSMemPut (OS_MEM  *pmem,
                 void    *pblk)
{
#if OS_CRITICAL_METHOD == 3u                     /* Allocate storage for CPU status register           */
    OS_CPU_SR  cpu_sr = 0u;
#endif


#if OS_ARG_CHK_EN > 0u
    if (pmem == (OS_MEM *)0) {                   /* Must point to a valid memory partition             */
        return (OS_ERR_MEM_INVALID_PMEM);
    }
    if (pblk == (void *)0) {                     /* Must release a valid block                         */
        return (OS_ERR_MEM_INVALID_PBLK);
    }
#endif

    OS_TRACE_MEM_PUT_ENTER(pmem, pblk);

    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree >= pmem->OSMemNBlks) {  /* Make sure all blocks not already returned          */
        OS_EXIT_CRITICAL();
        OS_TRACE_MEM_PUT_EXIT(OS_ERR_MEM_FULL);
        return (OS_ERR_MEM_FULL);
    }
    *(void **)pblk      = pmem->OSMemFreeList;   /* Insert released block into free block list         */
    pmem->OSMemFreeList = pblk;
    pmem->OSMemNFree++;                          /* One more memory block in this partition            */
    OS_EXIT_CRITICAL();
    OS_TRACE_MEM_PUT_EXIT(OS_ERR_NONE);
    return (OS_ERR_NONE);                        /* Notify caller that memory block was released       */
}
```

This function takes 2 parameters; **pmem** which is a pointer to the memory partition control block and **pblk** which is a pointer to the memory being released.

The initial check is performed on **pmem** and **pblk** such that the pointers point to valid memory partition and the block being released must be a valid block. Once the checks are done, the code disables all the interrupts by calling **OS_ENTER_CRITICAL**() and performs the operation to place that block in respective partition.

First, a condition operation is performed on the **OSMemNFree** and **OSMemNBlks** of **pmem** to make sure that all blocks in the partition have already been returned. If returned, then the function exits by throwing the error **OS_ERR_MEM_FULL**. If not , the released block is inserted into the free block list by updating the **pblk** ‘s content with the current value of **OSMemFreeList** of **pmem**. Then update the **OSMemFreeList** of **pmem** with

pblk indicating that the block is returned at the right partition. Once it is added to the partition, the number of blocks in the partition is increased by 1 by performing increment operation on **OSMemNFree** of **pmem**. Upon incrementing, the block is successfully placed back where it belonged in the partition and the function returns **OS_ERR_NONE** indicating that the memory block was released.

OSMemNameSet():

OSMemeNameSet() is used to assign a name to a memory partition.

```c
void  OSMemNameSet (OS_MEM  *pmem,
                    INT8U   *pname,
                    INT8U   *perr)
{
#if OS_CRITICAL_METHOD == 3u                     /* Allocate storage for CPU status register           */
    OS_CPU_SR  cpu_sr = 0u;
#endif



#ifdef OS_SAFETY_CRITICAL
    if (perr == (INT8U *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

#if OS_ARG_CHK_EN > 0u
    if (pmem == (OS_MEM *)0) {                   /* Is 'pmem' a NULL pointer?                          */
        *perr = OS_ERR_MEM_INVALID_PMEM;
        return;
    }
    if (pname == (INT8U *)0) {                   /* Is 'pname' a NULL pointer?                         */
        *perr = OS_ERR_PNAME_NULL;
        return;
    }
#endif
    if (OSIntNesting > 0u) {                     /* See if trying to call from an ISR                  */
        *perr = OS_ERR_NAME_SET_ISR;
        return;
    }
    OS_ENTER_CRITICAL();
    pmem->OSMemName = pname;
    OS_EXIT_CRITICAL();
    OS_TRACE_EVENT_NAME_SET(pmem, pname);
    *perr           = OS_ERR_NONE;
}
```

This function takes 3 parameters as its input, **pmem** which is a pointer to the memory partition, **pname** which is a pointer to an ASCII string that contains the name of the memory partition and **perr** which is a pointer to an error code.

After the initial checks the name is simply assigned to the memory partition by assigning **pname** to **OSMemName** of **pmem** structure. Once that’s done, the function returns **OS_ERR_NONE** to the caller indicating that the name has been successfully assigned to the memory partition.

OSMemNameGet():

This function is used to obtain the name assigned to a memory partition.

```
INT8U  OSMemNameGet (OS_MEM    *pmem,
                     INT8U   **pname,
                     INT8U    *perr)
{
    INT8U      len;
#if OS_CRITICAL_METHOD == 3u                      /* Allocate storage for CPU status register           */
    OS_CPU_SR  cpu_sr = 0u;
#endif



#ifdef OS_SAFETY_CRITICAL
    if (perr == (INT8U *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return (0u);
    }
#endif

#if OS_ARG_CHK_EN > 0u
    if (pmem == (OS_MEM *)0) {                    /* Is 'pmem' a NULL pointer?                          */
        *perr = OS_ERR_MEM_INVALID_PMEM;
        return (0u);
    }
    if (pname == (INT8U **)0) {                   /* Is 'pname' a NULL pointer?                         */
        *perr = OS_ERR_PNAME_NULL;
        return (0u);
    }
#endif
    if (OSIntNesting > 0u) {                      /* See if trying to call from an ISR                  */
        *perr = OS_ERR_NAME_GET_ISR;
        return (0u);
    }
    OS_ENTER_CRITICAL();
    *pname = pmem->OSMemName;
    len    = OS_StrLen(*pname);
    OS_EXIT_CRITICAL();
    *perr  = OS_ERR_NONE;
    return (len);
}
```

This function takes similar parameters to that of **OSMemSetName**(). **pmem** which is a pointer to the memory partition, **pname** which is a pointer to an ASCII string that will receive the name of the memory partition and **perr** which is a pointer to an error code.

The initial checks are performed to check if **pmem** is a valid memory location, if **pname** is a valid name and if this function is being called from an ISR. Once the checks are completed, the name is retrieved by assigning **pname** with **OSMemName** of **pmem** and also calculate the length of the string which would be returned to the caller function on successful retrieval of the memory partition name.

Note: The OSMemSetName() and OSMemGetName() are only available when OS_MEM_NAME_EN is enabled.

OSMemQuery():

This function is used to determine the number of free memory blocks and the number of used memory blocks from a memory partition. It is available when OS_MEM_QUERY_EN is enabled.

```
INT8U  OSMemQuery (OS_MEM       *pmem,
                   OS_MEM_DATA  *p_mem_data)
{
#if OS_CRITICAL_METHOD == 3u                     /* Allocate storage for CPU status register           */
    OS_CPU_SR  cpu_sr = 0u;
#endif



#if OS_ARG_CHK_EN > 0u
    if (pmem == (OS_MEM *)0) {                   /* Must point to a valid memory partition             */
        return (OS_ERR_MEM_INVALID_PMEM);
    }
    if (p_mem_data == (OS_MEM_DATA *)0) {        /* Must release a valid storage area for the data     */
        return (OS_ERR_MEM_INVALID_PDATA);
    }
#endif
    OS_ENTER_CRITICAL();
    p_mem_data->OSAddr     = pmem->OSMemAddr;
    p_mem_data->OSFreeList = pmem->OSMemFreeList;
    p_mem_data->OSBlkSize  = pmem->OSMemBlkSize;
    p_mem_data->OSNBlks    = pmem->OSMemNBlks;
    p_mem_data->OSNFree    = pmem->OSMemNFree;
    OS_EXIT_CRITICAL();
    p_mem_data->OSNUsed    = p_mem_data->OSNBlks - p_mem_data->OSNFree;
    return (OS_ERR_NONE);
}
```

OS_MemClr():

This function is used to clear a contiguous block of RAM.

```
void  OS_MemClr (INT8U  *pdest,
                 INT16U  size)
{
    while (size > 0u) {
        *pdest++ = (INT8U)0;
        size--;
    }
}
```

Here, pdest is the pointer to memory location in the RAM, the start of the RAM and size is the number of bytes to clear. Its use can be seen in OS_InitEventList(); used to clear the event table, in OS_InitTCBList() to clear the Task Control Blocks and priority table, in OS_FlagInit() to clear the flag group table and in many code section in uC/OS-II.

References: MicroC/OS-II The Real-Time Kernel, Second Edition, Jean J. Labrosse.

Article Written By: Yashwanth Naidu Tikkisetty

Follow **#oswithyash** to get updates related to Embedded Systems, Space and Technology.