BRTOS Reference Manual

Version 1.7x

May 2013

Summary

BRTOS Settings

BRTOSConfig.h file

```
/// Define if simulation or DEBUG
#define DEBUG 0

/// Define if verbose info is available
#define VERBOSE 0

/// Define if error check is available
#define ERROR_CHECK 0

/// Define if whatchdog is active
#define WATCHDOG 1

/// Define if compute cpu load is active
#define COMPUTES_CPU_LOAD 1

// The Nesting define must be set in the file HAL.h
// Example:
/// Define if nesting interrupt is active
//#define NESTING_INT 0

/// Define Number of Priorities
#define NUMBER_OF_PRIORITIES 32

/// Define the maximum number of Tasks to be Installed
#define NUMBER_OF_TASKS (INT8U)6

/// Defines the memory allocation and deallocation function to the dynamic queues
#define BRTOS_ALLOC   malloc
#define BRTOS_DEALLOC free

/// Define if OS Trace is active
#define OSTRACE 0

#if (OSTRACE == 1)
  #include "debug_stack.h"
#endif

/// Define if TimerHook function is active
#define TIMER_HOOK_EN                 1

/// Define if IdleHook function is active
#define IDLE_HOOK_EN                  0

/// Enable or disable semaphore controls
#define BRTOS_SEM_EN                  1

/// Enable or disable mutex controls
#define BRTOS_MUTEX_EN                1

/// Enable or disable mailbox controls
#define BRTOS_MBOX_EN                 1

/// Enable or disable queue controls
#define BRTOS_QUEUE_EN                1

/// Enable or disable dynamic queue controls
#define BRTOS_DYNAMIC_QUEUE_ENABLED    1
```

```
/// Enable or disable queue 16 bits controls
#define BRTOS_QUEUE_16_EN       1

/// Enable or disable queue 32 bits controls
#define BRTOS_QUEUE_32_EN       1

/// Defines the maximum number of semaphores\n
/// Limits the memory allocation for semaphores
#define BRTOS_MAX_SEM           4
/// Defines the maximum number of mutexes\n
/// Limits the memory allocation for mutex
#define BRTOS_MAX_MUTEX         4

/// Defines the maximum number of mailboxes\n
/// Limits the memory allocation mailboxes
#define BRTOS_MAX_MBOX          1

/// Defines the maximum number of queues\n
/// Limits the memory allocation for queues
#define BRTOS_MAX_QUEUE         3


/// TickTimer Defines
#define configCPU_CLOCK_HZ          (INT32U)25000000    ///< CPU clock in Hertz
#define configTICK_RATE_HZ          (INT32U)1000        ///< Tick timer rate in Hertz
#define configTIMER_PRE_SCALER      0         ///< Informs if there is a timer prescaler


/// Stack Size of the Idle Task
#define IDLE_STACK_SIZE             (INT16U)150

/// Stack Defines
/// 8KB of RAM: 32 * 128 bytes = 4KB of Virtual Stack
#define HEAP_SIZE 32*128


// Queue heap defines
// Configurado com 1KB p/ filas
#define QUEUE_HEAP_SIZE 8*128
```

- **DEBUG** - It is used on some platforms that have differences between the simulator and the real-time debugger, adapting the code to the desired state through precompiler directives. DEBUG = 1 indicates mode debugger / write code in the microcontroller.
- **VERBOSE** – enables additional task context information, which can be used in debugging. VERBOSE = 1 enables the extra information in the context of the tasks.
- **ERROR_CHECK** – enables additional checks for errors that may occur with the misuse of system calls, making the system less susceptible to failure. ERROR_CHECK = 1 enables error checking.
- **WATCHDOG** – enables the watchdog system. WATCHDOG = 1 indicates activated watchdog.
- **COMPUTES_CPU_LOAD** – enables CPU load computation. COMPUTES_CPU_LOAD = 1 enables the CPU load computation.

- **NUMBER_OF_PRIORITIES** – it is used to configure the maximum number of available priorities. The only valid values are 16 or 32. We recommend using 16 priorities for 8-bit microcontrollers and 32 priorities for 16 and 32 bits microcontrollers.
- **NUMBER_OF_TASKS** – configure how many tasks can be installed in an application. The maximum value for this setting is 31 for **NUMBER_OF_PRIORITIES** = 32 and 15 for **NUMBER_OF_PRIORITIES** = 16. This setting allows that a smaller amount of RAM memory be allocated for task control block (TCB). Use this feature in order to reduce the total amount of system RAM when a small number of tasks are installed.
- **BRTOS_ALLOC** – defines the memory allocation function that will be used in the BRTOS dynamic queues. By default this definition points to the standard C **malloc** function. OBS.: Note that to use this feature you must set the heap size (amount of memory available for dynamic memory allocation) in the used toolchain (linker configuration file). One can also utilize customized memory allocation functions.
- **BRTOS_DEALLOC** – defines the memory dealloc function that will be used in the BRTOS dynamic queues. By default this definition points to the standard C **free** function. The remarks for BRTOS_ALLOC setting are also valid for this definition.
- **OS_TRACE** - indicates whether the trace of the system is active or not. The system trace is used to monitor the behavior of the system.
- **TIMER_HOOK_EN** - indicates whether the timer anchor function is active. If so, you can run a small snippet of code from the tick timer interrupt. This is an important feature to implement tiny time checks, such as timeouts.
- **IDLE_HOOK_EN** – indicates whether the idle hook function is active. **WARNING**: The code executed in this function is the lowest priority in the system.
- **BRTOS_SEM_EN** – enables the BRTOS semaphores when BRTOS_SEM_EN = 1.
- **BRTOS_MUTEX_EN** – enables the BRTOS mutex (mutual exclusion control) when BRTOS_MUTEX_EN = 1.
- **BRTOS_MBOX_EN** – enables the BRTOS mailbox when BRTOS_MBOX_EN = 1.
- **BRTOS_QUEUE_EN** – enables the BRTOS byte queues when BRTOS_QUEUE_EN = 1.
- **BRTOS_DYNAMIC_QUEUE_ENABLED** – enables the BRTOS dynamic queues when BRTOS_DYNAMIC_QUEUE_ENABLED = 1. The dynamic queues support any data size. It is the only BRTOS queue service that supports to be deleted. In order to work properly, the BRTOS_DEALLOC and BRTOS_ALLOC definitions must point to functions that implement dynamic memory allocation and deallocation.
- **BRTOS_QUEUE_16_EN** – enables the BRTOS 16 bits queues when BRTOS_QUEUE_16_EN = 1.
- **BRTOS_QUEUE_32_EN** – enables the BRTOS 32 bits queues when BRTOS_QUEUE_32_EN = 1.
- **BRTOS_MAX_SEM** – defines the maximum number of semaphore control blocks that will be available in the system. Or, how many semaphores may be used in a given application.

- **BRTOS_MAX_MUTEX** - defines the maximum number of mutex control blocks that will be available in the system.
- **BRTOS_MAX_MBOX** - defines the maximum number of mailbox control blocks that will be available in the system.
- **BRTOS_MAX_QUEUE** - defines the maximum number of queue control blocks that will be available in the system. Note that such definition is valid for all types of queues. For example, if it is necessary two dynamic queues and three 16 bits queues, set this value to 5.
- **configCPU_CLOCK_HZ** - indicates the bus frequency used in the system in hertz.
- **configTICK_RATE_HZ** - defines the tick timer of the system (time stamp), ie, the time resolution of the RTOS time manager. Values between 1 ms (1000 Hz) and 10 ms (100 Hz) are recommended. Never forget that the resolution of the time management is +/- 1 tick.
- **configTIMER_PRE_SCALER** - can be used in the microcontroller port to configure a hardware prescaler for tick timer.
- **IDLE_STACK_SIZE** – determines the amount of memory allocated for the virtual stack of the Idle Task. The idle task will be installed in the BRTOSStart function, and the amount of memory used by this task will be allocated on the heap.
- **HEAP_SIZE** - determines the amount of memory allocated to the virtual stack of tasks (in bytes). Whenever a task is installed, the amount of memory used by the task will be allocated on this heap.
- **QUEUE_HEAP_SIZE** - determines the amount of memory allocated to the OS queues (in bytes). Whenever a queue is created, the amount of memory used by the queue will be allocated in this HEAP.

HAL.h file

```
/// Supported processors
#define COLDFIRE_V1          1u
#define HCS08                2u
#define MSP430               3u
#define ATMEGA               4u
#define PIC18                5u
#define RX600                6u
#define ARM_Cortex_M0        7u
#define ARM_Cortex_M3        8u
#define ARM_Cortex_M4        9u
#define ARM_Cortex_M4F       10u

/// Set a specific core as the default processor
#define PROCESSOR            ARM_Cortex_M4F


/// Defines the CPU type
#define OS_CPU_TYPE          INT32U

/// Defines MCU FPU hardware support
#define FPU_SUPPORT          1

/// Defines if the optimized scheduler will be used
#define OPTIMIZED_SCHEDULER 1
```

```
/// Defines if InstallTask function will support parameters
#define TASK_WITH_PARAMETERS 1

/// Defines if 32 bits register for tick timer will be used
#define TICK_TIMER_32BITS   1

/// Defines if nesting interrupt is active
#define NESTING_INT         1

/// Defines CPU Stack Pointer Size
#define SP_SIZE             32
```

- **PROCESSOR** – defines a specific core as the default processor. It is used to configure precompiler directives for this specific core (for example, pragmas).
- **OS_CPU_TYPE** – it is used to configure the processor base type. For example, in a 32-bit microcontroller this type must be a 32-bit unsigned integer. On the other hand, for an 8-bit microcontroller this type must be an unsigned char.
- **FPU_SUPPORT** – in the microcontrollers where there is a floating point unit, this setting enable the FPU unit and the additional save context on the BRTOS HAL.
- **OPTIMIZED_SCHEDULER** – this define must be set to 1 if the used processor has the necessary instructions to implement the scheduler acceleration. So for, only coldfire and ARM Cortex-M3/4 processors have these instructions (FF1 and CLZ, respectively).
- **TASK_WITH_PARAMETERS** – indicates to the BRTOS if the available HAL supports passing parameters in the task installation function (if set to 1). An example of passing parameters is provided in the *InstallTask* function description.
- **TICK_TIMER_32BITS** – the standard size of the tick counter register in the BRTOS is 16 bits. However, on some processors (such as ARM Cortex-Mx) the timer module register is 32 bits. In such case this setting should be set to 1.
- **NESTING_INT** – it is used to inform the BRTOS if the used processor supports nesting of interrupts (or, if this resource is enabled). NESTING_INT = 1 informs that there is support for nested interrupts.
- **SP_SIZE** – it is used to inform the size in bits of the stack pointer (SP) register. Valid values for the stack pointer are 16 or 32. For example, the stack pointer size for the HCS08 and MSP430 microcontrollers is 16 bits. On the other hand, for coldfire and ARM processors the size is 32 bits.

BRTOS Services

OSEnterCritical e OSExitCritical

Can be called from:	**Enabled by:**
Tasks and interrupts	Always available

The **OSEnterCritical** and **OSExitCritical** functions are used to disable and enable, respectively, the microcontroller interrupts. Note that for the port of some processors it is necessary to declare a special variable (OS_SR_SAVE_VAR), which is responsible for saving the current priority mask, as shown in the example below.

Exemplo:

```
void Example_Task (void)
{
   /* Task setup */
   OS_SR_SAVE_VAR     // Defines the variable used to save the current priority mask

   /* Task loop */
   for (;;)
   {
      // Disable interrupts
      OSEnterCritical();

      // Run your critical code here

      // Enable interrupts
      OSExitCritical();
   }
}
```

UserEnterCritical e UserExitCritical

Can be called from:	**Enabled by:**
Tasks	Always available

The **UserEnterCritical** and **UserExitCritical** functions are used to disable and enable, respectively, the microcontroller interrupts by the user code. The difference of the **OSEnterCritical** and **OSExitCritical** functions for this user functions is dependent on the hardware used. On processors that do not support nested interrupts the implementation of these functions is the same. However, on processors with support for nesting of interrupts, the **OSEnterCritical** function saves the current interrupt mask in order to restore it at the exit of the critical section by the **OSExitCritical** function. The use of the user critical region functions are the same of the system functions, with the caveat that it can only be used within tasks, never inside of interruptions. Normally the user critical region functions occupy less clock cycles that the system functions.

InstallTask

```
INT8U InstallTask(void(*FctPtr)(void),const CHAR8 *TaskName, INT16U
USER_STACKED_BYTES,INT8U iPriority)
```

or

```
INT8U InstallTask(void(*FctPtr)(void),const CHAR8 *TaskName, INT16U
USER_STACKED_BYTES,INT8U iPriority, void *parameters)
```

Can be called from:	**Enabled by:**
System initialization and task	Always available

The **InstallTask** function installs a task, allocating their information in an available task control block. A task can be installed before the start of the scheduler or for a task.

Arguments:

- Pointer of the task to be installed
- Task name
- Size of the task virtual stack
- Task priority
- Task parameters (only available if TASK_WITH_PARAMETERS = 1).

Return:

- **OK** - if the task was successfully installed.
- **STACK_SIZE_TOO_SMALL** – if the stack size set by the user is less than the minimum needed to initialize the task.
- **NO_MEMORY** – if the memory available to allocate the task virtual stack is not sufficient to the needed task stack.
- **END_OF_AVAILABLE_PRIORITIES** – if the specified task priority is greater than the system threshold.
- **BUSY_PRIORITY** – if the specified priority is allocated already.
- **CANNOT_ASSIGN_IDLE_TASK_PRIO** – if the specified priority is 0 (Idle Task priority).

Example without parameters:

```
void main (void)
{
        // BRTOS initialization
        BRTOS_Init();

        // Install a task
       if(InstallTask(&System_Time,"System Time",150,31) != OK)
       {
         // Should not enter here. Treat this exception !!!
         while(1){};
       };
}
```

Example with parameters:

```
// Demo of parameters
struct _parametros
{
  INT8U direction;
  INT8U data;
};

typedef struct _parametros Parametros;

Parametros ParamTest;

void main (void)
{
       // BRTOS initialization
       BRTOS_Init();

      // Specifies a parallel port configuration to be used inside the task
      ParamTest.direction = 3;
      ParamTest.data = 0;

       // Install a task
      if(InstallTask(&System_Time,"System Time",150,31,(void*)&ParamTest) != OK)
      {
        // Should not enter here. Treat this exception !!!
        while(1){};
      };
}


void System_Time(void *parameters)
{
   /* task setup */
   Parametros *param = (Parametros*)parameters;
   PTAD = param->data;
   PTADD = param->direction;

   /* task main loop */
   for (;;)
   {
      PTAD_PTAD1 = ~PTAD_PTAD1;
      DelayTask(100);
   }
}
```

OSGetTickCount

`INT16U OSGetTickCount(void)`

Can be called from:	**Enabled by:**
Tasks and interrupts	Always available

The **OSGetTickCount** function returns the current count of system time ticks. It is used, for example, in applications that need to know the runtime of a particular procedure.

Example:

```
void Example_Task (void)
{
   /* task setup */
   INT16U tick_count;

   /* Task loop */
   for (;;)
   {
       // Acquires the current count of system ticks
       tick_count = OSGetTickCount();
   }
}
```

DelayTask

`INT8U DelayTask(INT16U time_wait)`

Can be called from:	**Enabled by:**
Tasks	Always available

The **DelayTask** function allows a task to give up the processor for a integer number of timer ticks (typical values ranging from 1ms to 100ms). Valid values are from 1 to **TickCountOverFlow** (defined in the BRTOS.h file). A delay of 0 means that the task will not give up the processor. At the end of the time delay the system wakes up the task, which returns to compete for the processor.

Arguments:

- number of system ticks that the running task will give up the processor

Return:

- **OK** - if the function is successfully executed
- **NOT_VALID_TASK** – if the function is called before the system initialization
- **NO_TASK_DELAY** – if the required delay is 0

Example:

```
void Example_Task (void)
{
   /* task setup */

   /* task loop */
   for (;;)
   {
       // Give up the processor by 10 system ticks
       (void)DelayTask(10);
   }
}
```

DelayTaskHMSM

`INT8U DelayTaskHMSM(INT8U hours, INT8U minutes, INT8U seconds, INT16U miliseconds)`

Can be called from:	Enabled by:
Tasks	Always available

The **DelayTaskHMSM** function allows a task to give up the processor for for a certain time in hours, minutes, seconds and milliseconds. A delay of 0 means that the task will not give up the processor. At the end of the time delay the system wakes up the task, which returns to compete for the processor.

Arguments:

- number of hours that the running task will give up the processor. Valid values ranging from 0 to 255
- number of minutes that the running task will give up the processor. Valid values ranging from 0 to 59
- number of seconds that the running task will give up the processor. Valid values ranging from 0 to 59
- number of miliseconds that the running task will give up the processor. Valid values ranging from 0 to 999. Note that the resolution of this argument depends on the system tick. For example, if the system tick is 10ms, an argument of 3ms will be considered 0.

Return:

- **OK** - if the function is successfully executed
- **NO_TASK_DELAY** – if the required delay is 0
- **INVALID_TIME** – if one of the passed arguments is out of valid values

Example:

```
void Example_Task (void)
{
   /* task setup */


   /* task loop */
   for (;;)
   {
       // Give up the processor by 15 minutes and 30 seconds
       (void)DelayTaskHMSM(0,15,30,0);
   }
}
```

BlockPriority

`INT8U BlockPriority(INT8U iPriority)`

Can be called from:	**Enabled by:**
Tasks	Always available

The **BlockPriority** function removes the task associated with this priority from the ready list (this task will not compete for the processor anymore). The system will search for the highest priority task ready to run if the task blocks itself.

Arguments:

- priority of the task to be blocked

Return:

- **OK** - if the task is successfully blocked
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code

Example:

```
void Example_Task (void)
{
   /* task setup */


   /* task loop */
   for (;;)
   {
       // Blocks the task associated with priority 5
       (void)BlockPriority(5);
   }
}
```

UnBlockPriority

`INT8U UnBlockPriority(INT8U iPriority)`

Can be called from:	**Enabled by:**
Tasks	Always available

The **UnBlockPriority** function adds the task associated with this priority to the ready list (this task will compete for the processor again). The system will search for the highest priority task ready to run after running this function.

Arguments:

- priority of the task to be unblocked

Return:

- **OK** - if the task is successfully unblocked

Example:

```
void Example_Task (void)
{
   /* task setup */


   /* task loop */
   for (;;)
   {
       // Unblock the task associated with priority 5
       (void)UnBlockPriority(5);
   }
}
```

BlockTask

`INT8U BlockTask(INT8U iTaskNumber)`

Can be called from:	**Enabled by:**
Tasks	Always available

The **BlockTask** function removes a task from the ready list (this task will not compete for the processor anymore). The system will search for the highest priority task ready to run if the task blocks itself.

Arguments:

- ID of the task to be blocked

Return:

- **OK** - if the task is successfully blocked
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code

Example:

```
void Example_Task (void)
{
   /* task setup */


   /* task loop */
   for (;;)
   {
       // Blocks the task with ID = 5
       (void)BlockTask(5);
   }
}
```

UnBlockTask

```
INT8U UnBlockTask(INT8U iTaskNumber)
```

Can be called from:	**Enabled by:**
Tasks	Always available

The **UnBlockTask** function adds a task to the ready list (this task will compete for the processor again). The system will search for the highest priority task ready to run after running this function.

Arguments:

- ID of the task to be unblocked

Return:

- **OK** - if the task is successfully unblocked

Example:

```
void Example_Task (void)
{
   /* task setup */


   /* task loop */
   for (;;)
   {
       // Unblock the task with ID = 5
       (void)UnBlockTask(5);
   }
}
```

BlockMultipleTask

```
INT8U BlockMultipleTask(INT8U TaskStart, INT8U TaskNumber)
```

Can be called from:	**Enabled by:**
Tasks	Always available

The **BlockMultipleTask** function removes the specified tasks from the ready list (these tasks will not compete for the processor anymore). The running task will be ignored, in the case of being inside of the list of tasks to be blocked. Such function can be used as a **mutex**, blocking the tasks that may compete for one or more resources.

Arguments:

- ID of the first task to be blocked
- number of tasks to be blocked from the task defined in **TaskStart**

Return:

- **OK** - if the tasks are successfully blocked
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code

Example:

```
void Example_Task (void)
{
   /* task setup */


   /* task loop */
   for (;;)
   {
       // Block 4 tasks, starting from the task with ID = 3
       (void)BlockMultipleTask(3,4);
   }
}
```

UnBlockMultipleTask

```
INT8U UnBlockMultipleTask(INT8U TaskStart, INT8U TaskNumber)
```

Can be called from:	**Enabled by:**
Tasks	Always available

The **UnBlockMultipleTask** function adds the specified tasks to the ready list (these tasks will compete for the processor again).

Arguments:

- ID of the first task to be unblocked
- number of tasks to be unblocked from the task defined in **TaskStart**

Return:

- **OK** - if the tasks are successfully unblocked

Example:

```
void Example_Task (void)
{
   /* task setup */


   /* task loop */
   for (;;)
   {
       // Unblock 4 tasks, starting from the task with ID = 3
       (void)UnBlockMultipleTask(3,4);
   }
}
```

Semaphore

OSSemCreate

```
INT8U OSSemCreate (INT8U cnt, BRTOS_Sem **event)
```

Can be called from:	Enabled by:	Number of available semaphores defined by:
System initialization and task	BRTOS_SEM_EN (BRTOSConfig.h)	BRTOS_MAX_SEM (BRTOSConfig.h)

The **OSSemCreate** function creates and initializes a semaphore. A semaphore can be used to:

- synchronize tasks with other tasks and/or with interrupts
- signal the occurrence of an event

Arguments:

- Initial value of the semaphore (0 to 255)
- Address of a pointer of type “semaphore control block” (**BRTOS_Sem**) that will receive the address of the control block allocated for the created semaphore.

Return:

- **ALLOC_EVENT_OK** - if the semaphore is successfully created
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code
- **NO_AVAILABLE_EVENT** – if there is no semaphore control blocks available

Obs.: A semaphore should always be created before its first use.

Example:

```
BRTOS_Sem *TestSem;

void main (void)
{
        // BRTOS initialization
        BRTOS_Init();

        // Creates a semaphore
        if (OSSemCreate(0,&TestSem) != ALLOC_EVENT_OK)
        {
                // Semaphore allocation failed
                // Treat this error here !!!
        }
}
```

OSSemDelete

```
INT8U OSSemDelete (BRTOS_Sem **event)
```

Can be called from:	Enabled by:	Number of available semaphores defined by:
System initialization and task	BRTOS_SEM_EN (BRTOSConfig.h)	BRTOS_MAX_SEM (BRTOSConfig.h)

The **OSSemDelete** function deallocs a semaphore control block, which can be used by another task. Remember that in BRTOS the components (semaphores, mutexes, mailboxes, etc) are statically allocated in compilation time. However, you must "create" the component on the task in order to use it.

Arguments:

- Address of a pointer of type "semaphore control block" (**BRTOS_Sem**) that contains the address of the control block allocated for a previously created semaphore.

Return:

- **DELETE_EVENT_OK** - if the semaphore is successfully deleted
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code

Obs.: To delete a semaphore you must ensure that it is not in use for tasks and interruptions.

Example:

```
BRTOS_Sem *TestSem;

void Example_Task (void)
{
   /* Task setup */


   /* Task loop */
   for (;;)
   {

       // Semahore delete
       if (OSSemDelete(&TestSem) != DELETE_EVENT_OK)
       {
              // Semaphore delete failed
              // Treat this error here !!!
       }
   }
}
```

OSSemPend

`INT8U OSSemPend (BRTOS_Sem *pont_event, INT16U timeout)`

Can be called from:	Enabled by:	Number of available semaphores defined by:
Tasks	BRTOS_SEM_EN (BRTOSConfig.h)	BRTOS_MAX_SEM (BRTOSConfig.h)

The **OSSemPend** function decrements a semaphore if its counter is greater than zero. However, if the counter is zero, this function will put the current task on the semaphore waiting list. The task will wait for a semaphore post for a specified timeout (or indefinitely if timeout equal to 0). BRTOS will wake up the highest priority task waiting by the semaphore, if there is semaphore posting before the timeout expires. A blocked task may be added to the ready list by a semaphore, but will only run after its unblock.

Argumentos:

- Address of a pointer of type "semaphore control block" (**BRTOS_Sem**) that contains the address of the control block allocated to the semaphore in use.
- Timeout for receiving a semaphore post

Retorno:

- **OK** - if the semaphore is successfully decremented
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code
- **TIMEOUT** – if the timeout expires

Obs.: A semaphore should always be created before its first use.

Example:

```
BRTOS_Sem *TestSem;

void Example_Task (void)
{
   /* Task setup */


   /* Task loop */
   for (;;)
   {
      // Use of the function without return test
      (void)OSSemPend(TestSem,15);

      // Use of the function with return test
      if (OSSemPend(TestSem,15) != OK)
      {
           // Semaphore decrement failed
           // Treat this error here !!!
            // For example, timeout expired
      }
   }
}
```

OSSemPost

```
INT8U OSSemPost(BRTOS_Sem *pont_event)
```

Can be called from:	Enabled by:	Number of available semaphores defined by:
Tasks and interrupts	BRTOS_SEM_EN (BRTOSConfig.h)	BRTOS_MAX_SEM (BRTOSConfig.h)

The **OSSemPost** function increments the semaphore if there is no task on its wait list. If there is at least one waiting task, this function removes the highest priority waiting task and adds such task on the ready list. Then the scheduler is invoked to check whether the waked task is the higher priority ready to run.

Arguments:

- Address of a pointer of type "semaphore control block" (**BRTOS_Sem**) that contains the address of the control block allocated to the semaphore in use.

Return:

- **OK** - if the semaphore is successfully decremented
- **ERR_SEM_OVF** – if the semaphore counter overflows
- **NULL_EVENT_POINTER** – if the passed pointer for the semaphore control block is null
- **ERR_EVENT_NO_CREATED** – if the passed semaphore control block has not been correctly allocated (created).

Obs.: A semaphore should always be created before its first use.

Example:

```
BRTOS_Sem *TestSem;

void Example_Task (void)
{
   /* Task setup */

   /* Task loop */
   for (;;)
   {
      // Use of the function without return test
      (void)OSSemPost(TestSem);

      // Use of the function with return test
      if (OSSemPost(TestSem) != OK)
      {
           // Semaphore increment failed
           // Treat this error here !!!
           // For example, counter overflow
      }
   }
}
```

Mutex

OSMutexCreate

```
INT8U OSMutexCreate (BRTOS_Mutex **event, INT8U HigherPriority)
```

Can be called from:	**Enabled by:**	**Number of available mutexes defined by:**
System initialization and task	BRTOS_MUTEX_EN (BRTOSConfig.h)	BRTOS_MUTEX_SEM (BRTOSConfig.h)

The **OSMutexCreate** function creates and initializes a mutex. A mutex is typically used to manage shared resources.

Arguments:

- Address of a pointer of type "mutex control block" (**BRTOS_Mutex**) that will receive the address of the control block allocated for the created mutex.
- The priority that is associated with the mutex. This priority should always be higher than the highest priority of the tasks that will compete for a particular resource. For example, imagine a case where 3 tasks with priorities 4, 10 and 12 share a resource. In such case, the the appropriate value for the priority of the mutex is 13. The task that gets the resource passes its priority to the mutex priority (13), not being interrupted by tasks that compete for the same resource. The task back to its original priority when the mutex release function is called.

Return:

- **ALLOC_EVENT_OK** - if the mutex is successfully created
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code
- **NO_AVAILABLE_EVENT** – if there is no mutex control blocks available
- **BUSY_PRIORITY** – if the specified priority is already in use

Obs.: A mutex should always be created before its first use.

Example:

```
BRTOS_Mutex *TestMutex;

void main (void)
{
        // BRTOS initialization
        BRTOS_Init();

        // Creates a mutex with priority equal to 13
        if (OSMutexCreate(&TestMutex,13) != ALLOC_EVENT_OK)
        {
                // Mutex allocation failed
                // Treat this error here !!!
        }
}
```

OSMutexDelete

```
INT8U OSMutexDelete (BRTOS_Mutex **event)
```

Can be called from:	Enabled by:	Number of available mutexes defined by:
System initialization and task	BRTOS_MUTEX_EN (BRTOSConfig.h)	BRTOS_MUTEX_SEM (BRTOSConfig.h)

The **OSMutexDelete** function deallocs a mutex control block, which can be used by another task.

Arguments:

- Address of a pointer of type "mutex control block" (**BRTOS_Mutex**) that contains the address of the control block allocated for a previously created mutex.

Return:

- **DELETE_EVENT_OK** - if the mutex is successfully deleted
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code

Obs.: To delete a mutex you must ensure that it is not in use for tasks and interruptions.

Example:

```
BRTOS_Mutex *TestMutex;

void Example_Task (void)
{
   /* Task setup */


   /* Task loop */
   for (;;)
   {

       // Deletes a mutex
       if (OSMutexDelete(&TestMutex) != DELETE_EVENT_OK)
       {
              // Mutex delete failed
              // Treat this error here !!!
       }
   }
}
```

OSMutexAcquire

```
INT8U OSMutexAcquire(BRTOS_Mutex *pont_event)
```

Can be called from:	**Enabled by:**	**Number of available mutexes defined by:**
Tasks	BRTOS_MUTEX_EN (BRTOSConfig.h)	BRTOS_MUTEX_SEM (BRTOSConfig.h)

The **OSMutexAcquire** function initially checks whether the resource is available to be acquired. If so, it is immediately marked as busy. The task that called this function gets the temporary ownership of the resource (ie, only this task can release the acquired resource). Thus, the priority of the task that acquired the resource becomes the mutex priority. However, if the resource is busy, this function will put the calling task into the mutex waiting list. The task will wait indefinitely for the mutex release. BRTOS will wake up the task with the highest priority waiting for the mutex when the mutex release occurs. A blocked task can be added to the ready list by a mutex, but will only run after its unblock.

Arguments:

- Address of a pointer of type “mutex control block” (**BRTOS_Mutex**) that contains the address of the control block allocated to the mutex in use.

Return:

- **OK** - if the mutex is successfully acquired
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code

Obs.: A mutex should always be created before its first use.

Example:

```
BRTOS_Mutex *TestMutex;

void Example_Task (void)
{
   /* Task setup */


   /* Task loop */
   for (;;)
   {
      // Acquires a resource
      (void)OSMutexAcquire(TestMutex);

      // Performs the necessary operations with the acquired resource
   }
}
```

OSMutexRelease

```
INT8U OSMutexRelease(BRTOS_Mutex *pont_event)
```

Can be called from:	**Enabled by:**	**Number of available mutexes defined by:**
Tasks	BRTOS_MUTEX_EN (BRTOSConfig.h)	BRTOS_MUTEX_SEM (BRTOSConfig.h)

The **OSMutexRelease** function releases a shared resource to be used by other tasks, returning the priority of the task to its original priority. If there are tasks in the mutex wait list, the task with the highest priority is removed from the mutex wait list and added to the ready list (changing its priority to the mutex priority). The scheduler is invoked to check whether the waked up task is the task with the higher priority ready to run.

Arguments:

- Address of a pointer of type "mutex control block" (**BRTOS_Mutex**) that contains the address of the control block allocated to the mutex in use.

Return:

- **OK** - if the mutex is successfully released
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code
- **ERR_EVENT_OWNER** – if the task trying to release the mutex is not its temporary owner

Obs.: A mutex should always be created before its first use.

Example:

```
BRTOS_Mutex *TestMutex;

void Example_Task (void)
{
   /* Task setup */


   /* Task loop */
   for (;;)
   {
      // Use of the function without return test
      (void)OSMutexRelease(TestMutex);

      // Use of the function with return test
      if (OSMutexRelease(TestMutex) != OK)
      {
           // Mutex release failed
           // Treat this error here !!!
           // For example, task is not the temporary mutex owner
      }
   }
}
```

In some cases it is necessary that several resources are acquired to perform a particular procedure. In such cases care should be taken in the order used to acquire the resources. In BRTOS the resources should be released in the reverse order of their acquisition. Releasing mutexes in the mistaken order can lead to system inconsistencies.

Example:

```
BRTOS_Mutex *TestMutex1;
BRTOS_Mutex *TestMutex2;
BRTOS_Mutex *TestMutex3;


void Example_Task (void)
{
   /* Task setup */


   /* Task loop */
   for (;;)
   {
      // Acquires resource 1
      (void)OSMutexAcquire(TestMutex1);

      // Acquires resource 2
      (void)OSMutexAcquire(TestMutex2);

      // Acquires resource 3
      (void)OSMutexAcquire(TestMutex3);


      ////////////////////////////////////////////////////////////////////
      // Performs the necessary operations with the acquired resource   //
      ////////////////////////////////////////////////////////////////////


      // Releases resource 3
      (void)OSMutexRelease(TestMutex3);

      // Releases resource 2
      (void)OSMutexRelease(TestMutex2);

      // Releases resource 1
      (void)OSMutexRelease(TestMutex3);
   }
}
```

Mailbox

OSMboxCreate

```
INT8U OSMboxCreate (BRTOS_Mbox **event, void *message)
```

Can be called from:	**Enabled by:**	**Number of available mailboxes defined by:**
System initialization and task	BRTOS_MBOX_EN (BRTOSConfig.h)	BRTOS_MBOX_SEM (BRTOSConfig.h)

The **OSMboxCreate** function creates and initializes a mailbox. A mailbox must be used to pass values / pointer from interrupts to tasks or, from tasks to another tasks.

Arguments:

- Address of a pointer of type “mailbox control block” (**BRTOS_Mbox**) that will receive the address of the control block allocated for the created mailbox.
- Pointer of a message to be passed. If it is not required an initial message, you must pass NULL.

Return:

- **ALLOC_EVENT_OK** - if the mailbox is successfully created
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code
- **NO_AVAILABLE_EVENT** – if there is no mailbox control blocks available

Obs.: A mailbox should always be created before its first use.

Example:

```
BRTOS_Mbox *TestMbox;

void main (void)
{
        // BRTOS initialization
        BRTOS_Init();

        // Create mailbox
        if (OSMboxCreate(&TestMbox,NULL) != ALLOC_EVENT_OK)
        {
                // Mailbox allocation failed
                // Treat this error here !!!
        }
}
```

OSMboxDelete

```
INT8U OSMboxDelete (BRTOS_Mbox **event)
```

Can be called from:	**Enabled by:**	**Number of available mailboxes defined by:**
System initialization and task	BRTOS_MBOX_EN (BRTOSConfig.h)	BRTOS_MBOX_SEM (BRTOSConfig.h)

The **OSMboxDelete** function deallocs a mailbox control block, which can be used by another task.

Arguments:

- Address of a pointer of type “mailbox control block” (**BRTOS_Mbox**) that contains the address of the control block allocated for a previously created mailbox.

Return:

- **DELETE_EVENT_OK** - if the mailbox is successfully deleted
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code

Obs.: To delete a mailbox you must ensure that it is not in use for tasks and interruptions.

Example:

```
BRTOS_Mbox *TestMbox;

void Example_Task (void)
{
   /* Task setup */


   /* Task loop */
   for (;;)
   {

       // Deletes a mailbox
       if (OSMboxDelete(&TestMbox) != DELETE_EVENT_OK)
       {
              // Mailbox delete failed
              // Treat this error here !!!
       }
   }
}
```

OSMboxPend

```
INT8U OSMboxPend (BRTOS_Mbox *pont_event, void **Mail, INT16U timeout)
```

Pode ser chamado de:	**Habilitado por:**	**Número de caixas de mensagem disponíveis definido por:**
Tarefas	BRTOS_MBOX_EN (BRTOSConfig.h)	BRTOS_MAX_MBOX (BRTOSConfig.h)

The **OSMboxPend** function verifies if there is a available message. If so, it returns the message through a pointer (mail). However, if there is no message, this function will put the current task into the mailbox wait list. The task will wait for a message posting for a specified timeout (or indefinitely if timeout equal to 0). BRTOS will wake up the highest priority task waiting by a message, if there is a message posting before the timeout expires. A blocked task may be added to the ready list by a mailbox, but will only run after its unblock.

Arguments:

- Address of a pointer of type “mailbox control block” (**BRTOS_Mbox**) that contains the address of the control block allocated to the mailbox in use.
- Address of the pointer that will receive the message
- Timeout for receiving a message post

Return:

- **OK** - if the message is successfully received
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code
- **TIMEOUT** – if the timeout expires

Obs.: A mailbox should always be created before its first use.

Example:

```
BRTOS_Mbox *TestMbox;

void Example_Task (void)
{
   /* Task setup */
   INT16U *teste;

   /* Task loop */
   for (;;)
   {
      // Use of the function with return test
      if (OSMboxPend(TestMbox,(void **)&teste,0) != OK)
      {
           // Message delivery failed
           // Treat this error here !!!
           // For example, timeout expired
      }
   }
}
```

OSMboxPost

```
INT8U OSMboxPost(BRTOS_Mbox *pont_event, void *message)
```

Pode ser chamado de:	**Habilitado por:**	**Número de caixas de mensagem disponíveis definido por:**
Tarefas e interrupções	BRTOS_MBOX_EN (BRTOSConfig.h)	BRTOS_MAX_MBOX (BRTOSConfig.h)

The **OSMboxPost** function sends a message to the specified mailbox. If there is more than one task waiting for the message, this function removes the task with the highest priority of its wait list and adds such task to the ready list. Then the scheduler is invoked to check whether the waked task is the higher priority ready to run.

Arguments:

- Address of a pointer of type "mailbox control block" (**BRTOS_Mbox**) that contains the address of the control block allocated to the mailbox in use.
- Pointer of the sent message

Return:

- **OK** - if the message is successfully sent
- **NULL_EVENT_POINTER** – if the passed pointer for the mailbox control block is null
- **ERR_EVENT_NO_CREATED** – if the passed mailbox control block has not been correctly allocated (created).

Obs.: A mailbox should always be created before its first use.

Example:

```
BRTOS_Mbox *TestMbox;

void Example_Task (void)
{
   /* Task setup */
   INT16U teste = 0;

   /* Task loop */
   for (;;)
   {
      // Use of this function without return test
      (void)OSMboxPost(TestMbox,(void *)&teste);
   }
}
```

Example of passing a pointer through mailbox:

```
BRTOS_Mbox *TestMbox;


void Example_Task_1 (void)
{
   /* Task setup */
   INT16U *recebe_pont;
   INT16U recebe;

   /* Task loop */
   for (;;)
   {
      // Use of the function with return test
      if (OSMboxPend(TestMbox,(void **)&recebe_pont,0) != OK)
      {
           // Message delivery failed
           // Treat this error here !!!
            // For example, timeout expired
      }
      // Copies the value passed by the message, if necessary
      recebe = *recebe_pont;
   }
}


void Example_Task_2 (void)
{
   /* Task setup */
   INT16U envia = 0;

   /* Task loop */
   for (;;)
   {
      envia++;
      // Use of the function without return test
      (void)OSMboxPost(TestMbox,(void *)&envia);
   }
}
```

Example of passing a string pointer through mailbox:

```
BRTOS_Mbox *TestMbox;


void Example_Task_1 (void)
{
   /* Task setup */
   CHAR8 *recebe_pont;

   /* Task loop */
   for (;;)
   {
      // Use of the function with return test
      if (OSMboxPend(TestMbox,(void **)&recebe_pont,0) != OK)
      {
           // Message delivery failed
           // Treat this error here !!!
           // For example, timeout expired
      }

      // Uses the received message
      while(*recebe_pont)
      {
        Serial_Envia_Caracter(*recebe_pont);
        recebe_pont++;
      }
      Serial_Envia_Caracter(CR);
      Serial_Envia_Caracter(LF);
   }
}


void Example_Task_2 (void)
{
   /* Task setup */
   CHAR8 *envia = "Teste de envio";

   /* Task loop */
   for (;;)
   {
      // Use of the function without return test
      (void)OSMboxPost(TestMbox,(void *)envia);
   }
}
```

Example of passing value through mailbox:

```
BRTOS_Mbox *TestMbox;


void Example_Task_1 (void)
{
   /* Task setup */
   INT32U *recebe_pont;
   INT32U recebe;

   /* Task loop */
   for (;;)
   {
      // Use of the function with return test
      if (OSMboxPend(TestMbox,(void **)&recebe_pont,0) != OK)
      {
           // Message delivery failed
           // Treat this error here !!!
            // For example, timeout expired
      }

      // Copies the value passed by the message, if necessary
      recebe = (INT32U)recebe_pont;
   }
}


void Example_Task_2 (void)
{
   /* Task setup */
   INT32U *envia;
   INT32U inc = 0;

   /* Task loop */
   for (;;)
   {
      inc++;
      envia = (INT32U*)inc;

      // Use of the function without return test
      (void)OSMboxPost(TestMbox,(void *)envia);
   }
}
```

Queue

OSQueueXXCreate

`INT8U OSQueueXXCreate(OS_QUEUE_XX *cqueue, INT16U size, BRTOS_Queue **event)`

Can be called from:	Enabled by:	Number of available queues defined by:
System initialization and task	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSQueueXXCreate** function creates and initializes a queue, where XX specifies the size of data on the queue (8, 16 or 32 bits). Queues can be seen as mailbox vectors, being used to pass values / pointers from interrupts to tasks or from tasks to other tasks.

Arguments:

- The pointer of a system queue (**OS_QUEUE_8**, **OS_QUEUE_16** ou **OS_QUEUE_32**)
- Maximum amount of data that can be stored in the queue
- Address of a pointer of type "queue control block" (**BRTOS_Queue**) that will receive the address of the control block allocated for the created queue.

Return:

- **ALLOC_EVENT_OK** - if the queue is successfully created
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code
- **NO_MEMORY** – if the requested size to the queue is greater than the space available in the queue heap
- **NO_AVAILABLE_EVENT** – if there is no queue control blocks available

Obs.: A queue should always be created before its first use.

Example:

```
BRTOS_Queue *TestQueue;
OS_QUEUE_16 QueueBuffer;

void main (void)
{
       // BRTOS initialization
       BRTOS_Init();

       // Creates a queue with 128 positions of 16 bits (256 bytes)
       if (OSQueue16Create(&QueueBuffer, 128, &TestQueue) != ALLOC_EVENT_OK)
       {
              // Queue allocation failure
              // Treat this error here !!!
       }
}
```

OSQueuePend

```
INT8U OSQueuePend (BRTOS_Queue *pont_event, INT8U* pdata, INT16U timeout)
```

Can be called from:	Enabled by:	Number of available queues defined by:
Tasks	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSQueuePend** function verifies if there is available data in the queue. If so, it returns the first data available into the queue. However, if there is no data, this function will put the current task into the queue wait list. The task will wait for a queue posting for a specified timeout (or indefinitely if timeout equal to 0). BRTOS will wake up the highest priority task waiting for queue data, if there is a queue posting before the timeout expires. A blocked task may be added to the ready list by a queue, but will only run after its unblock.

Arguments:

- Address of a pointer of type "queue control block" (**BRTOS_Queue**) that contains the address of the control block allocated to the queue in use.
- Pointer of the variable that will receive the queue data
- Timeout for receiving a queue post

Return:

- **READ_BUFFER_OK** - if the queue data is successfully received
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code
- **TIMEOUT** – if the timeout expires

Obs 1.: A queue should always be created before its first use.
Obs 2.: Version 1.7x of BRTOS only supports Pend and Post function for 8 bits queues or dynamic queues.

Example:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Task setup */
   INT8U *teste;

   /* Task loop */
   for (;;)
   {
      // Use of the function with return test
      if(!OSQueuePend(TestQueue, &teste, 0))
      {
        switch(teste)
        {
           ....
        }
   }
}
```

OSQueuePost

```
INT8U OSQueuePost(BRTOS_Queue *pont_event, INT8U data)
```

Can be called from:	**Enabled by:**	**Number of available queues defined by:**
Tasks and interrupts	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSQueuePost** function sends data to the specified queue. If there is more than one task waiting for the queue data, this function removes the task with the highest priority of its wait list and adds such task to the ready list. Then the scheduler is invoked to check whether the waked task is the higher priority ready to run.

Arguments:

- Address of a pointer of type "queue control block" (**BRTOS_Queue**) that contains the address of the control block allocated to the queue in use.
- Data to be sent to the queue

Return:

- **WRITE_BUFFER_OK** - if the data is successfully sent to the queue
- **BUFFER_UNDERRUN** – if there is no more space available to copy data into the queue
- **NULL_EVENT_POINTER** – if the passed pointer for the queue control block is null
- **ERR_EVENT_NO_CREATED** – if the passed queue control block has not been correctly allocated (created).

Obs 1.: A queue should always be created before its first use.
Obs 2.: Version 1.7x of BRTOS only supports Pend and Post function for 8 bits queues or dynamic queues.

Example:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Task setup */
   INT16U teste = 0;

   /* Task loop */
   for (;;)
   {
      teste++;
      // Use of the function with return test
      if (OSQueuePost(TestQueue, teste) == BUFFER_UNDERRUN)
      {
         // Error: queue underrun
         OSCleanQueue(TestQueue);
      }
   }
}
```

OSCleanQueue

```
INT8U OSCleanQueue(BRTOS_Queue *pont_event)
```

Can be called from:	**Enabled by:**	**Number of available queues defined by:**
Tasks and interrupts	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSCleanQueue** function resets the pointers (in and out) and clears the counter of queue entries. This function can be used when the queue overflows.

Arguments:

- Address of a pointer of type "queue control block" (**BRTOS_Queue**) that contains the address of the control block allocated to the queue in use.

Return:

- **CLEAN_BUFFER_OK** - if the queue is successfully cleared
- **NULL_EVENT_POINTER** – if the passed pointer for the queue control block is null
- **ERR_EVENT_NO_CREATED** – if the passed queue control block has not been correctly allocated (created).

Obs.: A queue should always be created before its first use.

Example:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Task setup */

   /* Task loop */
   for (;;)
   {
      // Resets the queue TesteQueue
      OSCleanQueue(TestQueue);
   }
}
```

OSWQueueXX

```
INT8U OSWQueueXX(OS_QUEUE_XX *cqueue, INTXXU data)
```

Can be called from:	**Enabled by:**	**Number of available queues defined by:**
Tasks and interrupts	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSWQueueXX** function writes a data into the queue without using the event service of the system, where XX specifies the size of data in the queue (8, 16 or 32 bits)

Arguments:

- The pointer of a system queue (**OS_QUEUE_8**, **OS_QUEUE_16** or **OS_QUEUE_32**) that contains the address of the queue structure in use.
- The data to be sent to the queue

Return:

- **WRITE_BUFFER_OK** - if the data is successfully sent to the queue
- **BUFFER_UNDERRUN** – if there is no more space available to copy data into the queue

Obs.: A queue should always be created before its first use.

Example:

```
OS_QUEUE_16 QueueBuffer;

void Example_Task (void)
{
   /* Task setup */
   INT16U teste = 0;

   /* Task loop */
   for (;;)
   {
      teste++;
      // Use of the function without return test
      if (OSWQueue16(QueueBuffer, teste) == BUFFER_UNDERRUN)
      {
         // Error: buffer underrun
         OSCleanQueue(TestQueue);
      }
   }
}
```

OSRQueueXX

```
INT8U OSRQueueXX(OS_QUEUE_XX *cqueue, INTXXU *pdata)
```

Can be called from:	Enabled by:	Number of available queues defined by:
Tasks and interrupts	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSRQueueXX** function receives (copies) data from the specified queue without using the event service of the system, where XX specifies the size of data in the queue (8, 16 or 32 bits).

Arguments:
- The pointer of a system queue (**OS_QUEUE_8**, **OS_QUEUE_16** or **OS_QUEUE_32**) that contains the address of the queue structure in use.
- Pointer of the variable that will receive the queue data

Return:
- **READ_BUFFER_OK** - if the queue data is successfully received
- **NO_ENTRY_AVAILABLE** – if there is no data into the queue

Obs.: A queue should always be created before its first use.

Example:

```
OS_QUEUE_16 QueueBuffer;

void Example_Task (void)
{
   /* Task setup */
   INT16U teste = 0;

   /* Task loop */
   for (;;)
   {
      // Use of the function with return test
      if (OSRQueue16(QueueBuffer, &teste) == READ_BUFFER_OK)
      {
        switch(teste)
        {
           ....
        }
      }
   }
}
```

Dynamic Queues

OSDQueueCreate

```
INT8U OSDQueueCreate(INT16U queue_length, OS_CPU_TYPE type_size,
BRTOS_Queue **event)
```

Can be called from:	**Enabled by:**	**Number of available dynamic queues defined by:**
System initialization and task	BRTOS_DYNAMIC_QUEUE_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSDQueueCreate** function creates and initializes a dynamic queue. Such queues have two basic differences from the traditional BRTOS queues: 1. can be created and deleted; 2. The size of the data type that the queue sends and receives can be configured in its creation. Thus, it is possible, for example, create a queue of a data structure with several bytes.

Arguments:
- Maximum amount of data that can be stored in the queue
- The size in bytes of the data type to be stored in the queue
- Address of a pointer of type "queue control block" (**BRTOS_Queue**) that will receive the address of the control block allocated for the created queue.

Return:
- **ALLOC_EVENT_OK** - if the queue is successfully created
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code
- **NO_AVAILABLE_MEMORY** – if the requested size to the queue is greater than the space available in the dynamic queue heap
- **NO_AVAILABLE_EVENT** – if there is no queue control blocks available
- **INVALID_PARAMETERS** – if the amount of data to be stored into the queue and/or te dise in bytes of the data type is less than or equal to zero

Obs.: A queue should always be created before its first use.

Example:
```
BRTOS_Queue *TestQueue;

void main (void)
{
	// BRTOS initialization
	BRTOS_Init();

	// Creates a queue with 128 positions of a void pointer size
	if (OSDQueueCreate(128, sizeof(void*), &TestQueue) != ALLOC_EVENT_OK)
	{
		// Queue allocation failure
		// Treat this error here !!!
	}
}
```

OSDQueueDelete

```
INT8U OSDQueueDelete(BRTOS_Queue **pont_event)
```

Can be called from:	**Enabled by:**	**Number of available dynamic queues defined by:**
System initialization and task	BRTOS_DYNAMIC_QUEUE_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSDQueueDelete** function deallocates the memory used by the queue and by the queue control block.

Arguments:

- Address of a pointer of type "queue control block" (**BRTOS_Queue**) that contains the address of the control block allocated to the queue in use.

Return:

- **DELETE_EVENT_OK** - if the queue is successfully deleted
- **NULL_EVENT_POINTER** – if the passed pointer for the queue control block is null
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code

Obs.: To delete a queue you must ensure that it is not in use for tasks and interruptions.

Example:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Task setup */
   // Creates a queue with 128 positions of a void pointer size
   if (OSDQueueCreate(128, sizeof(void*), &TestQueue) != ALLOC_EVENT_OK)
   {
       // Queue allocation failure
       // Treat this error here !!!
   }

   /* Task loop */
   for (;;)
   {
      // Excludes queue TesteQueue
      OSDQueueDelete(&TestQueue);
   }
}
```

OSDQueuePend

`INT8U OSDQueuePend (BRTOS_Queue *pont_event, void *pdata, INT16U timeout)`

Can be called from:	**Enabled by:**	**Number of available dynamic queues defined by:**
Tasks	BRTOS_DYNAMIC_QUEUE_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSDQueuePend** function verifies if there is available data in the queue. If so, it returns the first data available into the queue. However, if there is no data, this function will put the current task into the queue wait list. The task will wait for a queue posting for a specified timeout (or indefinitely if timeout equal to 0). BRTOS will wake up the highest priority task waiting for queue data, if there is a queue posting before the timeout expires. A blocked task may be added to the ready list by a queue, but will only run after its unblock.

Arguments:

- Address of a pointer of type “queue control block” (**BRTOS_Queue**) that contains the address of the control block allocated to the queue in use.
- Pointer of the variable that will receive the queue data (if necessary, use *casting* for void*)
- Timeout for receiving a queue post

Return:

- **READ_BUFFER_OK** - if the queue data is successfully received
- **IRQ_PEND_ERR** – if the function is called in an interrupt handling code
- **TIMEOUT** – if the timeout expires

Obs.: A queue should always be created before its first use.

Example:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Task setup */
   INT32U *teste;

   /* Task loop */
   for (;;)
   {
      // Use of the function with return test
      if(!OSDQueuePend(TestQueue, (void*)&teste, 0))
      {
        switch(teste)
        {
           ....
        }
   }
}
```

OSDQueuePost

```
INT8U OSDQueuePost(BRTOS_Queue *pont_event, void *pdata)
```

Can be called from:	**Enabled by:**	**Number of available dynamic queues defined by:**
Tasks and interrupts	BRTOS_DYNAMIC_QUEUE_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSDQueuePost** function sends data to the specified queue. If there is more than one task waiting for the queue data, this function removes the task with the highest priority of its wait list and adds such task to the ready list. Then the scheduler is invoked to check whether the waked task is the higher priority ready to run.

Arguments:

- Address of a pointer of type "queue control block" (**BRTOS_Queue**) that contains the address of the control block allocated to the queue in use.
- Address of the data to be sent to the queue

Return:

- **WRITE_BUFFER_OK** - if the data is successfully sent to the queue
- **BUFFER_UNDERRUN** – if there is no more space available to copy data into the queue
- **NULL_EVENT_POINTER** – if the passed pointer for the queue control block is null
- **ERR_EVENT_NO_CREATED** – if the passed queue control block has not been correctly allocated (created).

Obs.: A queue should always be created before its first use.

Example:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Task setup */
   INT16U teste = 0;

   /* Task loop */
   for (;;)
   {
      teste++;
      // Use of the function with return test
      if (OSDQueuePost(TestQueue, (void*)&teste) == BUFFER_UNDERRUN)
      {
         // Error: Queue underrun
         OSCleanQueue(TestQueue);
      }
   }
}
```

OSDQueueClean

```
INT8U OSDCleanQueue(BRTOS_Queue *pont_event)
```

Can be called from:	**Enabled by:**	**Number of available dynamic queues defined by:**
Tasks and interrupts	BRTOS_DYNAMIC_QUEUE_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

The **OSDCleanQueue** function resets the pointers (in and out) and clears the counter of queue entries. This function can be used when the queue overflows.

Arguments:

- Address of a pointer of type "queue control block" (**BRTOS_Queue**) that contains the address of the control block allocated to the queue in use.

Return:

- **CLEAN_BUFFER_OK** - if the queue is successfully cleared
- **NULL_EVENT_POINTER** – if the passed pointer for the queue control block is null
- **ERR_EVENT_NO_CREATED** – if the passed queue control block has not been correctly allocated (created).

Obs.: A queue should always be created before its first use.

Example:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Task setup */

   /* Task loop */
   for (;;)
   {
      // Resets the queue TesteQueue
      OSDCleanQueue(TestQueue);
   }
}
```

BRTOS initialization example with two tasks

main.c file:

```
#include "tasks.h"

// Parameters
Porta_IO Dados_Porta_IO;

void main (void)
{
       // BRTOS initialization
       BRTOS_Init();

      // Setup of a I/O pin to be used into the task
      Dados_Porta_IO.direcao = 3;
      Dados_Porta_IO.dados = 0;


       // Installs a task with parameters
      if(InstallTask(&System_Time,"System Time",150,31,(void*)&ParamTest) != OK)
      {
        // Should not enter here. Treat this exception!!
        while(1){};
      };


       // Installs a second task without parameters
      if(InstallTask(&Test_Task,"Tarefa de teste",100,10,NULL) != OK)
      {
        // Should not enter here. Treat this exception!!
        while(1){};
      };


      // Starts the scheduler
      if(BRTOSStart() != OK)
      {
        // Should not enter here !!!
        // Should have jumped to the highest priority task
        while(1){};
      };


      for(;;)
      {
         // Nor should come here !!!
      }
}
```

tasks.h file:

```
struct _IO
{
  INT8U direcao;
  INT8U dados;
};

typedef struct _IO Porta_IO;

// Tasks prototypes
void System_Time(void *parameters);
void Test_Task(void *parameters);
```

tasks.c file:

```
#include "tasks.h"

void System_Time(void *parameters)
{
   /* task setup */
   Porta_IO *param = (Porta_IO*)parameters;
   PTAD = param->dados;
   PTADD = param->direcao;

   /* task main loop */
   for (;;)
   {
      PTAD_PTAD1 = ~PTAD_PTAD1;
      DelayTask(100);
   }
}



void Test_Task(void *parameters)
{
   /* task setup */
   INT8U count = 0;
   (void)parameters;   // ignores the parameters


   /* task main loop */
   for (;;)
   {
      count++;
      DelayTask(50);
   }
}
```