Manual de Referência do BRTOS

Versão 1.7x

Apoio

Outubro de 2012

Sumário

Configurações do BRTOS

Arquivo BRTOSConfig.h

```
/// 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)25165824    ///< 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
/// Coldfire with 8KB of RAM: 40 * 128 bytes = 5KB of Virtual Stack
#define HEAP_SIZE 46*128


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

- **DEBUG** - é utilizado em algumas plataformas que possuem diferenciações entre o simulador e o *real-time debugger*, adaptando o código a condição desejada através de diretivas de pré-compilador. DEBUG = 1 indica modo *debugger* / gravação do código no microcontrolador.
- **VERBOSE** – ativa informações adicionais no contexto das tarefas, que podem ser utilizadas na depuração dos projetos. VERBOSE = 1 ativa informações adicionais no contexto das tarefas.
- **ERROR_CHECK** – ativa verificações adicionais de erros que possam ocorrer com o uso incorreto das chamadas do sistema, tornando o sistema menos susceptível a falhas. ERROR_CHECK = 1 ativa a verificação de erros.
- **WATCHDOG** - indica se o *watchdog* do sistema estará ativo. WATCHDOG = 1 indica *watchdog* ativo.
- **COMPUTES_CPU_LOAD** – ativa o cálculo de ocupação de CPU do sistema. COMPUTES_CPU_LOAD = 1 ativa o cálculo de ocupação de CPU.

- **NUMBER_OF_PRIORITIES** - é utilizado para informar ao sistema a quantidade de prioridades disponíveis. Os valores válidos são 16 ou 32. Recomenda-se utilizar 16 prioridades para microcontroladores de 8 bits e 32 prioridades para microcontroladores de 16 e 32 bits.
- **NUMBER_OF_TASKS** - indica quantas tarefas podem ser instaladas em uma aplicação. O valor máximo para esta definição é 31 para **NUMBER_OF_PRIORITIES** = 32 e 15 para **NUMBER_OF_PRIORITIES** = 16. Esta definição permite que uma menor quantidade de memória seja alocada para estruturas de contexto de tarefas. Utilize este recurso para reduzir o consumo de memória do sistema quando um número pequeno de tarefas for instalado.
- **BRTOS_ALLOC** – define a função de alocação de memória utilizada na implementação de filas dinâmicas. Por padrão utiliza-se a função padrão da linguagem C, ***malloc***. OBS.: Note que para utilizar esta função deve-se definir o tamanho do *heap* (quantidade de memória disponível para a alocação dinâmica) no arquivo de *linker* do *toolchain* utilizado. Pode-se ainda utilizar funções de alocação personalizadas.
- **BRTOS_DEALLOC** – define a função de desalocação de memória utilizada na implementação de filas dinâmicas. Por padrão utiliza-se a função padrão da linguagem C, ***free***. As mesmas observações da definição BRTOS_ALLOC são válidas para esta definição.
- **OS_TRACE** - habilita ou desabilita o *trace* do sistema. Com este recurso habilitado é possível monitorar o comportamento das tarefas, interrupções e chamadas de sistema. Desta forma torna-se possível identificar comportamentos errôneos, chamadas indevidas, entre outros problemas que possam vir a ocorrer em uma determinada aplicação.
- **TIMER_HOOK_EN** - habilita ou desabilita uma função ancora no *Tick* do sistema. Esta função pode ser utilizada para implementar trechos curtos de código que dependam de um temporizador. Como estes trechos de código são curtos, não justificam a utilização de uma tarefa. **CUIDADO**: Se o sistema não estiver permitindo a utilização de aninhamento de interrupções, esta função estará em região com interrupções bloqueadas.
- **IDLE_HOOK_EN** - habilita ou desabilita uma função ancora no tarefa *Idle* do sistema. **CUIDADO**: O código executado nesta função é o de menor prioridade do sistema.
- **BRTOS_SEM_EN** – habilita os serviços de semáforo do sistema quando BRTOS_SEM_EN = 1.
- **BRTOS_MUTEX_EN** - habilita os serviços de *mutex* (controle de acesso exclusivo) do sistema quando BRTOS_MUTEX_EN = 1.
- **BRTOS_MBOX_EN** - habilita os serviços de caixa de mensagem do sistema quando BRTOS_MBOX_EN = 1.
- **BRTOS_QUEUE_EN** - habilita os serviços de fila (filas de bytes) do sistema quando BRTOS_QUEUE_EN = 1.
- **BRTOS_DYNAMIC_QUEUE_ENABLED** – habilita os serviços de filas dinâmicas do sistema quando BRTOS_DYNAMIC_QUEUE_ENABLED = 1. As filas dinâmicas suportam qualquer tamanho de dados. Ainda, é o único serviço de filas do BRTOS que suporta tanto a criação quando a exclusão de filas. Para sua correta implementação as definições BRTOS_ALLOC e BRTOS_DEALLOC devem apontar para funções que implementem a alocação e desalocação dinâmica de memória de dados.

- **BRTOS_QUEUE_16_EN** - habilita os serviços de fila (filas de words) do sistema quando BRTOS_QUEUE_16_EN = 1.
- **BRTOS_QUEUE_32_EN** - habilita os serviços de fila (filas de valores de 32 bits) do sistema quando BRTOS_QUEUE_32_EN = 1.
- **BRTOS_MAX_SEM** – define o número máximo de blocos de controle de semáforos disponíveis no sistema.
- **BRTOS_MAX_MUTEX** - define o número máximo de blocos de controle de *mutex* disponíveis no sistema.
- **BRTOS_MAX_MBOX** - define o número máximo de blocos de controle de caixas de mensagem disponíveis no sistema.
- **BRTOS_MAX_QUEUE** - define o número máximo de blocos de controle de filas disponíveis no sistema.
- **configCPU_CLOCK_HZ** - indica a frequência de barramento utilizada pelo microcontrolador em hertz.
- **configTICK_RATE_HZ** - define o *Timer Tick* (marca de tempo) do sistema, ou seja, a resolução do gerenciador de tempo do RTOS. Valores entre 1ms (1000 Hz) e 10ms (100 Hz) são recomendados. Nunca esqueça que a resolução do gerenciamento de tempo é de ±1 *Timer Tick*.
- **configTIMER_PRE_SCALER** - pode ser utilizado no *port* do sistema para a configuração do *hardware* responsável pelo *Timer Tick*.
- **IDLE_STACK_SIZE** – tamanho do *stack* virtual da tarefa *Idle* em bytes.
- **HEAP_SIZE** - determina a quantidade de memória alocada como pilha virtual das tarefas. Sempre que uma tarefa é instalada, a quantidade de memória utilizada pela tarefa será alocada no HEAP.
- **QUEUE_HEAP_SIZE** - determina a quantidade de memória alocada como pilha virtual das filas do sistema. Sempre que uma fila é criada, a quantidade de memória utilizada pela fila será alocada no QUEUE_HEAP.

Arquivo HAL.h

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

/// Enables or disables the coldfire core as the default processor
#define PROCESSOR            COLDFIRE_V1
```

```
/// Define the CPU type
#define OS_CPU_TYPE          INT32U

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

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

/// Define if InstallTask function will support parameters
#define TASK_WITH_PARAMETERS 1

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

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

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

- **Definição do processador utilizado (PROCESSOR)** – Esta definição pode ser utilizada para configurar diretivas de pré-compilador especificas de um processador (pragmas por exemplo).
- **OS_CPU_TYPE** – é utilizado para informar ao BRTOS o tipo base do processador utilizado. Por exemplo, em um microcontrolador de 32 bits o tipo base é um inteiro de 32 bits. Já em um microcontrolador de 8 bits o tipo base é *char*.
- **FPU_SUPPORT** – nos microcontroladores em que existe uma unidade de ponto flutuante, essa definição informa ao HAL do BRTOS que é necessário habilitar essa unidade e salvar os registradores específicos desta unidade.
- **OPTIMIZED_SCHEDULER** – deve ser configurado para 1 se o processador possuir as instruções necessárias para implementar a aceleração do escalonador. Até o momento, somente os processador Coldfire, e ARM-Cortex-M3 e 4 possuem estas instruções. Note que o escalonador otimizado deve estar definido no HAL.
- **TASK_WITH_PARAMETERS** – indica se o *port* disponível no HAL suportará a passagem de parâmetros na instalação de tarefas (ao ser configurado como 1). Um exemplo de passagem de parâmetros é disponibilizado na descrição da função *InstallTask*.
- **TICK_TIMER_32BITS** – o padrão de registrador para *tick* do sistema no BRTOS é 16 bits. No entanto, em alguns processadores (como os ARM-Cortex-Mx), o registrador de módulo do temporizador é de 32 bits, caso onde esta definição deve ser configurada para 1.
- **NESTING_INT** - é utilizado para informar ao BRTOS se o processador utilizado possui suporte a aninhamento de interrupções (ou se este recurso está ativado). NESTING_INT = 1 informa ao sistema que existe suporte a aninhamento de interrupções.
- **SP_SIZE** - é utilizado para informar ao BRTOS o tamanho do registrador *stack pointer* (SP) do processador em bits. Os valores válidos para *Stack Pointer* são 16 e 32. Por exemplo, os microcontroladores HCS08 e MSP430 utilizam SP_SIZE = 16. Já o microcontrolador Coldfire V1 utiliza SP_SIZE = 32.

Serviços do núcleo do BRTOS

OSEnterCritical e OSExitCritical

Pode ser chamado de:	**Habilitado por:**
Tarefas e interrupções	Sempre disponível

As funções **OSEnterCritical** e **OSExitCritical** são usadas para desabilitar e habilitar, respectivamente, as interrupções do processador. Note que no *port* de alguns processadores é necessário declarar a variável responsável por salvar a prioridade de bloqueio atual das interrupções ao utilizar estas funções, como demonstrado no exemplo abaixo.

Exemplo:

```
void Example_Task (void)
{
   /* Configuração da tarefa */
   OS_SR_SAVE_VAR     // Define variável utilizada para salvar o prioridade atual

   /* Laço da tarefa */
   for (;;)
   {
      // Desabilita as interrupções
      OSEnterCritical();

      // Execute aqui seu código de acesso crítico

      // Habilita as interrupções
      OSExitCritical();
   }
}
```

UserEnterCritical e UserExitCritical

Pode ser chamado de:	**Habilitado por:**
Tarefas	Sempre disponível

As funções **UserEnterCritical** e **UserExitCritical** são usadas para desabilitar e habilitar, respectivamente, as interrupções do processador por código de usuário. A diferença das funções **OSEnterCritical** e **OSExitCritical** para as funções **UserEnterCritical** e **UserExitCritical** é dependente do *hardware* utilizado. Em processadores que não suportam interrupções aninhadas a implementação destas funções é a mesma. Já em processadores com suporte a aninhamento de interrupções, a função **OSEnterCritical** salva o nível de interrupção em que o processador se encontra para restaurá-lo na saída da seção crítica através da função **OSExitCritical**. O modo de utilização de serviços de região crítica de usuário é o mesmo que o de sistema, com a ressalva de

que só pode ser utilizado dentro de tarefas, nunca dentro de interrupções. Normalmente as funções de região crítica de usuário ocupam menos ciclos de *clock* do que as funções do sistema.

InstallTask

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

ou

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

Pode ser chamado de:	**Habilitado por:**
Inicialização do sistema e tarefas	Sempre disponível

A função **InstallTask** instala uma tarefa, alocando suas informações em um bloco de controle de tarefa disponível. Uma tarefa pode ser instalada antes do inicio do escalonador (inicialização do sistema) ou por uma tarefa.

Argumentos:

- Ponteiro para a tarefa a ser instalada
- Nome da tarefa
- Tamanho do stack virtual da tarefa
- Prioridade da tarefa
- Parâmetros necessários para a instalação da tarefa (somente disponível se TASK_WITH_PARAMETERS = 1).

Retorno:

- **OK** - se a tarefa for instalada com sucesso
- **STACK_SIZE_TOO_SMALL** – se o tamanho do *stack* definido pelo usuário for menor que o mínimo necessário para inicializar a tarefa
- **NO_MEMORY** – se a memória disponível para alocar o *stack* virtual (HEAP do sistema) não for suficiente para alocar o *stack* virtual especificado
- **END_OF_AVAILABLE_PRIORITIES** – se a prioridade especificada for maior do que o limite do sistema
- **BUSY_PRIORITY** – se a prioridade especifica estiver ocupada
- **CANNOT_ASSIGN_IDLE_TASK_PRIO** – se a prioridade especificada for 0 (prioridade da *Idle Task*).

Exemplo sem parâmetros:

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

       // Instala uma tarefa
      if(InstallTask(&System_Time,"System Time",150,31) != OK)
      {
        // Não deveria entrar aqui. Trate esta exceção !!!
        while(1){};
      };
}
```

Exemplo com parâmetros:

```
// Demo of parameters
struct _parametros
{
  INT8U direcao;
  INT8U dados;
};

typedef struct _parametros Parametros;

Parametros ParamTest;

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

      // Determina a configuração de um pino de I/O dentro da tarefa
      ParamTest.direcao = 3;
      ParamTest.dados = 0;

       // Instala uma tarefa
      if(InstallTask(&System_Time,"System Time",150,31,(void*)&ParamTest) != OK)
      {
        // Não deveria entrar aqui. Trate esta exceção !!!
        while(1){};
      };
}


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

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

OSGetTickCount

`INT16U OSGetTickCount(void)`

Pode ser chamado de:	Habilitado por:
Tarefas e interrupções	Sempre disponível

A função **OSGetTickCount** retorna a contagem atual de *ticks* do sistema. É utilizada em aplicações que necessitem descobrir o tempo de execução de um determinado procedimento.

Exemplo:

```
void Example_Task (void)
{
   /* Configuração da tarefa */
   INT16U tick_count;

   /* Laço da tarefa */
   for (;;)
   {
      // Adquire a contagem atual de ticks do sistema
      tick_count = OSGetTickCount();
   }
}
```

DelayTask

`INT8U DelayTask(INT16U time_wait)`

Pode ser chamado de:	Habilitado por:
Tarefas	Sempre disponível

A função **DelayTask** permite que uma tarefa desista do processador por um número inteiro de *ticks* do sistema (valores típicos do *tick* do sistema vão de 1ms a 100ms). Os valores válidos de *delay* vão de 1 a **TickCountOverFlow** (definido no arquivo BRTOS.h). Um *delay* de 0 significa que a tarefa não irá desistir do processador. A tarefa volta a competir pelo processador tão logo seja acordada pelo sistema.

Argumentos:

- número de *ticks* do sistema que a tarefa atualmente em execução irá desistir do processador

Retorno:

- **OK** - se a função for executada com sucesso
- **NOT_VALID_TASK** – se a função for chamada antes da inicialização do escalonador
- **NO_TASK_DELAY** – se o valor de *ticks* passado para a função for 0.

Exemplo:

```
void Example_Task (void)
{
   /* Configuração da tarefa */

   /* Laço da tarefa */
   for (;;)
   {
       // Desiste do processador por 10 ticks do sistema
       (void)DelayTask(10);
   }
}
```

DelayTaskHMSM

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

Pode ser chamado de:	**Habilitado por:**
Tarefas	Sempre disponível

A função **DelayTaskHMSM** permite que uma tarefa desista do processador por um tempo determinado em horas, minutos, segundos e milisegundos. Um *delay* de 0 significa que a tarefa não irá desistir do processador. A tarefa volta a competir pelo processador tão logo seja acordada pelo sistema.

Argumentos:

- número de horas que a tarefa atualmente em execução irá desistir do processador. Valores válidos vão de 0 a 255
- número de minutos que a tarefa atualmente em execução irá desistir do processador. Valores válidos vão de 0 a 59
- número de segundos que a tarefa atualmente em execução irá desistir do processador. Valores válidos vão de 0 a 59
- número de milisegundos que a tarefa atualmente em execução irá desistir do processador. Valores válidos vão de 0 a 999. Note que a resolução deste argumento depende do tick do sistema. Por exemplo, se o *tick* do sistema for de 10ms, um argumento de 3ms será considerado 0.

Retorno:

- **OK** - se a função for executada com sucesso
- **NO_TASK_DELAY** – se o valor de tempo passado para a função for 0.
- **INVALID_TIME** – se um dos argumentos passados estiver fora dos valores válidos.

Exemplo:

```
void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
       // Desiste do processador por 15 minutos e 30 segundos
       (void)DelayTaskHMSM(0,15,30,0);
   }
}
```

BlockPriority

`INT8U BlockPriority(INT8U iPriority)`

Pode ser chamado de:	**Habilitado por:**
Tarefas	Sempre disponível

A função **BlockPriority** retira a tarefa associada a esta prioridade da concorrência pelo processador. Em alguns sistemas este procedimento é conhecido como suspensão da tarefa. O sistema irá procurar pela tarefa de maior prioridade pronta para executar se uma tarefa realizar seu próprio bloqueio.

Argumentos:

- prioridade da tarefa a ser bloqueada

Retorno:

- **OK** - se a tarefa for bloqueada com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção

Exemplo:

```
void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
       // Bloqueia a tarefa associada a prioridade 5
       (void)BlockPriority(5);
   }
}
```

UnBlockPriority

`INT8U UnBlockPriority(INT8U iPriority)`

Pode ser chamado de:	Habilitado por:
Tarefas	Sempre disponível

A função **UnBlockPriority** desbloqueia a tarefa associada a esta prioridade, permitindo que a mesma volta a concorrer pelo processador. Em alguns sistemas este procedimento é conhecido como resumo da tarefa. O sistema irá procurar pela tarefa de maior prioridade pronta para executar após executar este procedimento.

Argumentos:

- prioridade da tarefa a ser desbloqueada

Retorno:

- **OK** - se a tarefa for desbloqueada com sucesso

Exemplo:

```
void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
       // Desbloqueia a tarefa associada a prioridade 5
       (void)UnBlockPriority(5);
   }
}
```

BlockTask

`INT8U BlockTask(INT8U iTaskNumber)`

Pode ser chamado de:	Habilitado por:
Tarefas	Sempre disponível

A função **BlockTask** retira a tarefa especificada da concorrência pelo processador. Em alguns sistemas este procedimento é conhecido como suspensão da tarefa. O sistema irá procurar pela tarefa de maior prioridade pronta para executar se uma tarefa realizar seu próprio bloqueio.

Argumentos:

- ID da tarefa a ser bloqueada

Retorno:

- **OK** - se a tarefa for bloqueada com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção

Exemplo:

```
void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
       // Bloqueia a tarefa com ID igual a 5
       (void)BlockTask(5);
   }
}
```

UnBlockTask

`INT8U UnBlockTask(INT8U iTaskNumber)`

Pode ser chamado de:	**Habilitado por:**
Tarefas	Sempre disponível

A função **UnBlockTask** desbloqueia a tarefa especificada, permitindo que a mesma volta a concorrer pelo processador. Em alguns sistemas este procedimento é conhecido como resumo da tarefa. O sistema irá procurar pela tarefa de maior prioridade pronta para executar após executar este procedimento.

Argumentos:

- ID da tarefa a ser desbloqueada

Retorno:

- **OK** - se a tarefa for desbloqueada com sucesso

Exemplo:

```
void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
       // Desbloqueia a tarefa com ID igual a 5
       (void)UnBlockTask(5);
   }
}
```

BlockMultipleTask

`INT8U BlockMultipleTask(INT8U TaskStart, INT8U TaskNumber)`

Pode ser chamado de:	**Habilitado por:**
Tarefas	Sempre disponível

A função **BlockMultipleTask** retira as tarefas especificadas da concorrência pelo processador. A tarefa atualmente em execução é ignorada, caso esteja na lista de tarefas a serem bloqueadas. Esta função pode ser utilizada como um **mutex**, bloqueando as tarefas que podem vir a concorrer por um ou mais recursos.

Argumentos:

- ID da primeira tarefa a ser bloqueada
- número de tarefas a serem bloqueadas, a partir da tarefa definida em **TaskStart**

Retorno:

- **OK** - se as tarefas forem bloqueadas com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção

Exemplo:

```
void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
       // Bloqueia 4 tarefas, a partir da tarefa com ID 3
       (void)BlockMultipleTask(3,4);
   }
}
```

UnBlockMultipleTask

`INT8U UnBlockMultipleTask(INT8U TaskStart, INT8U TaskNumber)`

Pode ser chamado de:	**Habilitado por:**
Tarefas	Sempre disponível

A função **UnBlockMultipleTask** desbloqueia as tarefa especificadas, permitindo que as mesmas voltem a concorrer pelo processador.

Argumentos:

- ID da primeira tarefa a ser desbloqueada
- número de tarefas a serem desbloqueadas, a partir da tarefa definida em **TaskStart**

Retorno:

- **OK** - se as tarefas forem desbloqueadas com sucesso

Exemplo:

```
void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
       // Desbloqueia 4 tarefas, a partir da tarefa com ID 3
       (void)UnBlockMultipleTask(3,4);
   }
}
```

Semáforo

OSSemCreate

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

Pode ser chamado de:	Habilitado por:	Número de semáforos disponíveis definido por:
Inicialização do sistema e tarefas	BRTOS_SEM_EN (BRTOSConfig.h)	BRTOS_MAX_SEM (BRTOSConfig.h)

A função **OSSemCreate** cria e inicializa um semáforo. Um semáforo pode ser utilizado para:

- sincronizar tarefas com outras tarefas e/ou interrupções
- sinalizar a ocorrência de um evento

Argumentos:

- Valor inicial do semáforo (0 a 255)
- O endereço de um ponteiro do tipo "bloco de controle de semáforo" (**BRTOS_Sem**) que receberá o endereço do bloco de controle alocado para o semáforo criado

Retorno:

- **ALLOC_EVENT_OK** - se o semáforo foi criado com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção
- **NO_AVAILABLE_EVENT** – se não houver mais blocos de controle de semáforo disponíveis

Obs.: Um semáforo sempre deve ser criado antes de sua primeira utilização.

Exemplo:

```
BRTOS_Sem *TestSem;

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

	// Cria semáforo
	if (OSSemCreate(0,&TestSem) != ALLOC_EVENT_OK)
	{
		// Falha na alocação do semáforo
		// Trate este erro aqui !!!
	}
}
```

OSSemDelete

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

Pode ser chamado de:	Habilitado por:	Número de semáforos disponíveis definido por:
Inicialização do sistema e tarefas	BRTOS_SEM_EN (BRTOSConfig.h)	BRTOS_MAX_SEM (BRTOSConfig.h)

A função **OSSemDelete** desaloca um bloco de controle de semáforo, que poderá ser utilizado por outro semáforo.

Argumentos:

- O endereço de um ponteiro do tipo “bloco de controle de semáforo” (**BRTOS_Sem**) que receberá o endereço do bloco de controle alocado para o semáforo criado

Retorno:

- **DELETE_EVENT_OK** - se o semáforo foi apagado com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção

Obs.: Para apagar um semáforo deve-se garantir que o mesmo não está em uso por tarefas e interrupções.

Exemplo:

```
BRTOS_Sem *TestSem;

void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {

       // Apaga semáforo
       if (OSSemDelete(&TestSem) != DELETE_EVENT_OK)
       {
              // Falha ao apagar um semáforo
              // Função chamada dentro de uma interrupção
       }
   }
}
```

OSSemPend

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

Pode ser chamado de:	Habilitado por:	Número de semáforos disponíveis definido por:
Tarefas	BRTOS_SEM_EN (BRTOSConfig.h)	BRTOS_MAX_SEM (BRTOSConfig.h)

A função **OSSemPend** decrementa um semáforo se o seu contador for maior do que zero. No entanto, se o valor do contador do semáforo for zero, esta função irá colocar a tarefa que a chamou na lista de espera do semáforo. A tarefa irá esperar pela postagem de um semáforo até um tempo limite (ou por tempo ilimitado no caso de *timeout* igual a 0). O BRTOS irá acordar a tarefa de mais alta prioridade esperando pelo semáforo se houver postagem de semáforo antes do *timeout* expirar. Uma tarefa bloqueada pelo BRTOS pode ser adicionada a lista de prontos pelo semáforo, mas só será executada após seu desbloqueio.

Argumentos:

- Um ponteiro do tipo "bloco de controle de semáforo" (**BRTOS_Sem**) que contém o endereço do bloco de controle alocado para o semáforo em uso
- Tempo limite para receber a postagem de um semáforo

Retorno:

- **OK** - se o semáforo foi decrementado com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção
- **TIMEOUT** – se o *timeout* de pedido do semáforo expirar

Obs.: Um semáforo sempre deve ser criado antes de sua primeira utilização.

Exemplo:

```
BRTOS_Sem *TestSem;

void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função sem teste do retorno
      (void)OSSemPend(TestSem,15);

      // Uso da função com teste do retorno
      if (OSSemPend(TestSem,15) != OK)
      {
            // Falha no decremento do semáforo
            // Trate esta exceção aqui !!!
           // Por exemplo, saída por timeout
      }
   }
}
```

OSSemPost

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

Pode ser chamado de:	Habilitado por:	Número de semáforos disponíveis definido por:
Tarefas e interrupções	BRTOS_SEM_EN (BRTOSConfig.h)	BRTOS_MAX_SEM (BRTOSConfig.h)

A função **OSSemPost** incrementa o semáforo se não houver tarefas em sua lista de espera. Caso haja, esta função remove a tarefa de mais alta prioridade da sua lista de espera e adiciona esta tarefa a lista de tarefas prontas para execução. Em seguida o escalonador é chamado para verificar se a tarefa acordada é a tarefa de maior prioridade pronta para executar.

Argumentos:

- Um ponteiro do tipo "bloco de controle de semáforo" (**BRTOS_Sem**) que contém o endereço do bloco de controle alocado para o semáforo em uso

Retorno:

- **OK** - se o semáforo foi incrementado com sucesso
- **ERR_SEM_OVF** – se ocorrer estouro no contador do semáforo
- **NULL_EVENT_POINTER** – se o ponteiro do bloco de controle de semáforo passado pela função estiver vazio
- **ERR_EVENT_NO_CREATED** – se o bloco de controle passado pela função não foi alocado (criado) corretamente

Obs.: Um semáforo sempre deve ser criado antes de sua primeira utilização.

Exemplo:

```
BRTOS_Sem *TestSem;

void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função sem teste do retorno
      (void)OSSemPost(TestSem);

      // Uso da função com teste do retorno
      if (OSSemPost(TestSem) != OK)
      {
            // Falha no incremento do semáforo
            // Trate esta exceção aqui !!!
           // Por exemplo, estouro no contador do semáforo
      }
   }
}
```

Mutex

OSMutexCreate

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

Pode ser chamado de:	Habilitado por:	Número de mutexes disponíveis definido por:
Inicialização do sistema e tarefas	BRTOS_MUTEX_EN (BRTOSConfig.h)	BRTOS_MAX_MUTEX (BRTOSConfig.h)

A função **OSMutexCreate** cria e inicializa um mutex. Mutexes geralmente são utilizados para gerenciar recursos compartilhados.

Argumentos:

- O endereço de um ponteiro do tipo "bloco de controle de mutex" (**BRTOS_Mutex**) que receberá o endereço do bloco de controle alocado para o mutex criado
- A prioridade que será associada ao mutex. Esta prioridade deve ser sempre um nível maior do que a maior prioridade das tarefas que irão competir por um determinado recurso. Por exemplo, imagine um caso onde 3 tarefas com prioridades 4, 10 e 12 compartilham um recurso. Neste caso o valor apropriado para a prioridade do mutex é 13. A tarefa que recebe o recurso passa a prioridade 13, não sendo interrompida pelas tarefas que competem pelo mesmo recurso. A tarefa volta a sua prioridade original quando liberar este recurso.

Retorno:

- **ALLOC_EVENT_OK** - se o mutex foi criado com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção
- **NO_AVAILABLE_EVENT** – se não houver mais blocos de controle de mutex disponíveis
- **BUSY_PRIORITY** – se a prioridade especificada já estiver em uso

Obs.: Um mutex sempre deve ser criado antes de sua primeira utilização.

Exemplo:

```
BRTOS_Mutex *TestMutex;

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

	// Cria mutex com prioridade 13
	if (OSMutexCreate(&TestMutex,13) != ALLOC_EVENT_OK)
	{
		// Falha na alocação do mutex
		// Trate este erro aqui !!!
	}
}
```

OSMutexDelete

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

Pode ser chamado de:	Habilitado por:	Número de mutexes disponíveis definido por:
Inicialização do sistema e tarefas	BRTOS_MUTEX_EN (BRTOSConfig.h)	BRTOS_MAX_MUTEX (BRTOSConfig.h)

A função **OSMutexDelete** desaloca um bloco de controle de mutex, que poderá ser utilizado por outro mutex.

Argumentos:

- O endereço de um ponteiro do tipo "bloco de controle de mutex" (**BRTOS_Mutex**) que receberá o endereço do bloco de controle alocado para o mutex criado

Retorno:

- **DELETE_EVENT_OK** - se o mutex foi apagado com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção

Obs.: Para apagar um mutex deve-se garantir que o mesmo não está em uso por tarefas e interrupções.

Exemplo:

```
BRTOS_Mutex *TestMutex;

void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {

       // Apaga mutex
       if (OSMutexDelete(&TestMutex) != DELETE_EVENT_OK)
       {
              // Falha ao apagar um mutex
              // Função chamada dentro de uma interrupção
       }
   }
}
```

OSMutexAcquire

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

Pode ser chamado de:	Habilitado por:	Número de mutexes disponíveis definido por:
Tarefas	BRTOS_MUTEX_EN (BRTOSConfig.h)	BRTOS_MAX_MUTEX (BRTOSConfig.h)

A função **OSMutexAcquire** inicialmente verifica se o recurso a ser adquirido está disponível. Se o recurso estiver disponível, este é imediatamente colocado como ocupado. A tarefa que chamou a função recebe a propriedade temporária do recurso (ou seja, somente esta tarefa poderá liberá-lo). Ainda, a prioridade da tarefa que adquiriu o recurso para a ser a prioridade do mutex. No entanto, se o recurso estiver ocupado, esta função irá colocar a tarefa que a chamou na lista de espera do mutex. A tarefa irá esperar indefinidamente pela liberação do mutex. O BRTOS irá acordar a tarefa de mais alta prioridade esperando pelo mutex se houver liberação do recurso. Uma tarefa bloqueada pelo BRTOS pode ser adicionada a lista de prontos pelo mutex, mas só será executada após seu desbloqueio.

Argumentos:

- Um ponteiro do tipo “bloco de controle de mutex” (**BRTOS_Mutex**) que contém o endereço do bloco de controle alocado para o mutex em uso

Retorno:

- **OK** - se o mutex foi adquirido com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção

Obs.: Um mutex sempre deve ser criado antes de sua primeira utilização.

Exemplo:

```
BRTOS_Mutex *TestMutex;

void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
      // Adquire um recurso
      (void)OSMutexAcquire(TestMutex);

      // Realiza as operações necessárias com o recurso adquirido
   }
}
```

OSMutexRelease

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

Pode ser chamado de:	Habilitado por:	Número de mutexes disponíveis definido por:
Tarefas e interrupções	BRTOS_MUTEX_EN (BRTOSConfig.h)	BRTOS_MAX_MUTEX (BRTOSConfig.h)

A função **OSMutexRelease** libera um recurso compartilhado para ser utilizado por outras tarefas. Caso haja tarefas na lista de espera do recurso, esta função retorna a prioridade da tarefa que o possuía para seu valor original, liberando o recurso. Em seguida, remove a tarefa de mais alta prioridade da sua lista de espera e adiciona esta tarefa a lista de tarefas prontas para execução. O escalonador é chamado para verificar se a tarefa acordada é a tarefa de maior prioridade pronta para executar.

Argumentos:

- Um ponteiro do tipo "bloco de controle de mutex" (**BRTOS_Mutex**) que contém o endereço do bloco de controle alocado para o mutex em uso

Retorno:

- **OK** - se o recurso foi liberado com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção
- **ERR_EVENT_OWNER** – se a tarefa que está tentando liberar o recurso não for sua proprietária temporária

Obs.: Um mutex sempre deve ser criado antes de sua primeira utilização.

Exemplo:

```
BRTOS_Mutex *TestMutex;

void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função sem teste do retorno
      (void)OSMutexRelease(TestMutex);

      // Uso da função com teste do retorno
      if (OSMutexRelease(TestMutex) != OK)
      {
            // Falha ao liberar o recurso
            // Trate esta exceção aqui !!!
           // Por exemplo, a tarefa não possui o mutex (não é a dona temporária)
      }
   }
}
```

Em algumas tarefas é necessário que vários recursos sejam adquiridos para realizar um determinado procedimento. Nestes casos deve-se cuidar a ordem de aquisição dos recursos para realizar a liberação dos mesmos de maneira correta. **No BRTOS os recursos devem ser liberados na ordem inversa de sua aquisição**. A inversão na ordem de liberação dos recursos pode levar a inconsistências no sistema.

Exemplo:

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


void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {
      // Adquire o recurso 1
      (void)OSMutexAcquire(TestMutex1);

      // Adquire o recurso 2
      (void)OSMutexAcquire(TestMutex2);

      // Adquire o recurso 3
      (void)OSMutexAcquire(TestMutex3);


      ////////////////////////////////////////////////////////////
      // Realiza as operações necessárias com o recurso adquirido  //
      ////////////////////////////////////////////////////////////


      // Libera o recurso 3
      (void)OSMutexRelease(TestMutex3);

      // Libera o recurso 2
      (void)OSMutexRelease(TestMutex2);

      // Libera o recurso 1
      (void)OSMutexRelease(TestMutex3);
   }
}
```

Caixa de mensagem

OSMboxCreate

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

Pode ser chamado de:	Habilitado por:	Número de caixas de mensagem disponíveis definido por:
Inicialização do sistema e tarefas	BRTOS_MBOX_EN (BRTOSConfig.h)	BRTOS_MAX_MBOX (BRTOSConfig.h)

A função **OSMboxCreate** cria e inicializa uma caixa de mensagem. Um caixa de mensagem pode ser utilizado para passar valores / ponteiros de interrupções para tarefas ou de tarefa para tarefa.

Argumentos:

- O endereço de um ponteiro do tipo "bloco de controle de caixa de mensagem" (**BRTOS_Mbox**) que receberá o endereço do bloco de controle alocado para a caixa de mensagem criada
- O ponteiro de uma mensagem a ser passada. Caso não seja desejado uma mensagem inicial, deve-se passar NULL.

Retorno:

- **ALLOC_EVENT_OK** - se a caixa de mensagem foi criada com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção
- **NO_AVAILABLE_EVENT** – se não houver mais blocos de controle de caixa de mensagem disponíveis

Obs.: Uma caixa de mensagem sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
BRTOS_Mbox *TestMbox;

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

	// Cria caixa de mensagem
	if (OSMboxCreate(&TestMbox,NULL) != ALLOC_EVENT_OK)
	{
		// Falha na alocação da caixa de mensagem
		// Trate este erro aqui !!!
	}
}
```

OSMboxDelete

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

Pode ser chamado de:	Habilitado por:	Número de caixas de mensagem disponíveis definido por:
Inicialização do sistema e tarefas	BRTOS_MBOX_EN (BRTOSConfig.h)	BRTOS_MAX_MBOX (BRTOSConfig.h)

A função **OSMboxDelete** desaloca um bloco de controle de caixa de mensagem, que poderá ser utilizado por outra caixa de mensagem.

Argumentos:

- O endereço de um ponteiro do tipo "bloco de controle de caixa de mensagem" (**BRTOS_Mbox**) que receberá o endereço do bloco de controle alocado para a caixa de mensagem criada

Retorno:

- **DELETE_EVENT_OK** - se a caixa de mensagem foi apagada com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção

Obs.: Para apagar uma caixa de mensagem deve-se garantir que o mesma não está em uso por tarefas e interrupções.

Exemplo:

```
BRTOS_Mbox *TestMbox;

void Example_Task (void)
{
   /* Configuração da tarefa */


   /* Laço da tarefa */
   for (;;)
   {

       // Apaga caixa de mensagem
       if (OSMboxDelete(&TestMbox) != DELETE_EVENT_OK)
       {
              // Falha ao apagar uma caixa de mensagem
              // Função chamada dentro de uma interrupção
       }
   }
}
```

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)

A função **OSMboxPend** verifica se há uma mensagem disponível. Se sim, retorna a mensagem através de um ponteiro (Mail). No entanto, se não houver mensagem esta função irá colocar a tarefa que a chamou na lista de espera da caixa de mensagem. A tarefa irá esperar pela postagem de uma mensagem até um tempo limite (ou por tempo ilimitado no caso de *timeout* igual a 0). O BRTOS irá acordar a tarefa de mais alta prioridade esperando pela mensagem se houver postagem de uma mensagem antes do *timeout* expirar. Uma tarefa bloqueada pelo BRTOS pode ser adicionada a lista de prontos pela caixa de mensagem, mas só será executada após seu desbloqueio.

Argumentos:

- Um ponteiro do tipo "bloco de controle de caixa de mensagem" (**BRTOS_Mbox**) que contém o endereço do bloco de controle alocado para a caixa de mensagem em uso
- Endereço do ponteiro que receberá a mensagem
- Tempo limite para receber a postagem da mensagem

Retorno:

- **OK** - se a mensagem foi recebida com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção
- **TIMEOUT** – se o *timeout* de pedido da mensagem expirar

Obs.: Uma caixa de mensagem sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
BRTOS_Mbox *TestMbox;

void Example_Task (void)
{
   /* Configuração da tarefa */
   INT16U *teste;

   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função com teste do retorno
      if (OSMboxPend(TestMbox,(void **)&teste,0) != OK)
      {
            // Falha no recebimento da mensagem
            // Trate esta exceção aqui !!!
           // Por exemplo, saída por timeout
      }
   }
}
```

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)

A função **OSMboxPost** envia uma mensagem para a caixa de mensagem especificada. Se houver uma ou mais tarefas a espera da mensagem, esta função remove a tarefa de mais alta prioridade da sua lista de espera e adiciona esta tarefa a lista de tarefas prontas para execução. Em seguida o escalonador é chamado para verificar se a tarefa acordada é a tarefa de maior prioridade pronta para executar.

Argumentos:

- Um ponteiro do tipo "bloco de controle de caixa de mensagem" (**BRTOS_Mbox**) que contém o endereço do bloco de controle alocado para a caixa de mensagem em uso
- O ponteiro da mensagem enviada

Retorno:

- **OK** - se a mensagem foi enviada com sucesso
- **NULL_EVENT_POINTER** – se o ponteiro do bloco de controle de caixa de mensagem passado pela função estiver vazio
- **ERR_EVENT_NO_CREATED** – se o bloco de controle passado pela função não foi alocado (criado) corretamente

Obs.: Uma caixa de mensagem sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
BRTOS_Mbox *TestMbox;

void Example_Task (void)
{
   /* Configuração da tarefa */
   INT16U teste = 0;

   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função com teste do retorno
      (void)OSMboxPost(TestMbox,(void *)&teste);
   }
}
```

Exemplo de passagem de ponteiro por caixa de mensagem:

```
BRTOS_Mbox *TestMbox;



void Example_Task_1 (void)
{
   /* Configuração da tarefa */
   INT16U *recebe_pont;
   INT16U recebe;

   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função com teste do retorno
      if (OSMboxPend(TestMbox,(void **)&recebe_pont,0) != OK)
      {
            // Falha no recebimento da mensagem
            // Trate esta exceção aqui !!!
           // Por exemplo, saída por timeout
      }
      // Copia o valor passado pela mensagem, caso necessário
      recebe = *recebe_pont;
   }
}


void Example_Task_2 (void)
{
   /* Configuração da tarefa */
   INT16U envia = 0;

   /* Laço da tarefa */
   for (;;)
   {
      envia++;
      // Uso da função com teste do retorno
      (void)OSMboxPost(TestMbox,(void *)&envia);
   }
}
```

Exemplo de passagem de ponteiro de uma string por caixa de mensagem:

```
BRTOS_Mbox *TestMbox;



void Example_Task_1 (void)
{
   /* Configuração da tarefa */
   CHAR8 *recebe_pont;

   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função com teste do retorno
      if (OSMboxPend(TestMbox,(void **)&recebe_pont,0) != OK)
      {
            // Falha no recebimento da mensagem
            // Trate esta exceção aqui !!!
           // Por exemplo, saída por timeout
      }

      // Utiliza a mensagem recebida
      while(*recebe_pont)
      {
        Serial_Envia_Caracter(*recebe_pont);
        recebe_pont++;
      }
      Serial_Envia_Caracter(CR);
      Serial_Envia_Caracter(LF);
   }
}


void Example_Task_2 (void)
{
   /* Configuração da tarefa */
   CHAR8 *envia = "Teste de envio";

   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função com teste do retorno
      (void)OSMboxPost(TestMbox,(void *)envia);
   }
}
```

Exemplo de passagem de valor por caixa de mensagem:

```
BRTOS_Mbox *TestMbox;



void Example_Task_1 (void)
{
   /* Configuração da tarefa */
   INT32U *recebe_pont;
   INT32U recebe;

   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função com teste do retorno
      if (OSMboxPend(TestMbox,(void **)&recebe_pont,0) != OK)
      {
            // Falha no recebimento da mensagem
            // Trate esta exceção aqui !!!
           // Por exemplo, saída por timeout
      }
      // Copia o valor passado pela mensagem, caso necessário
      recebe = (INT32U)recebe_pont;
   }
}


void Example_Task_2 (void)
{
   /* Configuração da tarefa */
   INT32U *envia;
   INT32U inc = 0;

   /* Laço da tarefa */
   for (;;)
   {
      inc++;
      envia = (INT32U*)inc;
      // Uso da função com teste do retorno
      (void)OSMboxPost(TestMbox,(void *)envia);
   }
}
```

Filas

OSQueueXXCreate

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Inicialização do sistema e tarefas	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSQueueXXCreate** cria e inicializa uma fila, onde XX especifica o tamanho de uma mensagem na fila (8, 16 e 32 bits). Filas podem ser vistas como vetores de caixas de mensagem, sendo utilizadas para passar valores / ponteiros de interrupções para tarefas ou de tarefa para tarefa.

Argumentos:

- O ponteiro de uma fila do sistema (**OS_QUEUE_8**, **OS_QUEUE_16** ou **OS_QUEUE_32**)
- A quantidade máxima de dados que poderá ser armazenada na fila
- O endereço de um ponteiro do tipo "bloco de controle de fila" (**BRTOS_Queue**) que receberá o endereço do bloco de controle alocado para a fila criada

Retorno:

- **ALLOC_EVENT_OK** - se a fila foi criada com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção
- **NO_MEMORY** – se o tamanho solicitado para a fila for maior do que o espaço disponível no HEAP de filas do sistema
- **NO_AVAILABLE_EVENT** – se não houver mais blocos de controle de fila disponíveis

Obs.: Uma fila sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
BRTOS_Queue *TestQueue;
OS_QUEUE_16 QueueBuffer;

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

	// Cria fila com 128 valores de 16 bits (256 bytes)
	if (OSQueue16Create(&QueueBuffer, 128, &TestQueue) != ALLOC_EVENT_OK)
	{
		// Falha na alocação da fila
		// Trate este erro aqui !!!
	}
}
```

OSQueuePend

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Tarefas	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSQueuePend** verifica se há dados disponíveis na fila. Se sim, retorna o primeiro dado disponível na fila. No entanto, se não houver dados disponíveis, esta função irá colocar a tarefa que a chamou na lista de espera da fila. A tarefa irá esperar pela postagem de pelo menos um dado na fila até um tempo limite (ou por tempo ilimitado no caso de *timeout* igual a 0). O BRTOS irá acordar a tarefa de mais alta prioridade esperando pelo dado se houver postagem antes do *timeout* expirar. Uma tarefa bloqueada pelo BRTOS pode ser adicionada a lista de prontos pela fila, mas só será executada após seu desbloqueio.

Argumentos:

- Um ponteiro do tipo "bloco de controle de fila" (**BRTOS_Queue**) que contém o endereço do bloco de controle alocado para a fila em uso
- Endereço da variável que receberá o valor da fila
- Tempo limite para receber a postagem da fila

Retorno:

- **READ_BUFFER_OK** - se o valor da fila foi recebido com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção
- **TIMEOUT** – se o *timeout* de pedido da fila expirar

Obs 1.: Uma fila sempre deve ser criada antes de sua primeira utilização.
Obs 2.: A versão 1.7x do BRTOS somente suporta as funções Pend e Post para filas de 8 bits e filas dinâmicas.

Exemplo:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Configuração da tarefa */
   INT8U *teste;

   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função com teste do retorno
      if(!OSQueuePend(TestQueue, &teste, 0))
      {
        switch(teste)
        {
           ....
        }
   }
}
```

OSQueuePost

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Tarefas e interrupções	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSQueuePost** envia um dado para a fila especificada. Se houver uma ou mais tarefas a espera de dados na fila, esta função remove a tarefa de mais alta prioridade da sua lista de espera e adiciona esta tarefa a lista de tarefas prontas para execução. Em seguida o escalonador é chamado para verificar se a tarefa acordada é a tarefa de maior prioridade pronta para executar.

Argumentos:

- Um ponteiro do tipo "bloco de controle de fila" (**BRTOS_Queue**) que contém o endereço do bloco de controle alocado para a fila em uso
- O dado a ser enviado para a fila

Retorno:

- **WRITE_BUFFER_OK** - se o dado foi enviado com sucesso para a fila
- **BUFFER_UNDERRUN** – se não houver mais espaço na fila para alocar dados
- **NULL_EVENT_POINTER** – se o ponteiro do bloco de controle de fila passado pela função estiver vazio
- **ERR_EVENT_NO_CREATED** – se o bloco de controle passado pela função não foi alocado (criado) corretamente

Obs 1.: Uma fila sempre deve ser criada antes de sua primeira utilização.
Obs 2.: A versão 1.7x do BRTOS somente suporta as funções Pend e Post para filas de 8 bits e filas dinâmicas.

Exemplo:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Configuração da tarefa */
   INT16U teste = 0;

   /* Laço da tarefa */
   for (;;)
   {
      teste++;
      // Uso da função com teste do retorno
      if (OSQueuePost(TestQueue, teste) == BUFFER_UNDERRUN)
      {
         // Problema: Estouro da fila
         OSCleanQueue(TestQueue);
      }
   }
}
```

OSCleanQueue

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Tarefas e interrupções	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSCleanQueue** reinicia os ponteiros (entrada e saída) e zera o contador de entradas da fila. Esta função pode ser utilizada quando ocorrem problemas de *overflow* nas filas.

Argumentos:

- Um ponteiro do tipo "bloco de controle de fila" (**BRTOS_Queue**) que contém o endereço do bloco de controle alocado para a fila em uso

Retorno:

- **CLEAN_BUFFER_OK** - se a fila foi reiniciada com sucesso
- **NULL_EVENT_POINTER** – se o ponteiro do bloco de controle de fila passado pela função estiver vazio
- **ERR_EVENT_NO_CREATED** – se o bloco de controle passado pela função não foi alocado (criado) corretamente

Obs.: Uma fila sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Configuração da tarefa */

   /* Laço da tarefa */
   for (;;)
   {
      // Reinicia a fila TesteQueue
      OSCleanQueue(TestQueue);
   }
}
```

OSWQueueXX

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Tarefas e interrupções	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSWQueueXX** escreve um dado na fila sem utilizar o serviço de eventos do sistema, onde XX especifica o tamanho de uma mensagem na fila (8, 16 e 32 bits).

Argumentos:

- Um ponteiro do tipo "fila do sistema" (**OS_QUEUE_XX**) que contém o endereço da estrutura de fila alocada para a fila em uso
- O dado a ser enviado para a fila

Retorno:

- **WRITE_BUFFER_OK** - se o dado foi enviado com sucesso para a fila
- **BUFFER_UNDERRUN** – se não houver mais espaço na fila para alocar dados

Obs.: Uma fila sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
OS_QUEUE_16 QueueBuffer;

void Example_Task (void)
{
   /* Configuração da tarefa */
   INT16U teste = 0;

   /* Laço da tarefa */
   for (;;)
   {
      teste++;
      // Uso da função com teste do retorno
      if (OSWQueue16(QueueBuffer, teste) == BUFFER_UNDERRUN)
      {
         // Problema: Estouro da fila
         OSCleanQueue(TestQueue);
      }
   }
}
```

OSRQueueXX

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Tarefas e interrupções	BRTOS_QUEUE_XX_EN (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSRQueueXX** recebe (copia) um dado da fila especificada sem utilizar o serviço de eventos do sistema, onde XX especifica o tamanho de uma mensagem na fila (8, 16 e 32 bits).

Argumentos:

- Um ponteiro do tipo "fila do sistema" (**OS_QUEUE_XX**) que contém o endereço da estrutura de fila alocada para a fila em uso
- Endereço da variável que receberá o valor da fila

Retorno:

- **READ_BUFFER_OK** - se o valor da fila foi recebido com sucesso
- **NO_ENTRY_AVAILABLE** – se não houver dados na fila

Obs.: Uma fila sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
OS_QUEUE_16 QueueBuffer;

void Example_Task (void)
{
   /* Configuração da tarefa */
   INT16U teste = 0;

   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função com teste do retorno
      if (OSRQueue16(QueueBuffer, &teste) == READ_BUFFER_OK)
      {
        switch(teste)
        {
           ....
        }
      }
   }
}
```

Filas Dinâmicas

OSDQueueCreate

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Inicialização do sistema e tarefas	BRTOS_DYNAMIC_QUEUE_ENABLED (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSDQueueCreate** cria e inicializa uma fila dinâmica. Este tipo de fila possui duas diferenças básicas das filas tradicionais do BRTOS: 1. podem ser criadas e excluídas; 2. O tamanho do tipo de dado que a fila envia pode ser configurado em sua inicialização. Assim, é possível, por exemplo, criar uma fila de uma estrutura de dados com diversos bytes.

Argumentos:

- A quantidade máxima de dados que poderá ser armazenada na fila
- O tamanho em bytes do tipo de dado que será armazenado na fila
- O endereço de um ponteiro do tipo "bloco de controle de fila" (**BRTOS_Queue**) que receberá o endereço do bloco de controle alocado para a fila criada

Retorno:

- **ALLOC_EVENT_OK** - se a fila foi criada com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção
- **NO_AVAILABLE_MEMORY** – se o tamanho solicitado para a fila for maior do que o espaço disponível no HEAP de filas do sistema
- **NO_AVAILABLE_EVENT** – se não houver mais blocos de controle de fila disponíveis
- **INVALID_PARAMETERS** – se o quantidade de dados a ser armazenado na fila e/ou o tamanho em bytes do tipo de dado da fila for menor ou igual a zero.

Obs.: Uma fila sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
BRTOS_Queue *TestQueue;

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

	// Cria fila com 128 valores do tamanho de um ponteiro void
	if (OSDQueueCreate(128, sizeof(void*), &TestQueue) != ALLOC_EVENT_OK)
	{
		// Falha na alocação da fila
		// Trate este erro aqui !!!
	}
}
```

OSDQueueDelete

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Tarefas	BRTOS_DYNAMIC_QUEUE_ENABLED (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSDQueueDelete** desaloca a memória utilizada pela fila e exclui seus dados do bloco de controle de filas.

Argumentos:

- O endereço de um ponteiro do tipo “bloco de controle de fila” (**BRTOS_Queue**) que contém o endereço do bloco de controle alocado para a fila em uso

Retorno:

- **DELETE_EVENT_OK** - se a fila foi excluída com sucesso
- **NULL_EVENT_POINTER** – se o ponteiro do bloco de controle de fila passado pela função estiver vazio
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção

Obs.: Uma fila sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Configuração da tarefa */
   // Cria fila com 128 valores do tamanho de um ponteiro void
   if (OSDQueueCreate(128, sizeof(void*), &TestQueue) != ALLOC_EVENT_OK)
   {
       // Falha na alocação da fila
       // Trate este erro aqui !!!
   }

   /* Laço da tarefa */
   for (;;)
   {
      // Exclue a fila TesteQueue
      OSDQueueDelete(&TestQueue);
   }
}
```

OSDQueuePend

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Tarefas	BRTOS_DYNAMIC_QUEUE_ENABLED (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSDQueuePend** verifica se há dados disponíveis na fila. Se sim, retorna o primeiro dado disponível na fila. No entanto, se não houver dados disponíveis, esta função irá colocar a tarefa que a chamou na lista de espera da fila. A tarefa irá esperar pela postagem de pelo menos um dado na fila até um tempo limite (ou por tempo ilimitado no caso de *timeout* igual a 0). O BRTOS irá acordar a tarefa de mais alta prioridade esperando pelo dado se houver postagem antes do *timeout* expirar. Uma tarefa bloqueada pelo BRTOS pode ser adicionada a lista de prontos pela fila, mas só será executada após seu desbloqueio.

Argumentos:

- Um ponteiro do tipo “bloco de controle de fila” (**BRTOS_Queue**) que contém o endereço do bloco de controle alocado para a fila em uso
- Endereço da variável que receberá o valor da fila (se necessário utilize *casting* para void*)
- Tempo limite para receber a postagem da fila

Retorno:

- **READ_BUFFER_OK** - se o valor da fila foi recebido com sucesso
- **IRQ_PEND_ERR** – se a função for chamada no código de tratamento de uma interrupção
- **TIMEOUT** – se o *timeout* de pedido da fila expirar

Obs .: Uma fila sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Configuração da tarefa */
   INT32U *teste;

   /* Laço da tarefa */
   for (;;)
   {
      // Uso da função com teste do retorno
      if(!OSDQueuePend(TestQueue, (void*)&teste, 0))
      {
        switch(teste)
        {
           ....
        }
   }
}
```

OSDQueuePost

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Tarefas e interrupções	BRTOS_DYNAMIC_QUEUE_ENABLED (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSDQueuePost** envia um dado para a fila especificada. Se houver uma ou mais tarefas a espera de dados na fila, esta função remove a tarefa de mais alta prioridade da sua lista de espera e adiciona esta tarefa a lista de tarefas prontas para execução. Em seguida o escalonador é chamado para verificar se a tarefa acordada é a tarefa de maior prioridade pronta para executar.

Argumentos:

- Um ponteiro do tipo “bloco de controle de fila” (**BRTOS_Queue**) que contém o endereço do bloco de controle alocado para a fila em uso
- Endereço do dado a ser enviado para a fila

Retorno:

- **WRITE_BUFFER_OK** - se o dado foi enviado com sucesso para a fila
- **BUFFER_UNDERRUN** – se não houver mais espaço na fila para alocar dados
- **NULL_EVENT_POINTER** – se o ponteiro do bloco de controle de fila passado pela função estiver vazio
- **ERR_EVENT_NO_CREATED** – se o bloco de controle passado pela função não foi alocado (criado) corretamente

Obs .: Uma fila sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Configuração da tarefa */
   INT16U teste = 0;

   /* Laço da tarefa */
   for (;;)
   {
      teste++;
      // Uso da função com teste do retorno
      if (OSDQueuePost(TestQueue, (void*)&teste) == BUFFER_UNDERRUN)
      {
         // Problema: Estouro da fila
         OSCleanQueue(TestQueue);
      }
   }
}
```

OSDQueueClean

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

Pode ser chamado de:	Habilitado por:	Número de filas disponíveis definido por:
Tarefas e interrupções	BRTOS_DYNAMIC_QUEUE_ENABLED (BRTOSConfig.h)	BRTOS_MAX_QUEUE (BRTOSConfig.h)

A função **OSDCleanQueue** reinicia os ponteiros (entrada e saída) e zera o contador de entradas da fila. Esta função pode ser utilizada quando ocorrem problemas de *overflow* nas filas.

Argumentos:

- Um ponteiro do tipo "bloco de controle de fila" (**BRTOS_Queue**) que contém o endereço do bloco de controle alocado para a fila em uso

Retorno:

- **CLEAN_BUFFER_OK** - se a fila foi reiniciada com sucesso
- **NULL_EVENT_POINTER** – se o ponteiro do bloco de controle de fila passado pela função estiver vazio
- **ERR_EVENT_NO_CREATED** – se o bloco de controle passado pela função não foi alocado (criado) corretamente

Obs.: Uma fila sempre deve ser criada antes de sua primeira utilização.

Exemplo:

```
BRTOS_Queue *TestQueue;

void Example_Task (void)
{
   /* Configuração da tarefa */

   /* Laço da tarefa */
   for (;;)
   {
      // Reinicia a fila TesteQueue
      OSDCleanQueue(TestQueue);
   }
}
```

Exemplo de inicialização do BRTOS com duas tarefas

Arquivo main.c:

```
#include "tasks.h"

// Parâmetros
Porta_IO Dados_Porta_IO;

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

      // Determina a configuração de um pino de I/O dentro da tarefa
      Dados_Porta_IO.direcao = 3;
      Dados_Porta_IO.dados = 0;


       // Instala uma tarefa com parâmetros
      if(InstallTask(&System_Time,"System Time",150,31,(void*)&ParamTest) != OK)
      {
        // Não deveria entrar aqui. Trate esta exceção !!!
        while(1){};
      };


       // Instala uma segunda tarefa sem parâmetros
      if(InstallTask(&Test_Task,"Tarefa de teste",100,10,NULL) != OK)
      {
        // Não deveria entrar aqui. Trate esta exceção !!!
        while(1){};
      };


      // Inicia o escalonador
      if(BRTOSStart() != OK)
      {
        // Não deveria entrar aqui !!!
        // Deveria ter saltado para a tarefa de maior prioridade.
        while(1){};
      };


      for(;;)
      {
         // Também não deveria entrar aqui !!!
      }
}
```

Arquivo tasks.h:

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

typedef struct _IO Porta_IO;

// Protótipo das tarefas
void System_Time(void *parameters);
void Test_Task(void *parameters);
```

Arquivo tasks.c:

```
#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;   // ignora os parâmetros


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