Reference Manual

API Services

Rubus® OS

Real-Time Operating System

for

Dependable Real-Time Systems

Part 2

Arcticus Systems

Document version

Version 4.0 October, 2009

Arcticus Systems AB
Datavägen 9A
Box 530
175 26 JÄRFÄLLA
SWEDEN
Phone +46 8 580 31100
info@arcticus-systems.com
www.arcticus-systems.com

Table of Contents

1 Introduction

The *Rubus® OS* Reference Manual describes the Application Program Interface specification for the C-Language. The manual is divided into two parts: Part 1 describing the general concepts and a functional overview of Rubus OS. Part 2 contains detailed information about the services provided by Rubus OS.

This document describes the run-time services for the C-Language provided by Rubus OS categorised as follows:

- The *Hardware Adaptation Layer Services* (HAL) contains services to adapt Rubus OS for the different supported hardware platforms.
- The *Basic Services* contains services that are common for the Red, Blue and Green Kernel. These services include, clock and timer services, communication and analyse services.
- The *Green Run-Time Services*, external event triggered execution (interrupts). The Green Kernel provides the Green Run-Time Services.
- The *Red Run-Time Services*, time triggered execution, *mainly* to be used for applications that have *hard real-time requirements*, i.e. meeting their deadlines is critical to the operation. The Red Kernel provides the Red Run-Time Services.
- The *Blue Run-Time Services*, internal event triggered execution, *mainly* to be used for applications that have *soft real-time requirements*, i.e. meeting their deadlines is not critical to the operation. The Blue Kernel provides the Blue Run-Time Services.

2 HAL Services

Basic Timer Control

halBsTimerIntrEntry - Notify Basic Timer

Synopsis

```
#include <basic/bs_basic.h>
void halBsTimerIntrEntry (void);
```

Descriptions

The *halBsTimerIntrEntry()* function increments the Basic Clock and the Red Schedule Timer. The function checks if a running Red thread has meet its deadline and in such case call the function *redError()*. If there is a Red thread having release time equal to the value of the Red Timer, a context switch to the Red Kernel mode is done. Switch to the Red Stack, makes a stack frame and calls the released thread. At return from this function the stack frame is removed. If all stack frames are removed from the Red Stack, the stack of the currently interrupted Blue thread is resumed and the execution of the interrupted Blue thread is continued (exit the context of the Red kernel).

Before completion of the function *halBsTimerIntrEntry()*, the state of the Blue Kernel is checked. If pre-emption of a Blue thread shall be done before the currently interrupted Blue thread is resumed, the Blue Kernel is resumed managing execution of Blue threads due to the scheduling policy defined. In such a case the function *halBsTimerIntrEntry()* is completed when the Blue Kernel has resumed the currently interrupted Blue thread.

In a Red system (no Blue threads), the background loop is continued after completion of the function *halBsTimerIntrEntry()*.

Returns

No value is returned.

Errors

During execution of the functions *halBsTimerIntrEntry()* errors can be detected causing call of the function *redError()*.

Example

```
#include <basic/bs_basic.h>

interrupt void halBsTimerISR (void)
{
  halBsTimerIntrEntry();
}
```

Notes

If an error is detected by the functions *halBsTimerIntrEntry()*, the function *redError()* is called with the error as an argument. If *redError()* is completed the execution is resumed, but this may cause undesired system behaviour. The recommendation is to notify the error and restart the system.

Some target implementation of Rubus OS requires that the functions *halBsTimerIntrEntry()* shall be invoked directly from the Interrupt Vector. In such a case a user supplied service can be required to handle reload of the timer value named *halBsTimerReload()*. Please read appropriate target appendix.

References

halBsTimerDisable, halBsTimerEnable, redError.

halBsTimerDisable – Disable Basic Timer

Synopsis

```
#include <basic/bs_basic.h>
void halBsTimerDisable (void);
```

Descriptions

The *halBsTimerDisable* function disable the Basic Timer, normally by disable the Basic Timer Interrupt without stopping the timer.

This function shall be supplied by the user and to be adapted to the user's environment.

Returns

No value is returned.

Errors

Example

```
#include  <basic/bs_basic.h>
/* Infineon XC166 implementation */
void halBsTimerDisable(void)
{
  T14IE  = 0;
}
```

Notes

The user supplies these functions.

References

halBsIntrLock,halBsIntrUnlock, halBsTimerEnable

halBsTimerEnable – Enable Basic Timer

Synopsis

```
#include <basic/bs_basic.h>
void halBsTimerEnable (void);
```

Descriptions

The *halBsTimerEnable* function enables the Basic Timer.

This function shall be supplied by the user and to be adapted to the user's environment.

Returns

No value is returned.

Errors

Example

```
#include  <basic/bs_basic.h>
/* Infineon XC166 implementation */

void halBsTimerEnable(void)
{
  T14IE  = 1;
}
```

Notes

The user supplies these functions.

References

halBsIntrLock,halBsIntrUnlock,halBsTimerDisable.htm

halBsTimerStart – Start Basic Timer

Synopsis

```
#include <basic/bs_basic.h>
void halBsTimerStart (void);
```

Descriptions

The *halBsTimerStart()* function initialise the Basic Timer and enables the Basic Timer interrupt. The timers for execution time measurement and logging are also started.

This function shall be supplied by the user and to be adapted to the user's environment.

Returns

No value is returned.

Errors

Example

```
#include  <basic/bs_basic.h>
/* Infineon XC166 implementation */

void halBsTimerStart(void)
{
  rtcInit();

  T4    = 0;
  T4CON = 0x0000 | TX_PRE_FACTOR;
  T4R   = 1;

  T3    = 0;
  T3CON = 0x0000 | TX_PRE_FACTOR;
  T3R   = 1;

  IEN = 1;
  rtcTimerUnlock();
}
```

Notes

The function is called by the function *bsRubusStart()*.

The user supplies these functions.

References

bsRubusStart, halBsExecTimeStart, halBsExecTimeStop, halBsTimeLog.

Execution Time Measurement

halBsExecTimeStart - Start Execution Time Measurement

Synopsis

```
#include <basic/bs_basic.h>
uint_t halBsExecTimeStart (void);
```

Descriptions

The *halBsExecTimeStart* function support measurement of maximum execution time for the events Red threads and interrupts. The maximum execution time value is given in jiffies, a user defined resolution (range micro seconds).

The *halBsExecTimeStart* function returns current timer value.

This function shall be supplied by the user and to be adapted to the user's environment.

Returns

The *halBsExecTimeStart* function returns current timer value.

Errors

Example

```
#include  <basic/bs_basic.h>
/* Infineon XC166 implementation */

uint_t halBsExecTimeStart(void)
{
  return(T3);
}
```

Notes

The user supplies this function.

References

halBsExecTimeStop

halBsExecTimeStop - Stop Execution Time Measurement

Synopsis

```
#include <basic/bs_basic.h>
uint_t halBsExecTimeStop (uint_t value, uint_t adjust);
```

Descriptions

This HAL services support measurement of maximum, minimum or average execution time for the events as Blue, Red and Green threads. The maximum execution time value is given in jiffies, a user defined resolution (range micro seconds).

The *halBsExecTimeStop* function calculates the elapsed time by subtracting (if counting up) the start time given by *value* from current timer value decremented by *adjust* and sets the timer to *value* subtracted by *adjust*.

This function shall be supplied by the user and to be adapted to the user's environment.

Returns

The *halBsExecTimeStop* function returns the elapsed time.

Errors

Example

```
#include  <basic/bs_basic.h>
/* Infineon XC166 implementation */

uint_t halBsExecTimeStop(uint_t value, uint_t adjust)
{ uint_t  et;

  et = (T3 – value) - adjust;
  T3 = (value - adjust);
  return(et);
}
```

Notes

The user supplies these functions.

References

halBsExecTimeStart

halBsExecTimeGet - Get Execution Time Value

Synopsis

```
#include <basic/bs_basic.h>
uint_t halBsExecTimeGet (void);
```

Descriptions

The *halBsExecTimeGet* function returns the measurement timer value.

This function shall be supplied by the user and to be adapted to the user's environment.

Returns

The *halBsExecTimeGet* function returns current timer value.

Errors

Example

```
#include  <basic/bs_basic.h>
/* Infineon XC166 implementation */

uint_t halBsExecTimeGet(void)
{
  return(T3);
}
```

Notes

The user supplies these functions.

References

halBsExecTimeStart

halBsExecTimeResume - Resume Execution Time Measurement

Synopsis

```
#include <basic/bs_basic.h>
void halBsExecTimeResume (uint_t value, uint_t adjust);
```

Descriptions

The *halBsExecTimeResume* function resumes the timer value given by *value* subtracted by *adjust*.

This function shall be supplied by the user and to be adapted to the user's environment.

Returns

No value is returned.

Errors

Example

```
#include  <basic/bs_basic.h>
/* Infineon XC166 implementation */

void halBsExecTimeResume(uint_t value, uint_t adjust)
{
  T3 = (value - adjust);
}
```

Notes

The user supplies these functions.

References

halBsExecTimeStart

Free Running Timer

halBsFreeRunTimerGet - Get Time Value

Synopsis

```
#include <basic/bs_basic.h>
uint32_t halBsFreeRunTimerGet (void);
```

Descriptions

The *halBsFreeRunTimerGet* function returns the time stamp value.

This function shall be supplied by the user and to be adapted to the user's environment.

Returns

Returns current time value.

Errors

Example

```
#include <basic/bs_basic.h>
/* Freescale MPC55xx implementation */

uint32_t halBsFreeRunTimerGet (void)

{
  return(halRegTBLowRead());
}
```

Notes

The user supplies this function.

References

.

3 Basic Services

Rubus OS Control

bsRubusInit - Initiate Rubus OS

Synopsis

```
#include <basic/bs_basic.h>
int_t bsRubusInit(uint_t flags);
```

Descriptions

This function initialises Rubus OS.

The *flag* argument requests the initialisation of Rubus OS objects. The value of *flags* is the bitwise inclusive OR of values of the following list:

_R_READ	Do NOT initialize the Basic Log services.
_R_RESET	Initiate the memory space of the services to zero.

By not initialising the Basic Log services the contents of the Log may be read Post-Mortem.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	An invalid attribute of an object generated by Rubus ICE is detected.
R_ERROR_PRIORITY	Priority miss match between Rubus OS library and the configuration generated by Rubus ICE.
R_ERROR_CONTROL_BLOCK_SIZE	Miss match between the Control block size in Rubus OS and the control blocks generated by Rubus ICE.

Example

```
#include  <basic/bs_basic.h>

void func (void)
{
  halInit();
  bsRubusinit();
  :
  bsRubusStart();
}
```

References

bsRubusStart()

bsRubusStart – Start Execution of Rubus OS

Synopsis

```
#include <basic/bs_basic.h>
void bsRubusStart(void);
```

Descriptions

The *bsRubusStart()* function start execution of the Red and Blue Kernel. The Basic Timer is started by calling the function *halBsTimerStart()*. The execution of the default Red schedule is released as soon as the Basic timer interrupt is processed.

In case the Blue Kernel is enabled, the Blue thread assigned the highest priority will be executed first. From now on the Blue Kernel administrate the execution capacity of the processor not consumed by the Red Kernel and the Green Kernel.

Returns

Does not return.

Errors

Example

```
#include  <basic/bs_basic.h>
void main (void)
{
  halInit();
  bsRubusinit();
  :
  bsRubusStart();
}
```

Notes

In case the Blue Kernel is enabled, current User Stack (the C-stack) is released.

An error detected by the Blue Kernel during run-time is notified by calling the user defined function *blueError.*

References

bsRubusInit(), halBsTimerStart().

Common Services

bsClockGetTime - Get Clock

Synopsis

```
#include <basic/bs_basic.h>
void bsClockGetTime(uint32_t *tp);
```

Descriptions

The *bsClockGetTime* function returns the current value *tp* for the Basic Clock.

Returns

The functions always complete without error.

Errors

Example

```
#include  <basic/bs_basic.h>

void func (void)
{ uint32_t time;

  bsClockGetTime(&time);
    :
}
```

Notes

Only the Basic Clock is relevant for the set operation, since the value of the Basic Clock is copied into the Blue Clock, upon Blue Timer expiration (at Blue Timer notification).

The value of Blue Clock differs from the Basic Clock until the Blue Timer expires the first time after the set operation.

References

bsClockSetTime

bsClockSetTime - Set Clock

Synopsis

```
#include <basic/bs_basic.h>
void bsClockSetTime(uint32_t *tp);
```

Descriptions

The *bsClockSetTime* function shall set the Basic Clock to the value specified by *tp*.

Returns

The functions always complete without error.

Errors

Example

```
#include  <basic/bs_basic.h>

void func (void)
{ uint32_t time;
```

```
  time = 2;
  bsClockSetTime(&time);
   :
}
```

Notes

Only the Basic Clock is relevant for the set operation, since the value of the Basic Clock is copied into the Blue Clock, upon Blue Timer expiration (at Blue Timer notification).

The value of Blue Clock differs from the Basic Clock until the Blue Timer expires the first time after the set operation.

References

bsClockGetTime

bsResourceNext– Find all Resources

Synopsis

```
#include <basic/bs_basic.h>
int_t bsResourceNext(uint_t type, bsObject_t const **object);
```

Descriptions

This function finds all resources in Rubus OS of type specified by *type*. The various resource types are specified in <basic/bs_basic.h>.

To get the first resource of a specified type the *object* parameter shall point to the value NULL. Subsequent calls to *bsResourceNext* will get the next resource.

See also Static Allocation.

Returns

Upon successful completion, the function shall return R_OK. Otherwise, a negative value is returned indicating the error.

Errors

R_ERROR_EMPTY	No more objects
R_ERROR_INVALID_ATTRIBUTE	The object parameter points to an object with a different type then specified in *type*

Example

```
#include <blue/b_thread.h>

void func (void)
{ blueThreadAttr_t const *thread;

  thread = 0;
  while (bsResourceNext ((R_OBJECT_BLUE_THREAD | _R_THREAD_TYPE_STACK |
         _R_THREAD_TYPE_NO_STACK),(bsObject_t const **)&thread) == R_OK) {
     :
  }
  return(R_OK);
}
```

References

.

bsJiffiesToTv, - Convert between Jiffies and Time Value

Synopsis

```
#include <basic/bs_basic.h>
void bsJiffiesToTv(uint32_t jiffies, bsPosixTime_t *tp);
```

Descriptions

The *bsJiffiesToTv* function converts from number of timer ticks specified by *jiffies* to time value and store the result in *tp*.

Returns

The function always complete without error.

Errors

Example

```
#include  <basic/bs_basic.h>
bsPosixTime_t const Time = {1,0};

void func (void)
{ bsPosixTime_t   tv;
  uint32_t t;

  bsJiffiesToTv (t, &tv);
  :
}
```

References

bsTvToJiffies.

bsTvToJiffies - Convert between Time Value and Jiffies

Synopsis

```
#include <basic/bs_basic.h>
uint32_t bsTvToJiffies(bsPosixTime_t const *tp);
```

Descriptions

The *bsTvToJiffies* function converts a time value specified by *tp* to number of timer ticks.

Returns

Upon successful completion, the function shall return the number of timerticks. Otherwise a NULL value is returned indicating the error.

Errors

NULL	The time value is corrupt, the number of nano seconds is greater than or equal to 1000000000.

Example

```
#include  <basic/bs_basic.h>
bsPosixTime_t const Time = {1,0};

void func (void)
{  uint32_t t;

  t = bsTvToJiffies(&time);
  :
}
```

Notes

Conversion from time value to number of timer ticks may not be correct (*bsTvToJiffies*) if number of seconds in the time value exceeds the precision of the type specified by the macro uint32_t.

The number of timer ticks returned by the function *bsTvToJiffies* is normally used for Blue time supervision, which may require a signed value (lost precision).

References

bsJiffiesToTv.

bsStatus – Get Basic Status

Synopsis

```
#include <basic/bs_basic.h>
void bsStatus(bsStatus_t *status);
```

Descriptions

Get the status of the Basic Services and store the information in *status*.

The *bsStatus_t* type declares the following members:

execMode	R_MODE_RED	Indicate execution of Red threads (Red Kernel).
	R_MODE_GREEN	Indicates execution of Green threads, Green Kernel)
	R_MODE_BLUE	Indicates execution of Blue threads (Blue Kernel).
preemptionLevel		Pre-emption level related to the kernel mode.
lcetBasicTimer		Current execution time value (jiffies) of the Basic Timer interrupt.

Returns

The functions always complete without error.

Errors

Example

```
#include <basic/bs_basic.h>

void func (void)
{ bsStatus_t status;

  bsStatus(&status);
  :
}
```

References

Basic Message Queue

bsQueueInit - Initiate a Queue

Synopsis

```
#include <basic/bs_queue.h>
int_t bsQueueInit(bsQueueId_t queue, uint_t flags);
```

Descriptions

This function initialises a Basic Queue defined by the parameter *queue.* The queue is initialised to be empty. If _R_RESET is set in *flag*, the memory space for the queue is set to zero. If _R_OVERWRITE is set in *flag*, the oldest element of the queue is overwritten when full.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	The attribute *queue* is invalid.

Example

```
#include  <basic/bs_queue.h>
extern bsQueueId_t queue;

void func (void)
{
  bsQueueInit(&queue, _R_RESET);
}
```

References

bsInit

bsQueueDestroy - Destroy a Message Queue

Synopsis

```
#include <basic/bs_queue.h>
int_t bsQueueDestroy(bsQueueId_t queue);
```

Descriptions

This function destroys the message queue referred by the attribute *queue.* Since the memory space for the message descriptor and the message queue is statically allocated the memory space is not de-allocated.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *queue*.

Example

```
#include  <basic/bs_queue.h>
extern bsQueueAttr_t const queue;

void func (void)
{
  if (bsQueueInit(&queue)) == R_OK)
    { :
      bsQueueDestroy(&queue);
    }
}
```

References

bsQueueInit

bsQueueGet - Consuming Read of Basic Queue Element

Synopsis

```
#include <basic/bs_queue.h>
int_t bsQueueGet(bsQueueId_t queue, uint8_t *object);
```

Descriptions

This function removes the oldest element from the Basic Queue *queue*, by copying the contents of next element to be released into *object*. If the queue is empty, an error code is returned.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *queue*.
R_ERROR_INVALID_CONTROL_BLOCK	The control block of *queue* is invalid.
R_ERROR_EMPTY	The queue is empty

Example

```
#include  <basic/bs_queue.h>
extern bsQueueId_t queue;

void func(void)
{ user_t obj;

  if (bsQueueGet(&queue , (uint8_t*)&obj)) == R_OK)
   :
}
```

References

bsInit, bsQueueInit, bsQueuePut, bsQueueRead

bsQueuePut - Put a Basic Queue Element

Synopsis

```
#include <basic/bs_queue.h>
int_t bsQueuePut(bsQueueId_t queue, uint8_t *object);
```

Descriptions

This function inserts a queue object into the Basic Queue *queue*, by copying the contents of *object* into the queue. If the queue is full, an error code is returned.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *queue*.
R_ERROR_INVALID_CONTROL_BLOCK	The control block of *queue* is invalid.
R_ERROR_FULL	The queue is full, the flag _R_OVERWRITE is not set

Example

```
#include  <basic/bs_queue.h>
extern bsQueueId_t queue;

void func(void)
{ user_t obj;

  if (bsQueuePut(&queue, (uint8_t*)&obj) != R_OK) {
  :
}
```

References

bsInit, bsQueueInit, bsQueueGet, bsQueueRead

bsQueueRead - Non consuming Read of Basic Queue Elements

Synopsis

```
#include <basic/bs_queue.h>
int_t bsQueueRead(bsQueueId_t queue, uint8_t *object, uint_t flags);
```

Descriptions

This function copies an element from the Basic Queue *queue* into *object* without removing the element from the queue.

If _R_REWIND is set in *flags,* the oldest element is always read.

If the _R_REWIND is NOT set in *flags,* the *bsQueueRead* function reads the elements from the queue in descending order.

When trying to read beyond the last element of the queue, an error code is returned.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *queue*.
R_ERROR_INVALID_CONTROL_BLOCK	The control block of *queue* is invalid.
R_ERROR_EMPTY	The queue is empty.

Example

```
#include  <basic/bs_queue.h>
extern bsQueueId_t queue;

void func(void)
{ user_t obj;

  :
  while(bsQueueRead(&queue , (uint8_t*)&obj),0) == R_OK)
     :
}
```

References

bsInit, bsQueueInit, bsQueuePut, bsQueueGet.

.

Basic Memory Pool

bsPoolInit - Initiate a Memory Pool

Synopsis

```
#include <basic/bs_pool.h>
int_t bsPoolInit(bsPoolId_t pool, uint_t flags);
```

Descriptions

The *bsPoolInit* function initialises a Basic Memory Pool defined by the attribute *pool.* The free list of buffers is build from the reserved memory space for the pool.

If _R_RESET is set in *flags*, the memory space for the pools is set to zero.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	The attribute *pool* is invalid.

Example

```
#include  <basic/bs_pool.h>
extern bsPoolId_t pool;

void func (void)
{
  bsPoolInit(&pool,R_RESET);
}
```

References

bsInit

bsPoolDestroy - Destroy a Message Pool

Synopsis

```
#include <basic/bs_pool.h>
int_t bsPoolDestroy(bsPoolId_t pool);
```

Descriptions

This function destroys the message pool referred by the attribute *pool.* Since the memory space for the message descriptor and the message pool is statically allocated the memory space is not de-allocated.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *msg*.

Example

```
#include  <basic/bs_pool.h>
extern bsPoolAttr_t const pool;

void func (void)
{
  if (bsPoolInit(&pool,R_RESET)) == R_OK)
    { :
      bsPoolDestroy(&pool);
    }
}
```

References

bsPoolInit

bsPoolAllocate - Allocate a Memory Pool Buffer

Synopsis

```
#include <basic/bs_pool.h>
int_t bsPoolAllocate (bsPoolId_t pool, uint8_t **buffer);
```

Descriptions

This function removes a buffer from Basic Memory Pool *pool* list of free buffers and stores the buffer in the pointer pointed to by the argument *buffer*. If the Pool is empty, an error code is returned.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	The argument *pool* is invalid.
R_ERROR_INVALID_CONTROL_BLOCK	The control block of *pool* is invalid.
R_ERROR_EMPTY	The memory pool is empty

Example

```
#include  <basic/bs_pool.h>
extern bsPoolId_t pool;

void func(void)
{ uint8_t *obj;

  if (bsPoolAllocate(&pool , &obj)) == R_OK)
   :
}
```

References

bsPoolInit, bsPoolFree..

bsPoolFree - Free a Memory Pool Buffer

Synopsis

```
#include <basic/bs_pool.h>
int_t bsPoolFree (bsPoolId_t pool, uint8_t **buffer);
```

Descriptions

This function inserts the buffer pointer pointed to by the argument *buffer* in the list of free buffers of the Basic Memory Pool *pool*. The content of the pointer pointed to by the argument *buffer* is set to NULL. If the buffer does not belong to the memory pool *pool* an error is returned.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_PARAMETER	Invalid parameter, *buffer or *buffer is NULL*.
R_ERROR_INVALID_ATTRIBUTE	The argument *pool* is invalid.
R_ERROR_INVALID_CONTROL_BLOCK	The control block of *pool* is invalid.
R_ERROR_NOT_PERMITTED	The buffer don’t belong to this pool

Example

```
#include  <basic/bs_pool.h>
extern bsPoolId_t pool;

void func(void)
{ uint8_t *obj;

  if (bsPoolFree(&pool , &obj)) == R_OK)
    :
}
```

References

bsPoolInit, bsPoolAllocate.

Basic Mailbox

bsMailboxInit - Initiate a Mailbox

Synopsis

```
#include <basic/bs_mbox.h>
int_t bsMailboxInit(bsMailboxId_t mailbox);
```

Descriptions

The *bsMailboxInit* function initialises a Basic Mailbox defined by the attribute *mailbox*. The mailbox is intilised to be empty.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	The attribute *mailbox* is invalid.

Example

```
#include  <basic/bs_mbox.h>
extern bsMailboxId_t mailbox;

void func (void)
{
  bsMailboxInit(&mailbox);
}
```

References

bsInit

bsMailboxDestroy - Destroy a Mailbox

Synopsis

```
#include <basic/bs_mbox.h>
int_t bsMailboxDestroy(bsMailboxId_t mailbox);
```

Descriptions

This function destroys the mailbox referred by the attribute mailbox. Since the memory space for the message descriptor and the message pool is statically allocated the memory space is not de-allocated.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *mailbox*.

Example

```
#include  <basic/bs_mbox.h>
extern bsMailboxAttr_t const mailbox;

void func (void)
{
  if (bsMailboxInit(&mailbox)) == R_OK)
    { :
      bsMailboxDestroy(&mailbox);
    }
}
```

References

bsMailboxInit

bsMailboxSend - Send a message to a Mailbox

Synopsis

```
#include <basic/bs_mbox.h>
int_t bsMailboxSend (bsMailboxId_t mailbox, bsMailboxHdr_t *msgPtr);
```

Descriptions

This function adds the message pointed to by the argument *msgPtr* to the Basic Mailbox specified by *mailbox.* The buffer is inserted FIFO into the message list of the mailbox.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	The attribute *mailbox* is invalid.
R_ERROR_INVALID_PARAMETER	Invalid parameter *msgPtr*
R_ERROR_INVALID_CONTROL_BLOCK	The control block of *mailbox* is invalid.

Example

```
#include  <basic/bs_mbox.h>
extern bsMailboxId_t mailbox;

void func(void)
{ bsMailboxHdr_t *obj;

  :
  obj->type = 0x0100;
  if (bsMailboxSend(&mailbox , obj)) == R_OK)
}
```

References

bsMailboxInit, bsMailboxReceive

bsMailboxReceive - Receive a message from a Mailbox

Synopsis

```
#include <basic/bs_mbox.h>
int_t bsMailboxReceive (bsMailboxId_t mailbox, bsMailboxHdr_t **msgPtr, uint_t msgType);
```

Descriptions

This function is used to receive a message from the Basic Mailbox *mailbox*. If there is a message of specified message type *msgType*, the oldest message of that type is received from the mailbox and the message is stored in the pointer pointed to by the *msgPtr* argument.

The message type is a bit mask. If the message type is not NULL, the list of messages are scanned starting with the oldest. For each message in the list the intersection between *msgType* and the message type of the message (*bsMailboxHdr_t*) is checked. If the intersection is not NULL (and-operation) the message is received from the list.

If no message type is specified (*msgType* == NULL), the oldest message in the list is received.

If no message is available of appropriate message type, an error code is returned.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *mailbox*.
R_ERROR_INVALID_PARAMETER	Invalid parameter *buffer*.
R_ERROR_INVALID_CONTROL_BLOCK	The control block of *mailbox* is invalid.
R_ERROR_EMPTY	The mailbox is empty or does not contain a message of specified type

Example

```
#include  <basic/bs_mbox.h>
extern bsMailboxId_t mailbox;

void func(void)
{ bsMailboxHdr_t *obj;

  :
  if (bsMailboxReceive(&mailbox , &obj, 0x0120)) == R_OK)
    :
}
```

References

bsMailboxInit, bsMailboxSend..

bsMailboxFree – Free a Message buffer allocated from a Memory Pool

Synopsis

```
#include <basic/bs_mbox.h>
int_t bsMailboxFree (bsMailboxHdr_t **msgPtr)
```

Descriptions

This function frees a message buffer that has been allocated from a memory pool.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_PARAMETER	*msgPtr* or * *msgPtr* is NULL

See also error codes for *bsPoolFree*.

Example

```
#include <basic/bs_mbox.h>
extern bsMailboxId_t mailbox;

void func (void)
{
  if (bsMailboxFree(&mailbox) == R_OK)
  :
}
```

References

bsMailboxInit, bsPoolFree

Basic Event Log

bsLogSet – Set Event Log Mode of an Object

Synopsis

```
#include <basic/bs_log.h>
int_t bsLogSet (int_t id, uint8_t flags);
```

Descriptions

The *bsLogSet* function sets the Event Log Mode of the object specified by argument *id.* The *flags* attribute defines the log mode of the object. If the value `R_LOG_OBJECT_ON` is set in *flags* the events of the object is logged. If the value `R_LOG_OBJECT_OFF` is set the events of the object is not logged.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_PARAMETER	Invalid *id*.

Example

```
#include <basic/bs_log.h>

void func(void)
{
  bsLogSet(id, R_LOG_OBJECT_ON);
  :
  bsLogStart();
  :
  bsLogStop();
}
```

References

bsLogSet, bsLogSetAl, bsLogStart, bsLogStop.

bsLogSetAll – Set Event Log Mode of all Objects

Synopsis

```
#include <basic/bs_log.h>
void bsLogSetAll (uint_t flags);
```

Descriptions

The *bsLogSetAll* function sets the Event Log Mode of all objects. The *flags* attribute defines the log mode of the object. If the value `R_LOG_OBJECT_ON` is set in *flags* the events of the objects are logged. If the value `R_LOG_OBJECT_OFF` is set the events of the objects are not logged.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

This function returns nothing.

Example

```
#include <basic/bs_log.h>

void func(void)
{
  bsLogSetAll(R_LOG_OBJECT_ON);
  :
  bsLogStart();
  :
  bsLogStop();
}
```

References

bsLogSet, bsLogSetAl, bsLogStart, bsLogStop.

bsLogStart – Start the Event Log

Synopsis

```
#include <basic/bs_log.h>
int_t bsLogStart (uint_t flags);
```

Descriptions

The *bsLogStart* function starts logging the previous specified objects (*bsLogSet, bsLogSetAll*) The operation to be done when the Event Log becomes full is specified by argument *flags*. If the flag _R_OVERWRITE is set the oldest element in the log is overwritten

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_ACCESS_DENIED	Log services is not initialised
R_ERROR_INVALID_CONTROL_BLOCK	The Log control block is invalid.

Example

```
#include <basic/bs_log.h>
```

```
void func(void)
{
  bsLogSetAll(R_LOG_OBJECT_ON);
  :
  bsLogStart(_R_OVERWRITE);
  :
}
```

Notes

References

bsLogSet, bsLogSetAl, bsLogStart, bsLogStop.

bsLogStop – Stop the Event Log

Synopsis

```
#include <basic/bs_log.h>
int_t bsLogStop (void);
```

Descriptions

The *bsLogStop* function stops logging the events.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_ACCESS_DENIED	Log services is not initialised
R_ERROR_INVALID_CONTROL_BLOCK	The Log control block is invalid.

Example

```
#include <basic/bs_log.h>

void func(void)
{
  bsLogStart(_R_OVERWRITE);
  :
  bsLogStop();
}
```

Notes

References

bsLogSet, bsLogSetAl, bsLogStart, bsLogStop.

4 Green Services

Green Kernel

GreenError – Green Error Handling

Synopsis

```
#include <green/g_thread.h>
void greenError(int_t code, bsObject_t const * object);
```

Descriptions

The *greenError* function raises an error indicated by the argument code and denoting the related object via the argument.

The *greenError* function is invoked from the Green Kernel. The User must supply this function.

Returns

This function returns nothing.

Errors

Example

```
#include  <basic/bs_basic.h>
#include  <green/g_thread.h>

void greenError(int_t code, bsObject_t const * object)
{
  :
  halHalt();
}
```

Notes

The *greenError* function is utilised within the Green Kernel as well in order to indicate an error from an application thread.

Errors detected by the Green Kernel will generate a call to the *greenError* function.

Errors detected by Green Kernel are described in Error Handling.

The *greenError* function called from the Green Kernel may be completed or not. If it is completed, error recovering is done, which may lead to undesired system behaviour.

The user supplies this function.

References

blueError, redError

Interrupt Control

GreenIntrDisable - Disable Interrupts

Synopsis

```
#include <green/g_thread.h>
void greenIntrDisable (void);
```

Descriptions

The *greenIntrDisable()* function disables the interrupts by setting the interrupt level of the processor to the Green Ceiling Priority Level.

Returns

No value is returned.

Errors

Example

```
#include  <green/g_thread.h>

void func (void)
{
  greenIntrDisable();
}
```

Notes

References

greenIntrEnable, greenIntrSuspend, greenIntrResume

GreenIntrEnable - Enable Interrupts

Synopsis

```
#include <green/g_thread.h>
void greenIntrEnable (void);
```

Descriptions

The *greenIntrEnable()* function enables the interrupts by setting the interrupt level of the processor to allow all interrupts.

Returns

No value is returned.

Errors

Example

```
#include  <green/g_thread.h>

void func (void)
{
  greenIntrEnable();
}
```

Notes

References

greenIntrDisable, greenIntrSuspend, greenIntrResume

GreenIntrSuspend - Suspend Interrupts

Synopsis

```
#include <green/g_thread.h>
R_TYPE_GREEN_LEVEL greenIntrSuspend (void);
```

Descriptions

The *greenIntrSuspend()* function saves current interrupt level and disables the interrupts by setting the interrupt level of the processor to the Green Ceiling Priority Level.

Returns

The current interrupt level is returned, a processor specific value.

Errors

Example

```
#include  <green/g_thread.h>

{ R_TYPE_GREEN_LEVEL level;

  level = greenIntrSuspend();
   :
  greenIntrResume(level);
}
```

Notes

The returned interrupt level value is a processor specific value and shall not be modified by the user.

References

greenIntrDisable, greenIntrEnable, greenIntrResume

GreenIntrResume - Resume Interrupts

Synopsis

```
#include <green/g_thread.h>
void greenIntrResume (R_TYPE_GREEN_LEVEL level);
```

Descriptions

The *greenIntrResume()* function restores the interrupt level of the processor to the level specified by *level*.

Returns

No value is returned.

Errors

Example

```
#include  <green/g_thread.h>

void func (void)
{ R_TYPE_GREEN_LEVEL level;

  level = greenIntrSuspend();
   :
  greenIntrResume(level);
}
```

Notes

The interrupt level to restore is previous returned by the *greenIntrSuspend()* function.

References

greenIntrDisable, greenIntrEnable, greenIntrSuspend

5 Red Services

Red Kernel

redError – Red Error Handler

Synopsis

```
#include <red/r_thread.h>
void redError(int_t code, bsObject_t const * object);
```

Descriptions

The *redError* function raises an error indicated by the argument code and denoting the related object via the argument .

The *redError* function is invoked from the Red Kernel. The User must supply this function.

Returns

This function returns nothing.

Errors

Example

```
#include  <basic/bs_basic.h>
#include  <red/r_thread.h>

void redError(int_t code, bsObject_t const * object)
{
  :
  :
  halHalt();
    or
  redSetScheduleImmediate(&redSchedule_1);
}

void redfunc(void)
{
  redError(-4,redSelf());
}
```

Notes

The *redError* function is utilised within the Red Kernel as well in order to indicate an error from an application thread.

Errors detected by the Red Kernel Will generate a call to the *redError* function.

Errors detected by Red Kernel are described in Error Handling.

The *redError* function called from the Red Kernel may be completed or not. If it is completed, error recovering is done, which may lead to undesired system behaviour.

The user supplies this function.

References

blueError, greenError

redStatus – Get Red Kernel Status

Synopsis

```
#include <red/r_thread.h>
int_t redStatus(redStatus_t *status);
```

Description

The *redStatus* function retrieves the status of the Red Kernel.

The redStatus_t type declares the following members:

redTime	Current Red Time
schedule	Executing schedule
thread	Executing thread
lcetKernelBasic	Current execution time in jiffies, Red Kernel context switch
lcetKernelPreemption	Current execution time in jiffies, Red Kernel pre-emption

Returns

The function always shall return R_OK.

Errors

Example

```
#include  <red/r_thread.h>

void func(void)
{ redStatus_t status;

  if (redStatus(&status) == R_OK)
    :
}
```

Notes

References

redThreadStatus

redStackUsed – Get Red Stack Status

Synopsis

```
#include <red/r_thread.h>
int_t redStackUsed (int_t * stackUsed);
```

Descriptions

The *redStackUsed* function returns the status of the Red Stack and number of bytes used on the stack in *stackUsed*.

The condition of the Red Stack is controlled. The top and the bottom of the stack contain links, which are checked. If the links are not corrupt and the pattern of the top of the stack is not modified the Red stack is correct.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_STACK_INCONSISTENT	The stack is corrupt

Example

```
#include  <red/r_thread.h>

void func(void)
{ int_t size;

  if (redStackUsed(&size) == R_OK)
    :
}
```

Notes

A stack is initialised by a pattern at initialisation of the Red Kernel. By calculating the free space (containing the pattern) and subtract this figure from the total size of the stack the used size is found.

The Red stack is NOT continuously checked by the Red Kernel, as is the case with the Blue thread stacks.

References

redStatus

Red Schedule

redGetTime - Get Red Schedule Time

Synopsis

```
#include <red/r_thread.h>
uint16_t redGetTime(void);
```

Descriptions

Returns the Red Schedule Timer value given in number of timer ticks. The time represents the execution time relative the beginning of current Red schedule.

Returns

Returns the current timer value.

Errors

Example

```
#include <red/r_thread.h>

void func (void)
{ uint16_t time;

  time = redGetTime();
  :
}
```

References

bsGetTime.

redSetSchedule - Set Schedule

Synopsis

```
#include <red/r_thread.h>
int_t  redSetSchedule(redScheduleId_t schedule);
```

Descriptions

The *redSetSchedule* function sets the Red schedule specified by schedule. The execution of the specified schedule begins after completion of current schedule or if no schedule is active at the beginning of next timer notification.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid parameter *schedule*

Example

```
#include  <red/r_thread.h>

void func (void)
{
  if (R_OK != redSetSchedule(&redSchedule_1))
     ;
}
```

References

redSetScheduleImmediate.

redSetScheduleImmediate - Set Schedule Immediate

Synopsis

```
#include <red/r_thread.h>
int_t  redSetScheduleImmediate(redScheduleId_t schedule);
```

Descriptions

The *redSetScheduleImmediate* function sets the Red schedule specified by schedule. The execution of the specified schedule begins at the beginning of next Red Timer notification. Execution of current Red schedule is interrupted and the context of the Red Kernel is completed.

Returns

If the *redSetScheduleImmediate* function is called from a Red thread, it will not return. In such a case only if an error is detected this function will return.

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid parameter *schedule*

Example

```
#include  <red/r_thread.h>

void func (void)
{
  if (R_OK != redSetScheduleImmediate(&redSchedule_1))
     ;
}
```

Notes

The *redSetScheduleImmediate* function interrupts execution of current Red schedule if called from a Red thread. Normally used in an error situation.

References

redInit, redSetSchedule.

Red Thread

redSelf - Get Executing Red Thread's ID

Synopsis

```
#include <red/r_thread.h>
redThreadId_t redSelf(void);
```

Descriptions

Returns the identifier of the executing Red Thread.

Returns

Upon successful completion, the function returns the thread identifier of the calling thread. Otherwise the NULL pointer is returned.

Errors

NULL	No Red thread is running or the thread attribute is corrupt.

Example

```
#include  <red/r_thread.h>

void func (void)
{ redThreadId_t  thread;

  if ((thread = redSelf()) != 0)
     ;
}
```

References

redSetSchedule, redSetScheduleImmediate, redGetTime.

6 Blue Services

Blue Kernel

blueError – Blue Error Handling

Synopsis

```
#include <blue/b_thread.h>
void blueError(int_t code, bsObject_t const *object);
```

Descriptions

This function raises an error indicated by the argument *code* in the context of the running Blue thread. The argument *object* indicates the erroneous Blue Thread.

This function is invoked from the Blue Kernel. The User must supply this function.

Returns

This function returns nothing.

Errors

Example

```
#include  <basic/bs_basic.h>
#include  <blue/b_thread.h>

void blueError(int_t code, bsObject_t const *object)
{
  /* user code */
}

void bluefunc(void)
{
  blueError(-4, (bsObject_t const *)blueSelf());
}
```

Notes

The *blueError* function is utilised within the Blue Kernel as well in order to indicate an error from an application thread.

Errors detected by the Blue Kernel will generate a call to the *blueError* function.

Errors detected by Blue Kernel are described in Error Handling.

The *blueError* function called from the Blue Kernel may be completed or not. If it is completed, error recovering is done, which may lead to undesired system behaviour.

The user supplies this function.

References

redError,greenError

blueClockGetTime – Get Blue Clock

Synopsis

```
#include <blue/b_thread.h>
void blueClockGetTime (int32_t *tp);
```

Descriptions

The function stores current value for the Blue Clock in *tp*.

Returns

The function can not fail.

Errors

Example

```
#include  <blue/b_thread.h>

void bluefunc(void)
{ int32_t time;
  :
  blueClockGetTime(&time);
  :
}
```

Notes

The value of the Basic Clock is copied to the Blue Clock when the Blue Timer expires (at Blue Timer notification).

The function shall only be called from a Blue thread.

To obtain the true system time the *bsClockGetTime()* can also be used from a Blue thread.

References

blueStackUsed – Get Blue Stack Status

Synopsis

```
#include <blue/b_thread.h>
int_t blueStackUsed (int_t * stackUsed);
```

Descriptions

The *blueStackUsed* function returns the status of the Blue Shared Run-Time Stack and number of bytes used on the stack in *stackUsed*.

The condition of the Blue Stack is controlled. The top and the bottom of the stack contain links, which are checked. If the links are not corrupt and the pattern of the top of the stack is not modified the Blue stack is correct.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_STACK_INCONSISTENT	The stack is corrupt

Example

```
#include  <blue/b_thread.h>

void func(void)
{ int_t size;

  if (blueStackUsed(&size) == R_OK)
    :
}
```

Notes

A stack is initialised by a pattern at initialisation of the Blue Kernel. By calculating the free space (containing the pattern) and subtract this figure from the total size of the stack the used size is found.

The Blue stack is NOT continuously checked by the Blue Kernel, as is the case with the Blue thread stacks.

References

Blue Thread Management

blueCreate – Create a Blue Thread

Synopsis

```
#include <blue/b_thread.h>
int_t blueCreate(blueThreadId_t thread);
```

Descriptions

Creates a Blue thread, with attributes specified by *thread*. Upon successful completion, the control block of the created thread is initialised.

The thread is initialised to state *Blocked* and the execution is postponed until it is unblocked by the activation signal generated by the *blueActivate – Activate a Thread* function.

When the thread becomes the running thread the entry function *thread→entry(arg)*, with *arg* as its sole argument is called. If the function *entry* returns, the thread is set in state *Blocked* waiting for the activation signal.

If the thread has an own run-time stack assigned (_R_THREAD_TYPE_STACK flag set) the memory spaces for the Blue Thread stack are initialised to a default pattern. This pattern is used to measure the maximum stack usage

The set of signals pending for the thread shall be empty.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *thread*
R_ERROR_NOT_PERMITTED	Stack parameters is specified but flag _R_THREAD_TYPE_NO_STACK is set.

Example

```
#include  <blue/b_thread.h>
extern blueThreadAttr_t const blueT;

void func (void)
{
  if ((blueCreate (&blueT)) == R_OK)
      ;
}

void blueTBody(void)
{
  :
  :
} /* The thread is implicitly set in state BLOCKED when reaching the end statement
     waiting for the activation signal */
```

Notes

Rubus ICE allocates memory space for the thread control block and its stack.

References

blueInit, blueExit, blueActivate, blueSelf, blueYield.

blueExit - Terminate the Calling Blue Thread

Synopsis

```
#include <blue/b_thread.h>
int_t blueExit (void);
```

Descriptions

Terminates the calling thread. Thread termination does not release any application visible thread resources, including but not limited to, mutex and message queues.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue/b_thread.h>

void func (void)
{
  blueExit();
}
```

Notes

A terminated thread can be revived by calling the *blueCreate()* function.

References

blueCreate, blueSelf, blueYield.

blueSelf - Get Calling Thread's ID

Synopsis

```
#include <blue/b_thread.h>
blueThreadId_t blueSelf(void);
```

Descriptions

Return the thread ID of the calling thread.

Returns

Upon successful completion, the function returns the thread ID of the calling thread. Otherwise the NULL pointer is returned to indicate the error.

Errors

NULL	The Blue Thread Control Block is corrupt

Example

```
#include  <blue/b_thread.h>

void func (void)
{ blueThreadId_t  thread;

  if ((thread = blueSelf()) != 0)
     ;
}
```

References

blueCreate,blueExit, blueYield

blueYield - Yield the Processor

Synopsis

```
#include <blue/b_thread.h>
int_t blueYield(void);
```

Descriptions

The *blueYield* function forces the running thread to relinquish the processor until it again becomes the head of its thread list..

If there are threads ready for execution having the same priority, the calling thread becomes the tail of the Blue Thread List of its priority. In other case, the calling thread will continue its execution.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include <blue/b_thread.h>

void func (void)
{
  blueYield();
  :
}
```

References

blueCreate,blueExit, blueSelf

bluePreemptionLock - Lock Pre-emption

Synopsis

```
#include <blue/b_thread.h>
void bluePreemptionLock(void);
```

Descriptions

Locks the pre-emption of the calling Blue thread. The functions are recursive. To unlock pre-emption, an equivalent amount of calls to *bluePreemptionUnlock* and must be done by the calling thread.

Returns

This function returns nothing and never yields an error.

Errors

Example

```
#include  <blue/b_thread.h>

void func (void)
{
  bluePreemptionLock();
}
```

Notes

These functions are to be used with care. If the pre-emption is locked the scheduling of running Blue threads is prohibited until either the calling thread is blocked or the pre-emption is unlocked. If a Blue thread lock pre-emption, and is blocked, the pre-emption must be unlocked explicitly when the thread is unblocked.

References

blueGetPrio, bluePreemptionUnlock

bluePreemptionUnlock - Unlock Pre-emption

Synopsis

```
#include <blue/b_thread.h>
void bluePreemptionUnlock(void);
```

Descriptions

Unlocks the pre-emption of the calling Blue thread. The functions are recursive. To lock pre-emption, an equivalent amount of calls to *bluePreemptionLock* must be done by the calling thread.

Returns

This function returns nothing and never yields an error.

Errors

Example

```
#include  <blue/b_thread.h>

void func (void)
{
    :
  bluePreemptionUnlock();
}
```

Notes

These functions are to be used with care. If the pre-emption is locked the scheduling of running Blue threads is prohibited until either the calling thread is blocked or the pre-emption is unlocked. If a Blue thread lock pre-emption, and is blocked, the pre-emption must be unlocked explicitly when the thread is unblocked.

References

blueGetPrio, bluePreemptionLock.

blueThreadStackUsed – Get Status of a Blue Thread Stack

Synopsis

```
#include <blue/b_thread.h>
int_t  blueThreadStackUsed (blueThreadId_t thread, int_t *stackUsed)
```

Descriptions

The *blueThreadStackUsed* function retrieves the state of the stack of the Blue thread *thread* and the number of bytes currently used is stored in *stackUsed*.

The condition of the Blue thread stack is controlled. The top and the bottom of the stack contain links, which are checked. If the links are not corrupt and the pattern of the top of the stack is not modified the stack is correct.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_STACK_INCONSISTENT	The stack of *thread* is corrupt.

Example

```
#include <blue/b_thread.h>

void func (void)
{ blueThreadId_t     thread;
  int_t                size;
  if (blueThreadStackUsed(thread,&size) == R_OK)
  :
}
```

Notes

A stack is initialised by a pattern at creation of the thread. By calculating the free space (containing the pattern) and subtract this figure from the total size of the stack the used size is found (the member *stackUsed*).

The Blue thread stack is continuously checked by the Blue Kernel.

References

blueCreate, blueExit, blueSelf

Blue Signals

blueActivate – Activate a Thread

Synopsis

```
#include <blue/b_signal.h>
int_t blueActivate (blueThreadId_t thread);
```

Descriptions

The function directs an activation signal to be delivered to the specified *thread* blocked waiting for the activation signal. The thread is blocked waiting for the activation signal at return from the threads entry function see *blueCreate – Create a Blue Thread.*

If the signal is delivered prior to the entry function of the thread returns, the signals is included into the set of pending signals of the thread. Each activation signal is counted and the thread will be unblocked as many times as counted. The maximum signals queued is given by the value R_THREAD_ACTIVATION_MAX.

If the thread is blocked waiting for the signal, the thread is unblocked and the entry function of the thread is called.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *thread*.
R_ERROR_INVALID_CONTROL_BLOCK	Invalid control block of *thread*.
R_ERROR_CORRUPT_IRQ	The Interrupt Queue is corrupt (called from a Red or Green thread).
R_ERROR_FULL	Maximum number of activation queued signals is exceeded.

Example

```
#include  <blue/b_signal.h>
void blueThread/greenThread/redThread (void)

{ blueThreadId_t thread;
  :
  blueActivate(thread);
  :
}
```

Notes

If the service is called from a Red or Green thread, the examination of the interrupt notification is postponed to the exit of the Red or Green mode.

The number of activation signals sent to a thread is not counted (bit-OR).

References

blueCreate

blueSigSend - Send a Signal to a Thread

Synopsis

```
#include <blue/b_signal.h>
int_t blueSigSend (blueThreadId_t thread, blueSigSet_t set);
```

Descriptions

The function directs a set of signals to be delivered to the specified *thread*.

If a signal is delivered prior to the call to the *blueSigTimedWait* function, the signals is included into the set of pending signals of the thread.

If the thread is blocked waiting for one of the signals, the thread is unblocked and the function *blueSigTimedWait* returns.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *thread*.
R_ERROR_INVALID_PARAMETER	The *set* contains reserved signals.
R_ERROR_INVALID_CONTROL_BLOCK	Invalid control block of *thread*.
R_ERROR_CORRUPT_IRQ	The Interrupt Queue is corrupt (called from a Red or Green thread).
R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue/b_signal.h>

void blueThread/greeanThread/redThread (void)
{ blueThreadId_t thread;
  :
  blueSigSend(thread,0x0200);
  :
}
```

Notes

This function can also be used from a Green or Red Thread in order to send a signal to a Blue Thread.

If the service is called from a Red or Green thread, the examination of the interrupt notification is postponed to the exit of the Red or Green mode.

References

blueSigTimedWait, blueSigPending

blueSigEmptySet - Clear Pending Signals

Synopsis

```
#include <blue/b_signal.h>
int_t blueSigEmptySet (blueThreadId_t thread, blueSigSet_t  set);
```

Descriptions

This function clears the pending signals specified by *set* of the specified *thread*.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *thread*.
R_ERROR_INVALID_PARAMETER	The *set* contains reserved signals.
R_ERROR_INVALID_CONTROL_BLOCK	Invalid control block of *thread*.

Example

```
#include  <blue\b_signal.h>

void func (void)
{ blueSigSet_t sig;

  blueSigEmptySet(0);
    :
}
```

References

blueSigTimedWait, blueSigSend, blueSigPending, blueSleep

blueSigPending - Examine Pending Signals

Synopsis

```
#include <blue/b_signal.h>
int_t blueSigPending (blueThreadId_t thread, blueSigSet_t  *set);
```

Descriptions

The *blueSigPending* function shall store the set of signals that are blocked from delivery and are pending for the specified *thread* in the space pointed to by the argument *set*.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *thread*.
R_ERROR_INVALID_CONTROL_BLOCK	Invalid control block of *thread*.

Example

```
#include  <blue\b_signal.h>

void func (void)
{ blueSigSet_t sig;

  blueSigPending(blueSelf(),&sig);
    :
}
```

References

blueSigTimedWait, blueSigSend, blueSleep..

blueSigTimedWait - Wait for a Signal

Synopsis

```
#include <blue/b_signal.h>
int_t blueSigTimedWait (blueSigSet_t set, blueSigSet_t *rset, int32_t *timeout);
```

Descriptions

The *blueSgTimedWait* function selects the pending signals from the set specified by *set* and automatically clears the signals from the *set* of pending signals. The intersection of the *set* and the pending signals are stored in the space pointed to by the argument *rset*.

If no signals in the *set* are pending at the time of the call, the thread waits for the time interval specified by *timeout* until one or more signals becomes pending.

If the value of the *timeout* parameter is non-null, the wait for a signal to occur shall be terminated when the specified timeout expires. The timeout expires after the interval specified by *timeout* has elapsed relative to the time that the wait began. The resolution of the timeout is the resolution of the Blue Clock.

If the value of the *timeout* is R_TIME_NO_WAIT (0), and the signal is not pending, the operation fails without waiting.

If the *timeout* is the R_TIME_WAIT_INFINITY*,* an infinite time is denoted.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_PARAMETER	The *set* contains reserved signals.
R_ERROR_OPERATION_TIMEOUT	No signal specified by *set* was delivered within the specified time-out period.
R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue\b_signal.h>

bsPosixTime_t const time = {1,0};
```

```
void func (void)
{ blueSigSet_t     rset;
  int32_t tv;

  tv = bsTvToJiffies(&time);
  if (blueSigTimedWait(0x0200,&rset,&tv)) == R_OK)
   :
  blueSigTimedWait(0x0200,&rset,0);
}
```

Notes

No round up of the time-out value is done.

The suspension time may be longer than requested due to the scheduling of other activities by the system.

The suspension time may be shorter than requested, if the argument value is less than the resolution of the Blue Clock or if the value is not an even multiple of the resolution.

References

blueSigSend, blueSigPending.

blueSleep - High Resolution Sleep

Synopsis

```
#include <blue/b_signal.h>
int_t blueSleep (int32_t *timeout);
```

Descriptions

The *blueSleep* function causes the current thread to be suspended until the time interval specified by the *timeout* argument has elapsed. The resolution of the time-out is the resolution of the Blue Clock.

If the value of the *timeout* is R_TIME_NO_WAIT (0), the operation is completed without suspending the current thread from execution.

If the *timeout* is the R_TIME_WAIT_INFINITY*,* an infinite time is denoted.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue\b_signal.h>

bsPosixTime_t const time = {2,0};

void func (void)
```

```
{ int32_t tv;
   :
  tv = bsTvToJiffies(&time);
  blueSleep(&tv);
   :
}
```

Notes

No round up of the time-out value is done.

The suspension time may be longer than requested due to the scheduling of other activities by the system.

The suspension time may be shorter than requested, if the argument value is less than the resolution of the Blue Clock or if the value is not an even multiple of the resolution.

The *blueSleep* function can not be interrupted by a signal delivered to thread.

References

blueSigTimedWait, blueSigSend, blueSigPending,.

Blue Mutex

blueMutexInit - Initiate a Mutex

Synopsis

```
#include <blue/b_mutex.h>
int_t blueMutexInit(blueMutexId_t mutex);
```

Descriptions

This function is used to initialise the Mutex referred by the attribute specified by *mutex*. If the function fails, the Mutex is not initialised.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *mutex*.

Example

```
#include  <blue/b_mutex.h>
extern blueMutexAttr_t const mx;

void func (void)
{
  blueMutexInit(&mx);
  :
}
```

References

blueMutexTimedWait, blueMutexTryLock, blueMutexUnlock.

blueMutexDestroy - Destroy a Mutex

Synopsis

```
#include <blue/b_mutex.h>
int_t blueMutexDestroy(blueMutexId_t mutex);
```

Descriptions

This function destroys the Mutex referred by the attribute specified by *mutex.* Trying to destroy a locked Mutex will result in an error being returned without changing the state of the Mutex.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *mutex*.
R_ERROR_BUSY	An attempt is done to destroy *mutex* while it is locked or referenced

Example

```
#include  <blue/b_mutex.h>
extern blueMutexAttr_t const mx;

void func (void)
{
  :
  blueMutexDestroy(&mx);
}
```

References

blueMutexTimedWait, blueMutexTryLock, blueMutexUnlock.

blueMutexTryLock - Try Locking a Mutex

Synopsis

```
#include <blue/b_mutex.h>
int_t blueMutexTryLock(blueMutexId_t mutex);
```

Descriptions

The *blueMutexTryLock* locks the Mutex, if the Mutex currently is unlocked. Otherwise the function returns immediately with the error code set.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *mutex.*
R_ERROR_INVALID_CONTROL_BLOCK	Invalid control block of *mutex*.
R_ERROR_ACCESS_DENIED	Trying to lock the mutex recursively or calling thread has higher priority then ceiling priority
R_ERROR_OPERATION_TIMEOUT	The *mutex* could not be locked.because it is busy
R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue/b_mutex.h>

void func (void)
{
  if (blueMutexTryLock(mxId) == R_OK)
     ;
}
```

Notes

A Mutex must not be locked recursively.

References

blueMutexInit, blueMutexDestroy.

blueMutexTimedLock - Locking a Mutex with timeout

Synopsis

```
#include <blue/b_mutex.h>
int_t blueMutexTimedLock(blueMutexId_t mutex, int32_t *timeout);
```

Descriptions

The *blueMutexTimedLock* locks the Mutex object referred by *mutex*. If the Mutex is already locked, the calling thread blocks until the Mutex becomes available. This operation returns with the Mutex object referenced by mutex in the locked state with the calling thread as its owner.

If the Mutex object referenced by *mutex* is locked at the time of the call, the thread shall wait for the time interval specified by *timeout* for the owner to unlock the Mutex.

If the value of the *timeout* parameter is non-null, the wait for another thread to unlock the Mutex shall be terminated when the specified time-out expires. The time-out expires after the interval specified by *timeout* has elapsed relative to the time that the wait began. The resolution of the time-out is the resolution of the Blue Clock.

If the value of the *timeout* is R_TIME_NO_WAIT (0), and the signal is not pending, the operation fails without waiting.

If the *timeout* is the R_TIME_WAIT_INFINITY*,* an infinite time is denoted.

If more than one thread is waiting to lock a Mutex when the Mutex become unlocked, the scheduling policy is used to determine which thread shall acquire the Mutex.

Returns

Upon successful completion, the function shall return R_OK. Otherwise, a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *mutex.*
R_ERROR_INVALID_CONTROL_BLOCK	Invalid control block of *mutex*.
R_ERROR_ACCESS_DENIED	Trying to lock the *mutex* recursively or calling thread has higher priority then ceiling priority
R_ERROR_OPERATION_TIMEOUT	The *mutex* could not be locked before the specified time-out expired.
R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue/b_mutex.h>
```

```
void func (void)
{ int32_t timeout = 1;

  if (blueMutexTimedLock(mxId,0) == R_OK)
}
```

Notes

A Mutex must not be locked recursively.

No round up of the time-out value is done.

The suspension time may be longer than requested due to the scheduling of other activities by the system.

The suspension time may be shorter than requested, if the argument value is less than the resolution of the Blue Clock or if the value is not an even multiple of the resolution.

References

blueMutexInit, blueMutexDestroy.

blueMutexUnlock - Unlock a Mutex

Synopsis

```
#include <blue/b_mutex.h>
int_t blueMutexUnlock(blueMutexId_t mutex);
```

Descriptions

This function unlocks the Mutex object referenced by *mutex*.

Trying to unlock a Mutex, which is not locked or if the calling thread is not the owner, will result in an error being returned.

If there are threads blocked on the Mutex object referenced by mutex when *blueMutexUnlock* is called, the scheduling policy is used to determine which thread shall acquire the Mutex.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *mutex.*
R_ERROR_INVALID_CONTROL _BLOCK	Invalid control block of *mutex*.
R_ ERROR_NOT_PERMITTED	Trying to unlock a *mutex* which is not locked or the callin thread is not the owner.
R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue/b_mutex.h>

void func (void)
{
  if (blueMutexUnLock(mxId) == R_OK)
     ;
}
```

Notes

If a thread locks several Mutex in sequence, the unlocking sequence shall be in LIFO order (Last In First out) to preserve the execution priority correct. The Mutex locked last shall be the one first unlocked.

References

blueMutexInit, blueMutexDestroy.

Blue Message Queue

blueMsgInit - Initiate a Message Queue

Synopsis

```
#include <blue/b_msg.h>
int_t blueMsgInit(blueMsgId_t msg);
```

Descriptions

The *blueMsgInit* function is used to initialise the message queue referred by the attribute specified by *msg*. If the *blueMsgInit* function fails, the message queue is not initialised..

The attributes of the message queue, such as size, message size, are specified in Rubus ICE.

The message queue is open for both receiving and sending messages.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *msg*.

Example

```
#include  <blue/b_msg.h>
extern blueMsgAttr_t const mq;

void func (void)
{
  if (blueMsgInit(&mq)) == R_OK)
    { :
      blueMsgDestroy(&mq);
    }
}
```

References

blueMsgTimedReceive, blueMsgSend.

blueMsgDestroy - Destroy a Message Queue

Synopsis

```
#include <blue/b_msg.h>
int_t blueMsgDestroy(blueMsgId_t msg);
```

Descriptions

The *blueMsgDestroy* function destroys the message queue referred by the attribute *msg*. The message queue is closed unconditionally. Threads blocked, waiting for the message queue will be unblocked returning with the message queue is not a valid reference. Since the memory space for the message descriptor and the message queue is statically allocated the memory space is not de-allocated.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *msg*.

Example

```
#include  <blue/b_msg.h>
extern blueMsgAttr_t const mq;

void func (void)
{
  if (blueMsgInit(&mq)) == R_OK)
    { :
      blueMsgDestroy(&mq);
    }
}
```

References

blueMsgTimedReceive, blueMsgSend.

blueMsgTimedReceive - Receive a Message from a Message Queue

Synopsis

```
#include <blue/b_msg.h>
int_t blueMsgTimedReceive(blueMsgId_t msg,uint8_t *msgPtr,uint_t msgLen,
int32_t *timeout);
```

Descriptions

This function receives the oldest of the messages from the message queue specified by *msg,* the message received is removed from the message queue and copied to the buffer pointed to by the *msgPtr*.

If the size of the buffer, expressed in bytes and specified by the *msgLen* argument, is less than the *size* attribute of the message queue, the function fails and returns an error.

If the message queue is empty, *blueMsgTimedReceive* causes the calling thread to wait for the time interval specified by *timeout* until a message is enqueued on the message queue.

If the value of the *timeout* parameter is non-null, the wait for another thread to enqueue a message on the message queue shall be terminated when the specified time-out expires. The time-out expires after the interval specified by *timeout* has elapsed relative to the time that the wait began. The resolution of the time-out is the resolution of the Blue Clock.

If the value of the *timeout* is R_TIME_NO_WAIT (0), and the signal is not pending, the operation fails without waiting.

If the *timeout* is the R_TIME_WAIT_INFINITY an infinite time is denoted.

If more than one thread is waiting to receive a message when a message arrives at an empty queue, the thread at the head of the Receive Waiting List of the message queue is selected to receive the message.

The Receive Waiting List is ordered by priority. The thread of highest priority that has been waiting the longest shall be selected to receive the message if the ordering is based on priority. If the waiting threads have the same priority then the thread that has been waiting the longest will receive the message.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *msg.*
R_ERROR_INVALID_CONTROL_BLOCK	The Message Queue Control Block is corrupt.
R_ERROR_EMPTY	The *msg* is empty or the time-out expired before a message could be received..
R_ERROR_SIZE	The specified message length, *msgLen*, exceeds the message size attribute of the message queue.
R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue/b_msg.h>

void func (void)
{ uint8_t  rbuff[10];
  uint_t   length;
  int32_t  timeout = 1;

  length = blueMsgTimedReceive(mqId,&rbuff,10,0);
  {}
   :
  if ((blueMsgTimedReceive(mqId,&rbuff,10,&timeout)) != R_ERROR)
  {}
}
```

Notes

No round up of the time-out value is done.

The suspension time may be longer than requested due to the scheduling of other activities by the system.

The suspension time may be shorter than requested, if the argument value is less than the resolution of the Blue Clock or if the value is not an even multiple of the resolution.

References

blueMsgInit, blueMsgDestroy.

blueMsgSend - Send a Message to a Message Queue

Synopsis

```
#include <blue/b_msg.h>
int_t blueMsgSend(blueMsgId_t msg,uint8_t *msgPtr,uint_t msgLen);
```

Descriptions

This adds the message pointed to by the argument *msgPtr* to the message queue specified by *msg*.

If the size of the buffer, expressed in bytes and specified by the *msgLen* argument, is equal zero or greater than the *size* attribute of the message queue, the function fails and returns an error.

The message is inserted into the message queue in FIFO order.

If the specified message queue is full, the message is not queued and an error is returned.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *msg.*
R_ERROR_INVALID_CONTROL_BLOCK	The Message Queue Control Block is corrupt.
R_ERROR_FULL	The specified message queue, *msg* is full.
R_ERROR_SIZE	The specified message length, *msgLen*, exceeds the message size attribute of the message queue.
R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue/b_msg.h>

void func (void)
{ uint8_t sbuff[10];
  uint8_t rbuff[10];

  blueMsgSend(mqId,&sbuff,10);
  :
  blueMsgTimedReceive(mqId,&rbuff,10,0);
}
```

Notes

Priority among messages within a message queue is not supported due to performance reasons. A message is always inserted FIFO.

References

blueMsgInit, blueMsgDestroy, blueMsgTimedReceive.

Blue Semaphore

blueSemInit - Initiate a Semaphore

Synopsis

```
#include <blue/b_sem.h>
int_t blueSemInit(blueSemId_t sem, uint_t value);
```

Descriptions

This function is used to initialise the Semaphore referred by the attribute specified by *sem*. The initial value of the Semaphore is set to *value*. If the function fails, the Semaphore is not initialised.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *sem*.
R_ERROR_INVALID_PARAMETER	If *value* exceed R_BLUE_SEM_VALUE_MAX

Example

```
#include  <blue/b_sem.h>
extern blueSemAttr_t const sem;

void func (void)
{
  blueSemInit(&sem, R_BLUE_SEM_VALUE_LOCKED);
  :
}
```

References

blueSemTimedWait, blueSemTryLock, blueSemUnlock, blueSemStatus.

blueSemDestroy - Destroy a Semaphore

Synopsis

```
#include <blue/b_sem.h>
int_t blueSemDestroy(blueSemId_t sem);
```

Descriptions

This function destroys the Sem referred by the attribute specified by *sem.* Trying to destroy a Semaphore that Blue threads are blocked on will result in an error being returned without changing the state of the Semaphore.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *sem*.
R_ERROR_BUSY	An attempt is done to destroy *sem* while it is locked or referenced

Example

```
#include  <blue/b_sem.h>
extern blueSemAttr_t const sem;

void func (void)
{
  :
  blueSemDestroy(&sem);
}
```

References

blueSemTryLock, blueSemTimedWait, blueSemUnlock, blueSemStatus.

blueSemTryLock - Try Locking a Semaphore

Synopsis

```
#include <blue/b_sem.h>
int_t blueSemTryLock(blueSemId_t sem);
```

Descriptions

The *blueSemTryLock* locks the Semaphore, if the Semaphore currently is unlocked. Otherwise the function returns immediately with the error code set.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *sem.*
R_ERROR_INVALID_CONTROL_BLOCK	Invalid control block of *sem*.
R_ERROR_OPERATION_TIMEOUT	The *sem* could not be locked.because it is busy
R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue/b_sem.h>

void func (void)
{
  if (blueSemTryLock(&sem) == R_OK)
     ;
}
```

Notes

References

blueSemInit, blueSemDestroy, blueSemTimedWait, blueSemUnlock.

blueSemTimedLock - Locking a Semaphore with timeout

Synopsis

```
#include <blue/b_sem.h>
int_t blueSemTimedLock(blueSemId_t sem, int32_t *timeout);
```

Descriptions

The *blueSemTimedLock* locks the Semaphore object referred by *sem*. If the Semaphore is already locked, the calling thread blocks until the Semaphore becomes available. This operation returns with the Semaphore object referenced by *sem* in the locked state.

If the Semaphore object referenced by *sem* is locked at the time of the call, the thread shall wait for the time interval specified by *timeout* for the Semaphore to be unlocked.

If the value of the *timeout* parameter is non-null, the wait for another thread to unlock the Sem shall be terminated when the specified time-out expires. The time-out expires after the interval specified by *timeout* has elapsed relative to the time that the wait began. The resolution of the time-out is the resolution of the Blue Clock.

If the value of the *timeout* is R_TIME_NO_WAIT (0), and the signal is not pending, the operation fails without waiting.

If the *timeout* is the R_TIME_WAIT_INFINITY, an infinite time is denoted.

If more than one thread is waiting to lock a Semaphore when the Semaphore become unlocked, the scheduling policy is used to determine which thread shall acquire the Semaphore.

Returns

Upon successful completion, the function shall return R_OK. Otherwise, a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *sem.*
R_ERROR_INVALID_CONTROL_BLOCK	Invalid control block of *sem*.
R_ERROR_OPERATION_TIMEOUT	The *sem* could not be locked before the specified time-out expired.
R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue/b_sem.h>

void func (void)
{ int32_t timeout = 1;

  if (blueSemTimedLock(&sem,0) == R_OK)
}
```

Notes

A Semaphore can be locked recursively.

No round up of the time-out value is done.

The suspension time may be longer than requested due to the scheduling of other activities by the system.

The suspension time may be shorter than requested, if the argument value is less than the resolution of the Blue Clock or if the value is not an even multiple of the resolution.

References

blueSemInit, blueSemDestroy, blueSemTryLock, blueSemUnlock.

blueSemUnlock - Unlock a Semaphore

Synopsis

```
#include <blue/b_sem.h>
int_t blueSemUnlock(blueSemId_t sem);
```

Descriptions

This function unlocks the Semaphore object referenced by *sem*.

Trying to unlock a Semaphore, which is not locked, will result in an error being returned.

If there are threads blocked on the Semaphore object referenced by *sem* when *blueSemUnlock* is called, the scheduling policy is used to determine which thread shall acquire the Semaphore.

Returns

Upon successful completion, the function shall return R_OK. Otherwise a negative value is returned indicating the error.

Errors

R_ERROR_INVALID_ATTRIBUTE	Invalid attribute *sem.*
R_ERROR_INVALID_CONTROL _BLOCK	Invalid control block of *sem*.
R_ERROR_NOT_PERMITTED	Trying to unlock a *sem* which is not locked.
R_ERROR_ACCESS_DENIED	Access is denied for a Blue SSX thread

Example

```
#include  <blue/b_sem.h>

void func (void)
{
  if (blueSemUnLock(&sem) == R_OK)
     ;
}
```

Notes

References

blueSemInit, blueSemDestroy, blueSemTryLock, blueSemTimedWait.