TINOC2_freeRTOS

1.0

Generated by Doxygen 1.8.11

Contents

Chapter 1

RTOSDemo

Programa del TINOC2 basado en ejemplo de freeRTOS sobre el microcontrolador LPC1102.

Autores:

- GARCIA, Pablo (Div. Telemetría. CITEDEF)
- ROUX, Federico (Div. Pirotecnia. CITEDEF)

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Module Documentation

5.1 Definiciones correspondientes al manejo del IAP

Funciones, variables y ctes. de manejo de la memoria flash.

Macros

- #define IAP_LOCATION 0x1FFF1FF1

  Dirección de memoria de punto de entrada de la función IAP. El código de esta región está en modo Thumb, así que al entrar a esta función, el programa entrará en modo Thumb UM10429 pag. 173.
- #define IAP_COM_REINVOKE_ISP 57

  Numero de comando para invocar ISP.

Typedefs

- typedef void(∗ IAP) (uint32_t[ ], uint32_t[ ])

  definicion de tipo de dato: puntero a función para ingresar comandos IAP.

5.1.1 Detailed Description

Funciones, variables y ctes. de manejo de la memoria flash.

5.1.2 Macro Definition Documentation

5.1.2.1 #define IAP_COM_REINVOKE_ISP 57

Numero de comando para invocar ISP.

Definition at line 27 of file flash.h.

5.1.2.2 #define IAP_LOCATION 0x1FFF1FF1

Dirección de memoria de punto de entrada de la función IAP. El código de esta región está en modo Thumb, así que al entrar a esta función, el programa entrará en modo Thumb UM10429 pag. 173.

Definition at line 25 of file flash.h.

5.1.3 Typedef Documentation

5.1.3.1 IAP

definicion de tipo de dato: puntero a función para ingresar comandos IAP.

Definition at line 39 of file flash.h.

5.2 xCoRoutineCreate

croutine. h

```
BaseType_t xCoRoutineCreate(
                                 crCOROUTINE_CODE pxCoRoutineCode,
                                 UBaseType_t uxPriority,
                                 UBaseType_t uxIndex
                               );
```

Create a new co-routine and add it to the list of co-routines that are ready to run.

Parameters

pxCoRoutineCode	Pointer to the co-routine function. Co-routine functions require special syntax - see the co-routine section of the WEB documentation for more information.
uxPriority	The priority with respect to other co-routines at which the co-routine will run.
uxIndex	Used to distinguish between different co-routines that execute the same function. See the example below and the co-routine section of the WEB documentation for further information.

Returns

pdPASS if the co-routine was successfully created and added to a ready list, otherwise an error code defined with ProjDefs.h.

Example usage:

```
// Co-routine to be created.
void vFlashCoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking
// This may not be necessary for const variables.
static const char cLedToFlash[ 2 ] = { 5, 6 };
static const TickType_t uxFlashRates[ 2 ] = { 200, 400 };

// Must start every co-routine with a call to crSTART();
crSTART( xHandle );

for( ;; )
{
    // This co-routine just delays for a fixed period, then toggles
    // an LED.  Two co-routines are created using this function, so
    // the uxIndex parameter is used to tell the co-routine which
    // LED to flash and how int32_t to delay.  This assumes xQueue has
    // already been created.
    vParTestToggleLED( cLedToFlash[ uxIndex ] );
    crDELAY( xHandle, uxFlashRates[ uxIndex ] );
}

// Must end every co-routine with a call to crEND();
crEND();


}

// Function that creates two co-routines.
void vOtherFunction( void )
{
uint8_t ucParameterToPass;
TaskHandle_t xHandle;
```

```
// Create two co-routines at priority 0.  The first is given index 0
// so (from the code above) toggles LED 5 every 200 ticks.  The second
// is given index 1 so toggles LED 6 every 400 ticks.
for( uxIndex = 0; uxIndex < 2; uxIndex++ )
{
    xCoRoutineCreate( vFlashCoRoutine, 0, uxIndex );
}


}
```

5.3 vCoRoutineSchedule

croutine. h

```
void vCoRoutineSchedule( void );
```

Run a co-routine.

vCoRoutineSchedule() executes the highest priority co-routine that is able to run. The co-routine will execute until it either blocks, yields or is preempted by a task. Co-routines execute cooperatively so one co-routine cannot be preempted by another, but can be preempted by a task.

If an application comprises of both tasks and co-routines then vCoRoutineSchedule should be called from the idle task (in an idle task hook).

Example usage:

```
// This idle task hook will schedule a co-routine each time it is called.
// The rest of the idle task will execute between co-routine calls.
void vApplicationIdleHook( void )
{
   vCoRoutineSchedule();
}


// Alternatively, if you do not require any other part of the idle task to
// execute, the idle task hook can call vCoRoutineScheduler() within an
// infinite loop.
void vApplicationIdleHook( void )
{
   for( ;; )
   {
       vCoRoutineSchedule();
   }
}
```

5.4 crSTART

croutine. h

```
crSTART( CoRoutineHandle_t xHandle );
```

This macro MUST always be called at the start of a co-routine function.

Example usage:

```
// Co-routine to be created.
void vACoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking
static int32_t ulAVariable;

// Must start every co-routine with a call to crSTART();
crSTART( xHandle );

for( ;; )
{
     // Co-routine functionality goes here.
}

// Must end every co-routine with a call to crEND();
crEND();


}
```

croutine. h

```
crEND();
```

This macro MUST always be called at the end of a co-routine function.

Example usage:

```
// Co-routine to be created.
void vACoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking
static int32_t ulAVariable;

// Must start every co-routine with a call to crSTART();
crSTART( xHandle );

for( ;; )
{
     // Co-routine functionality goes here.
}

// Must end every co-routine with a call to crEND();
crEND();


}
```

5.5 crDELAY

croutine. h

```
crDELAY( CoRoutineHandle_t xHandle, TickType_t xTicksToDelay );
```

Delay a co-routine for a fixed period of time.

crDELAY can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.

Parameters

xHandle	The handle of the co-routine to delay. This is the xHandle parameter of the co-routine function.
xTickToDelay	The number of ticks that the co-routine should delay for. The actual amount of time this equates to is defined by configTICK_RATE_HZ (set in FreeRTOSConfig.h). The constant portTICK_PERIOD_MS can be used to convert ticks to milliseconds.

Example usage:

```
// Co-routine to be created.
void vACoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking
// This may not be necessary for const variables.
// We are to delay for 200ms.
static const xTickType xDelayTime = 200 / portTICK_PERIOD_MS;

// Must start every co-routine with a call to crSTART();
crSTART( xHandle );

for( ;; )
{
   // Delay for 200ms.
   crDELAY( xHandle, xDelayTime );

   // Do something here.
}

// Must end every co-routine with a call to crEND();
crEND();


}
```

5.6 crQUEUE_SEND

```
crQUEUE_SEND(
                  CoRoutineHandle_t xHandle,
                  QueueHandle_t pxQueue,
                  void *pvItemToQueue,
                  TickType_t xTicksToWait,
                  BaseType_t *pxResult
             )
```

The macro's crQUEUE_SEND() and crQUEUE_RECEIVE() are the co-routine equivalent to the xQueueSend() and xQueueReceive() functions used by tasks.

crQUEUE_SEND and crQUEUE_RECEIVE can only be used from a co-routine whereas xQueueSend() and xQueueReceive() can only be used from tasks.

crQUEUE_SEND can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

Parameters

xHandle	The handle of the calling co-routine. This is the xHandle parameter of the co-routine function.
pxQueue	The handle of the queue on which the data will be posted. The handle is obtained as the return value when the queue is created using the xQueueCreate() API function.
pvItemToQueue	A pointer to the data being posted onto the queue. The number of bytes of each queued item is specified when the queue is created. This number of bytes is copied from pvItemToQueue into the queue itself.
xTickToDelay	The number of ticks that the co-routine should block to wait for space to become available on the queue, should space not be available immediately. The actual amount of time this equates to is defined by configTICK_RATE_HZ (set in FreeRTOSConfig.h). The constant portTICK_PERIOD_MS can be used to convert ticks to milliseconds (see example below).
pxResult	The variable pointed to by pxResult will be set to pdPASS if data was successfully posted onto the queue, otherwise it will be set to an error defined within ProjDefs.h.

Example usage:

```
// Co-routine function that blocks for a fixed period then posts a number onto
// a queue.
static void prvCoRoutineFlashTask( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking
static BaseType_t xNumberToPost = 0;
static BaseType_t xResult;


   // Co-routines must begin with a call to crSTART().
   crSTART( xHandle );


   for( ;; )
   {
       // This assumes the queue has already been created.
       crQUEUE_SEND( xHandle, xCoRoutineQueue, &xNumberToPost, NO_DELAY, &xResult );
```

```
        if( xResult != pdPASS )
        {
            // The message was not posted!
        }


        // Increment the number to be posted onto the queue.
        xNumberToPost++;


        // Delay for 100 ticks.
        crDELAY( xHandle, 100 );
    }


    // Co-routines must end with a call to crEND().
    crEND();
}
```

5.7 crQUEUE_RECEIVE

croutine. h

```
crQUEUE_RECEIVE(
                    CoRoutineHandle_t xHandle,
                    QueueHandle_t pxQueue,
                    void *pvBuffer,
                    TickType_t xTicksToWait,
                    BaseType_t *pxResult
                )
```

The macro's crQUEUE_SEND() and crQUEUE_RECEIVE() are the co-routine equivalent to the xQueueSend() and xQueueReceive() functions used by tasks.

crQUEUE_SEND and crQUEUE_RECEIVE can only be used from a co-routine whereas xQueueSend() and xQueueReceive() can only be used from tasks.

crQUEUE_RECEIVE can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

Parameters

xHandle	The handle of the calling co-routine. This is the xHandle parameter of the co-routine function.
pxQueue	The handle of the queue from which the data will be received. The handle is obtained as the return value when the queue is created using the xQueueCreate() API function.
pvBuffer	The buffer into which the received item is to be copied. The number of bytes of each queued item is specified when the queue is created. This number of bytes is copied into pvBuffer.
xTickToDelay	The number of ticks that the co-routine should block to wait for data to become available from the queue, should data not be available immediately. The actual amount of time this equates to is defined by configTICK_RATE_HZ (set in FreeRTOSConfig.h). The constant portTICK_PERIOD_MS can be used to convert ticks to milliseconds (see the crQUEUE_SEND example).
pxResult	The variable pointed to by pxResult will be set to pdPASS if data was successfully retrieved from the queue, otherwise it will be set to an error code as defined within ProjDefs.h.

Example usage:

```
// A co-routine receives the number of an LED to flash from a queue.  It
// blocks on the queue until the number is received.
static void prvCoRoutineFlashWorkTask( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking
static BaseType_t xResult;
static UBaseType_t uxLEDToFlash;

    // All co-routines must start with a call to crSTART().
    crSTART( xHandle );

    for( ;; )
    {
        // Wait for data to become available on the queue.
        crQUEUE_RECEIVE( xHandle, xCoRoutineQueue, &uxLEDToFlash, portMAX_DELAY, &xResult );
```

```
        if( xResult == pdPASS )
        {
            // We received the LED to flash - flash it!
            vParTestToggleLED( uxLEDToFlash );
        }
    }


    crEND();
}
```

5.8 crQUEUE_SEND_FROM_ISR

croutine. h

```
crQUEUE_SEND_FROM_ISR(
                            QueueHandle_t pxQueue,
                            void *pvItemToQueue,
                            BaseType_t xCoRoutinePreviouslyWoken
                        )
```

The macro's crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() are the co-routine equivalent to the xQueueSendFromISR() and xQueueReceiveFromISR() functions used by tasks.

crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() can only be used to pass data between a co-routine and and ISR, whereas xQueueSendFromISR() and xQueueReceiveFromISR() can only be used to pass data between a task and and ISR.

crQUEUE_SEND_FROM_ISR can only be called from an ISR to send data to a queue that is being used from within a co-routine.

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xCoRoutinePreviouslyWoken	This is included so an ISR can post onto the same queue multiple times from a single interrupt. The first call should always pass in pdFALSE. Subsequent calls should pass in the value returned from the previous call.

Returns

pdTRUE if a co-routine was woken by posting onto the queue. This is used by the ISR to determine if a context switch may be required following the ISR.

Example usage:

```
// A co-routine that blocks on a queue waiting for characters to be received.
static void vReceivingCoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
char cRxedChar;
BaseType_t xResult;

// All co-routines must start with a call to crSTART().
crSTART( xHandle );

for( ;; )
{
    // Wait for data to become available on the queue.  This assumes the
    // queue xCommsRxQueue has already been created!
    crQUEUE_RECEIVE( xHandle, xCommsRxQueue, &uxLEDToFlash, portMAX_DELAY, &xResult );

    // Was a character received?
```

```
    if( xResult == pdPASS )
    {
        // Process the character here.
    }
}

// All co-routines must end with a call to crEND().
crEND();


}


// An ISR that uses a queue to send characters received on a serial port to
// a co-routine.
void vUART_ISR( void )
{
char cRxedChar;
BaseType_t xCRWokenByPost = pdFALSE;

// We loop around reading characters until there are none left in the UART.
while( UART_RX_REG_NOT_EMPTY() )
{
    // Obtain the character from the UART.
    cRxedChar = UART_RX_REG;

    // Post the character onto a queue.  xCRWokenByPost will be pdFALSE
    // the first time around the loop.  If the post causes a co-routine
    // to be woken (unblocked) then xCRWokenByPost will be set to pdTRUE.
    // In this manner we can ensure that if more than one co-routine is
    // blocked on the queue only one is woken by this ISR no matter how
    // many characters are posted to the queue.
    xCRWokenByPost = crQUEUE_SEND_FROM_ISR( xCommsRxQueue, &cRxedChar, xCRWokenByPost );
}


}
```

5.9 crQUEUE_RECEIVE_FROM_ISR

croutine. h

```
crQUEUE_SEND_FROM_ISR(
                            QueueHandle_t pxQueue,
                            void *pvBuffer,
                            BaseType_t * pxCoRoutineWoken
                        )
```

The macro's crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() are the co-routine equivalent to the xQueueSendFromISR() and xQueueReceiveFromISR() functions used by tasks.

crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() can only be used to pass data between a co-routine and and ISR, whereas xQueueSendFromISR() and xQueueReceiveFromISR() can only be used to pass data between a task and and ISR.

crQUEUE_RECEIVE_FROM_ISR can only be called from an ISR to receive data from a queue that is being used from within a co-routine (a co-routine posted to the queue).

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvBuffer	A pointer to a buffer into which the received item will be placed. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from the queue into pvBuffer.
pxCoRoutineWoken	A co-routine may be blocked waiting for space to become available on the queue. If crQUEUE_RECEIVE_FROM_ISR causes such a co-routine to unblock *pxCoRoutineWoken will get set to pdTRUE, otherwise *pxCoRoutineWoken will remain unchanged.

Returns

pdTRUE an item was successfully received from the queue, otherwise pdFALSE.

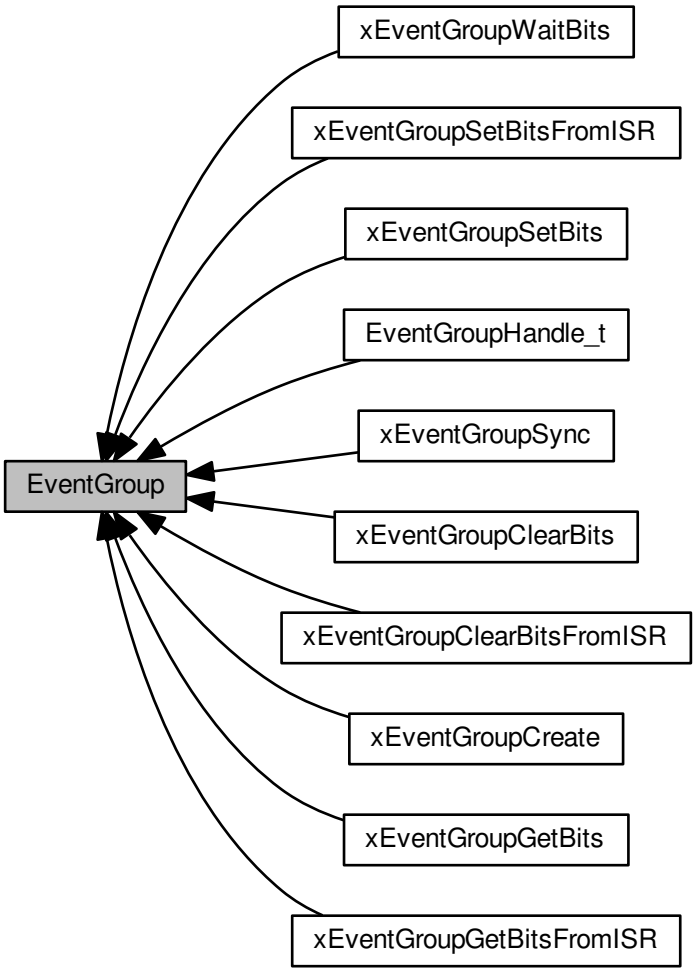
Example usage:

```
// A co-routine that posts a character to a queue then blocks for a fixed
// period.  The character is incremented each time.
static void vSendingCoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// cChar holds its value while this co-routine is blocked and must therefore
// be declared static.
static char cCharToTx = 'a';
BaseType_t xResult;

// All co-routines must start with a call to crSTART().
crSTART( xHandle );

for( ;; )
{
    // Send the next character to the queue.
    crQUEUE_SEND( xHandle, xCoRoutineQueue, &cCharToTx, NO_DELAY, &xResult );
```

```
    if( xResult == pdPASS )
    {
        // The character was successfully posted to the queue.
    }
 else
 {
    // Could not post the character to the queue.
 }

    // Enable the UART Tx interrupt to cause an interrupt in this
 // hypothetical UART.  The interrupt will obtain the character
 // from the queue and send it.
 ENABLE_RX_INTERRUPT();

 // Increment to the next character then block for a fixed period.
 // cCharToTx will maintain its value across the delay as it is
 // declared static.
 cCharToTx++;
 if( cCharToTx > 'x' )
 {
    cCharToTx = 'a';
 }
 crDELAY( 100 );
}

// All co-routines must end with a call to crEND().
crEND();


}


// An ISR that uses a queue to receive characters to send on a UART.
void vUART_ISR( void )
{
char cCharToTx;
BaseType_t xCRWokenByPost = pdFALSE;

while( UART_TX_REG_EMPTY() )
{
    // Are there any characters in the queue waiting to be sent?
 // xCRWokenByPost will automatically be set to pdTRUE if a co-routine
 // is woken by the post - ensuring that only a single co-routine is
 // woken no matter how many times we go around this loop.
    if( crQUEUE_RECEIVE_FROM_ISR( pxQueue, &cCharToTx, &xCRWokenByPost ) )
 {
     SEND_CHARACTER( cCharToTx );
 }
}


}
```

5.10 EventGroup

Collaboration diagram for EventGroup:



Modules

- EventGroupHandle_t
- xEventGroupCreate
- xEventGroupWaitBits
- xEventGroupClearBits
- xEventGroupClearBitsFromISR
- xEventGroupSetBits
- xEventGroupSetBitsFromISR
- xEventGroupSync
- xEventGroupGetBits
- xEventGroupGetBitsFromISR

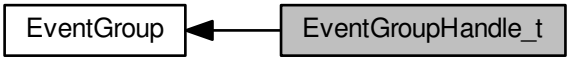
5.10.1 Detailed Description

An event group is a collection of bits to which an application can assign a meaning. For example, an application may create an event group to convey the status of various CAN bus related events in which bit 0 might mean "A CAN message has been received and is ready for processing", bit 1 might mean "The application has queued a message that is ready for sending onto the CAN network", and bit 2 might mean "It is time to send a SYNC message onto the CAN network" etc. A task can then test the bit values to see which events are active, and optionally enter the Blocked state to wait for a specified bit or a group of specified bits to be active. To continue the CAN bus example, a CAN controlling task can enter the Blocked state (and therefore not consume any processing time) until either bit 0, bit 1 or bit 2 are active, at which time the bit that was actually active would inform the task which action it had to take (process a received message, send a message, or send a SYNC).

The event groups implementation contains intelligence to avoid race conditions that would otherwise occur were an application to use a simple variable for the same purpose. This is particularly important with respect to when a bit within an event group is to be cleared, and when bits have to be set and then tested atomically - as is the case where event groups are used to create a synchronisation point between multiple tasks (a 'rendezvous').

5.11 EventGroupHandle_t

Collaboration diagram for EventGroupHandle_t:

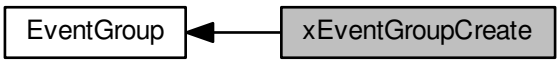


event_groups.h

Type by which event groups are referenced. For example, a call to xEventGroupCreate() returns an EventGroup↩ Handle_t variable that can then be used as a parameter to other event group functions.

5.12 xEventGroupCreate

Collaboration diagram for xEventGroupCreate:



event_groups.h

```
EventGroupHandle_t xEventGroupCreate( void );
```

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using xEventGropuCreate() then the required memory is automatically dynamically allocated inside the xEventGroupCreate() function. (see http://www.freertos.org/a00111.html). If an event group is created using xEventGropuCreateStatic() then the application writer must instead provide the memory that will get used by the event group. xEventGroupCreateStatic() therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the configUSE_16_BIT_TICKS setting in FreeRTOSConfig.h. If configUSE_16_BIT_TICKS is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If configUSE_16_BIT_TICKS is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The EventBits_t type is used to store event bits within an event group.

Returns

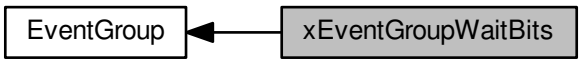
If the event group was created then a handle to the event group is returned. If there was insufficient FreeRTOS heap available to create the event group then NULL is returned. See http://www.freertos.org/a00111.html

Example usage:

```
// Declare a variable to hold the created event group.
EventGroupHandle_t xCreatedEventGroup;

// Attempt to create the event group.
xCreatedEventGroup = xEventGroupCreate();

// Was the event group created successfully?
if( xCreatedEventGroup == NULL )
{
 // The event group was not created because there was insufficient
 // FreeRTOS heap available.
}
else
{
 // The event group was created.
}
```

5.13 xEventGroupWaitBits

Collaboration diagram for xEventGroupWaitBits:



event_groups.h

```
EventGroupHandle_t xEventGroupCreateStatic( EventGroupHandle_t * pxEventGroupBuffer );
```

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using xEventGropuCreate() then the required memory is automatically dynamically allocated inside the xEventGroupCreate() function. (see http://www.freertos.org/a00111.html). If an event group is created using xEventGropuCreateStatic() then the application writer must instead provide the memory that will get used by the event group. xEventGroupCreateStatic() therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the configUSE_16_BIT_TICKS setting in FreeRTOSConfig.h. If configUSE_16_BIT_TICKS is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If configUSE_16_BIT_TICKS is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The EventBits_t type is used to store event bits within an event group.

Parameters

pxEventGroupBuffer	pxEventGroupBuffer must point to a variable of type StaticEventGroup_t, which will be then be used to hold the event group's data structures, removing the need for the memory to be allocated dynamically.

Returns

If the event group was created then a handle to the event group is returned. If pxEventGroupBuffer was NULL then NULL is returned.

Example usage:

```
// StaticEventGroup_t is a publicly accessible structure that has the same
// size and alignment requirements as the real event group structure.  It is
// provided as a mechanism for applications to know the size of the event
// group (which is dependent on the architecture and configuration file
// settings) without breaking the strict data hiding policy by exposing the
// real event group internals.  This StaticEventGroup_t variable is passed
// into the xSemaphoreCreateEventGroupStatic() function and is used to store
// the event group's data structures
StaticEventGroup_t xEventGroupBuffer;
```

```
// Create the event group without dynamically allocating any memory.
xEventGroup = xEventGroupCreateStatic( &xEventGroupBuffer );
```

event_groups.h

```
EventBits_t xEventGroupWaitBits(     EventGroupHandle_t xEventGroup,
                                const EventBits_t uxBitsToWaitFor,
                                const BaseType_t xClearOnExit,
                                const BaseType_t xWaitForAllBits,
                                const TickType_t xTicksToWait );
```

[Potentially] block to wait for one or more bits to be set within a previously created event group.

This function cannot be called from an interrupt.

Parameters

xEventGroup	The event group in which the bits are being tested. The event group must have previously been created using a call to xEventGroupCreate().
uxBitsToWaitFor	A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and/or bit 2 set uxBitsToWaitFor to 0x05. To wait for bits 0 and/or bit 1 and/or bit 2 set uxBitsToWaitFor to 0x07. Etc.
xClearOnExit	If xClearOnExit is set to pdTRUE then any bits within uxBitsToWaitFor that are set within the event group will be cleared before xEventGroupWaitBits() returns if the wait condition was met (if the function returns for a reason other than a timeout). If xClearOnExit is set to pdFALSE then the bits set in the event group are not altered when the call to xEventGroupWaitBits() returns.
xWaitForAllBits	If xWaitForAllBits is set to pdTRUE then xEventGroupWaitBits() will return when either all the bits in uxBitsToWaitFor are set or the specified block time expires. If xWaitForAllBits is set to pdFALSE then xEventGroupWaitBits() will return when any one of the bits set in uxBitsToWaitFor is set or the specified block time expires. The block time is specified by the xTicksToWait parameter.
xTicksToWait	The maximum amount of time (specified in 'ticks') to wait for one/all (depending on the xWaitForAllBits value) of the bits specified by uxBitsToWaitFor to become set.

Returns

The value of the event group at the time either the bits being waited for became set, or the block time expired. Test the return value to know which bits were set. If xEventGroupWaitBits() returned because its timeout expired then not all the bits being waited for will be set. If xEventGroupWaitBits() returned because the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared in the case that xClearOnExit parameter was set to pdTRUE.

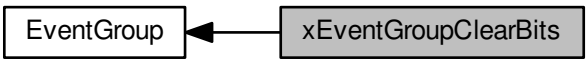
Example usage:

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )


void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;
const TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;
```

```
    // Wait a maximum of 100ms for either bit 0 or bit 4 to be set within
    // the event group.  Clear the bits before exiting.
    uxBits = xEventGroupWaitBits(
                xEventGroup,    // The event group being tested.
                BIT_0 | BIT_4,  // The bits within the event group to wait for.
                pdTRUE,         // BIT_0 and BIT_4 should be cleared before returning.
                pdFALSE,        // Don't wait for both bits, either bit will do.
                xTicksToWait ); // Wait a maximum of 100ms for either bit to be set.


    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        // xEventGroupWaitBits() returned because both bits were set.
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // xEventGroupWaitBits() returned because just BIT_0 was set.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // xEventGroupWaitBits() returned because just BIT_4 was set.
    }
    else
    {
        // xEventGroupWaitBits() returned because xTicksToWait ticks passed
        // without either BIT_0 or BIT_4 becoming set.
    }
}
```

5.14 xEventGroupClearBits

Collaboration diagram for xEventGroupClearBits:



event_groups.h

```
EventBits_t xEventGroupClearBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToCle
```

Clear bits within an event group. This function cannot be called from an interrupt.

Parameters

xEventGroup	The event group in which the bits are to be cleared.
uxBitsToClear	A bitwise value that indicates the bit or bits to clear in the event group. For example, to clear bit 3 only, set uxBitsToClear to 0x08. To clear bit 3 and bit 0 set uxBitsToClear to 0x09.

Returns

The value of the event group before the specified bits were cleared.

Example usage:

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )


void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;


  // Clear bit 0 and bit 4 in xEventGroup.
  uxBits = xEventGroupClearBits(
                          xEventGroup,    // The event group being updated.
                          BIT_0 | BIT_4 );// The bits being cleared.


  if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
  {
      // Both bit 0 and bit 4 were set before xEventGroupClearBits() was
      // called.  Both will now be clear (not set).
  }
  else if( ( uxBits & BIT_0 ) != 0 )
  {
```

```
        // Bit 0 was set before xEventGroupClearBits() was called.  It will
        // now be clear.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // Bit 4 was set before xEventGroupClearBits() was called.  It will
        // now be clear.
    }
    else
    {
        // Neither bit 0 nor bit 4 were set in the first place.
    }
}
```

5.15 xEventGroupClearBitsFromISR

Collaboration diagram for xEventGroupClearBitsFromISR:

event_groups.h

```
BaseType_t xEventGroupClearBitsFromISR( EventGroupHandle_t xEventGroup, const EventBits_t uxBit
```

A version of xEventGroupClearBits() that can be called from an interrupt.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow nondeterministic operations to be performed while interrupts are disabled, so protects event groups that are accessed from tasks by suspending the scheduler rather than disabling interrupts. As a result event groups cannot be accessed directly from an interrupt service routine. Therefore xEventGroupClearBitsFromISR() sends a message to the timer task to have the clear operation performed in the context of the timer task.

Parameters

xEventGroup	The event group in which the bits are to be cleared.
uxBitsToClear	A bitwise value that indicates the bit or bits to clear. For example, to clear bit 3 only, set uxBitsToClear to 0x08. To clear bit 3 and bit 0 set uxBitsToClear to 0x09.

Returns

If the request to execute the function was posted successfully then pdPASS is returned, otherwise pdFALSE is returned. pdFALSE will be returned if the timer service queue was full.

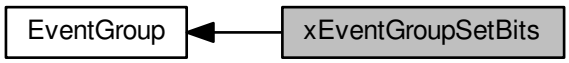
Example usage:

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

// An event group which it is assumed has already been created by a call to
// xEventGroupCreate().
EventGroupHandle_t xEventGroup;

void anInterruptHandler( void )
{
  // Clear bit 0 and bit 4 in xEventGroup.
  xResult = xEventGroupClearBitsFromISR(
                    xEventGroup,     // The event group being updated.
                    BIT_0 | BIT_4 ); // The bits being set.
```

```
    if( xResult == pdPASS )
    {
        // The message was posted successfully.
    }
}
```

5.16 xEventGroupSetBits

Collaboration diagram for xEventGroupSetBits:



event_groups.h

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet )
```

Set bits within an event group. This function cannot be called from an interrupt. xEventGroupSetBitsFromISR() is a version that can be called from an interrupt.

Setting bits in an event group will automatically unblock tasks that are blocked waiting for the bits.

Parameters

xEventGroup	The event group in which the bits are to be set.
uxBitsToSet	A bitwise value that indicates the bit or bits to set. For example, to set bit 3 only, set uxBitsToSet to 0x08. To set bit 3 and bit 0 set uxBitsToSet to 0x09.

Returns

The value of the event group at the time the call to xEventGroupSetBits() returns. There are two reasons why the returned value might have the bits specified by the uxBitsToSet parameter cleared. First, if setting a bit results in a task that was waiting for the bit leaving the blocked state then it is possible the bit will be cleared automatically (see the xClearBitOnExit parameter of xEventGroupWaitBits()). Second, any unblocked (or otherwise Ready state) task that has a priority above that of the task that called xEventGroupSetBits() will execute and may change the event group value before the call to xEventGroupSetBits() returns.

Example usage:

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )


void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;


  // Set bit 0 and bit 4 in xEventGroup.
  uxBits = xEventGroupSetBits(
                      xEventGroup,    // The event group being updated.
                      BIT_0 | BIT_4 );// The bits being set.
```

```
    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        // Both bit 0 and bit 4 remained set when the function returned.
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // Bit 0 remained set when the function returned, but bit 4 was
        // cleared.  It might be that bit 4 was cleared automatically as a
        // task that was waiting for bit 4 was removed from the Blocked
        // state.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // Bit 4 remained set when the function returned, but bit 0 was
        // cleared.  It might be that bit 0 was cleared automatically as a
        // task that was waiting for bit 0 was removed from the Blocked
        // state.
    }
    else
    {
        // Neither bit 0 nor bit 4 remained set.  It might be that a task
        // was waiting for both of the bits to be set, and the bits were
        // cleared as the task left the Blocked state.
    }
}
```

5.17 xEventGroupSetBitsFromISR

Collaboration diagram for xEventGroupSetBitsFromISR:

event_groups.h

```
BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup, const EventBits_t uxBits
```

A version of xEventGroupSetBits() that can be called from an interrupt.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow nondeterministic operations to be performed in interrupts or from critical sections. Therefore xEventGroupSetBitsFromISR() sends a message to the timer task to have the set operation performed in the context of the timer task - where a scheduler lock is used in place of a critical section.

Parameters

xEventGroup	The event group in which the bits are to be set.
uxBitsToSet	A bitwise value that indicates the bit or bits to set. For example, to set bit 3 only, set uxBitsToSet to 0x08. To set bit 3 and bit 0 set uxBitsToSet to 0x09.
pxHigherPriorityTaskWoken	As mentioned above, calling this function will result in a message being sent to the timer daemon task. If the priority of the timer daemon task is higher than the priority of the currently running task (the task the interrupt interrupted) then ∗pxHigherPriorityTaskWoken will be set to pdTRUE by xEventGroupSetBitsFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason ∗pxHigherPriorityTaskWoken must be initialised to pdFALSE. See the example code below.

Returns

If the request to execute the function was posted successfully then pdPASS is returned, otherwise pdFALSE is returned. pdFALSE will be returned if the timer service queue was full.

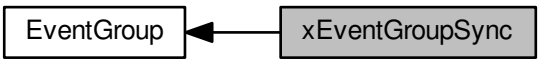
Example usage:

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )


// An event group which it is assumed has already been created by a call to
// xEventGroupCreate().
EventGroupHandle_t xEventGroup;
```

```
  void anInterruptHandler( void )
  {
  BaseType_t xHigherPriorityTaskWoken, xResult;


    // xHigherPriorityTaskWoken must be initialised to pdFALSE.
    xHigherPriorityTaskWoken = pdFALSE;


    // Set bit 0 and bit 4 in xEventGroup.
    xResult = xEventGroupSetBitsFromISR(
                        xEventGroup,    // The event group being updated.
                        BIT_0 | BIT_4   // The bits being set.
                        &xHigherPriorityTaskWoken );


    // Was the message posted successfully?
    if( xResult == pdPASS )
    {
        // If xHigherPriorityTaskWoken is now set to pdTRUE then a context
        // switch should be requested.  The macro used is port specific and
        // will be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() -
        // refer to the documentation page for the port being used.
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    }
}
```

5.18 xEventGroupSync

Collaboration diagram for xEventGroupSync:



event_groups.h

```
EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup,
                             const EventBits_t uxBitsToSet,
                             const EventBits_t uxBitsToWaitFor,
                             TickType_t xTicksToWait );
```

Atomically set bits within an event group, then wait for a combination of bits to be set within the same event group. This functionality is typically used to synchronise multiple tasks, where each task has to wait for the other tasks to reach a synchronisation point before proceeding.

This function cannot be used from an interrupt.

The function will return before its block time expires if the bits specified by the uxBitsToWait parameter are set, or become set within that time. In this case all the bits specified by uxBitsToWait will be automatically cleared before the function returns.

Parameters

xEventGroup	The event group in which the bits are being tested. The event group must have previously been created using a call to xEventGroupCreate().
uxBitsToSet	The bits to set in the event group before determining if, and possibly waiting for, all the bits specified by the uxBitsToWait parameter are set.
uxBitsToWaitFor	A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and bit 2 set uxBitsToWaitFor to 0x05. To wait for bits 0 and bit 1 and bit 2 set uxBitsToWaitFor to 0x07. Etc.
xTicksToWait	The maximum amount of time (specified in 'ticks') to wait for all of the bits specified by uxBitsToWaitFor to become set.

Returns

The value of the event group at the time either the bits being waited for became set, or the block time expired. Test the return value to know which bits were set. If xEventGroupSync() returned because its timeout expired then not all the bits being waited for will be set. If xEventGroupSync() returned because all the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared.

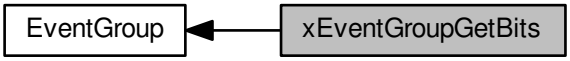
Example usage:

```
// Bits used by the three tasks.
#define TASK_0_BIT      ( 1 << 0 )
#define TASK_1_BIT      ( 1 << 1 )
#define TASK_2_BIT      ( 1 << 2 )


#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )


// Use an event group to synchronise three tasks.  It is assumed this event
// group has already been created elsewhere.
EventGroupHandle_t xEventBits;


void vTask0( void *pvParameters )
{
EventBits_t uxReturn;
TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

for( ;; )
{
// Perform task functionality here.

// Set bit 0 in the event flag to note this task has reached the
// sync point.  The other two tasks will set the other two bits defined
// by ALL_SYNC_BITS.  All three tasks have reached the synchronisation
// point when all the ALL_SYNC_BITS are set.  Wait a maximum of 100ms
// for this to happen.
uxReturn = xEventGroupSync( xEventBits, TASK_0_BIT, ALL_SYNC_BITS, xTicksToWait );

if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )
{
    // All three tasks reached the synchronisation point before the call
    // to xEventGroupSync() timed out.
}


   }
}


void vTask1( void *pvParameters )
{
    for( ;; )
    {
    // Perform task functionality here.


    // Set bit 1 in the event flag to note this task has reached the
    // synchronisation point.  The other two tasks will set the other two
    // bits defined by ALL_SYNC_BITS.  All three tasks have reached the
    // synchronisation point when all the ALL_SYNC_BITS are set.  Wait
    // indefinitely for this to happen.
    xEventGroupSync( xEventBits, TASK_1_BIT, ALL_SYNC_BITS, portMAX_DELAY );


    // xEventGroupSync() was called with an indefinite block time, so
    // this task will only reach here if the syncrhonisation was made by all
    // three tasks, so there is no need to test the return value.
    }
}


void vTask2( void *pvParameters )
{
    for( ;; )
    {
    // Perform task functionality here.
```

```
    // Set bit 2 in the event flag to note this task has reached the
    // synchronisation point.  The other two tasks will set the other two
    // bits defined by ALL_SYNC_BITS.  All three tasks have reached the
    // synchronisation point when all the ALL_SYNC_BITS are set.  Wait
    // indefinitely for this to happen.
    xEventGroupSync( xEventBits, TASK_2_BIT, ALL_SYNC_BITS, portMAX_DELAY );


    // xEventGroupSync() was called with an indefinite block time, so
    // this task will only reach here if the syncrhonisation was made by all
    // three tasks, so there is no need to test the return value.
   }
}
```

5.19 xEventGroupGetBits

Collaboration diagram for xEventGroupGetBits:



event_groups.h

```
EventBits_t xEventGroupGetBits( EventGroupHandle_t xEventGroup );
```

Returns the current value of the bits in an event group. This function cannot be used from an interrupt.

Parameters

xEventGroup	The event group being queried.

Returns

The event group bits at the time xEventGroupGetBits() was called.

5.20 xEventGroupGetBitsFromISR

Collaboration diagram for xEventGroupGetBitsFromISR:

event_groups.h

```
EventBits_t xEventGroupGetBitsFromISR( EventGroupHandle_t xEventGroup );
```

A version of xEventGroupGetBits() that can be called from an ISR.

Parameters

xEventGroup	The event group being queried.

Returns

The event group bits at the time xEventGroupGetBitsFromISR() was called.

5.21 xQueueCreate

queue. h

```
QueueHandle_t xQueueCreate(
                          UBaseType_t uxQueueLength,
                          UBaseType_t uxItemSize
                      );
```

Creates a new queue instance, and returns a handle by which the new queue can be referenced.

Internally, within the FreeRTOS implementation, queues use two blocks of memory. The first block is used to hold the queue's data structures. The second block is used to hold items placed into the queue. If a queue is created using xQueueCreate() then both blocks of memory are automatically dynamically allocated inside the xQueueCreate() function. (see http://www.freertos.org/a00111.html). If a queue is created using xQueueCreate↩ Static() then the application writer must provide the memory that will get used by the queue. xQueueCreateStatic() therefore allows a queue to be created without using any dynamic memory allocation.

http://www.FreeRTOS.org/Embedded-RTOS-Queues.html

Parameters

uxQueueLength	The maximum number of items that the queue can contain.
uxItemSize	The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

Returns

If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

Example usage:

```
struct AMessage
{
   char ucMessageID;
   char ucData[ 20 ];
};


void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;


   // Create a queue capable of containing 10 uint32_t values.
   xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );
   if( xQueue1 == 0 )
   {
    // Queue was not created and must not be used.
   }
```

```
   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue2 == 0 )
   {
    // Queue was not created and must not be used.
   }


   // ... Rest of task code.
}
```

5.22 xQueueCreateStatic

queue. h

```
QueueHandle_t xQueueCreateStatic(
                          UBaseType_t uxQueueLength,
                          UBaseType_t uxItemSize,
                          uint8_t *pucQueueStorageBuffer,
                          StaticQueue_t *pxQueueBuffer
                      );
```

Creates a new queue instance, and returns a handle by which the new queue can be referenced.

Internally, within the FreeRTOS implementation, queues use two blocks of memory. The first block is used to hold the queue's data structures. The second block is used to hold items placed into the queue. If a queue is created using xQueueCreate() then both blocks of memory are automatically dynamically allocated inside the xQueueCreate() function. (see http://www.freertos.org/a00111.html). If a queue is created using xQueueCreate↩ Static() then the application writer must provide the memory that will get used by the queue. xQueueCreateStatic() therefore allows a queue to be created without using any dynamic memory allocation.

http://www.FreeRTOS.org/Embedded-RTOS-Queues.html

Parameters

uxQueueLength	The maximum number of items that the queue can contain.
uxItemSize	The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.
pucQueueStorageBuffer	If uxItemSize is not zero then pucQueueStorageBuffer must point to a uint8_t array that is at least large enough to hold the maximum number of items that can be in the queue at any one time - which is (uxQueueLength * uxItemsSize) bytes. If uxItemSize is zero then pucQueueStorageBuffer can be NULL.
pxQueueBuffer	Must point to a variable of type StaticQueue_t, which will be used to hold the queue's data structure.

Returns

If the queue is created then a handle to the created queue is returned. If pxQueueBuffer is NULL then NULL is returned.

Example usage:

```
struct AMessage
{
   char ucMessageID;
   char ucData[ 20 ];
};


#define QUEUE_LENGTH 10
#define ITEM_SIZE sizeof( uint32_t )


// xQueueBuffer will hold the queue structure.
StaticQueue_t xQueueBuffer;
```

```
// ucQueueStorage will hold the items posted to the queue.  Must be at least
// [(queue length) * ( queue item size)] bytes long.
uint8_t ucQueueStorage[ QUEUE_LENGTH * ITEM_SIZE ];


void vATask( void *pvParameters )
{
QueueHandle_t xQueue1;


   // Create a queue capable of containing 10 uint32_t values.
   xQueue1 = xQueueCreate( QUEUE_LENGTH, // The number of items the queue can hold.
                           ITEM_SIZE     // The size of each item in the queue
                           &( ucQueueStorage[ 0 ] ), // The buffer that will hold the items in the qu
                           &xQueueBuffer ); // The buffer that will hold the queue structure.


   // The queue is guaranteed to be created successfully as no dynamic memory
   // allocation is used.  Therefore xQueue1 is now a handle to a valid queue.


   // ... Rest of task code.
}
```

5.23 xQueueSend

queue. h

```
BaseType_t xQueueSendToToFront(
                                QueueHandle_t    xQueue,
                                const void       *pvItemToQueue,
                                TickType_t       xTicksToWait
                            );
```

This is a macro that calls xQueueGenericSend().

Post an item to the front of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xTicksToWait	The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

Returns

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

```
struct AMessage
{
   char ucMessageID;
   char ucData[ 20 ];
} xMessage;


uint32_t ulVar = 10UL;


void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;


   // Create a queue capable of containing 10 uint32_t values.
   xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );


   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
```

```
    // ...


    if( xQueue1 != 0 )
    {
     // Send an uint32_t.  Wait for 10 ticks for space to become
     // available if necessary.
     if( xQueueSendToFront( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
     {
         // Failed to post the message, even after 10 ticks.
     }
    }


    if( xQueue2 != 0 )
    {
     // Send a pointer to a struct AMessage object.  Don't block if the
     // queue is already full.
     pxMessage = & xMessage;
     xQueueSendToFront( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
    }


    // ... Rest of task code.
}
```

queue. h

```
BaseType_t xQueueSendToBack(
                                QueueHandle_t    xQueue,
                                const void       *pvItemToQueue,
                                TickType_t       xTicksToWait
                            );
```

This is a macro that calls xQueueGenericSend().

Post an item to the back of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xTicksToWait	The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

Returns

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

```
struct AMessage
{
   char ucMessageID;
   char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

   // Create a queue capable of containing 10 uint32_t values.
   xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

   // ...

   if( xQueue1 != 0 )
   {
    // Send an uint32_t.  Wait for 10 ticks for space to become
    // available if necessary.
    if( xQueueSendToBack( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
    {
        // Failed to post the message, even after 10 ticks.
    }
   }

   if( xQueue2 != 0 )
   {
    // Send a pointer to a struct AMessage object.  Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSendToBack( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
   }

   // ... Rest of task code.
}
```

queue. h

```
BaseType_t xQueueSend(
                          QueueHandle_t xQueue,
                          const void * pvItemToQueue,
                          TickType_t xTicksToWait
                      );
```

This is a macro that calls xQueueGenericSend(). It is included for backward compatibility with versions of Free↩ RTOS.org that did not include the xQueueSendToFront() and xQueueSendToBack() macros. It is equivalent to xQueueSendToBack().

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xTicksToWait	The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

Returns

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

```
struct AMessage
{
   char ucMessageID;
   char ucData[ 20 ];
} xMessage;


uint32_t ulVar = 10UL;


void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;


   // Create a queue capable of containing 10 uint32_t values.
   xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );


   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );


   // ...


   if( xQueue1 != 0 )
   {
    // Send an uint32_t.  Wait for 10 ticks for space to become
    // available if necessary.
    if( xQueueSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
    {
        // Failed to post the message, even after 10 ticks.
    }
   }


   if( xQueue2 != 0 )
   {
    // Send a pointer to a struct AMessage object.  Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
   }
```

```
   // ... Rest of task code.
}
```

queue. h

```
BaseType_t xQueueGenericSend(
                                    QueueHandle_t xQueue,
                                    const void * pvItemToQueue,
                                    TickType_t xTicksToWait
                                    BaseType_t xCopyPosition
                                );
```

It is preferred that the macros xQueueSend(), xQueueSendToFront() and xQueueSendToBack() are used in place of calling this function directly.

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xTicksToWait	The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.
xCopyPosition	Can take the value queueSEND_TO_BACK to place the item at the back of the queue, or queueSEND_TO_FRONT to place the item at the front of the queue (for high priority messages).

Returns

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

```
struct AMessage
{
   char ucMessageID;
   char ucData[ 20 ];
} xMessage;


uint32_t ulVar = 10UL;


void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;
```

```
   // Create a queue capable of containing 10 uint32_t values.
   xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );


   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );


   // ...


   if( xQueue1 != 0 )
   {
    // Send an uint32_t.  Wait for 10 ticks for space to become
    // available if necessary.
    if( xQueueGenericSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10, queueSEND_TO_BACK ) != p
    {
        // Failed to post the message, even after 10 ticks.
    }
   }


   if( xQueue2 != 0 )
   {
    // Send a pointer to a struct AMessage object.  Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueGenericSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0, queueSEND_TO_BACK );
   }


   // ... Rest of task code.
}
```

5.24 xQueueOverwrite

queue. h

```
BaseType_t xQueueOverwrite(
                              QueueHandle_t xQueue,
                              const void * pvItemToQueue
                          );
```

Only for use with queues that have a length of one - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

This function must not be called from an interrupt service routine. See xQueueOverwriteFromISR () for an alternative which may be used in an ISR.

Parameters

xQueue	The handle of the queue to which the data is being sent.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.

Returns

xQueueOverwrite() is a macro that calls xQueueGenericSend(), and therefore has the same return values as xQueueSendToFront(). However, pdPASS is the only value that can be returned because xQueueOverwrite() will write to the queue even when the queue is already full.

Example usage:

```
void vFunction( void *pvParameters )
{
QueueHandle_t xQueue;
uint32_t ulVarToSend, ulValReceived;

    // Create a queue to hold one uint32_t value.  It is strongly
    // recommended  not to use xQueueOverwrite() on queues that can
    // contain more than one value, and doing so will trigger an assertion
    // if configASSERT() is defined.
    xQueue = xQueueCreate( 1, sizeof( uint32_t ) );

    // Write the value 10 to the queue using xQueueOverwrite().
    ulVarToSend = 10;
    xQueueOverwrite( xQueue, &ulVarToSend );

    // Peeking the queue should now return 10, but leave the value 10 in
    // the queue.  A block time of zero is used as it is known that the
    // queue holds a value.
    ulValReceived = 0;
    xQueuePeek( xQueue, &ulValReceived, 0 );
```

```
    if( ulValReceived != 10 )
    {
        // Error unless the item was removed by a different task.
    }


    // The queue is still full.  Use xQueueOverwrite() to overwrite the
    // value held in the queue with 100.
    ulVarToSend = 100;
    xQueueOverwrite( xQueue, &ulVarToSend );


    // This time read from the queue, leaving the queue empty once more.
    // A block time of 0 is used again.
    xQueueReceive( xQueue, &ulValReceived, 0 );


    // The value read should be the last value written, even though the
    // queue was already full when the value was written.
    if( ulValReceived != 100 )
    {
        // Error!
    }


    // ...
}
```

5.25 xQueueReceive

queue. h

```
BaseType_t xQueuePeek(
                          QueueHandle_t xQueue,
                          void *pvBuffer,
                          TickType_t xTicksToWait
                      );
```

This is a macro that calls the xQueueGenericReceive() function.

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to xQueue↩ Receive().

This macro must not be used in an interrupt service routine. See xQueuePeekFromISR() for an alternative that can be called from an interrupt service routine.

Parameters

xQueue	The handle to the queue from which the item is to be received.
pvBuffer	Pointer to the buffer into which the received item will be copied.
xTicksToWait	The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. xQueuePeek() will return immediately if xTicksToWait is 0 and the queue is empty.

Returns

pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Example usage:

```
struct AMessage
{
   char ucMessageID;
   char ucData[ 20 ];
} xMessage;


QueueHandle_t xQueue;


// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
struct AMessage *pxMessage;
```

```
   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue == 0 )
   {
    // Failed to create the queue.
   }


   // ...


   // Send a pointer to a struct AMessage object.  Don't block if the
   // queue is already full.
   pxMessage = & xMessage;
   xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );


   // ... Rest of task code.
}


// Task to peek the data from the queue.
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;


   if( xQueue != 0 )
   {
    // Peek a message on the created queue.  Block for 10 ticks if a
    // message is not immediately available.
    if( xQueuePeek( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
    {
        // pcRxedMessage now points to the struct AMessage variable posted
        // by vATask, but the item still remains on the queue.
    }
   }


   // ... Rest of task code.
}
```

queue. h

```
BaseType_t xQueueReceive(
                            QueueHandle_t xQueue,
                            void *pvBuffer,
                            TickType_t xTicksToWait
                        );
```

This is a macro that calls the xQueueGenericReceive() function.

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items are removed from the queue.

This function must not be used in an interrupt service routine. See xQueueReceiveFromISR for an alternative that can.

Parameters

xQueue	The handle to the queue from which the item is to be received.
pvBuffer	Pointer to the buffer into which the received item will be copied.
xTicksToWait	The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. xQueueReceive() will return immediately if xTicksToWait is zero and the queue is empty. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

Returns

pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Example usage:

```
struct AMessage
{
   char ucMessageID;
   char ucData[ 20 ];
} xMessage;


QueueHandle_t xQueue;


// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
struct AMessage *pxMessage;


   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue == 0 )
   {
    // Failed to create the queue.
   }


   // ...


   // Send a pointer to a struct AMessage object.  Don't block if the
   // queue is already full.
   pxMessage = & xMessage;
   xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );


   // ... Rest of task code.
}


// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;
```

```
   if( xQueue != 0 )
   {
    // Receive a message on the created queue.  Block for 10 ticks if a
    // message is not immediately available.
    if( xQueueReceive( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
    {
        // pcRxedMessage now points to the struct AMessage variable posted
        // by vATask.
    }
   }


   // ... Rest of task code.
}
```

queue. h

```
BaseType_t xQueueGenericReceive(
                                    QueueHandle_t    xQueue,
                                    void *pvBuffer,
                                    TickType_t   xTicksToWait
                                    BaseType_t   xJustPeek
                                );
```

It is preferred that the macro xQueueReceive() be used rather than calling this function directly.

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

This function must not be used in an interrupt service routine. See xQueueReceiveFromISR for an alternative that can.

Parameters

xQueue	The handle to the queue from which the item is to be received.
pvBuffer	Pointer to the buffer into which the received item will be copied.
xTicksToWait	The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. xQueueGenericReceive() will return immediately if the queue is empty and xTicksToWait is 0.
xJustPeek	When set to true, the item received from the queue is not actually removed from the queue - meaning a subsequent call to xQueueReceive() will return the same item. When set to false, the item being received from the queue is also removed from the queue.

Returns

pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Example usage:

```
struct AMessage
{
   char ucMessageID;
   char ucData[ 20 ];
} xMessage;
```

```
QueueHandle_t xQueue;


// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
struct AMessage *pxMessage;


   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue == 0 )
   {
    // Failed to create the queue.
   }


   // ...


   // Send a pointer to a struct AMessage object.  Don't block if the
   // queue is already full.
   pxMessage = & xMessage;
   xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );


   // ... Rest of task code.
}


// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;


   if( xQueue != 0 )
   {
    // Receive a message on the created queue.  Block for 10 ticks if a
    // message is not immediately available.
    if( xQueueGenericReceive( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
    {
        // pcRxedMessage now points to the struct AMessage variable posted
        // by vATask.
    }
   }


   // ... Rest of task code.
}
```

5.26 xQueuePeekFromISR

queue. h

```
BaseType_t xQueuePeekFromISR(
                                    QueueHandle_t xQueue,
                                    void *pvBuffer,
                                );
```

A version of xQueuePeek() that can be called from an interrupt service routine (ISR).

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to xQueue↩ Receive().

Parameters

xQueue	The handle to the queue from which the item is to be received.
pvBuffer	Pointer to the buffer into which the received item will be copied.

Returns

pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

5.27 uxQueueMessagesWaiting

queue. h

```
UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue );
```

Return the number of messages stored in a queue.

Parameters

xQueue	A handle to the queue being queried.

Returns

The number of messages available in the queue.

queue. h

```
UBaseType_t uxQueueSpacesAvailable( const QueueHandle_t xQueue );
```

Return the number of free spaces available in a queue. This is equal to the number of items that can be sent to the queue before the queue becomes full if no items are removed.

Parameters

xQueue	A handle to the queue being queried.

Returns

The number of spaces available in the queue.

5.28 vQueueDelete

queue. h

```
void vQueueDelete( QueueHandle_t xQueue );
```

Delete a queue - freeing all the memory allocated for storing of items placed on the queue.

Parameters

xQueue	A handle to the queue to be deleted.

5.29 xQueueSendFromISR

queue. h

```
BaseType_t xQueueSendToFrontFromISR(
                                      QueueHandle_t xQueue,
                                      const void *pvItemToQueue,
                                      BaseType_t *pxHigherPriorityTaskWoken
                                    );
```

This is a macro that calls xQueueGenericSendFromISR().

Post an item to the front of a queue. It is safe to use this macro from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueSendToFrontFromISR() will set ∗pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendToFromFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns

pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPrioritTaskWoken;


   // We have not woken a task at the start of the ISR.
   xHigherPriorityTaskWoken = pdFALSE;


   // Loop until the buffer is empty.
   do
   {
    // Obtain a byte from the buffer.
    cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );


    // Post the byte.
    xQueueSendToFrontFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );
```

```
    } while( portINPUT_BYTE( BUFFER_COUNT ) );


    // Now the buffer is empty we can switch context if necessary.
    if( xHigherPriorityTaskWoken )
    {
     taskYIELD ();
    }
}
```

queue. h

```
BaseType_t xQueueSendToBackFromISR(
                                      QueueHandle_t xQueue,
                                      const void *pvItemToQueue,
                                      BaseType_t *pxHigherPriorityTaskWoken
                                   );
```

This is a macro that calls xQueueGenericSendFromISR().

Post an item to the back of a queue. It is safe to use this macro from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueSendToBackFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendToBackFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns

pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWoken;


    // We have not woken a task at the start of the ISR.
    xHigherPriorityTaskWoken = pdFALSE;


    // Loop until the buffer is empty.
    do
    {
     // Obtain a byte from the buffer.
     cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );
```

```
    // Post the byte.
    xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );


   } while( portINPUT_BYTE( BUFFER_COUNT ) );


   // Now the buffer is empty we can switch context if necessary.
   if( xHigherPriorityTaskWoken )
   {
    taskYIELD ();
   }
}
```

queue. h

```
BaseType_t xQueueSendFromISR(
                                    QueueHandle_t xQueue,
                                    const void *pvItemToQueue,
                                    BaseType_t *pxHigherPriorityTaskWoken
                                );
```

This is a macro that calls xQueueGenericSendFromISR(). It is included for backward compatibility with versions of FreeRTOS.org that did not include the xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() macros.

Post an item to the back of a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueSendFromISR() will set ∗pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns

pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWoken;


   // We have not woken a task at the start of the ISR.
   xHigherPriorityTaskWoken = pdFALSE;
```

```
    // Loop until the buffer is empty.
    do
    {
     // Obtain a byte from the buffer.
     cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );


     // Post the byte.
     xQueueSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );


    } while( portINPUT_BYTE( BUFFER_COUNT ) );


    // Now the buffer is empty we can switch context if necessary.
    if( xHigherPriorityTaskWoken )
    {
     // Actual macro used here is port specific.
     portYIELD_FROM_ISR ();
    }
}
```

queue. h

```
BaseType_t xQueueGenericSendFromISR(
                                        QueueHandle_t        xQueue,
                                        const    void    *pvItemToQueue,
                                        BaseType_t   *pxHigherPriorityTaskWoken,
                                        BaseType_t   xCopyPosition
                                    );
```

It is preferred that the macros xQueueSendFromISR(), xQueueSendToFrontFromISR() and xQueueSendToBack↩FromISR() be used in place of calling this function directly. xQueueGiveFromISR() is an equivalent for use by semaphores that don't actually copy any data.

Post an item on a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueGenericSendFromISR() will set ∗pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueGenericSendFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.
xCopyPosition	Can take the value queueSEND_TO_BACK to place the item at the back of the queue, or queueSEND_TO_FRONT to place the item at the front of the queue (for high priority messages).

Returns

pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWokenByPost;


   // We have not woken a task at the start of the ISR.
   xHigherPriorityTaskWokenByPost = pdFALSE;


   // Loop until the buffer is empty.
   do
   {
    // Obtain a byte from the buffer.
    cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );


    // Post each byte.
    xQueueGenericSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWokenByPost, queueSEND_TO_BACK )


   } while( portINPUT_BYTE( BUFFER_COUNT ) );


   // Now the buffer is empty we can switch context if necessary.  Note that the
   // name of the yield function required is port specific.
   if( xHigherPriorityTaskWokenByPost )
   {
    taskYIELD_YIELD_FROM_ISR();
   }
}
```

5.30 xQueueOverwriteFromISR

queue. h

```
BaseType_t xQueueOverwriteFromISR(
                              QueueHandle_t xQueue,
                              const void * pvItemToQueue,
                              BaseType_t *pxHigherPriorityTaskWoken
                          );
```

A version of xQueueOverwrite() that can be used in an interrupt service routine (ISR).

Only for use with queues that can hold a single item - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

Parameters

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueOverwriteFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueOverwriteFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns

xQueueOverwriteFromISR() is a macro that calls xQueueGenericSendFromISR(), and therefore has the same return values as xQueueSendToFrontFromISR(). However, pdPASS is the only value that can be returned because xQueueOverwriteFromISR() will write to the queue even when the queue is already full.

Example usage:

```
QueueHandle_t xQueue;


void vFunction( void <em>pvParameters )
{
   // Create a queue to hold one uint32_t value.  It is strongly
   // recommended *not to use xQueueOverwriteFromISR() on queues that can
   // contain more than one value, and doing so will trigger an assertion
   // if configASSERT() is defined.
   xQueue = xQueueCreate( 1, sizeof( uint32_t ) );
}
```

```
void vAnInterruptHandler( void )
{
// xHigherPriorityTaskWoken must be set to pdFALSE before it is used.
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
uint32_t ulVarToSend, ulValReceived;

// Write the value 10 to the queue using xQueueOverwriteFromISR().
ulVarToSend = 10;
xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

// The queue is full, but calling xQueueOverwriteFromISR() again will still
// pass because the value held in the queue will be overwritten with the
// new value.
ulVarToSend = 100;
xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

// Reading from the queue will now return 100.

// ...

if( xHigherPrioritytaskWoken == pdTRUE )
{
    // Writing to the queue caused a task to unblock and the unblocked task
    // has a priority higher than or equal to the priority of the currently
    // executing task (the task this interrupt interrupted).  Perform a context
    // switch so this interrupt returns directly to the unblocked task.
    portYIELD_FROM_ISR(); // or portEND_SWITCHING_ISR() depending on the port.
}


}
```

5.31 xQueueReceiveFromISR

queue. h

```
BaseType_t xQueueReceiveFromISR(
                                    QueueHandle_t    xQueue,
                                    void *pvBuffer,
                                    BaseType_t *pxTaskWoken
                                );
```

Receive an item from a queue. It is safe to use this function from within an interrupt service routine.

Parameters

xQueue	The handle to the queue from which the item is to be received.
pvBuffer	Pointer to the buffer into which the received item will be copied.
pxTaskWoken	A task may be blocked waiting for space to become available on the queue. If xQueueReceiveFromISR causes such a task to unblock ∗pxTaskWoken will get set to pdTRUE, otherwise ∗pxTaskWoken will remain unchanged.

Returns

pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Example usage:

```
QueueHandle_t xQueue;


// Function to create a queue and post some values.
void vAFunction( void *pvParameters )
{
char cValueToPost;
const TickType_t xTicksToWait = ( TickType_t )0xff;


   // Create a queue capable of containing 10 characters.
   xQueue = xQueueCreate( 10, sizeof( char ) );
   if( xQueue == 0 )
   {
    // Failed to create the queue.
   }


   // ...


   // Post some characters that will be used within an ISR.  If the queue
   // is full then this task will block for xTicksToWait ticks.
   cValueToPost = 'a';
   xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
   cValueToPost = 'b';
   xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
```

```
   // ... keep posting characters ... this task may block when the queue
   // becomes full.


   cValueToPost = 'c';
   xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
}


// ISR that outputs all the characters received on the queue.
void vISR_Routine( void )
{
BaseType_t xTaskWokenByReceive = pdFALSE;
char cRxedChar;


   while( xQueueReceiveFromISR( xQueue, ( void * ) &cRxedChar, &xTaskWokenByReceive) )
   {
    // A character was received.  Output the character now.
    vOutputCharacter( cRxedChar );


    // If removing the character from the queue woke the task that was
    // posting onto the queue cTaskWokenByReceive will have been set to
    // pdTRUE.  No matter how many times this loop iterates only one
    // task will be woken.
   }


   if( cTaskWokenByPost != ( char ) pdFALSE;
   {
    taskYIELD ();
   }
}
```

5.32 vSemaphoreCreateBinary

semphr. h

```
vSemaphoreCreateBinary( SemaphoreHandle_t xSemaphore )
```

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! http://www.freertos.org/RTOS-task-notifications.html

This old vSemaphoreCreateBinary() macro is now deprecated in favour of the xSemaphoreCreateBinary() function. Note that binary semaphores created using the vSemaphoreCreateBinary() macro are created in a state such that the first call to 'take' the semaphore would pass, whereas binary semaphores created using xSemaphoreCreate↩ Binary() are created in a state such that the the semaphore must first be 'given' before it can be 'taken'.

Macro that implements a semaphore by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as we don't want to actually store any data - we just want to know if the queue is empty or full.

This type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task. The semaphore need not be given back once obtained, so one task/interrupt can continuously 'give' the semaphore while another continuously 'takes' the semaphore. For this reason this type of semaphore does not use a priority inheritance mechanism. For an alternative that does use priority inheritance see xSemaphoreCreateMutex().

Parameters

xSemaphore	Handle to the created semaphore. Should be of type SemaphoreHandle_t.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;


void vATask( void * pvParameters )
{
   // Semaphore cannot be used before a call to vSemaphoreCreateBinary ().
   // This is a macro so pass the variable in directly.
   vSemaphoreCreateBinary( xSemaphore );


   if( xSemaphore != NULL )
   {
       // The semaphore was created successfully.
       // The semaphore can now be used.
   }
}
```

5.33 xSemaphoreCreateBinary

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateBinary( void )
```

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! http://www.freertos.org/RTOS-task-notifications.html

Internally, within the FreeRTOS implementation, binary semaphores use a block of memory, in which the semaphore structure is stored. If a binary semaphore is created using xSemaphoreCreateBinary() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateBinary() function. (see http://www.freertos.org/a00111.html). If a binary semaphore is created using xSemaphoreCreateBinaryStatic() then the application writer must provide the memory. xSemaphoreCreateBinaryStatic() therefore allows a binary semaphore to be created without using any dynamic memory allocation.

The old vSemaphoreCreateBinary() macro is now deprecated in favour of this xSemaphoreCreateBinary() function. Note that binary semaphores created using the vSemaphoreCreateBinary() macro are created in a state such that the first call to 'take' the semaphore would pass, whereas binary semaphores created using xSemaphoreCreateBinary() are created in a state such that the the semaphore must first be 'given' before it can be 'taken'.

This type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task. The semaphore need not be given back once obtained, so one task/interrupt can continuously 'give' the semaphore while another continuously 'takes' the semaphore. For this reason this type of semaphore does not use a priority inheritance mechanism. For an alternative that does use priority inheritance see xSemaphoreCreateMutex().

Returns

Handle to the created semaphore, or NULL if the memory required to hold the semaphore's data structures could not be allocated.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;


void vATask( void * pvParameters )
{
   // Semaphore cannot be used before a call to xSemaphoreCreateBinary().
   // This is a macro so pass the variable in directly.
   xSemaphore = xSemaphoreCreateBinary();


   if( xSemaphore != NULL )
   {
       // The semaphore was created successfully.
       // The semaphore can now be used.
   }
}
```

5.34 xSemaphoreCreateBinaryStatic

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateBinaryStatic( StaticSemaphore_t *pxSemaphoreBuffer )
```

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

NOTE: In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! http://www.freertos.org/RTOS-task-notifications.html

Internally, within the FreeRTOS implementation, binary semaphores use a block of memory, in which the semaphore structure is stored. If a binary semaphore is created using xSemaphoreCreateBinary() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateBinary() function. (see http://www.freertos.org/a00111.html). If a binary semaphore is created using xSemaphoreCreateBinaryStatic() then the application writer must provide the memory. xSemaphoreCreateBinaryStatic() therefore allows a binary semaphore to be created without using any dynamic memory allocation.

This type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task. The semaphore need not be given back once obtained, so one task/interrupt can continuously 'give' the semaphore while another continuously 'takes' the semaphore. For this reason this type of semaphore does not use a priority inheritance mechanism. For an alternative that does use priority inheritance see xSemaphoreCreateMutex().

Parameters

pxSemaphoreBuffer	Must point to a variable of type StaticSemaphore_t, which will then be used to hold the semaphore's data structure, removing the need for the memory to be allocated dynamically.

Returns

If the semaphore is created then a handle to the created semaphore is returned. If pxSemaphoreBuffer is NULL then NULL is returned.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;
StaticSemaphore_t xSemaphoreBuffer;


void vATask( void * pvParameters )
{
   // Semaphore cannot be used before a call to xSemaphoreCreateBinary().
   // The semaphore's data structures will be placed in the xSemaphoreBuffer
   // variable, the address of which is passed into the function.  The
   // function's parameter is not NULL, so the function will not attempt any
   // dynamic memory allocation, and therefore the function will not return
   // return NULL.
   xSemaphore = xSemaphoreCreateBinary( &xSemaphoreBuffer );


   // Rest of task code goes here.
}
```

5.35 xSemaphoreTake

semphr. h

```
xSemaphoreTake(
                    SemaphoreHandle_t xSemaphore,
                    TickType_t xBlockTime
                )
```

Macro to obtain a semaphore. The semaphore must have previously been created with a call to xSemaphore↩ CreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting().

Parameters

xSemaphore	A handle to the semaphore being taken - obtained when the semaphore was created.
xBlockTime	The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. A block time of portMAX_DELAY can be used to block indefinitely (provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h).

Returns

pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;


// A task that creates a semaphore.
void vATask( void * pvParameters )
{
   // Create the semaphore to guard a shared resource.
   xSemaphore = xSemaphoreCreateBinary();
}


// A task that uses the semaphore.
void vAnotherTask( void * pvParameters )
{
   // ... Do other things.


   if( xSemaphore != NULL )
   {
       // See if we can obtain the semaphore.  If the semaphore is not available
       // wait 10 ticks to see if it becomes free.
       if( xSemaphoreTake( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
       {
           // We were able to obtain the semaphore and can now access the
           // shared resource.


           // ...
```

```
            // We have finished accessing the shared resource.  Release the
            // semaphore.
            xSemaphoreGive( xSemaphore );
        }
        else
        {
            // We could not obtain the semaphore and can therefore not access
            // the shared resource safely.
        }
    }
}
```

5.36 xSemaphoreTakeRecursive

semphr. h xSemaphoreTakeRecursive(SemaphoreHandle_t xMutex, TickType_t xBlockTime)

Macro to recursively obtain, or 'take', a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex();

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.

This macro must not be used on mutexes created using xSemaphoreCreateMutex().

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

Parameters

xMutex	A handle to the mutex being obtained. This is the handle returned by xSemaphoreCreateRecursiveMutex();
xBlockTime	The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. If the task already owns the semaphore then xSemaphoreTakeRecursive() will return immediately no matter what the value of xBlockTime.

Returns

pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

Example usage:

```
SemaphoreHandle_t xMutex = NULL;


// A task that creates a mutex.
void vATask( void * pvParameters )
{
   // Create the mutex to guard a shared resource.
   xMutex = xSemaphoreCreateRecursiveMutex();
}


// A task that uses the mutex.
void vAnotherTask( void * pvParameters )
{
   // ... Do other things.


   if( xMutex != NULL )
   {
       // See if we can obtain the mutex.  If the mutex is not available
       // wait 10 ticks to see if it becomes free.
       if( xSemaphoreTakeRecursive( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
       {
           // We were able to obtain the mutex and can now access the
           // shared resource.
```

```
            // ...
            // For some reason due to the nature of the code further calls to
         // xSemaphoreTakeRecursive() are made on the same mutex.  In real
         // code these would not be just sequential calls as this would make
         // no sense.  Instead the calls are likely to be buried inside
         // a more complex call structure.
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );


            // The mutex has now been 'taken' three times, so will not be
         // available to another task until it has also been given back
         // three times.  Again it is unlikely that real code would have
         // these calls sequentially, but instead buried in a more complex
         // call structure.  This is just for illustrative purposes.
            xSemaphoreGiveRecursive( xMutex );
         xSemaphoreGiveRecursive( xMutex );
         xSemaphoreGiveRecursive( xMutex );


         // Now the mutex can be taken by other tasks.
        }
        else
        {
            // We could not obtain the mutex and can therefore not access
            // the shared resource safely.
        }
    }
}
```

5.37 xSemaphoreGive

semphr. h

```
xSemaphoreGive( SemaphoreHandle_t xSemaphore )
```

Macro to release a semaphore. The semaphore must have previously been created with a call to xSemaphore↩ CreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting(). and obtained using sSemaphore↩ Take().

This macro must not be used from an ISR. See xSemaphoreGiveFromISR () for an alternative which can be used from an ISR.

This macro must also not be used on semaphores created using xSemaphoreCreateRecursiveMutex().

Parameters

xSemaphore	A handle to the semaphore being released. This is the handle returned when the semaphore was created.

Returns

pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;


void vATask( void * pvParameters )
{
   // Create the semaphore to guard a shared resource.
   xSemaphore = vSemaphoreCreateBinary();


   if( xSemaphore != NULL )
   {
       if( xSemaphoreGive( xSemaphore ) != pdTRUE )
       {
           // We would expect this call to fail because we cannot give
           // a semaphore without first "taking" it!
       }


       // Obtain the semaphore - don't block if the semaphore is not
       // immediately available.
       if( xSemaphoreTake( xSemaphore, ( TickType_t ) 0 ) )
       {
           // We now have the semaphore and can access the shared resource.


           // ...
```

```
            // We have finished accessing the shared resource so can free the
            // semaphore.
            if( xSemaphoreGive( xSemaphore ) != pdTRUE )
            {
                // We would not expect this call to fail because we must have
                // obtained the semaphore to get here.
            }
        }
    }
}
```

5.38 xSemaphoreGiveRecursive

semphr. h

```
xSemaphoreGiveRecursive( SemaphoreHandle_t xMutex )
```

Macro to recursively release, or 'give', a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex();

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.

This macro must not be used on mutexes created using xSemaphoreCreateMutex().

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

Parameters

xMutex	A handle to the mutex being released, or 'given'. This is the handle returned by xSemaphoreCreateMutex();

Returns

pdTRUE if the semaphore was given.

Example usage:

```
SemaphoreHandle_t xMutex = NULL;


// A task that creates a mutex.
void vATask( void * pvParameters )
{
   // Create the mutex to guard a shared resource.
   xMutex = xSemaphoreCreateRecursiveMutex();
}


// A task that uses the mutex.
void vAnotherTask( void * pvParameters )
{
   // ... Do other things.


   if( xMutex != NULL )
   {
       // See if we can obtain the mutex.  If the mutex is not available
       // wait 10 ticks to see if it becomes free.
       if( xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 ) == pdTRUE )
       {
           // We were able to obtain the mutex and can now access the
           // shared resource.
```

```
            // ...
            // For some reason due to the nature of the code further calls to
         // xSemaphoreTakeRecursive() are made on the same mutex.  In real
         // code these would not be just sequential calls as this would make
         // no sense.  Instead the calls are likely to be buried inside
         // a more complex call structure.
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );


            // The mutex has now been 'taken' three times, so will not be
         // available to another task until it has also been given back
         // three times.  Again it is unlikely that real code would have
         // these calls sequentially, it would be more likely that the calls
         // to xSemaphoreGiveRecursive() would be called as a call stack
         // unwound.  This is just for demonstrative purposes.
            xSemaphoreGiveRecursive( xMutex );
         xSemaphoreGiveRecursive( xMutex );
         xSemaphoreGiveRecursive( xMutex );


         // Now the mutex can be taken by other tasks.
        }
        else
        {
            // We could not obtain the mutex and can therefore not access
            // the shared resource safely.
        }
    }
}
```

5.39 xSemaphoreGiveFromISR

semphr. h

```
xSemaphoreGiveFromISR(
                          SemaphoreHandle_t xSemaphore,
                          BaseType_t *pxHigherPriorityTaskWoken
                      )
```

Macro to release a semaphore. The semaphore must have previously been created with a call to xSemaphore↩ CreateBinary() or xSemaphoreCreateCounting().

Mutex type semaphores (those created using a call to xSemaphoreCreateMutex()) must not be used with this macro.

This macro can be used from an ISR.

Parameters

xSemaphore	A handle to the semaphore being released. This is the handle returned when the semaphore was created.
pxHigherPriorityTaskWoken	xSemaphoreGiveFromISR() will set ∗pxHigherPriorityTaskWoken to pdTRUE if giving the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns

pdTRUE if the semaphore was successfully given, otherwise errQUEUE_FULL.

Example usage:

```
#define LONG_TIME 0xffff
#define TICKS_TO_WAIT  10
SemaphoreHandle_t xSemaphore = NULL;


// Repetitive task.
void vATask( void * pvParameters )
{
   for( ;; )
   {
       // We want this task to run every 10 ticks of a timer.  The semaphore
       // was created before this task was started.


       // Block waiting for the semaphore to become available.
       if( xSemaphoreTake( xSemaphore, LONG_TIME ) == pdTRUE )
       {
           // It is time to execute.


           // ...
```

```
            // We have finished our task.  Return to the top of the loop where
            // we will block on the semaphore until it is time to execute
            // again.  Note when using the semaphore for synchronisation with an
         // ISR in this manner there is no need to 'give' the semaphore back.
        }
    }
}


// Timer ISR
void vTimerISR( void * pvParameters )
{
static uint8_t ucLocalTickCount = 0;
static BaseType_t xHigherPriorityTaskWoken;


    // A timer tick has occurred.


    // ... Do other time functions.


    // Is it time for vATask () to run?
    xHigherPriorityTaskWoken = pdFALSE;
    ucLocalTickCount++;
    if( ucLocalTickCount >= TICKS_TO_WAIT )
    {
        // Unblock the task by releasing the semaphore.
        xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );


        // Reset the count so we release the semaphore again in 10 ticks time.
        ucLocalTickCount = 0;
    }


    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // We can force a context switch here.  Context switching from an
        // ISR uses port specific syntax.  Check the demo task for your port
        // to find the syntax required.
    }
}
```

5.40 xSemaphoreCreateMutex

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateMutex( void )
```

Creates a new mutex type semaphore instance, and returns a handle by which the new mutex can be referenced.

Internally, within the FreeRTOS implementation, mutex semaphores use a block of memory, in which the mutex structure is stored. If a mutex is created using xSemaphoreCreateMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateMutex() function. (see http://www.freertos.↩org/a00111.html). If a mutex is created using xSemaphoreCreateMutexStatic() then the application writer must provided the memory. xSemaphoreCreateMutexStatic() therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros. The xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros must not be used.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Returns

If the mutex was successfully created then a handle to the created semaphore is returned. If there was not enough heap to allocate the mutex data structures then NULL is returned.

Example usage:

```
SemaphoreHandle_t xSemaphore;


void vATask( void * pvParameters )
{
   // Semaphore cannot be used before a call to xSemaphoreCreateMutex().
   // This is a macro so pass the variable in directly.
   xSemaphore = xSemaphoreCreateMutex();


   if( xSemaphore != NULL )
   {
       // The semaphore was created successfully.
       // The semaphore can now be used.
   }
}
```

5.41 xSemaphoreCreateMutexStatic

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateMutexStatic( StaticSemaphore_t *pxMutexBuffer )
```

Creates a new mutex type semaphore instance, and returns a handle by which the new mutex can be referenced.

Internally, within the FreeRTOS implementation, mutex semaphores use a block of memory, in which the mutex structure is stored. If a mutex is created using xSemaphoreCreateMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateMutex() function. (see http://www.freertos.org/a00111.html). If a mutex is created using xSemaphoreCreateMutexStatic() then the application writer must provided the memory. xSemaphoreCreateMutexStatic() therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros. The xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros must not be used.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Parameters

pxMutexBuffer	Must point to a variable of type StaticSemaphore_t, which will be used to hold the mutex's data structure, removing the need for the memory to be allocated dynamically.

Returns

If the mutex was successfully created then a handle to the created mutex is returned. If pxMutexBuffer was NULL then NULL is returned.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;


void vATask( void * pvParameters )
{
   // A mutex cannot be used before it has been created.  xMutexBuffer is
   // into xSemaphoreCreateMutexStatic() so no dynamic memory allocation is
   // attempted.
   xSemaphore = xSemaphoreCreateMutexStatic( &xMutexBuffer );


   // As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
   // so there is no need to check it.
}
```

5.42 xSemaphoreCreateRecursiveMutex

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( void )
```

Creates a new recursive mutex type semaphore instance, and returns a handle by which the new recursive mutex can be referenced.

Internally, within the FreeRTOS implementation, recursive mutexs use a block of memory, in which the mutex structure is stored. If a recursive mutex is created using xSemaphoreCreateRecursiveMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateRecursiveMutex() function. (see http://www.freertos.org/a00111.html). If a recursive mutex is created using xSemaphoreCreate↩ RecursiveMutexStatic() then the application writer must provide the memory that will get used by the mutex. x↩ SemaphoreCreateRecursiveMutexStatic() therefore allows a recursive mutex to be created without using any dynamic memory allocation.

Mutexes created using this macro can be accessed using the xSemaphoreTakeRecursive() and xSemaphoreGive↩ Recursive() macros. The xSemaphoreTake() and xSemaphoreGive() macros must not be used.

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Returns

xSemaphore Handle to the created mutex semaphore. Should be of type SemaphoreHandle_t.

Example usage:

```
SemaphoreHandle_t xSemaphore;


void vATask( void * pvParameters )
{
   // Semaphore cannot be used before a call to xSemaphoreCreateMutex().
   // This is a macro so pass the variable in directly.
   xSemaphore = xSemaphoreCreateRecursiveMutex();


   if( xSemaphore != NULL )
   {
       // The semaphore was created successfully.
       // The semaphore can now be used.
   }
}
```

5.43 xSemaphoreCreateRecursiveMutexStatic

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutexStatic( StaticSemaphore_t *pxMutexBuffer )
```

Creates a new recursive mutex type semaphore instance, and returns a handle by which the new recursive mutex can be referenced.

Internally, within the FreeRTOS implementation, recursive mutexs use a block of memory, in which the mutex structure is stored. If a recursive mutex is created using xSemaphoreCreateRecursiveMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateRecursiveMutex() function. (see http://www.freertos.org/a00111.html). If a recursive mutex is created using xSemaphoreCreate↩ RecursiveMutexStatic() then the application writer must provide the memory that will get used by the mutex. x↩ SemaphoreCreateRecursiveMutexStatic() therefore allows a recursive mutex to be created without using any dynamic memory allocation.

Mutexes created using this macro can be accessed using the xSemaphoreTakeRecursive() and xSemaphoreGive↩ Recursive() macros. The xSemaphoreTake() and xSemaphoreGive() macros must not be used.

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Parameters

pxMutexBuffer	Must point to a variable of type StaticSemaphore_t, which will then be used to hold the recursive mutex's data structure, removing the need for the memory to be allocated dynamically.

Returns

If the recursive mutex was successfully created then a handle to the created recursive mutex is returned. If pxMutexBuffer was NULL then NULL is returned.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;


void vATask( void * pvParameters )
{
   // A recursive semaphore cannot be used before it is created.  Here a
```

```
    // recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic().
    // The address of xMutexBuffer is passed into the function, and will hold
    // the mutexes data structures - so no dynamic memory allocation will be
    // attempted.
    xSemaphore = xSemaphoreCreateRecursiveMutexStatic( &xMutexBuffer );


    // As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
    // so there is no need to check it.
}
```

5.44 xSemaphoreCreateCounting

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount, UBaseType_t uxInitialCount )
```

Creates a new counting semaphore instance, and returns a handle by which the new counting semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a counting semaphore! http://www.freertos.org/RTOS-task-notifications.html

Internally, within the FreeRTOS implementation, counting semaphores use a block of memory, in which the counting semaphore structure is stored. If a counting semaphore is created using xSemaphoreCreateCounting() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateCounting() function. (see http://www.freertos.org/a00111.html). If a counting semaphore is created using xSemaphore↩ CreateCountingStatic() then the application writer can instead optionally provide the memory that will get used by the counting semaphore. xSemaphoreCreateCountingStatic() therefore allows a counting semaphore to be created without using any dynamic memory allocation.

Counting semaphores are typically used for two things:

1) Counting events.

In this usage scenario an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2) Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

Parameters

uxMaxCount	The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'given'.
uxInitialCount	The count value assigned to the semaphore when it is created.

Returns

Handle to the created semaphore. Null if the semaphore could not be created.

Example usage:

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphore = NULL;
```

```
   // Semaphore cannot be used before a call to xSemaphoreCreateCounting().
   // The max value to which the semaphore can count should be 10, and the
   // initial value assigned to the count should be 0.
   xSemaphore = xSemaphoreCreateCounting( 10, 0 );


   if( xSemaphore != NULL )
   {
       // The semaphore was created successfully.
       // The semaphore can now be used.
   }
}
```

5.45 xSemaphoreCreateCountingStatic

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateCountingStatic( UBaseType_t uxMaxCount, UBaseType_t uxInitialCou
```

Creates a new counting semaphore instance, and returns a handle by which the new counting semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a counting semaphore! http://www.freertos.org/RTOS-task-notifications.html

Internally, within the FreeRTOS implementation, counting semaphores use a block of memory, in which the counting semaphore structure is stored. If a counting semaphore is created using xSemaphoreCreateCounting() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateCounting() function. (see http://www.freertos.org/a00111.html). If a counting semaphore is created using xSemaphore↩ CreateCountingStatic() then the application writer must provide the memory. xSemaphoreCreateCountingStatic() therefore allows a counting semaphore to be created without using any dynamic memory allocation.

Counting semaphores are typically used for two things:

1) Counting events.

In this usage scenario an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2) Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

Parameters

uxMaxCount	The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'given'.
uxInitialCount	The count value assigned to the semaphore when it is created.
pxSemaphoreBuffer	Must point to a variable of type StaticSemaphore_t, which will then be used to hold the semaphore's data structure, removing the need for the memory to be allocated dynamically.

Returns

If the counting semaphore was successfully created then a handle to the created counting semaphore is returned. If pxSemaphoreBuffer was NULL then NULL is returned.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xSemaphoreBuffer;
```

```
void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphore = NULL;


   // Counting semaphore cannot be used before they have been created.  Create
   // a counting semaphore using xSemaphoreCreateCountingStatic().  The max
   // value to which the semaphore can count is 10, and the initial value
   // assigned to the count will be 0.  The address of xSemaphoreBuffer is
   // passed in and will be used to hold the semaphore structure, so no dynamic
   // memory allocation will be used.
   xSemaphore = xSemaphoreCreateCounting( 10, 0, &xSemaphoreBuffer );


   // No memory allocation was attempted so xSemaphore cannot be NULL, so there
   // is no need to check its value.
}
```

5.46 vSemaphoreDelete

semphr. h

```
void vSemaphoreDelete( SemaphoreHandle_t xSemaphore );
```

Delete a semaphore. This function must be used with care. For example, do not delete a mutex type semaphore if the mutex is held by a task.

Parameters

xSemaphore	A handle to the semaphore to be deleted.

5.47 TaskHandle_t

task. h

Type by which tasks are referenced. For example, a call to xTaskCreate returns (via a pointer parameter) an Task↩ Handle_t variable that can then be used as a parameter to vTaskDelete to delete the task.

5.48 taskYIELD

task. h

Macro for forcing a context switch.

5.49 taskENTER_CRITICAL

task. h

Macro to mark the start of a critical code region. Preemptive context switches cannot occur when in a critical region.

NOTE: This may alter the stack (depending on the portable implementation) so must be used with care!

5.50 taskEXIT_CRITICAL

task. h

Macro to mark the end of a critical code region. Preemptive context switches cannot occur when in a critical region.

NOTE: This may alter the stack (depending on the portable implementation) so must be used with care!

5.51 taskDISABLE_INTERRUPTS

task. h

Macro to disable all maskable interrupts.

5.52 taskENABLE_INTERRUPTS

task. h

Macro to enable microcontroller interrupts.

5.53 xTaskCreate

task. h

```
BaseType_t xTaskCreate(
                            TaskFunction_t pvTaskCode,
                            const char * const pcName,
                            uint16_t usStackDepth,
                            void *pvParameters,
                            UBaseType_t uxPriority,
                            TaskHandle_t *pvCreatedTask
                        );
```

Create a new task and add it to the list of tasks that are ready to run.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using xTaskCreate() then both blocks of memory are automatically dynamically allocated inside the xTaskCreate() function. (see http↩://www.freertos.org/a00111.html). If a task is created using xTaskCreateStatic() then the application writer must provide the required memory. xTaskCreateStatic() therefore allows a task to be created without using any dynamic memory allocation.

See xTaskCreateStatic() for a version that does not use any dynamic memory allocation.

xTaskCreate() can only be used to create a task that has unrestricted access to the entire microcontroller memory map. Systems that include MPU support can alternatively create an MPU constrained task using xTaskCreate↩Restricted().

Parameters

pvTaskCode	Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
pcName	A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN - default is 16.
usStackDepth	The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage.
pvParameters	Pointer that will be used as the parameter for the task being created.
uxPriority	The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE_BIT).
pvCreatedTask	Used to pass back a handle by which the created task can be referenced.

Returns

pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

Example usage:

```
// Task to be created.
void vTaskCode( void * pvParameters )
```

```
{
    for( ;; )
    {
     // Task code goes here.
    }
}


// Function that creates a task.
void vOtherFunction( void )
{
static uint8_t ucParameterToPass;
TaskHandle_t xHandle = NULL;

// Create the task, storing the handle.  Note that the passed parameter ucParameterToPass
// must exist for the lifetime of the task, so in this case is declared static.  If it was just an
// an automatic stack variable it might no longer exist, or at least have been corrupted, by the time
// the new task attempts to access it.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_PRIORITY, &xHandle );
configASSERT( xHandle );

// Use the handle to delete the task.
if( xHandle != NULL )
{
    vTaskDelete( xHandle );
}


}
```

5.54 xTaskCreateStatic

task. h

```
TaskHandle_t xTaskCreateStatic( TaskFunction_t pvTaskCode,
                                const char * const pcName,
                                uint32_t ulStackDepth,
                                void *pvParameters,
                                UBaseType_t uxPriority,
                                StackType_t *pxStackBuffer,
                                StaticTask_t *pxTaskBuffer );
```

Create a new task and add it to the list of tasks that are ready to run.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using xTaskCreate() then both blocks of memory are automatically dynamically allocated inside the xTaskCreate() function. (see http↩://www.freertos.org/a00111.html). If a task is created using xTaskCreateStatic() then the application writer must provide the required memory. xTaskCreateStatic() therefore allows a task to be created without using any dynamic memory allocation.

Parameters

pvTaskCode	Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
pcName	A descriptive name for the task. This is mainly used to facilitate debugging. The maximum length of the string is defined by configMAX_TASK_NAME_LEN in FreeRTOSConfig.h.
ulStackDepth	The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 32-bits wide and ulStackDepth is defined as 100 then 400 bytes will be allocated for stack storage.
pvParameters	Pointer that will be used as the parameter for the task being created.
uxPriority	The priority at which the task will run.
pxStackBuffer	Must point to a StackType_t array that has at least ulStackDepth indexes - the array will then be used as the task's stack, removing the need for the stack to be allocated dynamically.
pxTaskBuffer	Must point to a variable of type StaticTask_t, which will then be used to hold the task's data structures, removing the need for the memory to be allocated dynamically.

Returns

If neither pxStackBuffer or pxTaskBuffer are NULL, then the task will be created and pdPASS is returned. If either pxStackBuffer or pxTaskBuffer are NULL then the task will not be created and errCOULD_NOT_ALL↩OCATE_REQUIRED_MEMORY is returned.

Example usage:

```
// Dimensions the buffer that the task being created will use as its stack.
// NOTE:  This is the number of words the stack will hold, not the number of
// bytes.  For example, if each stack item is 32-bits, and this is set to 100,
// then 400 bytes (100 * 32-bits) will be allocated.
#define STACK_SIZE 200
```

```
// Structure that will hold the TCB of the task being created.
StaticTask_t xTaskBuffer;


// Buffer that the task being created will use as its stack.  Note this is
// an array of StackType_t variables.  The size of StackType_t is dependent on
// the RTOS port.
StackType_t xStack[ STACK_SIZE ];


// Function that implements the task being created.
void vTaskCode( void * pvParameters )
{
    // The parameter value is expected to be 1 as 1 is passed in the
    // pvParameters value in the call to xTaskCreateStatic().
    configASSERT( ( uint32_t ) pvParameters == 1UL );


    for( ;; )
    {
        // Task code goes here.
    }
}


// Function that creates a task.
void vOtherFunction( void )
{
    TaskHandle_t xHandle = NULL;


    // Create the task without using any dynamic memory allocation.
    xHandle = xTaskCreateStatic(
                  vTaskCode,       // Function that implements the task.
                  "NAME",          // Text name for the task.
                  STACK_SIZE,      // Stack size in words, not bytes.
                  ( void * ) 1,    // Parameter passed into the task.
                  tskIDLE_PRIORITY,// Priority at which the task is created.
                  xStack,          // Array to use as the task's stack.
                  &xTaskBuffer );  // Variable to hold the task's data structure.


    // puxStackBuffer and pxTaskBuffer were not NULL, so the task will have
    // been created, and xHandle will be the task's handle.  Use the handle
    // to suspend the task.
    vTaskSuspend( xHandle );
}
```

5.55 xTaskCreateRestricted

task. h

```
BaseType_t xTaskCreateRestricted( TaskParameters_t *pxTaskDefinition, TaskHandle_t *pxCreatedTask
```

xTaskCreateRestricted() should only be used in systems that include an MPU implementation.

Create a new task and add it to the list of tasks that are ready to run. The function parameters define the memory regions and associated access permissions allocated to the task.

Parameters

pxTaskDefinition	Pointer to a structure that contains a member for each of the normal xTaskCreate() parameters (see the xTaskCreate() API documentation) plus an optional stack buffer and the memory region definitions.
pxCreatedTask	Used to pass back a handle by which the created task can be referenced.

Returns

pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

Example usage:

```
// Create an TaskParameters_t structure that defines the task to be created.
static const TaskParameters_t xCheckTaskParameters =
{
    vATask,     // pvTaskCode - the function that implements the task.
    "ATask",    // pcName - just a text name for the task to assist debugging.
    100,        // usStackDepth - the stack size DEFINED IN WORDS.
    NULL,       // pvParameters - passed into the task function as the function parameters.
    ( 1UL | portPRIVILEGE_BIT ),// uxPriority - task priority, set the portPRIVILEGE_BIT if the ta
    cStackBuffer,// puxStackBuffer - the buffer to be used as the task stack.


    // xRegions - Allocate up to three separate memory regions for access by
    // the task, with appropriate access permissions.  Different processors have
    // different memory alignment requirements - refer to the FreeRTOS documentation
    // for full information.
    {
        // Base address                 Length  Parameters
        { cReadWriteArray,              32,     portMPU_REGION_READ_WRITE },
        { cReadOnlyArray,               32,     portMPU_REGION_READ_ONLY },
        { cPrivilegedOnlyAccessArray,   128,    portMPU_REGION_PRIVILEGED_READ_WRITE }
    }
};


int main( void )
{
TaskHandle_t xHandle;
```

```
// Create a task from the const structure defined above.  The task handle
// is requested (the second parameter is not NULL) but in this case just for
// demonstration purposes as its not actually used.
xTaskCreateRestricted( &xRegTest1Parameters, &xHandle );

// Start the scheduler.
vTaskStartScheduler();

// Will only get here if there was insufficient memory to create the idle
// and/or timer task.
for( ;; );


}
```

task. h

```
void vTaskAllocateMPURegions( TaskHandle_t xTask, const MemoryRegion_t * const pxRegions );
```

Memory regions are assigned to a restricted task when the task is created by a call to xTaskCreateRestricted(). These regions can be redefined using vTaskAllocateMPURegions().

Parameters

xTask	The handle of the task being updated.
xRegions	A pointer to an MemoryRegion_t structure that contains the new memory region definitions.

Example usage:

```
// Define an array of MemoryRegion_t structures that configures an MPU region
// allowing read/write access for 1024 bytes starting at the beginning of the
// ucOneKByte array.  The other two of the maximum 3 definable regions are
// unused so set to zero.
static const MemoryRegion_t xAltRegions[ portNUM_CONFIGURABLE_REGIONS ] =
{
    // Base address     Length      Parameters
    { ucOneKByte,       1024,       portMPU_REGION_READ_WRITE },
    { 0,                0,          0 },
    { 0,                0,          0 }
};


void vATask( void *pvParameters )
{
    // This task was created such that it has access to certain regions of
    // memory as defined by the MPU configuration.  At some point it is
    // desired that these MPU regions are replaced with that defined in the
    // xAltRegions const struct above.  Use a call to vTaskAllocateMPURegions()
    // for this purpose.  NULL is used as the task handle to indicate that this
    // function should modify the MPU regions of the calling task.
    vTaskAllocateMPURegions( NULL, xAltRegions );


    // Now the task can continue its function, but from this point on can only
    // access its stack and the ucOneKByte array (unless any other statically
    // defined or shared regions have been declared elsewhere).
}
```

5.56 vTaskDelete

task. h

```
void vTaskDelete( TaskHandle_t xTask );
```

INCLUDE_vTaskDelete must be defined as 1 for this function to be available. See the configuration section for more information.

Remove a task from the RTOS real time kernel's management. The task being deleted will be removed from all ready, blocked, suspended and event lists.

NOTE: The idle task is responsible for freeing the kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to vTaskDelete (). Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

See the demo application file death.c for sample code that utilises vTaskDelete ().

Parameters

xTask	The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.

Example usage:

```
void vOtherFunction( void )
{
TaskHandle_t xHandle;

// Create the task, storing the handle.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

// Use the handle to delete the task.
vTaskDelete( xHandle );


}
```

5.57 vTaskDelay

task. h

```
void vTaskDelay( const TickType_t xTicksToDelay );
```

Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

INCLUDE_vTaskDelay must be defined as 1 for this function to be available. See the configuration section for more information.

vTaskDelay() specifies a time at which the task wishes to unblock relative to the time at which vTaskDelay() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after vTaskDelay() is called. vTaskDelay() does not therefore provide a good method of controlling the frequency of a periodic task as the path taken through the code, as well as other task and interrupt activity, will effect the frequency at which vTaskDelay() gets called and therefore the time at which the task next executes. See vTaskDelayUntil() for an alternative API function designed to facilitate fixed frequency execution. It does this by specifying an absolute time (rather than a relative time) at which the calling task should unblock.

Parameters

xTicksToDelay	The amount of time, in tick periods, that the calling task should block.

Example usage:

void vTaskFunction(void * pvParameters) { Block for 500ms. const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

```
for( ;; )
{
```

Simply toggle the LED every 500ms, blocking between each toggle. vToggleLED(); vTaskDelay(xDelay); } }

5.58 vTaskDelayUntil

task. h

```
void vTaskDelayUntil( TickType_t *pxPreviousWakeTime, const TickType_t xTimeIncrement );
```

INCLUDE_vTaskDelayUntil must be defined as 1 for this function to be available. See the configuration section for more information.

Delay a task until a specified time. This function can be used by periodic tasks to ensure a constant execution frequency.

This function differs from vTaskDelay () in one important aspect: vTaskDelay () will cause a task to block for the specified number of ticks from the time vTaskDelay () is called. It is therefore difficult to use vTaskDelay () by itself to generate a fixed execution frequency as the time between a task starting to execute and that task calling vTask↩ Delay () may not be fixed [the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes].

Whereas vTaskDelay () specifies a wake time relative to the time at which the function is called, vTaskDelayUntil () specifies the absolute (exact) time at which it wishes to unblock.

The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

Parameters

pxPreviousWakeTime	Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialised with the current time prior to its first use (see the example below). Following this the variable is automatically updated within vTaskDelayUntil ().
xTimeIncrement	The cycle time period. The task will be unblocked at time ∗pxPreviousWakeTime + xTimeIncrement. Calling vTaskDelayUntil with the same xTimeIncrement parameter value will cause the task to execute with a fixed interface period.

Example usage:

```
// Perform an action every 10 ticks.
void vTaskFunction( void * pvParameters )
{
TickType_t xLastWakeTime;
const TickType_t xFrequency = 10;

// Initialise the xLastWakeTime variable with the current time.
xLastWakeTime = xTaskGetTickCount ();
for( ;; )
{
 // Wait for the next cycle.
 vTaskDelayUntil( &xLastWakeTime, xFrequency );

 // Perform action here.
}


}
```

5.59 xTaskAbortDelay

task. h

```
BaseType_t xTaskAbortDelay( TaskHandle_t xTask );
```

INCLUDE_xTaskAbortDelay must be defined as 1 in FreeRTOSConfig.h for this function to be available.

A task will enter the Blocked state when it is waiting for an event. The event it is waiting for can be a temporal event (waiting for a time), such as when vTaskDelay() is called, or an event on an object, such as when xQueueReceive() or ulTaskNotifyTake() is called. If the handle of a task that is in the Blocked state is used in a call to xTaskAbort↩ Delay() then the task will leave the Blocked state, and return from whichever function call placed the task into the Blocked state.

Parameters

xTask	The handle of the task to remove from the Blocked state.

Returns

If the task referenced by xTask was not in the Blocked state then pdFAIL is returned. Otherwise pdPASS is returned.

5.60 uxTaskPriorityGet

task. h

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t xTask );
```

INCLUDE_uxTaskPriorityGet must be defined as 1 for this function to be available. See the configuration section for more information.

Obtain the priority of any task.

Parameters

xTask	Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

Returns

The priority of xTask.

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

// Create a task, storing the handle.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

// ...

// Use the handle to obtain the priority of the created task.
// It was created with tskIDLE_PRIORITY, but may have changed
// it itself.
if( uxTaskPriorityGet( xHandle ) != tskIDLE_PRIORITY )
{
 // The task has changed it's priority.
}

// ...

// Is our priority higher than the created task?
if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )
{
 // Our priority (obtained using NULL handle) is higher.
}


}
```

5.61 vTaskGetInfo

task. h

```
void vTaskGetInfo( TaskHandle_t xTask, TaskStatus_t *pxTaskStatus, BaseType_t xGetFreeStackSpace,
```

configUSE_TRACE_FACILITY must be defined as 1 for this function to be available. See the configuration section for more information.

Populates a TaskStatus_t structure with information about a task.

Parameters

xTask	Handle of the task being queried. If xTask is NULL then information will be returned about the calling task.
pxTaskStatus	A pointer to the TaskStatus_t structure that will be filled with information about the task referenced by the handle passed using the xTask parameter.

The TaskStatus_t structure contains a member to report the stack high water mark of the task being queried. Calculating the stack high water mark takes a relatively long time, and can make the system temporarily unresponsive - so the xGetFreeStackSpace parameter is provided to allow the high water mark checking to be skipped. The high watermark value will only be written to the TaskStatus_t structure if xGetFreeStackSpace is not set to pdFALSE;

Parameters

eState	The TaskStatus_t structure contains a member to report the state of the task being queried. Obtaining the task state is not as fast as a simple assignment - so the eState parameter is provided to allow the state information to be omitted from the TaskStatus_t structure. To obtain state information then set eState to eInvalid - otherwise the value passed in eState will be reported as the task state in the TaskStatus_t structure.

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;
TaskStatus_t xTaskDetails;


   // Obtain the handle of a task from its name.
   xHandle = xTaskGetHandle( "Task_Name" );


   // Check the handle is not NULL.
   configASSERT( xHandle );


   // Use the handle to obtain further information about the task.
   vTaskGetInfo( xHandle,
                 &xTaskDetails,
                 pdTRUE, // Include the high water mark in xTaskDetails.
                 eInvalid ); // Include the task state in xTaskDetails.
}
```

5.62 vTaskPrioritySet

task. h

```
void vTaskPrioritySet( TaskHandle_t xTask, UBaseType_t uxNewPriority );
```

INCLUDE_vTaskPrioritySet must be defined as 1 for this function to be available. See the configuration section for more information.

Set the priority of any task.

A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

Parameters

xTask	Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set.
uxNewPriority	The priority to which the task will be set.

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

// Create a task, storing the handle.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

// ...

// Use the handle to raise the priority of the created task.
vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );

// ...

// Use a NULL handle to raise our priority to the same value.
vTaskPrioritySet( NULL, tskIDLE_PRIORITY + 1 );


}
```

5.63 vTaskSuspend

task. h

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend );
```

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

Suspend any task. When suspended a task will never get any microcontroller processing time, no matter what its priority.

Calls to vTaskSuspend are not accumulative - i.e. calling vTaskSuspend () twice on the same task still only requires one call to vTaskResume () to ready the suspended task.

Parameters

xTaskToSuspend	Handle to the task being suspended. Passing a NULL handle will cause the calling task to be suspended.

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

// Create a task, storing the handle.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

// ...

// Use the handle to suspend the created task.
vTaskSuspend( xHandle );

// ...

// The created task will not run during this period, unless
// another task calls vTaskResume( xHandle ).

//...


// Suspend ourselves.
vTaskSuspend( NULL );

// We cannot get here unless another task calls vTaskResume
// with our handle as the parameter.


}
```

5.64 vTaskResume

task. h

```
void vTaskResume( TaskHandle_t xTaskToResume );
```

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

Resumes a suspended task.

A task that has been suspended by one or more calls to vTaskSuspend () will be made available for running again by a single call to vTaskResume ().

Parameters

xTaskToResume	Handle to the task being readied.

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

// Create a task, storing the handle.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

// ...

// Use the handle to suspend the created task.
vTaskSuspend( xHandle );

// ...

// The created task will not run during this period, unless
// another task calls vTaskResume( xHandle ).

//...


// Resume the suspended task ourselves.
vTaskResume( xHandle );

// The created task will once again get microcontroller processing
// time in accordance with its priority within the system.


}
```

5.65 vTaskResumeFromISR

task. h

```
void xTaskResumeFromISR( TaskHandle_t xTaskToResume );
```

INCLUDE_xTaskResumeFromISR must be defined as 1 for this function to be available. See the configuration section for more information.

An implementation of vTaskResume() that can be called from within an ISR.

A task that has been suspended by one or more calls to vTaskSuspend () will be made available for running again by a single call to xTaskResumeFromISR ().

xTaskResumeFromISR() should not be used to synchronise a task with an interrupt if there is a chance that the interrupt could arrive prior to the task being suspended - as this can lead to interrupts being missed. Use of a semaphore as a synchronisation mechanism would avoid this eventuality.

Parameters

xTaskToResume	Handle to the task being readied.

Returns

pdTRUE if resuming the task should result in a context switch, otherwise pdFALSE. This is used by the ISR to determine if a context switch may be required following the ISR.

5.66 vTaskStartScheduler

task. h

```
void vTaskStartScheduler( void );
```

Starts the real time kernel tick processing. After calling the kernel has control over which tasks are executed and when.

See the demo application file main.c for an example of creating tasks and starting the kernel.

Example usage:

```
void vAFunction( void )
{
    // Create at least one task before starting the kernel.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL );


    // Start the real time kernel with preemption.
    vTaskStartScheduler ();


    // Will not get here unless a task calls vTaskEndScheduler ()
}
```

5.67 vTaskEndScheduler

task. h

```
void vTaskEndScheduler( void );
```

NOTE: At the time of writing only the x86 real mode port, which runs on a PC in place of DOS, implements this function.

Stops the real time kernel tick. All created tasks will be automatically deleted and multitasking (either preemptive or cooperative) will stop. Execution then resumes from the point where vTaskStartScheduler () was called, as if vTaskStartScheduler () had just returned.

See the demo application file main. c in the demo/PC directory for an example that uses vTaskEndScheduler ().

vTaskEndScheduler () requires an exit function to be defined within the portable layer (see vPortEndScheduler () in port. c for the PC port). This performs hardware specific operations such as stopping the kernel tick.

vTaskEndScheduler () will cause all of the resources allocated by the kernel to be freed - but will not free resources allocated by application tasks.

Example usage:

```
void vTaskCode( void * pvParameters )
{
    for( ;; )
    {
     // Task code goes here.


     // At some point we want to end the real time kernel processing
     // so call ...
     vTaskEndScheduler ();
    }
}


void vAFunction( void )
{
    // Create at least one task before starting the kernel.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL );


    // Start the real time kernel with preemption.
    vTaskStartScheduler ();


    // Will only get here when the vTaskCode () task has called
    // vTaskEndScheduler ().  When we get here we are back to single task
    // execution.
}
```

5.68 vTaskSuspendAll

task. h

```
void vTaskSuspendAll( void );
```

Suspends the scheduler without disabling interrupts. Context switches will not occur while the scheduler is suspended.

After calling vTaskSuspendAll () the calling task will continue to execute without risk of being swapped out until a call to xTaskResumeAll () has been made.

API functions that have the potential to cause a context switch (for example, vTaskDelayUntil(), xQueueSend(), etc.) must not be called while the scheduler is suspended.

Example usage:

```
void vTask1( void * pvParameters )
{
    for( ;; )
    {
     // Task code goes here.


     // ...


     // At some point the task wants to perform a long operation during
     // which it does not want to get swapped out.  It cannot use
     // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
     // operation may cause interrupts to be missed - including the
     // ticks.


     // Prevent the real time kernel swapping out the task.
     vTaskSuspendAll ();


     // Perform the operation here.  There is no need to use critical
     // sections as we have all the microcontroller processing time.
     // During this time interrupts will still operate and the kernel
     // tick count will be maintained.


     // ...


     // The operation is complete.  Restart the kernel.
     xTaskResumeAll ();
    }
}
```

5.69 xTaskResumeAll

task. h

```
BaseType_t xTaskResumeAll( void );
```

Resumes scheduler activity after it was suspended by a call to vTaskSuspendAll().

xTaskResumeAll() only resumes the scheduler. It does not unsuspend tasks that were previously suspended by a call to vTaskSuspend().

Returns

If resuming the scheduler caused a context switch then pdTRUE is returned, otherwise pdFALSE is returned.

Example usage:

```
void vTask1( void * pvParameters )
{
    for( ;; )
    {
     // Task code goes here.


     // ...


     // At some point the task wants to perform a long operation during
     // which it does not want to get swapped out.  It cannot use
     // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
     // operation may cause interrupts to be missed - including the
     // ticks.


     // Prevent the real time kernel swapping out the task.
     vTaskSuspendAll ();


     // Perform the operation here.  There is no need to use critical
     // sections as we have all the microcontroller processing time.
     // During this time interrupts will still operate and the real
     // time kernel tick count will be maintained.


     // ...


     // The operation is complete.  Restart the kernel.  We want to force
     // a context switch - but there is no point if resuming the scheduler
     // caused a context switch already.
     if( !xTaskResumeAll () )
     {
          taskYIELD ();
     }
    }
}
```

5.70 xTaskGetTickCount

task. h

```
TickType_t xTaskGetTickCount( void );
```

Returns

The count of ticks since vTaskStartScheduler was called.

5.71 xTaskGetTickCountFromISR

task. h

```
TickType_t xTaskGetTickCountFromISR( void );
```

Returns

The count of ticks since vTaskStartScheduler was called.

This is a version of xTaskGetTickCount() that is safe to be called from an ISR - provided that TickType_t is the natural word size of the microcontroller being used or interrupt nesting is either not supported or not being used.

5.72 uxTaskGetNumberOfTasks

task. h

```
uint16_t uxTaskGetNumberOfTasks( void );
```

Returns

The number of tasks that the real time kernel is currently managing. This includes all ready, blocked and suspended tasks. A task that has been deleted but not yet freed by the idle task will also be included in the count.

5.73 pcTaskGetName

task. h

```
char *pcTaskGetName( TaskHandle_t xTaskToQuery );
```

Returns

The text (human readable) name of the task referenced by the handle xTaskToQuery. A task can query its own name by either passing in its own handle, or by setting xTaskToQuery to NULL.

5.74 pcTaskGetHandle

task. h

```
TaskHandle_t xTaskGetHandle( const char *pcNameToQuery );
```

NOTE: This function takes a relatively long time to complete and should be used sparingly.

Returns

The handle of the task that has the human readable name pcNameToQuery. NULL is returned if no matching name is found. INCLUDE_xTaskGetHandle must be set to 1 in FreeRTOSConfig.h for pcTaskGetHandle() to be available.

5.75 vTaskList

task. h

```
void vTaskList( char *pcWriteBuffer );
```

configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. See the configuration section of the FreeRTOS.org website for more information.

NOTE 1: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

Lists all the current tasks, along with their current state and stack usage high water mark.

Tasks are reported as blocked ('B'), ready ('R'), deleted ('D') or suspended ('S').

PLEASE NOTE:

This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

vTaskList() calls uxTaskGetSystemState(), then formats part of the uxTaskGetSystemState() output into a human readable table that displays task names, states and stack usage.

vTaskList() has a dependency on the sprintf() C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of sprintf() is provided in many of the FreeRTOS/Demo sub-directories in a file called printf-stdarg.c (note printf-stdarg.c does not provide a full snprintf() implementation!).

It is recommended that production systems call uxTaskGetSystemState() directly to get access to raw stats data, rather than indirectly through a call to vTaskList().

Parameters

pcWriteBuffer	A buffer into which the above mentioned details will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

5.76 vTaskGetRunTimeStats

task. h

```
void vTaskGetRunTimeStats( char *pcWriteBuffer );
```

configGENERATE_RUN_TIME_STATS and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. The application must also then provide definitions for portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and portGET_RUN_TIME_COUNTER_VALUE() to configure a peripheral timer/counter and return the timers current count value respectively. The counter should be at least 10 times the frequency of the tick count.

NOTE 1: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

Setting configGENERATE_RUN_TIME_STATS to 1 will result in a total accumulated execution time being stored for each task. The resolution of the accumulated time value depends on the frequency of the timer configured by the portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() macro. Calling vTaskGetRunTimeStats() writes the total execution time of each task into a buffer, both as an absolute count value and as a percentage of the total system execution time.

NOTE 2:

This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

vTaskGetRunTimeStats() calls uxTaskGetSystemState(), then formats part of the uxTaskGetSystemState() output into a human readable table that displays the amount of time each task has spent in the Running state in both absolute and percentage terms.

vTaskGetRunTimeStats() has a dependency on the sprintf() C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of sprintf() is provided in many of the FreeRTOS/Demo sub-directories in a file called printf-stdarg.c (note printf-stdarg.c does not provide a full snprintf() implementation!).

It is recommended that production systems call uxTaskGetSystemState() directly to get access to raw stats data, rather than indirectly through a call to vTaskGetRunTimeStats().

Parameters

pcWriteBuffer	A buffer into which the execution times will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

5.77 xTaskNotify

task. h

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction );
```

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

When configUSE_TASK_NOTIFICATIONS is set to one each task has its own private "notification value", which is a 32-bit unsigned integer (uint32_t).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWait() or ulTaskNotifyTake(). If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use xTaskNotifyWait() to [optionally] block to wait for a notification to be pending, or ulTaskNotifyTake() to [optionally] block to wait for its notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

See http://www.FreeRTOS.org/RTOS-task-notifications.html for details.

Parameters

xTaskToNotify	The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle().
ulValue	Data that can be sent with the notification. How the data is used depends on the value of the eAction parameter.
eAction	Specifies how the notification updates the task's notification value, if at all. Valid values for eAction are as follows:

eSetBits - The task's notification value is bitwise ORed with ulValue. xTaskNofify() always returns pdPASS in this case.

eIncrement - The task's notification value is incremented. ulValue is not used and xTaskNotify() always returns pdPASS in this case.

eSetValueWithOverwrite - The task's notification value is set to the value of ulValue, even if the task being notified had not yet processed the previous notification (the task already had a notification pending). xTaskNotify() always returns pdPASS in this case.

eSetValueWithoutOverwrite - If the task being notified did not already have a notification pending then the task's notification value is set to ulValue and xTaskNotify() will return pdPASS. If the task being notified already had a notification pending then no action is performed and pdFAIL is returned.

eNoAction - The task receives a notification without its notification value being updated. ulValue is not used and xTaskNotify() always returns pdPASS in this case.

pulPreviousNotificationValue - Can be used to pass out the subject task's notification value before any bits are modified by the notify function.

Returns

Dependent on the value of eAction. See the description of the eAction parameter.

task. h

```
BaseType_t xTaskNotifyFromISR( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction
```

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

When configUSE_TASK_NOTIFICATIONS is set to one each task has its own private "notification value", which is a 32-bit unsigned integer (uint32_t).

A version of xTaskNotify() that can be used from an interrupt service routine (ISR).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWait() or ulTaskNotifyTake(). If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use xTaskNotifyWait() to [optionally] block to wait for a notification to be pending, or ulTaskNotifyTake() to [optionally] block to wait for its notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

See http://www.FreeRTOS.org/RTOS-task-notifications.html for details.

Parameters

xTaskToNotify	The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle().
ulValue	Data that can be sent with the notification. How the data is used depends on the value of the eAction parameter.
eAction	Specifies how the notification updates the task's notification value, if at all. Valid values for eAction are as follows:

eSetBits - The task's notification value is bitwise ORed with ulValue. xTaskNofify() always returns pdPASS in this case.

eIncrement - The task's notification value is incremented. ulValue is not used and xTaskNotify() always returns pdPASS in this case.

eSetValueWithOverwrite - The task's notification value is set to the value of ulValue, even if the task being notified had not yet processed the previous notification (the task already had a notification pending). xTaskNotify() always returns pdPASS in this case.

eSetValueWithoutOverwrite - If the task being notified did not already have a notification pending then the task's notification value is set to ulValue and xTaskNotify() will return pdPASS. If the task being notified already had a notification pending then no action is performed and pdFAIL is returned.

eNoAction - The task receives a notification without its notification value being updated. ulValue is not used and xTaskNotify() always returns pdPASS in this case.

Parameters

pxHigherPriorityTaskWoken	xTaskNotifyFromISR() will set ∗pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused the task to which the notification was sent to leave the Blocked state, and the unblocked task has a priority higher than the currently running task. If xTaskNotifyFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. How a context switch is requested from an ISR is dependent on the port - see the documentation page for the port in use.

Returns

Dependent on the value of eAction. See the description of the eAction parameter.

5.78 xTaskNotifyWait

task. h

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t
```

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

When configUSE_TASK_NOTIFICATIONS is set to one each task has its own private "notification value", which is a 32-bit unsigned integer (uint32_t).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWait() or ulTaskNotifyTake(). If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use xTaskNotifyWait() to [optionally] block to wait for a notification to be pending, or ulTaskNotifyTake() to [optionally] block to wait for its notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

See http://www.FreeRTOS.org/RTOS-task-notifications.html for details.

Parameters

ulBitsToClearOnEntry	Bits that are set in ulBitsToClearOnEntry value will be cleared in the calling task's notification value before the task checks to see if any notifications are pending, and optionally blocks if no notifications are pending. Setting ulBitsToClearOnEntry to ULONG_MAX (if limits.h is included) or 0xffffffffUL (if limits.h is not included) will have the effect of resetting the task's notification value to 0. Setting ulBitsToClearOnEntry to 0 will leave the task's notification value unchanged.
ulBitsToClearOnExit	If a notification is pending or received before the calling task exits the xTaskNotifyWait() function then the task's notification value (see the xTaskNotify() API function) is passed out using the pulNotificationValue parameter. Then any bits that are set in ulBitsToClearOnExit will be cleared in the task's notification value (note *pulNotificationValue is set before any bits are cleared). Setting ulBitsToClearOnExit to ULONG_MAX (if limits.h is included) or 0xffffffffUL (if limits.h is not included) will have the effect of resetting the task's notification value to 0 before the function exits. Setting ulBitsToClearOnExit to 0 will leave the task's notification value unchanged when the function exits (in which case the value passed out in pulNotificationValue will match the task's notification value).
pulNotificationValue	Used to pass the task's notification value out of the function. Note the value passed out will not be effected by the clearing of any bits caused by ulBitsToClearOnExit being non-zero.
xTicksToWait	The maximum amount of time that the task should wait in the Blocked state for a notification to be received, should a notification not already be pending when xTaskNotifyWait() was called. The task will not consume any processing time while it is in the Blocked state. This is specified in kernel ticks, the macro pdMS_TO_TICSK(value_in_ms) can be used to convert a time specified in milliseconds to a time specified in ticks.

Returns

If a notification was received (including notifications that were already pending when xTaskNotifyWait was called) then pdPASS is returned. Otherwise pdFAIL is returned.

task. h

```
void vTaskNotifyGiveFromISR( TaskHandle_t xTaskHandle, BaseType_t *pxHigherPriorityTaskWoken );
```

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this macro to be available.

When configUSE_TASK_NOTIFICATIONS is set to one each task has its own private "notification value", which is a 32-bit unsigned integer (uint32_t).

A version of xTaskNotifyGive() that can be called from an interrupt service routine (ISR).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

vTaskNotifyGiveFromISR() is intended for use when task notifications are used as light weight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given from an ISR using the xSemaphoreGiveFromISR() API function, the equivalent action that instead uses a task notification is vTaskNotifyGiveFromISR().

When task notifications are being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the ulTaskNotificationTake() API function rather than the xTaskNotifyWait() API function.

See http://www.FreeRTOS.org/RTOS-task-notifications.html for more details.

Parameters

xTaskToNotify & The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle().
pxHigherPriorityTaskWoken & vTaskNotifyGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused the task to which the notification was sent to leave the Blocked state, and the unblocked task has a priority higher than the currently running task. If vTaskNotifyGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. How a context switch is

5.79 xTaskNotifyGive

task. h

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this macro to be available.

When configUSE_TASK_NOTIFICATIONS is set to one each task has its own private "notification value", which is a 32-bit unsigned integer (uint32_t).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

xTaskNotifyGive() is a helper macro intended for use when task notifications are used as light weight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given using the xSemaphoreGive() API function, the equivalent action that instead uses a task notification is xTaskNotifyGive().

When task notifications are being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the ulTaskNotificationTake() API function rather than the xTaskNotifyWait() API function.

See http://www.FreeRTOS.org/RTOS-task-notifications.html for more details.

Parameters

xTaskToNotify	The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle().

Returns

xTaskNotifyGive() is a macro that calls xTaskNotify() with the eAction parameter set to eIncrement - so pdP↩ ASS is always returned.

5.80 ulTaskNotifyTake

task. h

```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit, TickType_t xTicksToWait );
```

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

When configUSE_TASK_NOTIFICATIONS is set to one each task has its own private "notification value", which is a 32-bit unsigned integer (uint32_t).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

ulTaskNotifyTake() is intended for use when a task notification is used as a faster and lighter weight binary or counting semaphore alternative. Actual FreeRTOS semaphores are taken using the xSemaphoreTake() API function, the equivalent action that instead uses a task notification is ulTaskNotifyTake().

When a task is using its notification value as a binary or counting semaphore other tasks should send notifications to it using the xTaskNotifyGive() macro, or xTaskNotify() function with the eAction parameter set to eIncrement.

ulTaskNotifyTake() can either clear the task's notification value to zero on exit, in which case the notification value acts like a binary semaphore, or decrement the task's notification value on exit, in which case the notification value acts like a counting semaphore.

A task can use ulTaskNotifyTake() to [optionally] block to wait for a the task's notification value to be non-zero. The task does not consume any CPU time while it is in the Blocked state.

Where as xTaskNotifyWait() will return when a notification is pending, ulTaskNotifyTake() will return when the task's notification value is not zero.

See http://www.FreeRTOS.org/RTOS-task-notifications.html for details.

Parameters

xClearCountOnExit	if xClearCountOnExit is pdFALSE then the task's notification value is decremented when the function exits. In this way the notification value acts like a counting semaphore. If xClearCountOnExit is not pdFALSE then the task's notification value is cleared to zero when the function exits. In this way the notification value acts like a binary semaphore.
xTicksToWait	The maximum amount of time that the task should wait in the Blocked state for the task's notification value to be greater than zero, should the count not already be greater than zero when ulTaskNotifyTake() was called. The task will not consume any processing time while it is in the Blocked state. This is specified in kernel ticks, the macro pdMS_TO_TICSK(value_in_ms) can be used to convert a time specified in milliseconds to a time specified in ticks.

Returns

The task's notification count before it is either cleared to zero or decremented (see the xClearCountOnExit parameter).

5.81 xTaskNotifyStateClear

task. h

```
BaseType_t xTaskNotifyStateClear( TaskHandle_t xTask );
```

If the notification state of the task referenced by the handle xTask is eNotified, then set the task's notification state to eNotWaitingNotification. The task's notification value is not altered. Set xTask to NULL to clear the notification state of the calling task.

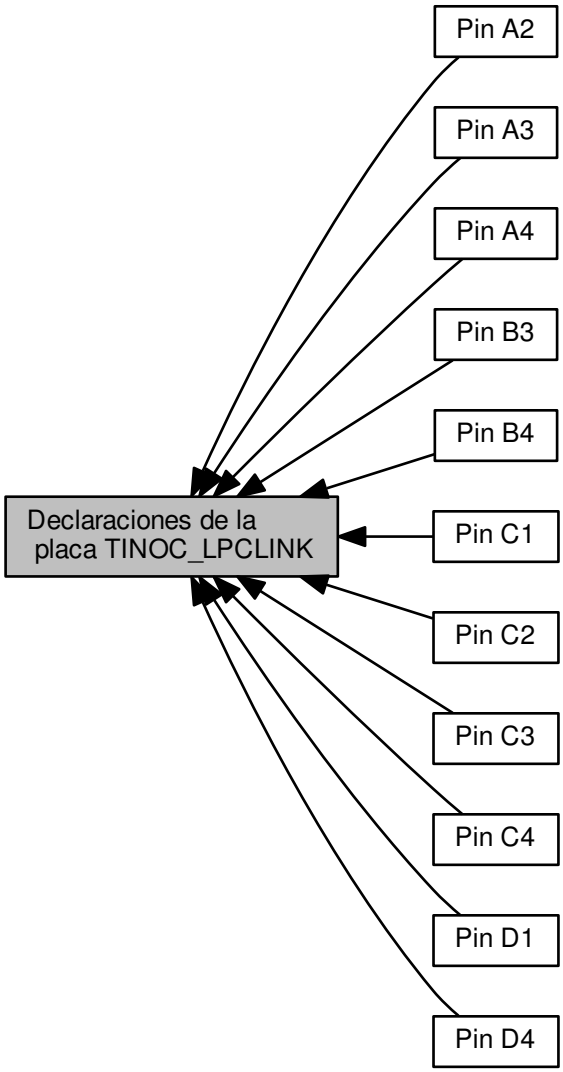
Returns

pdTRUE if the task's notification state was set to eNotWaitingNotification, otherwise pdFALSE.

5.82 Declaraciones de la placa TINOC_LPCLINK

Constantes asignadas a los pines de la placa.

Collaboration diagram for Declaraciones de la placa TINOC_LPCLINK:



Modules

- Pin C1
- Pin A2
- Pin A3
- Pin A4
- Pin B4
- Pin B3

- Pin C4
- Pin C3
- Pin D4
- Pin C2
- Pin D1

5.82.1 Detailed Description

Constantes asignadas a los pines de la placa.

5.83 Pin C1

Collaboration diagram for Pin C1:

Macros

- #define LPC1102_C1_BIT (1UL << 0UL)

  Defino el bit del puerto ocupado por C1.
- #define LPC1102_C1_PORT (0)

  Defino el port del puerto ocupado por C1.
- #define LPC1102_C1_GPIO LPC_GPIO0

  Defino el registro GPIO ocupado por C1.

5.83.1 Detailed Description

5.83.2 Macro Definition Documentation

5.83.2.1 #define LPC1102_C1_BIT (1UL << 0UL)

Defino el bit del puerto ocupado por C1.

Definition at line 28 of file tinoc_lpclink.h.

5.83.2.2 #define LPC1102_C1_GPIO LPC_GPIO0

Defino el registro GPIO ocupado por C1.

Definition at line 32 of file tinoc_lpclink.h.

5.83.2.3 #define LPC1102_C1_PORT (0)

Defino el port del puerto ocupado por C1.

Definition at line 30 of file tinoc_lpclink.h.

5.84 Pin A2

Collaboration diagram for Pin A2:

Macros

- #define LPC1102_A2_BIT (1UL << 8UL)

  Defino el port y bit del puerto ocupado por A2.
- #define LPC1102_A2_PORT (0)

  Defino el port del puerto ocupado por A2.
- #define LPC1102_A2_GPIO LPC_GPIO0

  Defino el registro GPIO ocupado por A2.

5.84.1 Detailed Description

5.84.2 Macro Definition Documentation

5.84.2.1 #define LPC1102_A2_BIT (1UL << 8UL)

Defino el port y bit del puerto ocupado por A2.

Definition at line 44 of file tinoc_lpclink.h.

5.84.2.2 #define LPC1102_A2_GPIO LPC_GPIO0

Defino el registro GPIO ocupado por A2.

Definition at line 48 of file tinoc_lpclink.h.

5.84.2.3 #define LPC1102_A2_PORT (0)

Defino el port del puerto ocupado por A2.

Definition at line 46 of file tinoc_lpclink.h.

5.85 Pin A3

Collaboration diagram for Pin A3:

Macros

- #define LPC1102_A3_BIT (1UL << 9UL)

  Defino el port y bit del puerto ocupado por A3.
- #define LPC1102_A3_PORT (0)

  Defino el port del puerto ocupado por A3.
- #define LPC1102_A3_GPIO LPC_GPIO0

  Defino el registro GPIO ocupado por A3.

5.85.1 Detailed Description

5.85.2 Macro Definition Documentation

5.85.2.1 #define LPC1102_A3_BIT (1UL << 9UL)

Defino el port y bit del puerto ocupado por A3.

Definition at line 61 of file tinoc_lpclink.h.

5.85.2.2 #define LPC1102_A3_GPIO LPC_GPIO0

Defino el registro GPIO ocupado por A3.

Definition at line 65 of file tinoc_lpclink.h.

5.85.2.3 #define LPC1102_A3_PORT (0)

Defino el port del puerto ocupado por A3.

Definition at line 63 of file tinoc_lpclink.h.

5.86 Pin A4

Collaboration diagram for Pin A4:

Macros

- #define LPC1102_A4_BIT (1UL << 10UL)

  Defino el port y bit del puerto ocupado por A4.
- #define LPC1102_A4_PORT (0)

  Defino el port del puerto ocupado por A4.
- #define LPC1102_A4_GPIO LPC_GPIO0

  Defino el registro GPIO ocupado por A4.

5.86.1 Detailed Description

5.86.2 Macro Definition Documentation

5.86.2.1 #define LPC1102_A4_BIT (1UL << 10UL)

Defino el port y bit del puerto ocupado por A4.

Definition at line 79 of file tinoc_lpclink.h.

5.86.2.2 #define LPC1102_A4_GPIO LPC_GPIO0

Defino el registro GPIO ocupado por A4.

Definition at line 83 of file tinoc_lpclink.h.

5.86.2.3 #define LPC1102_A4_PORT (0)

Defino el port del puerto ocupado por A4.

Definition at line 81 of file tinoc_lpclink.h.

5.87 Pin B4

Collaboration diagram for Pin B4:

Macros

- #define LPC1102_B4_BIT (1UL << 11UL)

  Defino el port y bit del puerto ocupado por B4.
- #define LPC1102_B4_PORT (0)

  Defino el port del puerto ocupado por B4.
- #define LPC1102_B4_GPIO LPC_GPIO0

  Defino el registro GPIO ocupado por B4.

5.87.1 Detailed Description

5.87.2 Macro Definition Documentation

5.87.2.1 #define LPC1102_B4_BIT (1UL << 11UL)

Defino el port y bit del puerto ocupado por B4.

Definition at line 95 of file tinoc_lpclink.h.

5.87.2.2 #define LPC1102_B4_GPIO LPC_GPIO0

Defino el registro GPIO ocupado por B4.

Definition at line 99 of file tinoc_lpclink.h.

5.87.2.3 #define LPC1102_B4_PORT (0)

Defino el port del puerto ocupado por B4.

Definition at line 97 of file tinoc_lpclink.h.

5.88 Pin B3

Collaboration diagram for Pin B3:

Macros

- #define LPC1102_B3_BIT (1UL << 0UL)

  Defino el port y bit del puerto ocupado por B3.
- #define LPC1102_B3_PORT (1)

  Defino el port del puerto ocupado por B3.
- #define LPC1102_B3_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por B3.

5.88.1 Detailed Description

5.88.2 Macro Definition Documentation

5.88.2.1 #define LPC1102_B3_BIT (1UL << 0UL)

Defino el port y bit del puerto ocupado por B3.

Definition at line 112 of file tinoc_lpclink.h.

5.88.2.2 #define LPC1102_B3_GPIO LPC_GPIO1

Defino el registro GPIO ocupado por B3.

Definition at line 116 of file tinoc_lpclink.h.

5.88.2.3 #define LPC1102_B3_PORT (1)

Defino el port del puerto ocupado por B3.

Definition at line 114 of file tinoc_lpclink.h.

5.89 Pin C4

Collaboration diagram for Pin C4:

Macros

- #define LPC1102_C4_BIT (1UL << 1UL)

  Defino el port y bit del puerto ocupado por C4.
- #define LPC1102_C4_PORT (1)

  Defino el port del puerto ocupado por C4.
- #define LPC1102_C4_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por C4.

5.89.1 Detailed Description

5.89.2 Macro Definition Documentation

5.89.2.1 #define LPC1102_C4_BIT (1UL << 1UL)

Defino el port y bit del puerto ocupado por C4.

Definition at line 128 of file tinoc_lpclink.h.

5.89.2.2 #define LPC1102_C4_GPIO LPC_GPIO1

Defino el registro GPIO ocupado por C4.

Definition at line 132 of file tinoc_lpclink.h.

5.89.2.3 #define LPC1102_C4_PORT (1)

Defino el port del puerto ocupado por C4.

Definition at line 130 of file tinoc_lpclink.h.

5.90 Pin C3

Collaboration diagram for Pin C3:

Macros

- #define LPC1102_C3_BIT (1UL << 2UL)

  Defino el port y bit del puerto ocupado por C3.
- #define LPC1102_C3_PORT (1)

  Defino el port del puerto ocupado por C3.
- #define LPC1102_C3_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por C3.

5.90.1 Detailed Description

5.90.2 Macro Definition Documentation

5.90.2.1 #define LPC1102_C3_BIT (1UL << 2UL)

Defino el port y bit del puerto ocupado por C3.

Definition at line 144 of file tinoc_lpclink.h.

5.90.2.2 #define LPC1102_C3_GPIO LPC_GPIO1

Defino el registro GPIO ocupado por C3.

Definition at line 148 of file tinoc_lpclink.h.

5.90.2.3 #define LPC1102_C3_PORT (1)

Defino el port del puerto ocupado por C3.

Definition at line 146 of file tinoc_lpclink.h.

5.91 Pin D4

Collaboration diagram for Pin D4:

Macros

- #define LPC1102_D4_BIT (1UL << 3UL)

  Defino el port y bit del puerto ocupado por D4.
- #define LPC1102_D4_PORT (1)

  Defino el port del puerto ocupado por D4.
- #define LPC1102_D4_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por D4.

5.91.1 Detailed Description

5.91.2 Macro Definition Documentation

5.91.2.1 #define LPC1102_D4_BIT (1UL << 3UL)

Defino el port y bit del puerto ocupado por D4.

Definition at line 160 of file tinoc_lpclink.h.

5.91.2.2 #define LPC1102_D4_GPIO LPC_GPIO1

Defino el registro GPIO ocupado por D4.

Definition at line 164 of file tinoc_lpclink.h.

5.91.2.3 #define LPC1102_D4_PORT (1)

Defino el port del puerto ocupado por D4.

Definition at line 162 of file tinoc_lpclink.h.

5.92 Pin C2

Collaboration diagram for Pin C2:

Macros

- #define LPC1102_C2_BIT (1UL << 6UL)

  Defino el port y bit del puerto ocupado por C2.
- #define LPC1102_C2_PORT (1)

  Defino el port del puerto ocupado por C2.
- #define LPC1102_C2_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por C2.

5.92.1 Detailed Description

5.92.2 Macro Definition Documentation

5.92.2.1 #define LPC1102_C2_BIT (1UL << 6UL)

Defino el port y bit del puerto ocupado por C2.

Definition at line 176 of file tinoc_lpclink.h.

5.92.2.2 #define LPC1102_C2_GPIO LPC_GPIO1

Defino el registro GPIO ocupado por C2.

Definition at line 180 of file tinoc_lpclink.h.

5.92.2.3 #define LPC1102_C2_PORT (1)

Defino el port del puerto ocupado por C2.

Definition at line 178 of file tinoc_lpclink.h.

5.93 Pin D1

Collaboration diagram for Pin D1:

Macros

- #define LPC1102_D1_BIT (1UL << 7UL)

  Defino bit del puerto ocupado por D1.
- #define LPC1102_D1_PORT (1)

  Defino el port del puerto ocupado por D1.
- #define LPC1102_D1_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por D1.

5.93.1 Detailed Description

5.93.2 Macro Definition Documentation

5.93.2.1 #define LPC1102_D1_BIT (1UL << 7UL)

Defino bit del puerto ocupado por D1.

Definition at line 192 of file tinoc_lpclink.h.

5.93.2.2 #define LPC1102_D1_GPIO LPC_GPIO1

Defino el registro GPIO ocupado por D1.

Definition at line 196 of file tinoc_lpclink.h.

5.93.2.3 #define LPC1102_D1_PORT (1)

Defino el port del puerto ocupado por D1.

Definition at line 194 of file tinoc_lpclink.h.

5.94 Definiciones del hardware asociado a los leds.

Asociacion de cada LED a pines físicos del MCU.

Macros

- #define LED1_IOCON LPC_IOCON_B4
  Registro IOCON asociado al LED1.
- #define LED1_IOCON_FUNC LPC_IOCON_B4_FUNC_PIO0_11
  Función asociada al registro IOCON.
- #define LED1_DIR_BIT (11UL)
  Bit asociado al LED1.
- #define LED1_DIR_MAS (0x01 << LED1_DIR_BIT)
  Dirección del bit LED1.
- #define LED1_DIR_REG LPC_GPIO0->DIR
  Registro de dirección.
- #define LED1_MAS_REG LPC_GPIO0->MASKED_ACCESS
  Registro IOCON asociado al LED1.

5.94.1 Detailed Description

Asociacion de cada LED a pines físicos del MCU.

5.94.2 Macro Definition Documentation

5.94.2.1 #define LED1_DIR_BIT (11UL)

Bit asociado al LED1.

Definition at line 37 of file leds.h.

5.94.2.2 #define LED1_DIR_MAS (0x01 << LED1_DIR_BIT)

Dirección del bit LED1.

Definition at line 39 of file leds.h.

5.94.2.3 #define LED1_DIR_REG LPC_GPIO0->DIR

Registro de dirección.

Definition at line 41 of file leds.h.

5.94.2.4 #define LED1_IOCON LPC_IOCON_B4

Registro IOCON asociado al LED1.

Definition at line 32 of file leds.h.

5.94.2.5 #define LED1_IOCON_FUNC LPC_IOCON_B4_FUNC_PIO0_11

Función asociada al registro IOCON.

Definition at line 34 of file leds.h.

5.94.2.6 #define LED1_MAS_REG LPC_GPIO0->MASKED_ACCESS

Registro IOCON asociado al LED1.

Definition at line 43 of file leds.h.

5.95 Declaracion de constantes de prioridades

Prioridad asignada a cada tarea.

Macros

- #define PRIORIDAD_TAREA_DELAY (tskIDLE_PRIORITY + 2)
- #define PRIORIDAD_TAREA_PARPADEAR (tskIDLE_PRIORITY + 1)

5.95.1 Detailed Description

Prioridad asignada a cada tarea.

5.95.2 Macro Definition Documentation

5.95.2.1 #define PRIORIDAD_TAREA_DELAY (tskIDLE_PRIORITY + 2)

Definition at line 56 of file leds.h.

5.95.2.2 #define PRIORIDAD_TAREA_PARPADEAR (tskIDLE_PRIORITY + 1)

Definition at line 58 of file leds.h.

5.96 Funciones "Hook" del FreeRTOS

Funciones handler de eventos de excepcion.

Functions

- void vApplicationMallocFailedHook (void)

  vApplicationMallocFailedHook() will only be called if
- void vApplicationIdleHook (void)

  vApplicationIdleHook() will only be called if configUSE_IDLE_HOOK is set
- void vApplicationStackOverflowHook (TaskHandle_t pxTask, char ∗pcTaskName)

  Run time stack overflow checking is performed if.
- void vApplicationTickHook (void)

  This function will be called by each tick interrupt if configUSE_TICK_HOOK is set to 1 in FreeRTOSConfig.h. User code can be added here, but the tick hook is called from an interrupt context, so code must not attempt to block, and only the interrupt safe FreeRTOS API functions can be used (those that end in FromISR()). Manually check the last few bytes of the interrupt stack to check they have not been overwritten. Note - the task stacks are automatically checked for overflow if configCHECK_FOR_STACK_OVERFLOW is set to 1 or 2 in FreeRTOSConifg.h, but the interrupt stack is not.

5.96.1 Detailed Description

Funciones handler de eventos de excepcion.

5.96.2 Function Documentation

5.96.2.1 void vApplicationIdleHook (void)

vApplicationIdleHook() will only be called if configUSE_IDLE_HOOK is set

void vApplicationIdleHook(void) to 1 in FreeRTOSConfig.h. It will be called on each iteration of the idle task. It is essential that code added to this hook function never attempts to block in any way (for example, call xQueue↩ Receive() with a block time specified, or call vTaskDelay()). If the application makes use of the vTaskDelete() API function (as this demo application does) then it is also important that vApplicationIdleHook() is permitted to return to its calling function, because it is the responsibility of the idle task to clean up memory allocated by the kernel to any task that has since been deleted.

Author

NXP

Definition at line 296 of file main.c.

5.96.2.2 void vApplicationMallocFailedHook (void)

vApplicationMallocFailedHook() will only be called if

void vApplicationMallocFailedHook(void) configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h. It is a hook function that will get called if a call to pvPortMalloc() fails. pvPortMalloc() is called internally by the kernel whenever a task, queue, timer or semaphore is created. It is also called by various parts of the demo application. If heap_1.c or heap_2.c are used, then the size of the heap available to pvPortMalloc() is defined by configTO↩ TAL_HEAP_SIZE in FreeRTOSConfig.h, and the xPortGetFreeHeapSize() API function can be used to query the size of free heap space that remains (although it does not provide information on how the remaining heap might be fragmented). memory, etc. are configured before main() is called.

Author

NXP

Definition at line 274 of file main.c.

5.96.2.3 void vApplicationStackOverflowHook (TaskHandle_t *pxTask*, char * *pcTaskName*)

Run time stack overflow checking is performed if.

void vApplicationStackOverflowHook(TaskHandle_t pxTask, char *pcTaskName) configCHECK_FOR_STACK_↩ OVERFLOW is defined to 1 or 2. This hook function is called if a stack overflow is detected.

Author

NXP

Definition at line 311 of file main.c.

5.96.2.4 void vApplicationTickHook (void)

This function will be called by each tick interrupt if configUSE_TICK_HOOK is set to 1 in FreeRTOSConfig.h. User code can be added here, but the tick hook is called from an interrupt context, so code must not attempt to block, and only the interrupt safe FreeRTOS API functions can be used (those that end in FromISR()). Manually check the last few bytes of the interrupt stack to check they have not been overwritten. Note - the task stacks are automatically checked for overflow if configCHECK_FOR_STACK_OVERFLOW is set to 1 or 2 in FreeRTOSConifg.h, but the interrupt stack is not.

vApplicationTickHook

Author

NXP

Definition at line 341 of file main.c.

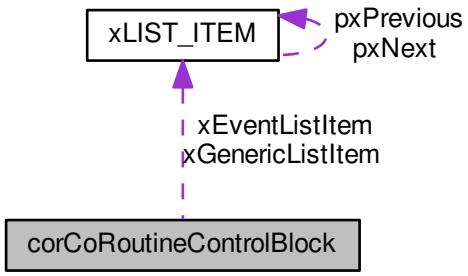
Chapter 6

Class Documentation

6.1 corCoRoutineControlBlock Struct Reference

`#include <croutine.h>`

Collaboration diagram for corCoRoutineControlBlock:



Public Attributes

- crCOROUTINE_CODE pxCoRoutineFunction
- ListItem_t xGenericListItem
- ListItem_t xEventListItem
- UBaseType_t uxPriority
- UBaseType_t uxIndex
- uint16_t uxState

6.1.1 Detailed Description

Definition at line 91 of file croutine.h.

6.1.2 Member Data Documentation

6.1.2.1 crCOROUTINE_CODE corCoRoutineControlBlock::pxCoRoutineFunction

Definition at line 93 of file croutine.h.

6.1.2.2 UBaseType_t corCoRoutineControlBlock::uxIndex

Definition at line 97 of file croutine.h.

6.1.2.3 UBaseType_t corCoRoutineControlBlock::uxPriority

Definition at line 96 of file croutine.h.

6.1.2.4 uint16_t corCoRoutineControlBlock::uxState

Definition at line 98 of file croutine.h.

6.1.2.5 ListItem_t corCoRoutineControlBlock::xEventListItem

Definition at line 95 of file croutine.h.

6.1.2.6 ListItem_t corCoRoutineControlBlock::xGenericListItem

Definition at line 94 of file croutine.h.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/croutine.h

6.2 COUNT_SEM_STRUCT Struct Reference

Public Attributes

- SemaphoreHandle_t xSemaphore
- UBaseType_t uxExpectedStartCount
- UBaseType_t uxLoopCounter

6.2.1 Detailed Description

Definition at line 129 of file countsem.c.

6.2.2 Member Data Documentation

6.2.2.1 UBaseType_t COUNT_SEM_STRUCT::uxExpectedStartCount

Definition at line 137 of file countsem.c.

6.2.2.2 UBaseType_t COUNT_SEM_STRUCT::uxLoopCounter

Definition at line 141 of file countsem.c.

6.2.2.3 SemaphoreHandle_t COUNT_SEM_STRUCT::xSemaphore

Definition at line 132 of file countsem.c.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/Common_↩ Demo_Tasks/countsem.c

6.3 HeapRegion Struct Reference

`#include <portable.h>`

Public Attributes

- uint8_t * pucStartAddress
- size_t xSizeInBytes

6.3.1 Detailed Description

Definition at line 148 of file portable.h.

6.3.2 Member Data Documentation

6.3.2.1 uint8_t* HeapRegion::pucStartAddress

Definition at line 150 of file portable.h.

6.3.2.2 size_t HeapRegion::xSizeInBytes

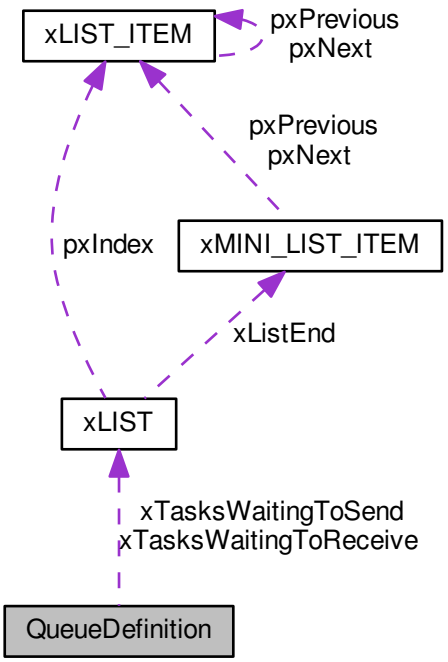
Definition at line 151 of file portable.h.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/portable.h

6.4 QueueDefinition Struct Reference

Collaboration diagram for QueueDefinition:



Public Attributes

- int8_t * pcHead
- int8_t * pcTail
- int8_t * pcWriteTo
- union {
  int8_t * pcReadFrom
  UBaseType_t uxRecursiveCallCount
  } u

- List_t xTasksWaitingToSend
- List_t xTasksWaitingToReceive
- volatile UBaseType_t uxMessagesWaiting
- UBaseType_t uxLength
- UBaseType_t uxItemSize
- volatile int8_t cRxLock
- volatile int8_t cTxLock

6.4.1 Detailed Description

Definition at line 130 of file queue.c.

6.4.2 Member Data Documentation

6.4.2.1 volatile int8_t QueueDefinition::cRxLock

Definition at line 149 of file queue.c.

6.4.2.2 volatile int8_t QueueDefinition::cTxLock

Definition at line 150 of file queue.c.

6.4.2.3 int8_t* QueueDefinition::pcHead

Definition at line 132 of file queue.c.

6.4.2.4 int8_t* QueueDefinition::pcReadFrom

Definition at line 138 of file queue.c.

6.4.2.5 int8_t* QueueDefinition::pcTail

Definition at line 133 of file queue.c.

6.4.2.6 int8_t* QueueDefinition::pcWriteTo

Definition at line 134 of file queue.c.

6.4.2.7 union { ... } QueueDefinition::u

6.4.2.8 UBaseType_t QueueDefinition::uxItemSize

Definition at line 147 of file queue.c.

6.4.2.9 UBaseType_t QueueDefinition::uxLength

Definition at line 146 of file queue.c.

6.4.2.10 volatile UBaseType_t QueueDefinition::uxMessagesWaiting

Definition at line 145 of file queue.c.

6.4.2.11 UBaseType_t QueueDefinition::uxRecursiveCallCount

Definition at line 139 of file queue.c.

6.4.2.12 List_t QueueDefinition::xTasksWaitingToReceive

Definition at line 143 of file queue.c.

6.4.2.13 List_t QueueDefinition::xTasksWaitingToSend

Definition at line 142 of file queue.c.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/queue.c

6.5 t_cola Struct Reference

`#include <colas.h>`

Public Attributes

- uint8_t buffer [BUFFER_N]

  Buffer de la cola.
- uint8_t ini
- uint8_t fin
- t_estado_buffer estado_buffer
- uint8_t datos_nuevos

6.5.1 Detailed Description

Definition at line 47 of file colas.h.

6.5.2 Member Data Documentation

6.5.2.1 uint8_t t_cola::buffer[BUFFER_N]

Buffer de la cola.

Definition at line 50 of file colas.h.

6.5.2.2 uint8_t t_cola::datos_nuevos

Definition at line 53 of file colas.h.

6.5.2.3 t_estado_buffer t_cola::estado_buffer

Definition at line 52 of file colas.h.

6.5.2.4 uint8_t t_cola::fin

Definition at line 51 of file colas.h.

6.5.2.5 uint8_t t_cola::ini

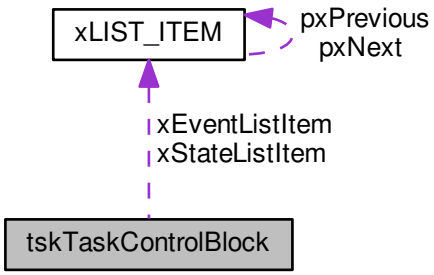
Definition at line 51 of file colas.h.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/UART/colas.h

6.6 tskTaskControlBlock Struct Reference

Collaboration diagram for tskTaskControlBlock:



Public Attributes

- volatile StackType_t * pxTopOfStack
- ListItem_t xStateListItem
- ListItem_t xEventListItem
- UBaseType_t uxPriority
- StackType_t * pxStack
- char pcTaskName [configMAX_TASK_NAME_LEN]

6.6.1 Detailed Description

Definition at line 293 of file tasks.c.

6.6.2 Member Data Documentation

6.6.2.1 char tskTaskControlBlock::pcTaskName[configMAX_TASK_NAME_LEN]

Definition at line 305 of file tasks.c.

6.6.2.2 StackType_t* tskTaskControlBlock::pxStack

Definition at line 304 of file tasks.c.

6.6.2.3 volatile StackType_t* tskTaskControlBlock::pxTopOfStack

Definition at line 295 of file tasks.c.

6.6.2.4 UBaseType_t tskTaskControlBlock::uxPriority

Definition at line 303 of file tasks.c.

6.6.2.5 ListItem_t tskTaskControlBlock::xEventListItem

Definition at line 302 of file tasks.c.

6.6.2.6 ListItem_t tskTaskControlBlock::xStateListItem

Definition at line 301 of file tasks.c.

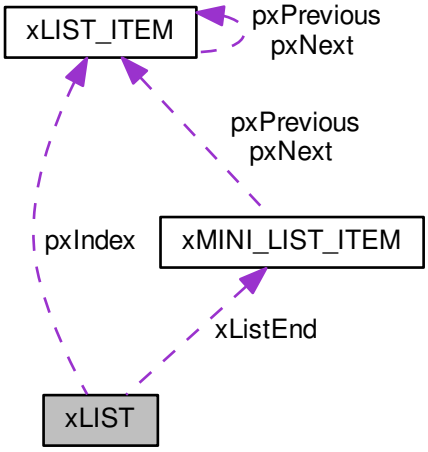
The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/tasks.c

6.7 xLIST Struct Reference

`#include <list.h>`

Collaboration diagram for xLIST:



Public Attributes

- listFIRST_LIST_INTEGRITY_CHECK_VALUE configLIST_VOLATILE UBaseType_t uxNumberOfItems
- ListItem_t ∗configLIST_VOLATILE pxIndex
- MiniListItem_t xListEnd

6.7.1 Detailed Description

Definition at line 205 of file list.h.

6.7.2 Member Data Documentation

6.7.2.1 ListItem_t∗ configLIST_VOLATILE xLIST::pxIndex

Definition at line 209 of file list.h.

6.7.2.2 listFIRST_LIST_INTEGRITY_CHECK_VALUE configLIST_VOLATILE UBaseType_t xLIST::uxNumberOfItems

Definition at line 208 of file list.h.

6.7.2.3 MiniListItem_t xLIST::xListEnd

Definition at line 210 of file list.h.

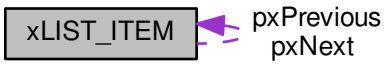
The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/list.h

6.8 xLIST_ITEM Struct Reference

`#include <list.h>`

Collaboration diagram for xLIST_ITEM:



Public Attributes

- listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE configLIST_VOLATILE TickType_t xItemValue
- struct xLIST_ITEM ∗configLIST_VOLATILE pxNext
- struct xLIST_ITEM ∗configLIST_VOLATILE pxPrevious
- void ∗ pvOwner
- void ∗configLIST_VOLATILE pvContainer

6.8.1 Detailed Description

Definition at line 181 of file list.h.

6.8.2 Member Data Documentation

6.8.2.1 void∗ configLIST_VOLATILE xLIST_ITEM::pvContainer

Definition at line 188 of file list.h.

6.8.2.2 void∗ xLIST_ITEM::pvOwner

Definition at line 187 of file list.h.

6.8.2.3 struct xLIST_ITEM* configLIST_VOLATILE xLIST_ITEM::pxNext

Definition at line 185 of file list.h.

6.8.2.4 struct xLIST_ITEM* configLIST_VOLATILE xLIST_ITEM::pxPrevious

Definition at line 186 of file list.h.

6.8.2.5 listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE configLIST_VOLATILE TickType_t xLIST_ITEM::xItemValue

Definition at line 184 of file list.h.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/list.h

6.9 xMEMORY_REGION Struct Reference

`#include <task.h>`

Public Attributes

- void * pvBaseAddress
- uint32_t ulLengthInBytes
- uint32_t ulParameters

6.9.1 Detailed Description

Definition at line 144 of file task.h.

6.9.2 Member Data Documentation

6.9.2.1 void* xMEMORY_REGION::pvBaseAddress

Definition at line 146 of file task.h.

6.9.2.2 uint32_t xMEMORY_REGION::ulLengthInBytes

Definition at line 147 of file task.h.

6.9.2.3 uint32_t xMEMORY_REGION::ulParameters

Definition at line 148 of file task.h.

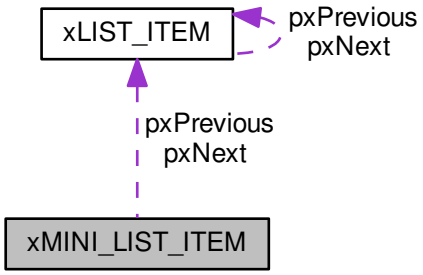
The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/task.h

6.10 xMINI_LIST_ITEM Struct Reference

`#include <list.h>`

Collaboration diagram for xMINI_LIST_ITEM:



Public Attributes

- listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE configLIST_VOLATILE TickType_t xItemValue
- struct xLIST_ITEM ∗configLIST_VOLATILE pxNext
- struct xLIST_ITEM ∗configLIST_VOLATILE pxPrevious

6.10.1 Detailed Description

Definition at line 193 of file list.h.

6.10.2 Member Data Documentation

6.10.2.1 struct xLIST_ITEM∗ configLIST_VOLATILE xMINI_LIST_ITEM::pxNext

Definition at line 197 of file list.h.

6.10.2.2 struct xLIST_ITEM* configLIST_VOLATILE xMINI_LIST_ITEM::pxPrevious

Definition at line 198 of file list.h.

6.10.2.3 listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE configLIST_VOLATILE TickType_t xMINI_LIST_ITEM::xItemValue

Definition at line 196 of file list.h.

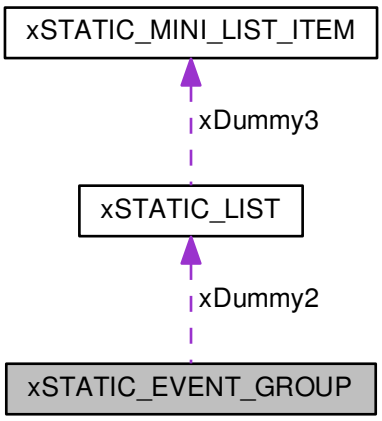
The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/list.h

6.11 xSTATIC_EVENT_GROUP Struct Reference

`#include <FreeRTOS.h>`

Collaboration diagram for xSTATIC_EVENT_GROUP:



Public Attributes

- TickType_t xDummy1
- StaticList_t xDummy2

6.11.1 Detailed Description

Definition at line 1012 of file FreeRTOS.h.

6.11.2 Member Data Documentation

6.11.2.1 TickType_t xSTATIC_EVENT_GROUP::xDummy1

Definition at line 1014 of file FreeRTOS.h.

6.11.2.2 StaticList_t xSTATIC_EVENT_GROUP::xDummy2

Definition at line 1015 of file FreeRTOS.h.

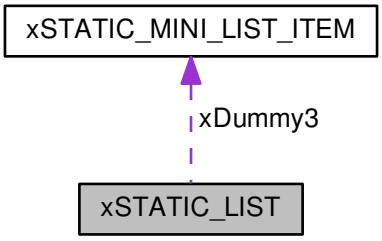
The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/FreeRTOS.h

6.12 xSTATIC_LIST Struct Reference

`#include <FreeRTOS.h>`

Collaboration diagram for xSTATIC_LIST:



Public Attributes

- UBaseType_t uxDummy1
- void * pvDummy2
- StaticMiniListItem_t xDummy3

6.12.1 Detailed Description

Definition at line 890 of file FreeRTOS.h.

6.12.2 Member Data Documentation

6.12.2.1 void* xSTATIC_LIST::pvDummy2

Definition at line 893 of file FreeRTOS.h.

6.12.2.2 UBaseType_t xSTATIC_LIST::uxDummy1

Definition at line 892 of file FreeRTOS.h.

6.12.2.3 StaticMiniListItem_t xSTATIC_LIST::xDummy3

Definition at line 894 of file FreeRTOS.h.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/FreeRTOS.h

6.13 xSTATIC_LIST_ITEM Struct Reference

`#include <FreeRTOS.h>`

Public Attributes

- TickType_t xDummy1
- void * pvDummy2 [4]

6.13.1 Detailed Description

Definition at line 874 of file FreeRTOS.h.

6.13.2 Member Data Documentation

6.13.2.1 void* xSTATIC_LIST_ITEM::pvDummy2[4]

Definition at line 877 of file FreeRTOS.h.

6.13.2.2 TickType_t xSTATIC_LIST_ITEM::xDummy1

Definition at line 876 of file FreeRTOS.h.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/FreeRTOS.h

6.14 xSTATIC_MINI_LIST_ITEM Struct Reference

`#include <FreeRTOS.h>`

Public Attributes

- TickType_t xDummy1
- void * pvDummy2 [2]

6.14.1 Detailed Description

Definition at line 882 of file FreeRTOS.h.

6.14.2 Member Data Documentation

6.14.2.1 void* xSTATIC_MINI_LIST_ITEM::pvDummy2[2]

Definition at line 885 of file FreeRTOS.h.

6.14.2.2 TickType_t xSTATIC_MINI_LIST_ITEM::xDummy1

Definition at line 884 of file FreeRTOS.h.

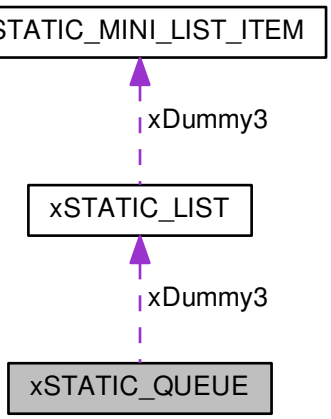
The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/FreeRTOS.h

6.15 xSTATIC_QUEUE Struct Reference

`#include <FreeRTOS.h>`

Collaboration diagram for xSTATIC_QUEUE:



Public Attributes

- void * pvDummy1 [3]
- union {
  void * pvDummy2
  UBaseType_t uxDummy2
  } u

- StaticList_t xDummy3 [2]
- UBaseType_t uxDummy4 [3]
- uint8_t ucDummy5 [2]

6.15.1 Detailed Description

Definition at line 968 of file FreeRTOS.h.

6.15.2 Member Data Documentation

6.15.2.1 void* xSTATIC_QUEUE::pvDummy1[3]

Definition at line 970 of file FreeRTOS.h.

6.15.2.2 void* xSTATIC_QUEUE::pvDummy2

Definition at line 974 of file FreeRTOS.h.

6.15.2.3 union { ... } xSTATIC_QUEUE::u

6.15.2.4 uint8_t xSTATIC_QUEUE::ucDummy5[2]

Definition at line 980 of file FreeRTOS.h.

6.15.2.5 UBaseType_t xSTATIC_QUEUE::uxDummy2

Definition at line 975 of file FreeRTOS.h.

6.15.2.6 UBaseType_t xSTATIC_QUEUE::uxDummy4[3]

Definition at line 979 of file FreeRTOS.h.

6.15.2.7 StaticList_t xSTATIC_QUEUE::xDummy3[2]

Definition at line 978 of file FreeRTOS.h.

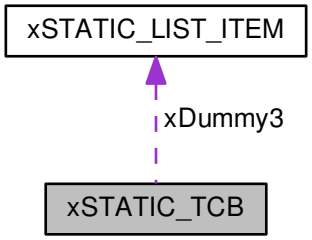
The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/FreeRTOS.h

6.16 xSTATIC_TCB Struct Reference

`#include <FreeRTOS.h>`

Collaboration diagram for xSTATIC_TCB:



Public Attributes

- void * pxDummy1
- StaticListItem_t xDummy3 [2]
- UBaseType_t uxDummy5
- void * pxDummy6
- uint8_t ucDummy7 [configMAX_TASK_NAME_LEN]
- uint32_t ulDummy18
- uint8_t ucDummy19

6.16.1 Detailed Description

Definition at line 910 of file FreeRTOS.h.

6.16.2 Member Data Documentation

6.16.2.1 void* xSTATIC_TCB::pxDummy1

Definition at line 912 of file FreeRTOS.h.

6.16.2.2 void* xSTATIC_TCB::pxDummy6

Definition at line 918 of file FreeRTOS.h.

6.16.2.3 uint8_t xSTATIC_TCB::ucDummy19

Definition at line 946 of file FreeRTOS.h.

6.16.2.4 uint8_t xSTATIC_TCB::ucDummy7[configMAX_TASK_NAME_LEN]

Definition at line 919 of file FreeRTOS.h.

6.16.2.5 uint32_t xSTATIC_TCB::ulDummy18

Definition at line 945 of file FreeRTOS.h.

6.16.2.6 UBaseType_t xSTATIC_TCB::uxDummy5

Definition at line 917 of file FreeRTOS.h.

6.16.2.7 StaticListItem_t xSTATIC_TCB::xDummy3[2]

Definition at line 916 of file FreeRTOS.h.

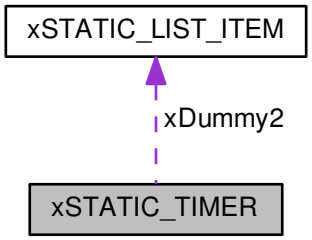
The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/FreeRTOS.h

6.17 xSTATIC_TIMER Struct Reference

`#include <FreeRTOS.h>`

Collaboration diagram for xSTATIC_TIMER:



Public Attributes

- void * pvDummy1
- StaticListItem_t xDummy2
- TickType_t xDummy3
- UBaseType_t uxDummy4
- void * pvDummy5 [2]

6.17.1 Detailed Description

Definition at line 1041 of file FreeRTOS.h.

6.17.2 Member Data Documentation

6.17.2.1 void* xSTATIC_TIMER::pvDummy1

Definition at line 1043 of file FreeRTOS.h.

6.17.2.2 void* xSTATIC_TIMER::pvDummy5[2]

Definition at line 1047 of file FreeRTOS.h.

6.17.2.3 UBaseType_t xSTATIC_TIMER::uxDummy4

Definition at line 1046 of file FreeRTOS.h.

6.17.2.4 StaticListItem_t xSTATIC_TIMER::xDummy2

Definition at line 1044 of file FreeRTOS.h.

6.17.2.5 TickType_t xSTATIC_TIMER::xDummy3

Definition at line 1045 of file FreeRTOS.h.

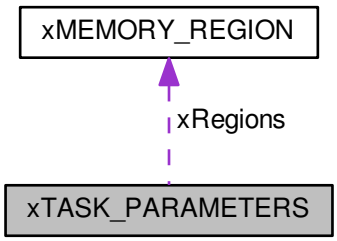
The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/FreeRTOS.h

6.18 xTASK_PARAMETERS Struct Reference

`#include <task.h>`

Collaboration diagram for xTASK_PARAMETERS:



Public Attributes

- TaskFunction_t pvTaskCode
- const char ∗const pcName
- uint16_t usStackDepth
- void ∗ pvParameters
- UBaseType_t uxPriority
- StackType_t ∗ puxStackBuffer
- MemoryRegion_t xRegions [portNUM_CONFIGURABLE_REGIONS]

6.18.1 Detailed Description

Definition at line 154 of file task.h.

6.18.2 Member Data Documentation

6.18.2.1 const char∗ const xTASK_PARAMETERS::pcName

Definition at line 157 of file task.h.

6.18.2.2 StackType_t∗ xTASK_PARAMETERS::puxStackBuffer

Definition at line 161 of file task.h.

6.18.2.3 void∗ xTASK_PARAMETERS::pvParameters

Definition at line 159 of file task.h.

6.18.2.4 TaskFunction_t xTASK_PARAMETERS::pvTaskCode

Definition at line 156 of file task.h.

6.18.2.5 uint16_t xTASK_PARAMETERS::usStackDepth

Definition at line 158 of file task.h.

6.18.2.6 UBaseType_t xTASK_PARAMETERS::uxPriority

Definition at line 160 of file task.h.

6.18.2.7 MemoryRegion_t xTASK_PARAMETERS::xRegions[portNUM_CONFIGURABLE_REGIONS]

Definition at line 162 of file task.h.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/task.h

6.19 xTASK_STATUS Struct Reference

`#include <task.h>`

Public Attributes

- TaskHandle_t xHandle
- const char * pcTaskName
- UBaseType_t xTaskNumber
- eTaskState eCurrentState
- UBaseType_t uxCurrentPriority
- UBaseType_t uxBasePriority
- uint32_t ulRunTimeCounter
- StackType_t * pxStackBase
- uint16_t usStackHighWaterMark

6.19.1 Detailed Description

Definition at line 167 of file task.h.

6.19.2 Member Data Documentation

6.19.2.1 eTaskState xTASK_STATUS::eCurrentState

Definition at line 172 of file task.h.

6.19.2.2 const char* xTASK_STATUS::pcTaskName

Definition at line 170 of file task.h.

6.19.2.3 StackType_t* xTASK_STATUS::pxStackBase

Definition at line 176 of file task.h.

6.19.2.4 uint32_t xTASK_STATUS::ulRunTimeCounter

Definition at line 175 of file task.h.

6.19.2.5 uint16_t xTASK_STATUS::usStackHighWaterMark

Definition at line 177 of file task.h.

6.19.2.6 UBaseType_t xTASK_STATUS::uxBasePriority

Definition at line 174 of file task.h.

6.19.2.7 UBaseType_t xTASK_STATUS::uxCurrentPriority

Definition at line 173 of file task.h.

6.19.2.8 TaskHandle_t xTASK_STATUS::xHandle

Definition at line 169 of file task.h.

6.19.2.9 UBaseType_t xTASK_STATUS::xTaskNumber

Definition at line 171 of file task.h.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/task.h

6.20 xTIME_OUT Struct Reference

`#include <task.h>`

Public Attributes

- BaseType_t xOverflowCount
- TickType_t xTimeOnEntering

6.20.1 Detailed Description

Definition at line 135 of file task.h.

6.20.2 Member Data Documentation

6.20.2.1 BaseType_t xTIME_OUT::xOverflowCount

Definition at line 137 of file task.h.

6.20.2.2 TickType_t xTIME_OUT::xTimeOnEntering

Definition at line 138 of file task.h.

The documentation for this struct was generated from the following file:

- /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_↩ Source/include/task.h

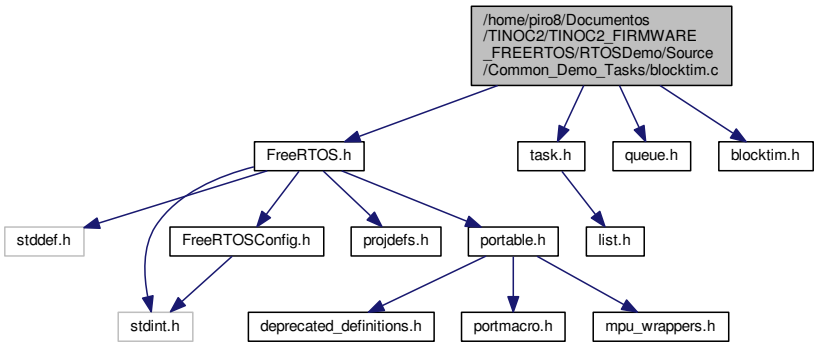
Chapter 7

File Documentation

7.1 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/Common_Demo_Tasks/blocktim.c File Reference

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "blocktim.h"
```

Include dependency graph for blocktim.c:



Macros

- #define bktPRIMARY_PRIORITY (configMAX_PRIORITIES - 3)
- #define bktSECONDARY_PRIORITY (configMAX_PRIORITIES - 4)
- #define bktQUEUE_LENGTH (5)
- #define bktSHORT_WAIT pdMS_TO_TICKS((TickType_t) 20)
- #define bktPRIMARY_BLOCK_TIME (10)
- #define bktALLOWABLE_MARGIN (15)
- #define bktTIME_TO_BLOCK (175)
- #define bktDONT_BLOCK ((TickType_t) 0)
- #define bktRUN_INDICATOR ((UBaseType_t) 0x55)
- #define configTIMER_TASK_PRIORITY (configMAX_PRIORITIES - 1)

Functions

- static void vPrimaryBlockTimeTestTask (void *pvParameters)
- static void vSecondaryBlockTimeTestTask (void *pvParameters)
- static void prvBasicDelayTests (void)
- void vCreateBlockTimeTasks (void)
- BaseType_t xAreBlockTimeTestTasksStillRunning (void)

Variables

- static QueueHandle_t xTestQueue
- static TaskHandle_t xSecondary
- static volatile BaseType_t xPrimaryCycles = 0
- static volatile BaseType_t xSecondaryCycles = 0
- static volatile BaseType_t xErrorOccurred = pdFALSE
- static volatile UBaseType_t xRunIndicator

7.1.1 Macro Definition Documentation

7.1.1.1 #define bktALLOWABLE_MARGIN (15)

Definition at line 97 of file blocktim.c.

7.1.1.2 #define bktDONT_BLOCK ((TickType_t) 0)

Definition at line 99 of file blocktim.c.

7.1.1.3 #define bktPRIMARY_BLOCK_TIME (10)

Definition at line 96 of file blocktim.c.

7.1.1.4 #define bktPRIMARY_PRIORITY (configMAX_PRIORITIES - 3)

Definition at line 86 of file blocktim.c.

7.1.1.5 #define bktQUEUE_LENGTH (5)

Definition at line 94 of file blocktim.c.

7.1.1.6 #define bktRUN_INDICATOR ((UBaseType_t) 0x55)

Definition at line 100 of file blocktim.c.

7.1.1.7 #define bktSECONDARY_PRIORITY (configMAX_PRIORITIES - 4)

Definition at line 90 of file blocktim.c.

7.1.1.8 #define bktSHORT_WAIT pdMS_TO_TICKS((TickType_t) 20)

Definition at line 95 of file blocktim.c.

7.1.1.9 #define bktTIME_TO_BLOCK (175)

Definition at line 98 of file blocktim.c.

7.1.1.10 #define configTIMER_TASK_PRIORITY (configMAX_PRIORITIES - 1)

Definition at line 105 of file blocktim.c.

7.1.2 Function Documentation

7.1.2.1 static void prvBasicDelayTests (void) [static]

Definition at line 510 of file blocktim.c.

7.1.2.2 void vCreateBlockTimeTasks (void)

Definition at line 140 of file blocktim.c.

7.1.2.3 static void vPrimaryBlockTimeTestTask (void * *pvParameters*) [static]

Definition at line 162 of file blocktim.c.

7.1.2.4 static void vSecondaryBlockTimeTestTask (void * *pvParameters*) [static]

Definition at line 420 of file blocktim.c.

7.1.2.5 BaseType_t xAreBlockTimeTestTasksStillRunning (void)

Definition at line 556 of file blocktim.c.

7.1.3 Variable Documentation

7.1.3.1 volatile BaseType_t xErrorOccurred = pdFALSE [static]

Definition at line 132 of file blocktim.c.

7.1.3.2 volatile BaseType_t xPrimaryCycles = 0 [static]

Definition at line 131 of file blocktim.c.

7.1.3.3 volatile UBaseType_t xRunIndicator [static]

Definition at line 136 of file blocktim.c.

7.1.3.4 TaskHandle_t xSecondary [static]

Definition at line 128 of file blocktim.c.

7.1.3.5 volatile BaseType_t xSecondaryCycles = 0 [static]

Definition at line 131 of file blocktim.c.

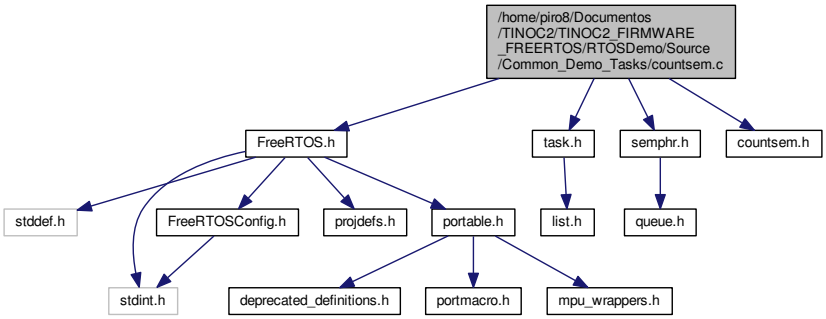
7.1.3.6 QueueHandle_t xTestQueue [static]

Definition at line 124 of file blocktim.c.

7.2 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/Common_Demo_Tasks/countsem.c File Reference

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "countsem.h"
```

Include dependency graph for countsem.c:



Classes

- struct COUNT_SEM_STRUCT

Macros

- #define countMAX_COUNT_VALUE (200)
- #define countSTART_AT_MAX_COUNT (0xaa)
- #define countSTART_AT_ZERO (0x55)
- #define countNUM_TEST_TASKS (2)
- #define countDONT_BLOCK (0)

Typedefs

- typedef struct COUNT_SEM_STRUCT xCountSemStruct

Functions

- static void prvCountingSemaphoreTask (void *pvParameters)
- static void prvIncrementSemaphoreCount (SemaphoreHandle_t xSemaphore, UBaseType_t *puxLoop↩
Counter)
- static void prvDecrementSemaphoreCount (SemaphoreHandle_t xSemaphore, UBaseType_t *puxLoop↩
Counter)
- void vStartCountingSemaphoreTasks (void)
- BaseType_t xAreCountingSemaphoreTasksStillRunning (void)

Variables

- static volatile BaseType_t xErrorDetected = pdFALSE
- static volatile xCountSemStruct xParameters [countNUM_TEST_TASKS]

7.2.1 Macro Definition Documentation

7.2.1.1 #define countDONT_BLOCK (0)

Definition at line 96 of file countsem.c.

7.2.1.2 #define countMAX_COUNT_VALUE (200)

Definition at line 84 of file countsem.c.

7.2.1.3 #define countNUM_TEST_TASKS (2)

Definition at line 95 of file countsem.c.

7.2.1.4 #define countSTART_AT_MAX_COUNT (0xaa)

Definition at line 90 of file countsem.c.

7.2.1.5 #define countSTART_AT_ZERO (0x55)

Definition at line 91 of file countsem.c.

7.2.2 Typedef Documentation

7.2.2.1 typedef struct COUNT_SEM_STRUCT xCountSemStruct

7.2.3 Function Documentation

7.2.3.1 static void prvCountingSemaphoreTask (void * *pvParameters*) [static]

Definition at line 258 of file countsem.c.

7.2.3.2 static void prvDecrementSemaphoreCount (SemaphoreHandle_t *xSemaphore,* UBaseType_t * *puxLoopCounter*) [static]

Definition at line 181 of file countsem.c.

7.2.3.3 static void prvIncrementSemaphoreCount (SemaphoreHandle_t *xSemaphore,* UBaseType_t * *puxLoopCounter*) [static]

Definition at line 220 of file countsem.c.

7.2.3.4 void vStartCountingSemaphoreTasks (void)

Definition at line 149 of file countsem.c.

7.2.3.5 BaseType_t xAreCountingSemaphoreTasksStillRunning (void)

Definition at line 296 of file countsem.c.

7.2.4 Variable Documentation

7.2.4.1 volatile BaseType_t xErrorDetected = pdFALSE [static]

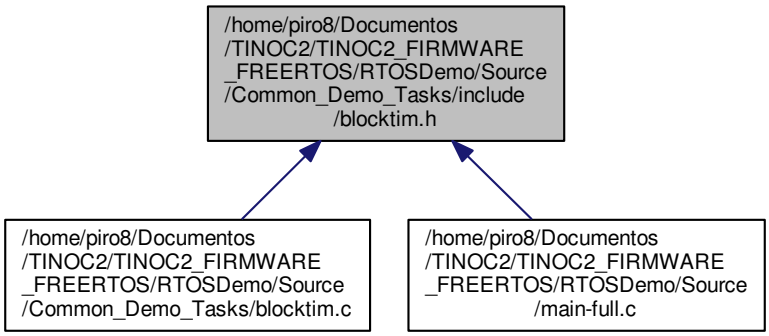
Definition at line 102 of file countsem.c.

7.2.4.2 volatile xCountSemStruct xParameters[countNUM_TEST_TASKS] [static]

Definition at line 145 of file countsem.c.

7.3 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/Common_Demo_Tasks/include/blocktim.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void vCreateBlockTimeTasks (void)
- BaseType_t xAreBlockTimeTestTasksStillRunning (void)

7.3.1 Function Documentation

7.3.1.1 void vCreateBlockTimeTasks (void)

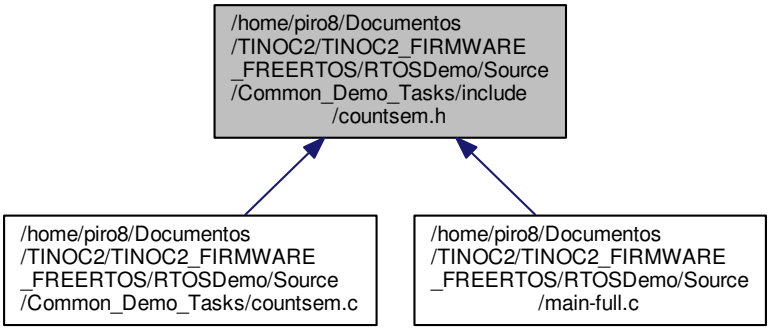
Definition at line 140 of file blocktim.c.

7.3.1.2 BaseType_t xAreBlockTimeTestTasksStillRunning (void)

Definition at line 556 of file blocktim.c.

7.4 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/Common_Demo_Tasks/include/countsem.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void vStartCountingSemaphoreTasks (void)
- BaseType_t xAreCountingSemaphoreTasksStillRunning (void)

7.4.1 Function Documentation

7.4.1.1 void vStartCountingSemaphoreTasks (void)

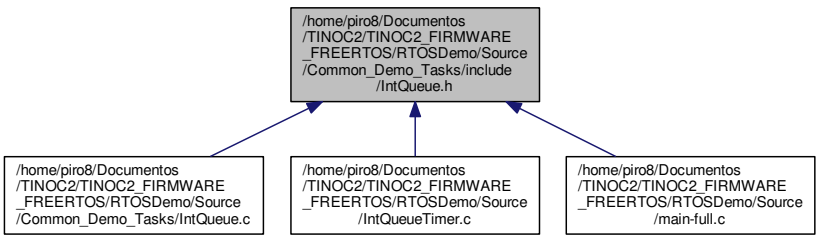
Definition at line 149 of file countsem.c.

7.4.1.2 BaseType_t xAreCountingSemaphoreTasksStillRunning (void)

Definition at line 296 of file countsem.c.

7.5 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/Common_Demo_Tasks/include/IntQueue.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void vStartInterruptQueueTasks (void)
- BaseType_t xAreIntQueueTasksStillRunning (void)
- BaseType_t xFirstTimerHandler (void)
- BaseType_t xSecondTimerHandler (void)

7.5.1 Function Documentation

7.5.1.1 void vStartInterruptQueueTasks (void)

Definition at line 239 of file IntQueue.c.

7.5.1.2 BaseType_t xAreIntQueueTasksStillRunning (void)

Definition at line 728 of file IntQueue.c.

7.5.1.3 BaseType_t xFirstTimerHandler (void)

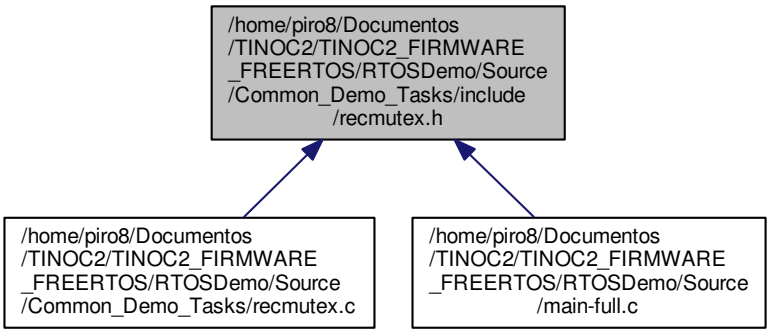
Definition at line 668 of file IntQueue.c.

7.5.1.4 BaseType_t xSecondTimerHandler (void)

Definition at line 696 of file IntQueue.c.

7.6 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/Common_Demo_Tasks/include/recmutex.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void vStartRecursiveMutexTasks (void)
- BaseType_t xAreRecursiveMutexTasksStillRunning (void)

7.6.1 Function Documentation

7.6.1.1 void vStartRecursiveMutexTasks (void)

Definition at line 148 of file recmutex.c.

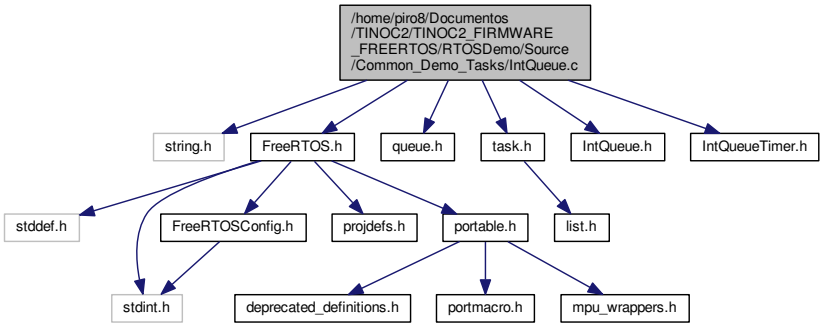
7.6.1.2 BaseType_t xAreRecursiveMutexTasksStillRunning (void)

Definition at line 405 of file recmutex.c.

7.7 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/Common_Demo_Tasks/IntQueue.c File Reference

```
#include <string.h>
#include "FreeRTOS.h"
#include "queue.h"
#include "task.h"
#include "IntQueue.h"
#include "IntQueueTimer.h"
```

Include dependency graph for IntQueue.c:



Macros

- #define intqHIGHER_PRIORITY (configMAX_PRIORITIES - 2)
- #define intqLOWER_PRIORITY (tskIDLE_PRIORITY)
- #define intqNUM_VALUES_TO_LOG (200)
- #define intqSHORT_DELAY (140)
- #define intqVALUE_OVERRUN (50)
- #define intqONE_TICK_DELAY (1)
- #define intqHIGH_PRIORITY_TASK1 ((UBaseType_t) 1)
- #define intqHIGH_PRIORITY_TASK2 ((UBaseType_t) 2)
- #define intqLOW_PRIORITY_TASK ((UBaseType_t) 3)
- #define intqFIRST_INTERRUPT ((UBaseType_t) 4)
- #define intqSECOND_INTERRUPT ((UBaseType_t) 5)
- #define intqQUEUE_LENGTH ((UBaseType_t) 10)
- #define intqMIN_ACCEPTABLE_TASK_COUNT (5)
- #define timerNORMALLY_EMPTY_TX()
- #define timerNORMALLY_FULL_TX()
- #define timerNORMALLY_EMPTY_RX()
- #define timerNORMALLY_FULL_RX()

Functions

- static void prvLowerPriorityNormallyEmptyTask (void *pvParameters)
- static void prvLowerPriorityNormallyFullTask (void *pvParameters)
- static void prvHigherPriorityNormallyEmptyTask (void *pvParameters)
- static void prv1stHigherPriorityNormallyFullTask (void *pvParameters)
- static void prv2ndHigherPriorityNormallyFullTask (void *pvParameters)
- static void prvRecordValue_NormallyEmpty (UBaseType_t uxValue, UBaseType_t uxSource)
- static void prvRecordValue_NormallyFull (UBaseType_t uxValue, UBaseType_t uxSource)
- static void prvQueueAccessLogError (UBaseType_t uxLine)
- void vStartInterruptQueueTasks (void)
- BaseType_t xFirstTimerHandler (void)
- BaseType_t xSecondTimerHandler (void)
- BaseType_t xAreIntQueueTasksStillRunning (void)

Variables

- static QueueHandle_t xNormallyEmptyQueue
- static QueueHandle_t xNormallyFullQueue
- static volatile UBaseType_t uxHighPriorityLoops1 = 0
- static volatile UBaseType_t uxHighPriorityLoops2 = 0
- static volatile UBaseType_t uxLowPriorityLoops1 = 0
- static volatile UBaseType_t uxLowPriorityLoops2 = 0
- static BaseType_t xErrorStatus = pdPASS
- static volatile UBaseType_t xErrorLine = (UBaseType_t) 0
- static BaseType_t xWasSuspended = pdFALSE
- static volatile UBaseType_t uxValueForNormallyEmptyQueue = 0
- static volatile UBaseType_t uxValueForNormallyFullQueue = 0
- TaskHandle_t xHighPriorityNormallyEmptyTask1
- TaskHandle_t xHighPriorityNormallyEmptyTask2
- TaskHandle_t xHighPriorityNormallyFullTask1
- TaskHandle_t xHighPriorityNormallyFullTask2
- static uint8_t ucNormallyEmptyReceivedValues [intqNUM_VALUES_TO_LOG] = { 0 }
- static uint8_t ucNormallyFullReceivedValues [intqNUM_VALUES_TO_LOG] = { 0 }

7.7.1 Macro Definition Documentation

7.7.1.1 #define intqFIRST_INTERRUPT ((UBaseType_t) 4)

Definition at line 129 of file IntQueue.c.

7.7.1.2 #define intqHIGH_PRIORITY_TASK1 ((UBaseType_t) 1)

Definition at line 126 of file IntQueue.c.

7.7.1.3 #define intqHIGH_PRIORITY_TASK2 ((UBaseType_t) 2)

Definition at line 127 of file IntQueue.c.

7.7.1.4 #define intqHIGHER_PRIORITY (configMAX_PRIORITIES - 2)

Definition at line 103 of file IntQueue.c.

7.7.1.5 #define intqLOW_PRIORITY_TASK ((UBaseType_t) 3)

Definition at line 128 of file IntQueue.c.

7.7.1.6 #define intqLOWER_PRIORITY (tskIDLE_PRIORITY)

Definition at line 105 of file IntQueue.c.

7.7.1.7 #define intqMIN_ACCEPTABLE_TASK_COUNT (5)

Definition at line 135 of file IntQueue.c.

7.7.1.8 #define intqNUM_VALUES_TO_LOG (200)

Definition at line 109 of file IntQueue.c.

7.7.1.9 #define intqONE_TICK_DELAY (1)

Definition at line 121 of file IntQueue.c.

7.7.1.10 #define intqQUEUE_LENGTH ((UBaseType_t) 10)

Definition at line 131 of file IntQueue.c.

7.7.1.11 #define intqSECOND_INTERRUPT ((UBaseType_t) 5)

Definition at line 130 of file IntQueue.c.

7.7.1.12 #define intqSHORT_DELAY (140)

Definition at line 110 of file IntQueue.c.

7.7.1.13 #define intqVALUE_OVERRUN (50)

Definition at line 117 of file IntQueue.c.

7.7.1.14 #define timerNORMALLY_EMPTY_RX( )

Value:

```
if( xQueueReceiveFromISR( xNormallyEmptyQueue, &uxRxedValue, &
      xHigherPriorityTaskWoken ) != pdPASS )   \
    {
      \
        prvQueueAccessLogError( __LINE__ );
      \
    }
      \
    else
      \
    {
      \
        prvRecordValue_NormallyEmpty( uxRxedValue,
      intqSECOND_INTERRUPT );                                  \
    }
```

Definition at line 173 of file IntQueue.c.

7.7.1.15 #define timerNORMALLY_EMPTY_TX( )

Value:

```
if( xQueueIsQueueFullFromISR( xNormallyEmptyQueue ) !=
      pdTRUE )                                                          \
    {
      \
    UBaseType_t uxSavedInterruptStatus;
      \
        uxSavedInterruptStatus = portSET_INTERRUPT_MASK_FROM_ISR();
      \
        {
      \
            uxValueForNormallyEmptyQueue++;
      \
            if( xQueueSendFromISR( xNormallyEmptyQueue, ( void * ) &
        uxValueForNormallyEmptyQueue, &xHigherPriorityTaskWoken ) !=
        pdPASS ) \
            {
      \
                uxValueForNormallyEmptyQueue--;
      \
            }
      \
        }
      \
        portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedInterruptStatus );
      \
    }
      \
```

Definition at line 139 of file IntQueue.c.

7.7.1.16 #define timerNORMALLY_FULL_RX( )

Value:

```
if( xQueueReceiveFromISR( xNormallyFullQueue, &uxRxedValue, &
      xHigherPriorityTaskWoken ) == pdPASS )     \
    {
      \
        prvRecordValue_NormallyFull( uxRxedValue,
      intqSECOND_INTERRUPT );                                \
    }
      \
```

Definition at line 185 of file IntQueue.c.

7.7.1.17 #define timerNORMALLY_FULL_TX( )

Value:

```
if( xQueueIsQueueFullFromISR( xNormallyFullQueue ) !=
      pdTRUE )                                                    \
    {
      \
    UBaseType_t uxSavedInterruptStatus;
      \
        uxSavedInterruptStatus = portSET_INTERRUPT_MASK_FROM_ISR();
      \
        {
      \
            uxValueForNormallyFullQueue++;
      \
            if( xQueueSendFromISR( xNormallyFullQueue, ( void * ) &
      uxValueForNormallyFullQueue, &xHigherPriorityTaskWoken ) !=
      pdPASS ) \
            {
      \
                uxValueForNormallyFullQueue--;
      \
            }
      \
        }
      \
        portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedInterruptStatus );
      \
    }
      \
```

Definition at line 156 of file IntQueue.c.

7.7.2 Function Documentation

7.7.2.1 static void prv1stHigherPriorityNormallyFullTask (void * *pvParameters*) [static]

Definition at line 478 of file IntQueue.c.

7.7.2.2 static void prv2ndHigherPriorityNormallyFullTask (void * *pvParameters*) [static]

Definition at line 581 of file IntQueue.c.

7.7.2.3 static void prvHigherPriorityNormallyEmptyTask (void * *pvParameters*) [static]

Definition at line 307 of file IntQueue.c.

7.7.2.4 static void prvLowerPriorityNormallyEmptyTask (void * *pvParameters*) [static]

Definition at line 430 of file IntQueue.c.

7.7.2.5 static void prvLowerPriorityNormallyFullTask (void * *pvParameters*) [static]

Definition at line 627 of file IntQueue.c.

7.7.2.6 static void prvQueueAccessLogError (UBaseType_t *uxLine*) [static]

Definition at line 299 of file IntQueue.c.

7.7.2.7 static void prvRecordValue_NormallyEmpty (UBaseType_t *uxValue,* UBaseType_t *uxSource*) [static]

Definition at line 282 of file IntQueue.c.

7.7.2.8 static void prvRecordValue_NormallyFull (UBaseType_t *uxValue,* UBaseType_t *uxSource*) [static]

Definition at line 265 of file IntQueue.c.

7.7.2.9 void vStartInterruptQueueTasks (void)

Definition at line 239 of file IntQueue.c.

7.7.2.10 BaseType_t xAreIntQueueTasksStillRunning (void)

Definition at line 728 of file IntQueue.c.

7.7.2.11 BaseType_t xFirstTimerHandler (void)

Definition at line 668 of file IntQueue.c.

7.7.2.12 BaseType_t xSecondTimerHandler (void)

Definition at line 696 of file IntQueue.c.

7.7.3 Variable Documentation

7.7.3.1 uint8_t ucNormallyEmptyReceivedValues[intqNUM_VALUES_TO_LOG] = { 0 } [static]

Definition at line 219 of file IntQueue.c.

7.7.3.2 uint8_t ucNormallyFullReceivedValues[intqNUM_VALUES_TO_LOG] = { 0 } [static]

Definition at line 220 of file IntQueue.c.

7.7.3.3 volatile UBaseType_t uxHighPriorityLoops1 = 0 [static]

Definition at line 198 of file IntQueue.c.

7.7.3.4 volatile UBaseType_t uxHighPriorityLoops2 = 0 [static]

Definition at line 198 of file IntQueue.c.

7.7.3.5 volatile UBaseType_t uxLowPriorityLoops1 = 0 [static]

Definition at line 198 of file IntQueue.c.

7.7.3.6 volatile UBaseType_t uxLowPriorityLoops2 = 0 [static]

Definition at line 198 of file IntQueue.c.

7.7.3.7 volatile UBaseType_t uxValueForNormallyEmptyQueue = 0 [static]

Definition at line 210 of file IntQueue.c.

7.7.3.8 volatile UBaseType_t uxValueForNormallyFullQueue = 0 [static]

Definition at line 210 of file IntQueue.c.

7.7.3.9 volatile UBaseType_t xErrorLine = (UBaseType_t) 0 [static]

Definition at line 203 of file IntQueue.c.

7.7.3.10 BaseType_t xErrorStatus = pdPASS [static]

Definition at line 202 of file IntQueue.c.

7.7.3.11 TaskHandle_t xHighPriorityNormallyEmptyTask1

Definition at line 213 of file IntQueue.c.

7.7.3.12 TaskHandle_t xHighPriorityNormallyEmptyTask2

Definition at line 213 of file IntQueue.c.

7.7.3.13 TaskHandle_t xHighPriorityNormallyFullTask1

Definition at line 213 of file IntQueue.c.

7.7.3.14 TaskHandle_t xHighPriorityNormallyFullTask2

Definition at line 213 of file IntQueue.c.

7.7.3.15 QueueHandle_t xNormallyEmptyQueue [static]

Definition at line 195 of file IntQueue.c.

7.7.3.16 QueueHandle_t xNormallyFullQueue [static]

Definition at line 195 of file IntQueue.c.

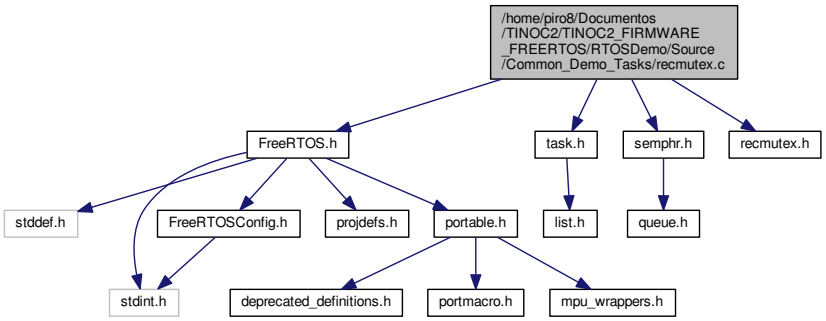
7.7.3.17 BaseType_t xWasSuspended = pdFALSE [static]

Definition at line 206 of file IntQueue.c.

7.8 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/Common_Demo_Tasks/recmutex.c File Reference

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "recmutex.h"
```

Include dependency graph for recmutex.c:



Macros

- #define recmuCONTROLLING_TASK_PRIORITY (tskIDLE_PRIORITY + 2)
- #define recmuBLOCKING_TASK_PRIORITY (tskIDLE_PRIORITY + 1)
- #define recmuPOLLING_TASK_PRIORITY (tskIDLE_PRIORITY + 0)
- #define recmuMAX_COUNT (10)
- #define recmuSHORT_DELAY (pdMS_TO_TICKS(20))
- #define recmuNO_DELAY ((TickType_t) 0)
- #define recmu15ms_DELAY (pdMS_TO_TICKS(15))

Functions

- static void prvRecursiveMutexControllingTask (void *pvParameters)
- static void prvRecursiveMutexBlockingTask (void *pvParameters)
- static void prvRecursiveMutexPollingTask (void *pvParameters)
- void vStartRecursiveMutexTasks (void)
- BaseType_t xAreRecursiveMutexTasksStillRunning (void)

Variables

- static SemaphoreHandle_t xMutex
- static volatile BaseType_t xErrorOccurred = pdFALSE
- static volatile BaseType_t xControllingIsSuspended = pdFALSE
- static volatile BaseType_t xBlockingIsSuspended = pdFALSE
- static volatile UBaseType_t uxControllingCycles = 0
- static volatile UBaseType_t uxBlockingCycles = 0
- static volatile UBaseType_t uxPollingCycles = 0
- static TaskHandle_t xControllingTaskHandle
- static TaskHandle_t xBlockingTaskHandle

7.8.1 Macro Definition Documentation

7.8.1.1 #define recmu15ms_DELAY (pdMS_TO_TICKS(15))

Definition at line 128 of file recmutex.c.

7.8.1.2 #define recmuBLOCKING_TASK_PRIORITY (tskIDLE_PRIORITY + 1)

Definition at line 119 of file recmutex.c.

7.8.1.3 #define recmuCONTROLLING_TASK_PRIORITY (tskIDLE_PRIORITY + 2)

Definition at line 117 of file recmutex.c.

7.8.1.4 #define recmuMAX_COUNT (10)

Definition at line 123 of file recmutex.c.

7.8.1.5 #define recmuNO_DELAY ((TickType_t) 0)

Definition at line 127 of file recmutex.c.

7.8.1.6 #define recmuPOLLING_TASK_PRIORITY (tskIDLE_PRIORITY + 0)

Definition at line 120 of file recmutex.c.

7.8.1.7 #define recmuSHORT_DELAY (pdMS_TO_TICKS(20))

Definition at line 126 of file recmutex.c.

7.8.2 Function Documentation

7.8.2.1 static void prvRecursiveMutexBlockingTask (void * *pvParameters*) [static]

Definition at line 254 of file recmutex.c.

7.8.2.2 static void prvRecursiveMutexControllingTask (void * *pvParameters*) [static]

Definition at line 171 of file recmutex.c.

7.8.2.3 static void prvRecursiveMutexPollingTask (void * *pvParameters*) [static]

Definition at line 310 of file recmutex.c.

7.8.2.4 void vStartRecursiveMutexTasks (void)

Definition at line 148 of file recmutex.c.

7.8.2.5 BaseType_t xAreRecursiveMutexTasksStillRunning (void)

Definition at line 405 of file recmutex.c.

7.8.3 Variable Documentation

7.8.3.1 volatile UBaseType_t uxBlockingCycles = 0 [static]

Definition at line 140 of file recmutex.c.

7.8.3.2 volatile UBaseType_t uxControllingCycles = 0 [static]

Definition at line 140 of file recmutex.c.

7.8.3.3 volatile UBaseType_t uxPollingCycles = 0 [static]

Definition at line 140 of file recmutex.c.

7.8.3.4 volatile BaseType_t xBlockingIsSuspended = pdFALSE [static]

Definition at line 139 of file recmutex.c.

7.8.3.5 TaskHandle_t xBlockingTaskHandle [static]

Definition at line 144 of file recmutex.c.

7.8.3.6 volatile BaseType_t xControllingIsSuspended = pdFALSE [static]

Definition at line 139 of file recmutex.c.

7.8.3.7 TaskHandle_t xControllingTaskHandle [static]

Definition at line 144 of file recmutex.c.

7.8.3.8 volatile BaseType_t xErrorOccurred = pdFALSE [static]

Definition at line 139 of file recmutex.c.

7.8.3.9 SemaphoreHandle_t xMutex [static]

Definition at line 136 of file recmutex.c.

7.9 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/cr_startup_lpc11.c File Reference

Macros

- #define WEAK __attribute__ ((weak))
- #define ALIAS(f) __attribute__ ((weak, alias (#f)))

Functions

- void ResetISR (void)
- WEAK void NMI_Handler (void)
- WEAK void HardFault_Handler (void)
- WEAK void SVCall_Handler (void)
- WEAK void PendSV_Handler (void)
- WEAK void SysTick_Handler (void)
- WEAK void IntDefaultHandler (void)
- void CAN_IRQHandler (void SSP1_IRQHandler() ALIAS(IntDefaultHandler) void)
- __attribute__ ((section(".after_vectors")))
- void pop_registers_from_fault_stack (unsigned int *hardfault_args)

Variables

- unsigned int __data_section_table
- unsigned int __data_section_table_end
- unsigned int __bss_section_table
- unsigned int __bss_section_table_end

7.9.1 Macro Definition Documentation

7.9.1.1 #define ALIAS(*f*) __attribute__ ((weak, alias (#f)))

Definition at line 46 of file cr_startup_lpc11.c.

7.9.1.2 #define WEAK __attribute__ ((weak))

Definition at line 45 of file cr_startup_lpc11.c.

7.9.2 Function Documentation

7.9.2.1 __attribute__ ((section(".after_vectors")))

Definition at line 200 of file cr_startup_lpc11.c.

7.9.2.2 void CAN_IRQHandler (void SSP1_IRQHandler () ALIAS(IntDefaultHandler) *void*)

Definition at line 82 of file cr_startup_lpc11.c.

7.9.2.3 WEAK void HardFault_Handler (void)

7.9.2.4 WEAK void IntDefaultHandler (void)

7.9.2.5 WEAK void NMI_Handler (void)

7.9.2.6 WEAK void PendSV_Handler (void)

7.9.2.7 void pop_registers_from_fault_stack (unsigned int * *hardfault_args*)

Definition at line 339 of file cr_startup_lpc11.c.

7.9.2.8 void ResetISR (void)

7.9.2.9 WEAK void SVCall_Handler (void)

7.9.2.10 WEAK void SysTick_Handler (void)

7.9.3 Variable Documentation

7.9.3.1 unsigned int __bss_section_table

7.9.3.2 unsigned int __bss_section_table_end

7.9.3.3 unsigned int __data_section_table

7.9.3.4 unsigned int __data_section_table_end

7.10 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/flash/flash.c File Reference

Funciones de manejo de memoria flash.

7.10.1 Detailed Description

Funciones de manejo de memoria flash.

Date

Mar 5, 2018

Author

Roux, Federico G. (froux@citedef.gob.ar)

7.11 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/flash/flash.h File Reference

Header de flash.c.

Macros

- #define IAP_LOCATION 0x1FFF1FF1

  Dirección de memoria de punto de entrada de la función IAP. El código de esta región está en modo Thumb, así que al entrar a esta función, el programa entrará en modo Thumb UM10429 pag. 173.
- #define IAP_COM_REINVOKE_ISP 57

  Numero de comando para invocar ISP.

Typedefs

- typedef void(∗ IAP) (uint32_t[ ], uint32_t[ ])

  definicion de tipo de dato: puntero a función para ingresar comandos IAP.

7.11.1 Detailed Description

Header de flash.c.

Date

Mar 5, 2018

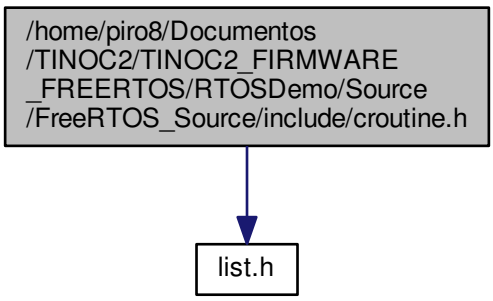
Author

Roux, Federico G. (froux@citedef.gob.ar)

7.12 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/include/croutine.h File Reference

```
#include "list.h"
```

Include dependency graph for croutine.h:



Classes

- struct corCoRoutineControlBlock

Macros

- #define crSTART(pxCRCB) switch(((CRCB_t *)(pxCRCB))->uxState) { case 0:
- #define crEND() }
- #define crSET_STATE0(xHandle) ((CRCB_t *)(xHandle))->uxState = (__LINE__ * 2); return; case (__LINE__ * 2):
- #define crSET_STATE1(xHandle) ((CRCB_t *)(xHandle))->uxState = ((__LINE__ * 2)+1); return; case ((__LINE__ * 2)+1):
- #define crDELAY(xHandle, xTicksToDelay)
- #define crQUEUE_SEND(xHandle, pxQueue, pvItemToQueue, xTicksToWait, pxResult)
- #define crQUEUE_RECEIVE(xHandle, pxQueue, pvBuffer, xTicksToWait, pxResult)
- #define crQUEUE_SEND_FROM_ISR(pxQueue, pvItemToQueue, xCoRoutinePreviouslyWoken) xQueue↩CRSendFromISR((pxQueue), (pvItemToQueue), (xCoRoutinePreviouslyWoken))
- #define crQUEUE_RECEIVE_FROM_ISR(pxQueue, pvBuffer, pxCoRoutineWoken) xQueueCRReceive↩FromISR((pxQueue), (pvBuffer), (pxCoRoutineWoken))

Typedefs

- typedef void * CoRoutineHandle_t
- typedef void(* crCOROUTINE_CODE) (CoRoutineHandle_t, UBaseType_t)
- typedef struct corCoRoutineControlBlock CRCB_t

Functions

- BaseType_t xCoRoutineCreate (crCOROUTINE_CODE pxCoRoutineCode, UBaseType_t uxPriority, U↩BaseType_t uxIndex)
- void vCoRoutineSchedule (void)
- void vCoRoutineAddToDelayedList (TickType_t xTicksToDelay, List_t *pxEventList)
- BaseType_t xCoRoutineRemoveFromEventList (const List_t *pxEventList)

7.12.1 Macro Definition Documentation

7.12.1.1 #define crDELAY(*xHandle, xTicksToDelay*)

Value:

```
if( ( xTicksToDelay ) > 0 )                                                 \
    {
      \
        vCoRoutineAddToDelayedList( ( xTicksToDelay ), NULL );
      \
    }
      \
    crSET_STATE0( ( xHandle ) );
```

Definition at line 332 of file croutine.h.

7.12.1.2 #define crEND( ) }

Definition at line 277 of file croutine.h.

7.12.1.3 #define crQUEUE_RECEIVE(*xHandle, pxQueue, pvBuffer, xTicksToWait, pxResult*)

Value:

```
{                                                                                   \
    *( pxResult ) = xQueueCRReceive( ( pxQueue) , ( pvBuffer ), ( xTicksToWait ) );  \
    if( *( pxResult ) == errQUEUE_BLOCKED )                                         \
    {                                                                               \
      \
        crSET_STATE0( ( xHandle ) );                                                \
        *( pxResult ) = xQueueCRReceive( ( pxQueue) , ( pvBuffer ), 0 );            \
    }                                                                               \
    if( *( pxResult ) == errQUEUE_YIELD )                                           \
    {                                                                               \
      \
        crSET_STATE1( ( xHandle ) );                                                \
        *( pxResult ) = pdPASS;                                                         \
    }                                                                               \
}
```

Definition at line 514 of file croutine.h.

7.12.1.4 #define crQUEUE_RECEIVE_FROM_ISR(*pxQueue, pvBuffer, pxCoRoutineWoken*) xQueueCRReceiveFromISR((pxQueue), (pvBuffer), (pxCoRoutineWoken))

Definition at line 736 of file croutine.h.

7.12.1.5 #define crQUEUE_SEND(*xHandle, pxQueue, pvItemToQueue, xTicksToWait, pxResult*)

Value:

```
{                                                                                   \
    *( pxResult ) = xQueueCRSend( ( pxQueue) , ( pvItemToQueue) , ( xTicksToWait ) );  \
    if( *( pxResult ) == errQUEUE_BLOCKED )                                         \
    {                                                                               \
      \
        crSET_STATE0( ( xHandle ) );                                                \
        *pxResult = xQueueCRSend( ( pxQueue ), ( pvItemToQueue ), 0 );              \
    }                                                                               \
    if( *pxResult == errQUEUE_YIELD )                                               \
    {                                                                               \
      \
        crSET_STATE1( ( xHandle ) );                                                \
        *pxResult = pdPASS;                                                             \
    }                                                                               \
}
```

Definition at line 422 of file croutine.h.

7.12.1.6 #define crQUEUE_SEND_FROM_ISR(*pxQueue, pvItemToQueue, xCoRoutinePreviouslyWoken*) xQueueCRSendFromISR((pxQueue), (pvItemToQueue), (xCoRoutinePreviouslyWoken))

Definition at line 623 of file croutine.h.

7.12.1.7 #define crSET_STATE0(*xHandle*) ((CRCB_t *)(xHandle))->uxState = (__LINE__ * 2); return; case (__LINE__ * 2):

Definition at line 283 of file croutine.h.

7.12.1.8 #define crSET_STATE1(*xHandle*) ((CRCB_t *)(xHandle))->uxState = ((__LINE__ * 2)+1); return; case ((__LINE__ * 2)+1):

Definition at line 284 of file croutine.h.

7.12.1.9 #define crSTART(*pxCRCB*) switch(((CRCB_t *)(pxCRCB))->uxState) { case 0:

Definition at line 246 of file croutine.h.

7.12.2 Typedef Documentation

7.12.2.1 typedef void* CoRoutineHandle_t

Definition at line 86 of file croutine.h.

7.12.2.2 typedef struct corCoRoutineControlBlock CRCB_t

7.12.2.3 typedef void(* crCOROUTINE_CODE) (CoRoutineHandle_t, UBaseType_t)

Definition at line 89 of file croutine.h.

7.12.3 Function Documentation

7.12.3.1 void vCoRoutineAddToDelayedList (TickType_t *xTicksToDelay,* List_t * *pxEventList*)

7.12.3.2 void vCoRoutineSchedule (void)

7.12.3.3 BaseType_t xCoRoutineCreate (crCOROUTINE_CODE *pxCoRoutineCode,* UBaseType_t *uxPriority,* UBaseType_t *uxIndex*)

7.12.3.4 BaseType_t xCoRoutineRemoveFromEventList (const List_t * *pxEventList*)

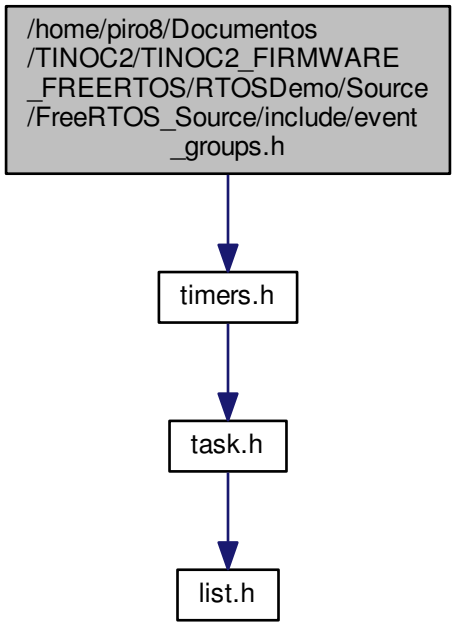
7.13 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/include/deprecated_definitions.h File Reference

This graph shows which files directly or indirectly include this file:

7.14 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/include/event_groups.h File Reference

```
#include "timers.h"
```

Include dependency graph for event_groups.h:



Macros

- #define xEventGroupClearBitsFromISR(xEventGroup, uxBitsToClear) xTimerPendFunctionCallFromISR(v↩EventGroupClearBitsCallback, (void *) xEventGroup, (uint32_t) uxBitsToClear, NULL)
- #define xEventGroupSetBitsFromISR(xEventGroup, uxBitsToSet, pxHigherPriorityTaskWoken) xTimer↩PendFunctionCallFromISR(vEventGroupSetBitsCallback, (void *) xEventGroup, (uint32_t) uxBitsToSet, pxHigherPriorityTaskWoken)
- #define xEventGroupGetBits(xEventGroup) xEventGroupClearBits(xEventGroup, 0)

Typedefs

- typedef void * EventGroupHandle_t
- typedef TickType_t EventBits_t

Functions

- EventBits_t xEventGroupWaitBits (EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits, TickType_t xTicksToWait) PRIVILEGED_FUNCTION
- EventBits_t xEventGroupClearBits (EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToClear) PRIVILEGED_FUNCTION
- EventBits_t xEventGroupSetBits (EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet) PRIVILEGED_FUNCTION
- EventBits_t xEventGroupSync (EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet, const EventBits_t uxBitsToWaitFor, TickType_t xTicksToWait) PRIVILEGED_FUNCTION
- EventBits_t xEventGroupGetBitsFromISR (EventGroupHandle_t xEventGroup) PRIVILEGED_FUNCTION
- void vEventGroupDelete (EventGroupHandle_t xEventGroup) PRIVILEGED_FUNCTION
- void vEventGroupSetBitsCallback (void *pvEventGroup, const uint32_t ulBitsToSet) PRIVILEGED_FUNCTION
- void vEventGroupClearBitsCallback (void *pvEventGroup, const uint32_t ulBitsToClear) PRIVILEGED_FUNCTION

7.14.1 Macro Definition Documentation

7.14.1.1 #define xEventGroupClearBitsFromISR(*xEventGroup, uxBitsToClear*) xTimerPendFunctionCallFromISR(vEventGroupClearBitsCallback, (void *) xEventGroup, (uint32_t) uxBitsToClear, NULL)

Definition at line 451 of file event_groups.h.

7.14.1.2 #define xEventGroupGetBits(*xEventGroup*) xEventGroupClearBits(xEventGroup, 0)

Definition at line 749 of file event_groups.h.

7.14.1.3 #define xEventGroupSetBitsFromISR(*xEventGroup, uxBitsToSet, pxHigherPriorityTaskWoken*) xTimerPendFunctionCallFromISR(vEventGroupSetBitsCallback, (void *) xEventGroup, (uint32_t) uxBitsToSet, pxHigherPriorityTaskWoken)

Definition at line 603 of file event_groups.h.

7.14.2 Typedef Documentation

7.14.2.1 typedef TickType_t EventBits_t

Definition at line 133 of file event_groups.h.

7.14.2.2 typedef void* EventGroupHandle_t

Definition at line 123 of file event_groups.h.

7.14.3 Function Documentation

7.14.3.1 void vEventGroupClearBitsCallback (void * *pvEventGroup,* const uint32_t *ulBitsToClear*)

7.14.3.2 void vEventGroupDelete (EventGroupHandle_t *xEventGroup*)

event_groups.h

```
void xEventGroupDelete( EventGroupHandle_t xEventGroup );
```

Delete an event group that was previously created by a call to xEventGroupCreate(). Tasks that are blocked on the event group will be unblocked and obtain 0 as the event group's value.

Parameters

xEventGroup	The event group being deleted.

7.14.3.3 void vEventGroupSetBitsCallback (void * *pvEventGroup,* const uint32_t *ulBitsToSet*)

7.14.3.4 EventBits_t xEventGroupClearBits (EventGroupHandle_t *xEventGroup,* const EventBits_t *uxBitsToClear*)

7.14.3.5 EventBits_t xEventGroupGetBitsFromISR (EventGroupHandle_t *xEventGroup*)

7.14.3.6 EventBits_t xEventGroupSetBits (EventGroupHandle_t *xEventGroup,* const EventBits_t *uxBitsToSet*)

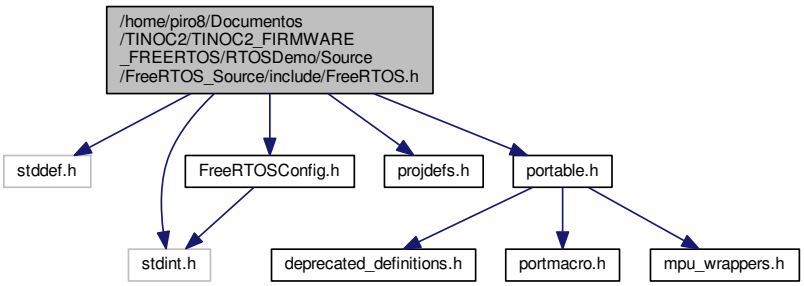
7.14.3.7 EventBits_t xEventGroupSync (EventGroupHandle_t *xEventGroup,* const EventBits_t *uxBitsToSet,* const EventBits_t *uxBitsToWaitFor,* TickType_t *xTicksToWait*)

7.14.3.8 EventBits_t xEventGroupWaitBits (EventGroupHandle_t *xEventGroup,* const EventBits_t *uxBitsToWaitFor,* const BaseType_t *xClearOnExit,* const BaseType_t *xWaitForAllBits,* TickType_t *xTicksToWait*)

7.15 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/FreeRTOS_Source/include/FreeRTOS.h File Reference

```
#include <stddef.h>
#include <stdint.h>
#include "FreeRTOSConfig.h"
#include "projdefs.h"
#include "portable.h"
```

Include dependency graph for FreeRTOS.h:



This graph shows which files directly or indirectly include this file:

Classes

- struct xSTATIC_LIST_ITEM
- struct xSTATIC_MINI_LIST_ITEM
- struct xSTATIC_LIST
- struct xSTATIC_TCB
- struct xSTATIC_QUEUE
- struct xSTATIC_EVENT_GROUP
- struct xSTATIC_TIMER

Macros

- #define configUSE_NEWLIB_REENTRANT 0
- #define configUSE_CO_ROUTINES 0
- #define INCLUDE_vTaskPrioritySet 0
- #define INCLUDE_uxTaskPriorityGet 0
- #define INCLUDE_vTaskDelete 0
- #define INCLUDE_vTaskSuspend 0
- #define INCLUDE_vTaskDelayUntil 0
- #define INCLUDE_vTaskDelay 0
- #define INCLUDE_xTaskGetIdleTaskHandle 0
- #define INCLUDE_xTaskAbortDelay 0
- #define INCLUDE_xQueueGetMutexHolder 0
- #define INCLUDE_xSemaphoreGetMutexHolder INCLUDE_xQueueGetMutexHolder
- #define INCLUDE_xTaskGetHandle 0
- #define INCLUDE_uxTaskGetStackHighWaterMark 0
- #define INCLUDE_eTaskGetState 0
- #define INCLUDE_xTaskResumeFromISR 1
- #define INCLUDE_xTimerPendFunctionCall 0
- #define INCLUDE_xTaskGetSchedulerState 0

- #define INCLUDE_xTaskGetCurrentTaskHandle 0
- #define configUSE_DAEMON_TASK_STARTUP_HOOK 0
- #define configUSE_APPLICATION_TASK_TAG 0
- #define configNUM_THREAD_LOCAL_STORAGE_POINTERS 0
- #define configUSE_RECURSIVE_MUTEXES 0
- #define configUSE_MUTEXES 0
- #define configUSE_TIMERS 0
- #define configUSE_COUNTING_SEMAPHORES 0
- #define configUSE_ALTERNATIVE_API 0
- #define portCRITICAL_NESTING_IN_TCB 0
- #define configMAX_TASK_NAME_LEN 16
- #define configIDLE_SHOULD_YIELD 1
- #define configASSERT(x)
- #define configASSERT_DEFINED 0
- #define portSET_INTERRUPT_MASK_FROM_ISR() 0
- #define portCLEAR_INTERRUPT_MASK_FROM_ISR(uxSavedStatusValue) (void) uxSavedStatusValue
- #define portCLEAN_UP_TCB(pxTCB) (void) pxTCB
- #define portPRE_TASK_DELETE_HOOK(pvTaskToDelete, pxYieldPending)
- #define portSETUP_TCB(pxTCB) (void) pxTCB
- #define configQUEUE_REGISTRY_SIZE 0U
- #define vQueueAddToRegistry(xQueue, pcName)
- #define vQueueUnregisterQueue(xQueue)
- #define pcQueueGetName(xQueue)
- #define portPOINTER_SIZE_TYPE uint32_t
- #define traceSTART()
- #define traceEND()
- #define traceTASK_SWITCHED_IN()
- #define traceINCREASE_TICK_COUNT(x)
- #define traceLOW_POWER_IDLE_BEGIN()
- #define traceLOW_POWER_IDLE_END()
- #define traceTASK_SWITCHED_OUT()
- #define traceTASK_PRIORITY_INHERIT(pxTCBOfMutexHolder, uxInheritedPriority)
- #define traceTASK_PRIORITY_DISINHERIT(pxTCBOfMutexHolder, uxOriginalPriority)
- #define traceBLOCKING_ON_QUEUE_RECEIVE(pxQueue)
- #define traceBLOCKING_ON_QUEUE_SEND(pxQueue)
- #define configCHECK_FOR_STACK_OVERFLOW 0
- #define traceMOVED_TASK_TO_READY_STATE(pxTCB)
- #define tracePOST_MOVED_TASK_TO_READY_STATE(pxTCB)
- #define traceQUEUE_CREATE(pxNewQueue)
- #define traceQUEUE_CREATE_FAILED(ucQueueType)
- #define traceCREATE_MUTEX(pxNewQueue)
- #define traceCREATE_MUTEX_FAILED()
- #define traceGIVE_MUTEX_RECURSIVE(pxMutex)
- #define traceGIVE_MUTEX_RECURSIVE_FAILED(pxMutex)
- #define traceTAKE_MUTEX_RECURSIVE(pxMutex)
- #define traceTAKE_MUTEX_RECURSIVE_FAILED(pxMutex)
- #define traceCREATE_COUNTING_SEMAPHORE()
- #define traceCREATE_COUNTING_SEMAPHORE_FAILED()
- #define traceQUEUE_SEND(pxQueue)
- #define traceQUEUE_SEND_FAILED(pxQueue)
- #define traceQUEUE_RECEIVE(pxQueue)
- #define traceQUEUE_PEEK(pxQueue)
- #define traceQUEUE_PEEK_FROM_ISR(pxQueue)
- #define traceQUEUE_RECEIVE_FAILED(pxQueue)
- #define traceQUEUE_SEND_FROM_ISR(pxQueue)

- #define traceQUEUE_SEND_FROM_ISR_FAILED(pxQueue)
- #define traceQUEUE_RECEIVE_FROM_ISR(pxQueue)
- #define traceQUEUE_RECEIVE_FROM_ISR_FAILED(pxQueue)
- #define traceQUEUE_PEEK_FROM_ISR_FAILED(pxQueue)
- #define traceQUEUE_DELETE(pxQueue)
- #define traceTASK_CREATE(pxNewTCB)
- #define traceTASK_CREATE_FAILED()
- #define traceTASK_DELETE(pxTaskToDelete)
- #define traceTASK_DELAY_UNTIL(x)
- #define traceTASK_DELAY()
- #define traceTASK_PRIORITY_SET(pxTask, uxNewPriority)
- #define traceTASK_SUSPEND(pxTaskToSuspend)
- #define traceTASK_RESUME(pxTaskToResume)
- #define traceTASK_RESUME_FROM_ISR(pxTaskToResume)
- #define traceTASK_INCREMENT_TICK(xTickCount)
- #define traceTIMER_CREATE(pxNewTimer)
- #define traceTIMER_CREATE_FAILED()
- #define traceTIMER_COMMAND_SEND(xTimer, xMessageID, xMessageValueValue, xReturn)
- #define traceTIMER_EXPIRED(pxTimer)
- #define traceTIMER_COMMAND_RECEIVED(pxTimer, xMessageID, xMessageValue)
- #define traceMALLOC(pvAddress, uiSize)
- #define traceFREE(pvAddress, uiSize)
- #define traceEVENT_GROUP_CREATE(xEventGroup)
- #define traceEVENT_GROUP_CREATE_FAILED()
- #define traceEVENT_GROUP_SYNC_BLOCK(xEventGroup, uxBitsToSet, uxBitsToWaitFor)
- #define traceEVENT_GROUP_SYNC_END(xEventGroup, uxBitsToSet, uxBitsToWaitFor, xTimeout↩ Occurred) (void) xTimeoutOccurred
- #define traceEVENT_GROUP_WAIT_BITS_BLOCK(xEventGroup, uxBitsToWaitFor)
- #define traceEVENT_GROUP_WAIT_BITS_END(xEventGroup, uxBitsToWaitFor, xTimeoutOccurred) (void) xTimeoutOccurred
- #define traceEVENT_GROUP_CLEAR_BITS(xEventGroup, uxBitsToClear)
- #define traceEVENT_GROUP_CLEAR_BITS_FROM_ISR(xEventGroup, uxBitsToClear)
- #define traceEVENT_GROUP_SET_BITS(xEventGroup, uxBitsToSet)
- #define traceEVENT_GROUP_SET_BITS_FROM_ISR(xEventGroup, uxBitsToSet)
- #define traceEVENT_GROUP_DELETE(xEventGroup)
- #define tracePEND_FUNC_CALL(xFunctionToPend, pvParameter1, ulParameter2, ret)
- #define tracePEND_FUNC_CALL_FROM_ISR(xFunctionToPend, pvParameter1, ulParameter2, ret)
- #define traceQUEUE_REGISTRY_ADD(xQueue, pcQueueName)
- #define traceTASK_NOTIFY_TAKE_BLOCK()
- #define traceTASK_NOTIFY_TAKE()
- #define traceTASK_NOTIFY_WAIT_BLOCK()
- #define traceTASK_NOTIFY_WAIT()
- #define traceTASK_NOTIFY()
- #define traceTASK_NOTIFY_FROM_ISR()
- #define traceTASK_NOTIFY_GIVE_FROM_ISR()
- #define configGENERATE_RUN_TIME_STATS 0
- #define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()
- #define configUSE_MALLOC_FAILED_HOOK 0
- #define portPRIVILEGE_BIT ((UBaseType_t) 0x00)
- #define portYIELD_WITHIN_API portYIELD
- #define portSUPPRESS_TICKS_AND_SLEEP(xExpectedIdleTime)
- #define configEXPECTED_IDLE_TIME_BEFORE_SLEEP 2
- #define configUSE_TICKLESS_IDLE 0
- #define configPRE_SLEEP_PROCESSING(x)
- #define configPOST_SLEEP_PROCESSING(x)

- #define configUSE_QUEUE_SETS 0
- #define portTASK_USES_FLOATING_POINT()
- #define configUSE_TIME_SLICING 1
- #define configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS 0
- #define configUSE_STATS_FORMATTING_FUNCTIONS 0
- #define portASSERT_IF_INTERRUPT_PRIORITY_INVALID()
- #define configUSE_TRACE_FACILITY 0
- #define mtCOVERAGE_TEST_MARKER()
- #define mtCOVERAGE_TEST_DELAY()
- #define portASSERT_IF_IN_ISR()
- #define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
- #define configAPPLICATION_ALLOCATED_HEAP 0
- #define configUSE_TASK_NOTIFICATIONS 1
- #define portTICK_TYPE_IS_ATOMIC 0
- #define configSUPPORT_STATIC_ALLOCATION 0
- #define configSUPPORT_DYNAMIC_ALLOCATION 1
- #define portTICK_TYPE_ENTER_CRITICAL() portENTER_CRITICAL()
- #define portTICK_TYPE_EXIT_CRITICAL() portEXIT_CRITICAL()
- #define portTICK_TYPE_SET_INTERRUPT_MASK_FROM_ISR() portSET_INTERRUPT_MASK_FROM↩_ISR()
- #define portTICK_TYPE_CLEAR_INTERRUPT_MASK_FROM_ISR(x) portCLEAR_INTERRUPT_MASK_↩FROM_ISR((x))
- #define configENABLE_BACKWARD_COMPATIBILITY 1
- #define eTaskStateGet eTaskGetState
- #define portTickType TickType_t
- #define xTaskHandle TaskHandle_t
- #define xQueueHandle QueueHandle_t
- #define xSemaphoreHandle SemaphoreHandle_t
- #define xQueueSetHandle QueueSetHandle_t
- #define xQueueSetMemberHandle QueueSetMemberHandle_t
- #define xTimeOutType TimeOut_t
- #define xMemoryRegion MemoryRegion_t
- #define xTaskParameters TaskParameters_t
- #define xTaskStatusType TaskStatus_t
- #define xTimerHandle TimerHandle_t
- #define xCoRoutineHandle CoRoutineHandle_t
- #define pdTASK_HOOK_CODE TaskHookFunction_t
- #define portTICK_RATE_MS portTICK_PERIOD_MS
- #define pcTaskGetTaskName pcTaskGetName
- #define pcTimerGetTimerName pcTimerGetName
- #define pcQueueGetQueueName pcQueueGetName
- #define vTaskGetTaskInfo vTaskGetInfo
- #define tmrTIMER_CALLBACK TimerCallbackFunction_t
- #define pdTASK_CODE TaskFunction_t
- #define xListItem ListItem_t
- #define xList List_t
- #define configUSE_TASK_FPU_SUPPORT 1

Typedefs

- typedef struct xSTATIC_LIST_ITEM StaticListItem_t
- typedef struct xSTATIC_MINI_LIST_ITEM StaticMiniListItem_t
- typedef struct xSTATIC_LIST StaticList_t
- typedef struct xSTATIC_TCB StaticTask_t
- typedef struct xSTATIC_QUEUE StaticQueue_t
- typedef StaticQueue_t StaticSemaphore_t
- typedef struct xSTATIC_EVENT_GROUP StaticEventGroup_t
- typedef struct xSTATIC_TIMER StaticTimer_t

7.15.1 Macro Definition Documentation

7.15.1.1 #define configAPPLICATION_ALLOCATED_HEAP 0

Definition at line 764 of file FreeRTOS.h.

7.15.1.2 #define configASSERT(*x*)

Definition at line 276 of file FreeRTOS.h.

7.15.1.3 #define configASSERT_DEFINED 0

Definition at line 277 of file FreeRTOS.h.

7.15.1.4 #define configCHECK_FOR_STACK_OVERFLOW 0

Definition at line 408 of file FreeRTOS.h.

7.15.1.5 #define configENABLE_BACKWARD_COMPATIBILITY 1

Definition at line 820 of file FreeRTOS.h.

7.15.1.6 #define configEXPECTED_IDLE_TIME_BEFORE_SLEEP 2

Definition at line 700 of file FreeRTOS.h.

7.15.1.7 #define configGENERATE_RUN_TIME_STATS 0

Definition at line 662 of file FreeRTOS.h.

7.15.1.8 #define configIDLE_SHOULD_YIELD 1

Definition at line 268 of file FreeRTOS.h.

7.15.1.9 #define configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS 0

Definition at line 732 of file FreeRTOS.h.

7.15.1.10 #define configMAX_TASK_NAME_LEN 16

Definition at line 264 of file FreeRTOS.h.

7.15.1.11 #define configNUM_THREAD_LOCAL_STORAGE_POINTERS 0

Definition at line 236 of file FreeRTOS.h.

7.15.1.12 #define configPOST_SLEEP_PROCESSING(*x*)

Definition at line 716 of file FreeRTOS.h.

7.15.1.13 #define configPRE_SLEEP_PROCESSING(*x*)

Definition at line 712 of file FreeRTOS.h.

7.15.1.14 #define configQUEUE_REGISTRY_SIZE 0U

Definition at line 320 of file FreeRTOS.h.

7.15.1.15 #define configSUPPORT_DYNAMIC_ALLOCATION 1

Definition at line 782 of file FreeRTOS.h.

7.15.1.16 #define configSUPPORT_STATIC_ALLOCATION 0

Definition at line 777 of file FreeRTOS.h.

7.15.1.17 #define configUSE_ALTERNATIVE_API 0

Definition at line 256 of file FreeRTOS.h.

7.15.1.18 #define configUSE_APPLICATION_TASK_TAG 0

Definition at line 232 of file FreeRTOS.h.

7.15.1.19 #define configUSE_CO_ROUTINES 0

Definition at line 150 of file FreeRTOS.h.

7.15.1.20 #define configUSE_COUNTING_SEMAPHORES 0

Definition at line 252 of file FreeRTOS.h.

7.15.1.21 #define configUSE_DAEMON_TASK_STARTUP_HOOK 0

Definition at line 228 of file FreeRTOS.h.

7.15.1.22 #define configUSE_MALLOC_FAILED_HOOK 0

Definition at line 684 of file FreeRTOS.h.

7.15.1.23 #define configUSE_MUTEXES 0

Definition at line 244 of file FreeRTOS.h.

7.15.1.24 #define configUSE_NEWLIB_REENTRANT 0

Definition at line 108 of file FreeRTOS.h.

7.15.1.25 #define configUSE_PORT_OPTIMISED_TASK_SELECTION 0

Definition at line 760 of file FreeRTOS.h.

7.15.1.26 #define configUSE_QUEUE_SETS 0

Definition at line 720 of file FreeRTOS.h.

7.15.1.27 #define configUSE_RECURSIVE_MUTEXES 0

Definition at line 240 of file FreeRTOS.h.

7.15.1.28 #define configUSE_STATS_FORMATTING_FUNCTIONS 0

Definition at line 736 of file FreeRTOS.h.

7.15.1.29 #define configUSE_TASK_FPU_SUPPORT 1

Definition at line 861 of file FreeRTOS.h.

7.15.1.30 #define configUSE_TASK_NOTIFICATIONS 1

Definition at line 768 of file FreeRTOS.h.

7.15.1.31 #define configUSE_TICKLESS_IDLE 0

Definition at line 708 of file FreeRTOS.h.

7.15.1.32 #define configUSE_TIME_SLICING 1

Definition at line 728 of file FreeRTOS.h.

7.15.1.33 #define configUSE_TIMERS 0

Definition at line 248 of file FreeRTOS.h.

7.15.1.34 #define configUSE_TRACE_FACILITY 0

Definition at line 744 of file FreeRTOS.h.

7.15.1.35 #define eTaskStateGet eTaskGetState

Definition at line 824 of file FreeRTOS.h.

7.15.1.36 #define INCLUDE_eTaskGetState 0

Definition at line 202 of file FreeRTOS.h.

7.15.1.37 #define INCLUDE_uxTaskGetStackHighWaterMark 0

Definition at line 198 of file FreeRTOS.h.

7.15.1.38 #define INCLUDE_uxTaskPriorityGet 0

Definition at line 158 of file FreeRTOS.h.

7.15.1.39 #define INCLUDE_vTaskDelay 0

Definition at line 174 of file FreeRTOS.h.

7.15.1.40 #define INCLUDE_vTaskDelayUntil 0

Definition at line 170 of file FreeRTOS.h.

7.15.1.41 #define INCLUDE_vTaskDelete 0

Definition at line 162 of file FreeRTOS.h.

7.15.1.42 #define INCLUDE_vTaskPrioritySet 0

Definition at line 154 of file FreeRTOS.h.

7.15.1.43 #define INCLUDE_vTaskSuspend 0

Definition at line 166 of file FreeRTOS.h.

7.15.1.44 #define INCLUDE_xQueueGetMutexHolder 0

Definition at line 186 of file FreeRTOS.h.

7.15.1.45 #define INCLUDE_xSemaphoreGetMutexHolder INCLUDE_xQueueGetMutexHolder

Definition at line 190 of file FreeRTOS.h.

7.15.1.46 #define INCLUDE_xTaskAbortDelay 0

Definition at line 182 of file FreeRTOS.h.

7.15.1.47 #define INCLUDE_xTaskGetCurrentTaskHandle 0

Definition at line 218 of file FreeRTOS.h.

7.15.1.48 #define INCLUDE_xTaskGetHandle 0

Definition at line 194 of file FreeRTOS.h.

7.15.1.49 #define INCLUDE_xTaskGetIdleTaskHandle 0

Definition at line 178 of file FreeRTOS.h.

7.15.1.50 #define INCLUDE_xTaskGetSchedulerState 0

Definition at line 214 of file FreeRTOS.h.

7.15.1.51 #define INCLUDE_xTaskResumeFromISR 1

Definition at line 206 of file FreeRTOS.h.

7.15.1.52 #define INCLUDE_xTimerPendFunctionCall 0

Definition at line 210 of file FreeRTOS.h.

7.15.1.53 #define mtCOVERAGE_TEST_DELAY( )

Definition at line 752 of file FreeRTOS.h.

7.15.1.54 #define mtCOVERAGE_TEST_MARKER( )

Definition at line 748 of file FreeRTOS.h.

7.15.1.55 #define pcQueueGetName(xQueue)

Definition at line 326 of file FreeRTOS.h.

7.15.1.56 #define pcQueueGetQueueName pcQueueGetName

Definition at line 841 of file FreeRTOS.h.

7.15.1.57 #define pcTaskGetTaskName pcTaskGetName

Definition at line 839 of file FreeRTOS.h.

7.15.1.58 #define pcTimerGetTimerName pcTimerGetName

Definition at line 840 of file FreeRTOS.h.

7.15.1.59 #define pdTASK_CODE TaskFunction_t

Definition at line 847 of file FreeRTOS.h.

7.15.1.60 #define pdTASK_HOOK_CODE TaskHookFunction_t

Definition at line 837 of file FreeRTOS.h.

7.15.1.61 #define portASSERT_IF_IN_ISR( )

Definition at line 756 of file FreeRTOS.h.

7.15.1.62 #define portASSERT_IF_INTERRUPT_PRIORITY_INVALID( )

Definition at line 740 of file FreeRTOS.h.

7.15.1.63 #define portCLEAN_UP_TCB(*pxTCB*) (void) pxTCB

Definition at line 308 of file FreeRTOS.h.

7.15.1.64 #define portCLEAR_INTERRUPT_MASK_FROM_ISR(*uxSavedStatusValue*) (void) uxSavedStatusValue

Definition at line 304 of file FreeRTOS.h.

7.15.1.65 #define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS( )

Definition at line 680 of file FreeRTOS.h.

7.15.1.66 #define portCRITICAL_NESTING_IN_TCB 0

Definition at line 260 of file FreeRTOS.h.

7.15.1.67 #define portPOINTER_SIZE_TYPE uint32_t

Definition at line 330 of file FreeRTOS.h.

7.15.1.68 #define portPRE_TASK_DELETE_HOOK(*pvTaskToDelete, pxYieldPending*)

Definition at line 312 of file FreeRTOS.h.

7.15.1.69 #define portPRIVILEGE_BIT ((UBaseType_t) 0x00)

Definition at line 688 of file FreeRTOS.h.

7.15.1.70 #define portSET_INTERRUPT_MASK_FROM_ISR( ) 0

Definition at line 300 of file FreeRTOS.h.

7.15.1.71 #define portSETUP_TCB(*pxTCB*) (void) pxTCB

Definition at line 316 of file FreeRTOS.h.

7.15.1.72 #define portSUPPRESS_TICKS_AND_SLEEP(*xExpectedIdleTime*)

Definition at line 696 of file FreeRTOS.h.

7.15.1.73 #define portTASK_USES_FLOATING_POINT( )

Definition at line 724 of file FreeRTOS.h.

7.15.1.74 #define portTICK_RATE_MS portTICK_PERIOD_MS

Definition at line 838 of file FreeRTOS.h.

7.15.1.75 #define portTICK_TYPE_CLEAR_INTERRUPT_MASK_FROM_ISR(*x*) portCLEAR_INTERRUPT_MASK_FR↩ OM_ISR((x))

Definition at line 807 of file FreeRTOS.h.

7.15.1.76 #define portTICK_TYPE_ENTER_CRITICAL( ) portENTER_CRITICAL()

Definition at line 804 of file FreeRTOS.h.

7.15.1.77 #define portTICK_TYPE_EXIT_CRITICAL( ) portEXIT_CRITICAL()

Definition at line 805 of file FreeRTOS.h.

7.15.1.78 #define portTICK_TYPE_IS_ATOMIC 0

Definition at line 772 of file FreeRTOS.h.

7.15.1.79 #define portTICK_TYPE_SET_INTERRUPT_MASK_FROM_ISR( ) portSET_INTERRUPT_MASK_FROM_ISR()

Definition at line 806 of file FreeRTOS.h.

7.15.1.80 #define portTickType TickType_t

Definition at line 825 of file FreeRTOS.h.

7.15.1.81 #define portYIELD_WITHIN_API portYIELD

Definition at line 692 of file FreeRTOS.h.

7.15.1.82 #define tmrTIMER_CALLBACK TimerCallbackFunction_t

Definition at line 846 of file FreeRTOS.h.

7.15.1.83 #define traceBLOCKING_ON_QUEUE_RECEIVE(*pxQueue*)

Definition at line 396 of file FreeRTOS.h.

7.15.1.84 #define traceBLOCKING_ON_QUEUE_SEND(*pxQueue*)

Definition at line 404 of file FreeRTOS.h.

7.15.1.85 #define traceCREATE_COUNTING_SEMAPHORE( )

Definition at line 454 of file FreeRTOS.h.

7.15.1.86 #define traceCREATE_COUNTING_SEMAPHORE_FAILED( )

Definition at line 458 of file FreeRTOS.h.

7.15.1.87 #define traceCREATE_MUTEX(*pxNewQueue*)

Definition at line 430 of file FreeRTOS.h.

7.15.1.88 #define traceCREATE_MUTEX_FAILED( )

Definition at line 434 of file FreeRTOS.h.

7.15.1.89 #define traceEND( )

Definition at line 343 of file FreeRTOS.h.

7.15.1.90 #define traceEVENT_GROUP_CLEAR_BITS(*xEventGroup*, *uxBitsToClear*)

Definition at line 602 of file FreeRTOS.h.

7.15.1.91 #define traceEVENT_GROUP_CLEAR_BITS_FROM_ISR(*xEventGroup, uxBitsToClear*)

Definition at line 606 of file FreeRTOS.h.

7.15.1.92 #define traceEVENT_GROUP_CREATE(*xEventGroup*)

Definition at line 578 of file FreeRTOS.h.

7.15.1.93 #define traceEVENT_GROUP_CREATE_FAILED( )

Definition at line 582 of file FreeRTOS.h.

7.15.1.94 #define traceEVENT_GROUP_DELETE(*xEventGroup*)

Definition at line 618 of file FreeRTOS.h.

7.15.1.95 #define traceEVENT_GROUP_SET_BITS(*xEventGroup, uxBitsToSet*)

Definition at line 610 of file FreeRTOS.h.

7.15.1.96 #define traceEVENT_GROUP_SET_BITS_FROM_ISR(*xEventGroup, uxBitsToSet*)

Definition at line 614 of file FreeRTOS.h.

7.15.1.97 #define traceEVENT_GROUP_SYNC_BLOCK(*xEventGroup, uxBitsToSet, uxBitsToWaitFor*)

Definition at line 586 of file FreeRTOS.h.

7.15.1.98 #define traceEVENT_GROUP_SYNC_END(*xEventGroup, uxBitsToSet, uxBitsToWaitFor, xTimeoutOccurred*) (void) xTimeoutOccurred

Definition at line 590 of file FreeRTOS.h.

7.15.1.99 #define traceEVENT_GROUP_WAIT_BITS_BLOCK(*xEventGroup, uxBitsToWaitFor*)

Definition at line 594 of file FreeRTOS.h.

7.15.1.100 #define traceEVENT_GROUP_WAIT_BITS_END(*xEventGroup, uxBitsToWaitFor, xTimeoutOccurred*) (void) xTimeoutOccurred

Definition at line 598 of file FreeRTOS.h.

7.15.1.101 #define traceFREE(*pvAddress, uiSize*)

Definition at line 574 of file FreeRTOS.h.

7.15.1.102 #define traceGIVE_MUTEX_RECURSIVE(*pxMutex*)

Definition at line 438 of file FreeRTOS.h.

7.15.1.103 #define traceGIVE_MUTEX_RECURSIVE_FAILED(*pxMutex*)

Definition at line 442 of file FreeRTOS.h.

7.15.1.104 #define traceINCREASE_TICK_COUNT(*x*)

Definition at line 355 of file FreeRTOS.h.

7.15.1.105 #define traceLOW_POWER_IDLE_BEGIN( )

Definition at line 360 of file FreeRTOS.h.

7.15.1.106 #define traceLOW_POWER_IDLE_END( )

Definition at line 365 of file FreeRTOS.h.

7.15.1.107 #define traceMALLOC(*pvAddress, uiSize*)

Definition at line 570 of file FreeRTOS.h.

7.15.1.108 #define traceMOVED_TASK_TO_READY_STATE(*pxTCB*)

Definition at line 414 of file FreeRTOS.h.

7.15.1.109 #define tracePEND_FUNC_CALL(*xFunctionToPend, pvParameter1, ulParameter2, ret*)

Definition at line 622 of file FreeRTOS.h.

7.15.1.110 #define tracePEND_FUNC_CALL_FROM_ISR(*xFunctionToPend, pvParameter1, ulParameter2, ret*)

Definition at line 626 of file FreeRTOS.h.

7.15.1.111 #define tracePOST_MOVED_TASK_TO_READY_STATE(*pxTCB*)

Definition at line 418 of file FreeRTOS.h.

7.15.1.112 #define traceQUEUE_CREATE(*pxNewQueue*)

Definition at line 422 of file FreeRTOS.h.

7.15.1.113 #define traceQUEUE_CREATE_FAILED(*ucQueueType*)

Definition at line 426 of file FreeRTOS.h.

7.15.1.114 #define traceQUEUE_DELETE(*pxQueue*)

Definition at line 506 of file FreeRTOS.h.

7.15.1.115 #define traceQUEUE_PEEK(*pxQueue*)

Definition at line 474 of file FreeRTOS.h.

7.15.1.116 #define traceQUEUE_PEEK_FROM_ISR(*pxQueue*)

Definition at line 478 of file FreeRTOS.h.

7.15.1.117 #define traceQUEUE_PEEK_FROM_ISR_FAILED(*pxQueue*)

Definition at line 502 of file FreeRTOS.h.

7.15.1.118 #define traceQUEUE_RECEIVE(*pxQueue*)

Definition at line 470 of file FreeRTOS.h.

7.15.1.119 #define traceQUEUE_RECEIVE_FAILED(*pxQueue*)

Definition at line 482 of file FreeRTOS.h.

7.15.1.120 #define traceQUEUE_RECEIVE_FROM_ISR(*pxQueue*)

Definition at line 494 of file FreeRTOS.h.

7.15.1.121 #define traceQUEUE_RECEIVE_FROM_ISR_FAILED(*pxQueue*)

Definition at line 498 of file FreeRTOS.h.

7.15.1.122 #define traceQUEUE_REGISTRY_ADD(**xQueue**, *pcQueueName*)

Definition at line 630 of file FreeRTOS.h.

7.15.1.123 #define traceQUEUE_SEND(*pxQueue*)

Definition at line 462 of file FreeRTOS.h.

7.15.1.124 #define traceQUEUE_SEND_FAILED(*pxQueue*)

Definition at line 466 of file FreeRTOS.h.

7.15.1.125 #define traceQUEUE_SEND_FROM_ISR(*pxQueue*)

Definition at line 486 of file FreeRTOS.h.

7.15.1.126 #define traceQUEUE_SEND_FROM_ISR_FAILED(*pxQueue*)

Definition at line 490 of file FreeRTOS.h.

7.15.1.127 #define traceSTART( )

Definition at line 337 of file FreeRTOS.h.

7.15.1.128 #define traceTAKE_MUTEX_RECURSIVE(*pxMutex*)

Definition at line 446 of file FreeRTOS.h.

7.15.1.129 #define traceTAKE_MUTEX_RECURSIVE_FAILED(*pxMutex*)

Definition at line 450 of file FreeRTOS.h.

7.15.1.130 #define traceTASK_CREATE(*pxNewTCB*)

Definition at line 510 of file FreeRTOS.h.

7.15.1.131 #define traceTASK_CREATE_FAILED( )

Definition at line 514 of file FreeRTOS.h.

7.15.1.132 #define traceTASK_DELAY( )

Definition at line 526 of file FreeRTOS.h.

7.15.1.133 #define traceTASK_DELAY_UNTIL(*x*)

Definition at line 522 of file FreeRTOS.h.

7.15.1.134 #define traceTASK_DELETE(*pxTaskToDelete*)

Definition at line 518 of file FreeRTOS.h.

7.15.1.135 #define traceTASK_INCREMENT_TICK(xTickCount)

Definition at line 546 of file FreeRTOS.h.

7.15.1.136 #define traceTASK_NOTIFY( )

Definition at line 650 of file FreeRTOS.h.

7.15.1.137 #define traceTASK_NOTIFY_FROM_ISR( )

Definition at line 654 of file FreeRTOS.h.

7.15.1.138 #define traceTASK_NOTIFY_GIVE_FROM_ISR( )

Definition at line 658 of file FreeRTOS.h.

7.15.1.139 #define traceTASK_NOTIFY_TAKE( )

Definition at line 638 of file FreeRTOS.h.

7.15.1.140 #define traceTASK_NOTIFY_TAKE_BLOCK( )

Definition at line 634 of file FreeRTOS.h.

7.15.1.141 #define traceTASK_NOTIFY_WAIT( )

Definition at line 646 of file FreeRTOS.h.

7.15.1.142 #define traceTASK_NOTIFY_WAIT_BLOCK( )

Definition at line 642 of file FreeRTOS.h.

7.15.1.143 #define traceTASK_PRIORITY_DISINHERIT(*pxTCBOfMutexHolder, uxOriginalPriority*)

Definition at line 388 of file FreeRTOS.h.

7.15.1.144 #define traceTASK_PRIORITY_INHERIT(*pxTCBOfMutexHolder, uxInheritedPriority*)

Definition at line 380 of file FreeRTOS.h.

7.15.1.145 #define traceTASK_PRIORITY_SET(*pxTask, uxNewPriority*)

Definition at line 530 of file FreeRTOS.h.

7.15.1.146 #define traceTASK_RESUME(*pxTaskToResume*)

Definition at line 538 of file FreeRTOS.h.

7.15.1.147 #define traceTASK_RESUME_FROM_ISR(*pxTaskToResume*)

Definition at line 542 of file FreeRTOS.h.

7.15.1.148 #define traceTASK_SUSPEND(*pxTaskToSuspend*)

Definition at line 534 of file FreeRTOS.h.

7.15.1.149 #define traceTASK_SWITCHED_IN( )

Definition at line 349 of file FreeRTOS.h.

7.15.1.150 #define traceTASK_SWITCHED_OUT( )

Definition at line 371 of file FreeRTOS.h.

7.15.1.151 #define traceTIMER_COMMAND_RECEIVED(*pxTimer, xMessageID, xMessageValue*)

Definition at line 566 of file FreeRTOS.h.

7.15.1.152 #define traceTIMER_COMMAND_SEND(*xTimer, xMessageID, xMessageValueValue, xReturn*)

Definition at line 558 of file FreeRTOS.h.

7.15.1.153 #define traceTIMER_CREATE(*pxNewTimer*)

Definition at line 550 of file FreeRTOS.h.

7.15.1.154 #define traceTIMER_CREATE_FAILED( )

Definition at line 554 of file FreeRTOS.h.

7.15.1.155 #define traceTIMER_EXPIRED(*pxTimer*)

Definition at line 562 of file FreeRTOS.h.

7.15.1.156 #define vQueueAddToRegistry(**xQueue**, *pcName*)

Definition at line 324 of file FreeRTOS.h.

7.15.1.157 #define vQueueUnregisterQueue(**xQueue**)

Definition at line 325 of file FreeRTOS.h.

7.15.1.158 #define vTaskGetTaskInfo **vTaskGetInfo**

Definition at line 842 of file FreeRTOS.h.

7.15.1.159 #define xCoRoutineHandle **CoRoutineHandle_t**

Definition at line 836 of file FreeRTOS.h.

7.15.1.160 #define xList **List_t**

Definition at line 849 of file FreeRTOS.h.

7.15.1.161 #define xListItem ListItem_t

Definition at line 848 of file FreeRTOS.h.

7.15.1.162 #define xMemoryRegion MemoryRegion_t

Definition at line 832 of file FreeRTOS.h.

7.15.1.163 #define xQueueHandle QueueHandle_t

Definition at line 827 of file FreeRTOS.h.

7.15.1.164 #define xQueueSetHandle QueueSetHandle_t

Definition at line 829 of file FreeRTOS.h.

7.15.1.165 #define xQueueSetMemberHandle QueueSetMemberHandle_t

Definition at line 830 of file FreeRTOS.h.

7.15.1.166 #define xSemaphoreHandle SemaphoreHandle_t

Definition at line 828 of file FreeRTOS.h.

7.15.1.167 #define xTaskHandle TaskHandle_t

Definition at line 826 of file FreeRTOS.h.

7.15.1.168 #define xTaskParameters TaskParameters_t

Definition at line 833 of file FreeRTOS.h.

7.15.1.169 #define xTaskStatusType TaskStatus_t

Definition at line 834 of file FreeRTOS.h.

7.15.1.170 #define xTimeOutType TimeOut_t

Definition at line 831 of file FreeRTOS.h.

7.15.1.171 #define xTimerHandle TimerHandle_t

Definition at line 835 of file FreeRTOS.h.

7.15.2 Typedef Documentation

7.15.2.1 typedef struct xSTATIC_EVENT_GROUP StaticEventGroup_t

7.15.2.2 typedef struct xSTATIC_LIST StaticList_t

7.15.2.3 typedef struct xSTATIC_LIST_ITEM StaticListItem_t

Definition at line 879 of file FreeRTOS.h.

7.15.2.4 typedef struct xSTATIC_MINI_LIST_ITEM StaticMiniListItem_t

Definition at line 887 of file FreeRTOS.h.

7.15.2.5 typedef struct xSTATIC_QUEUE StaticQueue_t

7.15.2.6 typedef StaticQueue_t StaticSemaphore_t

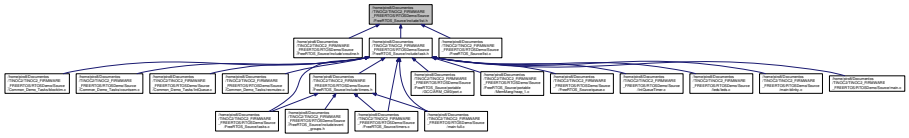
Definition at line 996 of file FreeRTOS.h.

7.15.2.7 typedef struct xSTATIC_TCB StaticTask_t

7.15.2.8 typedef struct xSTATIC_TIMER StaticTimer_t

7.16 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/FreeRTOS_Source/include/list.h File Reference

This graph shows which files directly or indirectly include this file:



Classes

- struct xLIST_ITEM
- struct xMINI_LIST_ITEM
- struct xLIST

Macros

- #define configLIST_VOLATILE
- #define listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE
- #define listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE
- #define listFIRST_LIST_INTEGRITY_CHECK_VALUE
- #define listSECOND_LIST_INTEGRITY_CHECK_VALUE
- #define listSET_FIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE(pxItem)
- #define listSET_SECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE(pxItem)
- #define listSET_LIST_INTEGRITY_CHECK_1_VALUE(pxList)
- #define listSET_LIST_INTEGRITY_CHECK_2_VALUE(pxList)
- #define listTEST_LIST_ITEM_INTEGRITY(pxItem)
- #define listTEST_LIST_INTEGRITY(pxList)
- #define listSET_LIST_ITEM_OWNER(pxListItem, pxOwner) ((pxListItem)->pvOwner = (void *) (pxOwner))
- #define listGET_LIST_ITEM_OWNER(pxListItem) ((pxListItem)->pvOwner)
- #define listSET_LIST_ITEM_VALUE(pxListItem, xValue) ((pxListItem)->xItemValue = (xValue))
- #define listGET_LIST_ITEM_VALUE(pxListItem) ((pxListItem)->xItemValue)
- #define listGET_ITEM_VALUE_OF_HEAD_ENTRY(pxList) (((pxList)->xListEnd).pxNext->xItemValue)
- #define listGET_HEAD_ENTRY(pxList) (((pxList)->xListEnd).pxNext)
- #define listGET_NEXT(pxListItem) ((pxListItem)->pxNext)
- #define listGET_END_MARKER(pxList) ((ListItem_t const *) (&((pxList)->xListEnd)))
- #define listLIST_IS_EMPTY(pxList) ((BaseType_t) ((pxList)->uxNumberOfItems == (UBaseType_t) 0))
- #define listCURRENT_LIST_LENGTH(pxList) ((pxList)->uxNumberOfItems)
- #define listGET_OWNER_OF_NEXT_ENTRY(pxTCB, pxList)
- #define listGET_OWNER_OF_HEAD_ENTRY(pxList) ((&((pxList)->xListEnd))->pxNext->pvOwner)
- #define listIS_CONTAINED_WITHIN(pxList, pxListItem) ((BaseType_t) ((pxListItem)->pvContainer == (void *) (pxList)))
- #define listLIST_ITEM_CONTAINER(pxListItem) ((pxListItem)->pvContainer)
- #define listLIST_IS_INITIALISED(pxList) ((pxList)->xListEnd.xItemValue == portMAX_DELAY)

Typedefs

- typedef struct xLIST_ITEM ListItem_t
- typedef struct xMINI_LIST_ITEM MiniListItem_t
- typedef struct xLIST List_t

Functions

- void vListInitialise (List_t *const pxList) PRIVILEGED_FUNCTION
- void vListInitialiseItem (ListItem_t *const pxItem) PRIVILEGED_FUNCTION
- void vListInsert (List_t *const pxList, ListItem_t *const pxNewListItem) PRIVILEGED_FUNCTION
- void vListInsertEnd (List_t *const pxList, ListItem_t *const pxNewListItem) PRIVILEGED_FUNCTION
- UBaseType_t uxListRemove (ListItem_t *const pxItemToRemove) PRIVILEGED_FUNCTION

7.16.1 Macro Definition Documentation

7.16.1.1 #define configLIST_VOLATILE

Definition at line 134 of file list.h.

7.16.1.2 #define listCURRENT_LIST_LENGTH(*pxList*) ((pxList)->uxNumberOfItems)

Definition at line 296 of file list.h.

7.16.1.3 #define listFIRST_LIST_INTEGRITY_CHECK_VALUE

Definition at line 150 of file list.h.

7.16.1.4 #define listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE

Definition at line 148 of file list.h.

7.16.1.5 #define listGET_END_MARKER(*pxList*) ((ListItem_t const *) (&((pxList)->xListEnd)))

Definition at line 282 of file list.h.

7.16.1.6 #define listGET_HEAD_ENTRY(*pxList*) (((pxList)->xListEnd).pxNext)

Definition at line 266 of file list.h.

7.16.1.7 #define listGET_ITEM_VALUE_OF_HEAD_ENTRY(*pxList*) (((pxList)->xListEnd).pxNext->xItemValue)

Definition at line 258 of file list.h.

7.16.1.8 #define listGET_LIST_ITEM_OWNER(*pxListItem*) ((pxListItem)->pvOwner)

Definition at line 230 of file list.h.

7.16.1.9 #define listGET_LIST_ITEM_VALUE(*pxListItem*) ((pxListItem)->xItemValue)

Definition at line 249 of file list.h.

7.16.1.10 #define listGET_NEXT(*pxListItem*) ((pxListItem)->pxNext)

Definition at line 274 of file list.h.

7.16.1.11 #define listGET_OWNER_OF_HEAD_ENTRY(*pxList*) ((&((pxList)->xListEnd))->pxNext->pvOwner)

Definition at line 348 of file list.h.

7.16.1.12 #define listGET_OWNER_OF_NEXT_ENTRY(*pxTCB, pxList*)

Value:

```
{
      \
List_t * const pxConstList = ( pxList );                                                \
    /* Increment the index to the next item and return the item, ensuring */              \
    /* we don't return the marker used at the end of the list.  */                        \
    ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;                          \
    if( ( void * ) ( pxConstList )->pxIndex == ( void * ) &( ( pxConstList )->xListEnd ) )  \
    {                                                                                     \
        ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;                      \
    }                                                                                       \
    ( pxTCB ) = ( pxConstList )->pxIndex->pvOwner;                                           \
}
```

Definition at line 318 of file list.h.

7.16.1.13 #define listIS_CONTAINED_WITHIN(*pxList, pxListItem*) ((BaseType_t) ((pxListItem)->pvContainer == (void *) (pxList)))

Definition at line 359 of file list.h.

7.16.1.14 #define listLIST_IS_EMPTY(*pxList*) ((BaseType_t) ((pxList)->uxNumberOfItems == (UBaseType_t) 0))

Definition at line 291 of file list.h.

7.16.1.15 #define listLIST_IS_INITIALISED(*pxList*) ((pxList)->xListEnd.xItemValue == portMAX_DELAY)

Definition at line 374 of file list.h.

7.16.1.16 #define listLIST_ITEM_CONTAINER(*pxListItem*) ((pxListItem)->pvContainer)

Definition at line 367 of file list.h.

7.16.1.17 #define listSECOND_LIST_INTEGRITY_CHECK_VALUE

Definition at line 151 of file list.h.

7.16.1.18 #define listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE

Definition at line 149 of file list.h.

7.16.1.19 #define listSET_FIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE(*pxItem*)

Definition at line 152 of file list.h.

7.16.1.20 #define listSET_LIST_INTEGRITY_CHECK_1_VALUE(*pxList*)

Definition at line 154 of file list.h.

7.16.1.21 #define listSET_LIST_INTEGRITY_CHECK_2_VALUE(*pxList*)

Definition at line 155 of file list.h.

7.16.1.22 #define listSET_LIST_ITEM_OWNER(*pxListItem, pxOwner*) ((pxListItem)->pvOwner = (void *) (pxOwner))

Definition at line 221 of file list.h.

7.16.1.23 #define listSET_LIST_ITEM_VALUE(*pxListItem, xValue*) ((pxListItem)->xItemValue = (xValue))

Definition at line 239 of file list.h.

7.16.1.24 #define listSET_SECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE(*pxItem*)

Definition at line 153 of file list.h.

7.16.1.25 #define listTEST_LIST_INTEGRITY(*pxList*)

Definition at line 157 of file list.h.

7.16.1.26 #define listTEST_LIST_ITEM_INTEGRITY(*pxItem*)

Definition at line 156 of file list.h.

7.16.2 Typedef Documentation

7.16.2.1 typedef struct xLIST List_t

7.16.2.2 typedef struct xLIST_ITEM ListItem_t

Definition at line 191 of file list.h.

7.16.2.3 typedef struct xMINI_LIST_ITEM MiniListItem_t

Definition at line 200 of file list.h.

7.16.3 Function Documentation

7.16.3.1 UBaseType_t uxListRemove (ListItem_t *const *pxItemToRemove*)

Definition at line 212 of file list.c.

7.16.3.2 void vListInitialise (List_t *const *pxList*)

Definition at line 79 of file list.c.

7.16.3.3 void vListInitialiseItem (ListItem_t *const *pxItem*)

Definition at line 104 of file list.c.

7.16.3.4 void vListInsert (List_t *const *pxList*, ListItem_t *const *pxNewListItem*)

Definition at line 145 of file list.c.

7.16.3.5 void vListInsertEnd (List_t *const *pxList*, ListItem_t *const *pxNewListItem*)

Definition at line 116 of file list.c.

7.17 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/include/mpu_prototypes.h File Reference

Functions

- BaseType_t MPU_xTaskCreate (TaskFunction_t pxTaskCode, const char *const pcName, const uint16_t usStackDepth, void *const pvParameters, UBaseType_t uxPriority, TaskHandle_t *const pxCreatedTask)
- TaskHandle_t MPU_xTaskCreateStatic (TaskFunction_t pxTaskCode, const char *const pcName, const uint32_t ulStackDepth, void *const pvParameters, UBaseType_t uxPriority, StackType_t *const puxStackBuffer, StaticTask_t *const pxTaskBuffer)
- BaseType_t MPU_xTaskCreateRestricted (const TaskParameters_t *const pxTaskDefinition, TaskHandle_t *pxCreatedTask)
- void MPU_vTaskAllocateMPURegions (TaskHandle_t xTask, const MemoryRegion_t *const pxRegions)
- void MPU_vTaskDelete (TaskHandle_t xTaskToDelete)
- void MPU_vTaskDelay (const TickType_t xTicksToDelay)
- void MPU_vTaskDelayUntil (TickType_t *const pxPreviousWakeTime, const TickType_t xTimeIncrement)
- BaseType_t MPU_xTaskAbortDelay (TaskHandle_t xTask)
- UBaseType_t MPU_uxTaskPriorityGet (TaskHandle_t xTask)
- eTaskState MPU_eTaskGetState (TaskHandle_t xTask)
- void MPU_vTaskGetInfo (TaskHandle_t xTask, TaskStatus_t *pxTaskStatus, BaseType_t xGetFreeStackSpace, eTaskState eState)
- void MPU_vTaskPrioritySet (TaskHandle_t xTask, UBaseType_t uxNewPriority)
- void MPU_vTaskSuspend (TaskHandle_t xTaskToSuspend)
- void MPU_vTaskResume (TaskHandle_t xTaskToResume)

- void MPU_vTaskStartScheduler (void)
- void MPU_vTaskSuspendAll (void)
- BaseType_t MPU_xTaskResumeAll (void)
- TickType_t MPU_xTaskGetTickCount (void)
- UBaseType_t MPU_uxTaskGetNumberOfTasks (void)
- char * MPU_pcTaskGetName (TaskHandle_t xTaskToQuery)
- TaskHandle_t MPU_xTaskGetHandle (const char *pcNameToQuery)
- UBaseType_t MPU_uxTaskGetStackHighWaterMark (TaskHandle_t xTask)
- void MPU_vTaskSetApplicationTaskTag (TaskHandle_t xTask, TaskHookFunction_t pxHookFunction)
- TaskHookFunction_t MPU_xTaskGetApplicationTaskTag (TaskHandle_t xTask)
- void MPU_vTaskSetThreadLocalStoragePointer (TaskHandle_t xTaskToSet, BaseType_t xIndex, void *pv↩Value)
- void * MPU_pvTaskGetThreadLocalStoragePointer (TaskHandle_t xTaskToQuery, BaseType_t xIndex)
- BaseType_t MPU_xTaskCallApplicationTaskHook (TaskHandle_t xTask, void *pvParameter)
- TaskHandle_t MPU_xTaskGetIdleTaskHandle (void)
- UBaseType_t MPU_uxTaskGetSystemState (TaskStatus_t *const pxTaskStatusArray, const UBaseType_↩t uxArraySize, uint32_t *const pulTotalRunTime)
- void MPU_vTaskList (char *pcWriteBuffer)
- void MPU_vTaskGetRunTimeStats (char *pcWriteBuffer)
- BaseType_t MPU_xTaskGenericNotify (TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction, uint32_t *pulPreviousNotificationValue)
- BaseType_t MPU_xTaskNotifyWait (uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_↩t *pulNotificationValue, TickType_t xTicksToWait)
- uint32_t MPU_ulTaskNotifyTake (BaseType_t xClearCountOnExit, TickType_t xTicksToWait)
- BaseType_t MPU_xTaskNotifyStateClear (TaskHandle_t xTask)
- BaseType_t MPU_xTaskIncrementTick (void)
- TaskHandle_t MPU_xTaskGetCurrentTaskHandle (void)
- void MPU_vTaskSetTimeOutState (TimeOut_t *const pxTimeOut)
- BaseType_t MPU_xTaskCheckForTimeOut (TimeOut_t *const pxTimeOut, TickType_t *const pxTicksToWait)
- void MPU_vTaskMissedYield (void)
- BaseType_t MPU_xTaskGetSchedulerState (void)
- BaseType_t MPU_xQueueGenericSend (QueueHandle_t xQueue, const void *const pvItemToQueue, Tick↩Type_t xTicksToWait, const BaseType_t xCopyPosition)
- BaseType_t MPU_xQueueGenericReceive (QueueHandle_t xQueue, void *const pvBuffer, TickType_t x↩TicksToWait, const BaseType_t xJustPeek)
- UBaseType_t MPU_uxQueueMessagesWaiting (const QueueHandle_t xQueue)
- UBaseType_t MPU_uxQueueSpacesAvailable (const QueueHandle_t xQueue)
- void MPU_vQueueDelete (QueueHandle_t xQueue)
- QueueHandle_t MPU_xQueueCreateMutex (const uint8_t ucQueueType)
- QueueHandle_t MPU_xQueueCreateMutexStatic (const uint8_t ucQueueType, StaticQueue_t *pxStatic↩Queue)
- QueueHandle_t MPU_xQueueCreateCountingSemaphore (const UBaseType_t uxMaxCount, const UBase↩Type_t uxInitialCount)
- QueueHandle_t MPU_xQueueCreateCountingSemaphoreStatic (const UBaseType_t uxMaxCount, const UBaseType_t uxInitialCount, StaticQueue_t *pxStaticQueue)
- void * MPU_xQueueGetMutexHolder (QueueHandle_t xSemaphore)
- BaseType_t MPU_xQueueTakeMutexRecursive (QueueHandle_t xMutex, TickType_t xTicksToWait)
- BaseType_t MPU_xQueueGiveMutexRecursive (QueueHandle_t pxMutex)
- void MPU_vQueueAddToRegistry (QueueHandle_t xQueue, const char *pcName)
- void MPU_vQueueUnregisterQueue (QueueHandle_t xQueue)
- const char * MPU_pcQueueGetName (QueueHandle_t xQueue)
- QueueHandle_t MPU_xQueueGenericCreate (const UBaseType_t uxQueueLength, const UBaseType_t ux↩ItemSize, const uint8_t ucQueueType)
- QueueHandle_t MPU_xQueueGenericCreateStatic (const UBaseType_t uxQueueLength, const UBase↩Type_t uxItemSize, uint8_t *pucQueueStorage, StaticQueue_t *pxStaticQueue, const uint8_t ucQueueType)

- QueueSetHandle_t MPU_xQueueCreateSet (const UBaseType_t uxEventQueueLength)
- BaseType_t MPU_xQueueAddToSet (QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet)
- BaseType_t MPU_xQueueRemoveFromSet (QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet)
- QueueSetMemberHandle_t MPU_xQueueSelectFromSet (QueueSetHandle_t xQueueSet, const TickType_t xTicksToWait)
- BaseType_t MPU_xQueueGenericReset (QueueHandle_t xQueue, BaseType_t xNewQueue)
- void MPU_vQueueSetQueueNumber (QueueHandle_t xQueue, UBaseType_t uxQueueNumber)
- UBaseType_t MPU_uxQueueGetQueueNumber (QueueHandle_t xQueue)
- uint8_t MPU_ucQueueGetQueueType (QueueHandle_t xQueue)
- TimerHandle_t MPU_xTimerCreate (const char *const pcTimerName, const TickType_t xTimerPeriodInTicks, const UBaseType_t uxAutoReload, void *const pvTimerID, TimerCallbackFunction_t pxCallbackFunction)
- TimerHandle_t MPU_xTimerCreateStatic (const char *const pcTimerName, const TickType_t xTimerPeriodInTicks, const UBaseType_t uxAutoReload, void *const pvTimerID, TimerCallbackFunction_t pxCallbackFunction, StaticTimer_t *pxTimerBuffer)
- void * MPU_pvTimerGetTimerID (const TimerHandle_t xTimer)
- void MPU_vTimerSetTimerID (TimerHandle_t xTimer, void *pvNewID)
- BaseType_t MPU_xTimerIsTimerActive (TimerHandle_t xTimer)
- TaskHandle_t MPU_xTimerGetTimerDaemonTaskHandle (void)
- BaseType_t MPU_xTimerPendFunctionCall (PendedFunction_t xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, TickType_t xTicksToWait)
- const char * MPU_pcTimerGetName (TimerHandle_t xTimer)
- TickType_t MPU_xTimerGetPeriod (TimerHandle_t xTimer)
- TickType_t MPU_xTimerGetExpiryTime (TimerHandle_t xTimer)
- BaseType_t MPU_xTimerCreateTimerTask (void)
- BaseType_t MPU_xTimerGenericCommand (TimerHandle_t xTimer, const BaseType_t xCommandID, const TickType_t xOptionalValue, BaseType_t *const pxHigherPriorityTaskWoken, const TickType_t xTicksToWait)
- EventGroupHandle_t MPU_xEventGroupCreate (void)
- EventGroupHandle_t MPU_xEventGroupCreateStatic (StaticEventGroup_t *pxEventGroupBuffer)
- EventBits_t MPU_xEventGroupWaitBits (EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits, TickType_t xTicksToWait)
- EventBits_t MPU_xEventGroupClearBits (EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToClear)
- EventBits_t MPU_xEventGroupSetBits (EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet)
- EventBits_t MPU_xEventGroupSync (EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet, const EventBits_t uxBitsToWaitFor, TickType_t xTicksToWait)
- void MPU_vEventGroupDelete (EventGroupHandle_t xEventGroup)
- UBaseType_t MPU_uxEventGroupGetNumber (void *xEventGroup)

7.17.1 Function Documentation

7.17.1.1 eTaskState MPU_eTaskGetState (TaskHandle_t *xTask*)

7.17.1.2 const char* MPU_pcQueueGetName (QueueHandle_t *xQueue*)

7.17.1.3 char* MPU_pcTaskGetName (TaskHandle_t *xTaskToQuery*)

7.17.1.4 const char* MPU_pcTimerGetName (TimerHandle_t *xTimer*)

7.17.1.5 void* MPU_pvTaskGetThreadLocalStoragePointer (TaskHandle_t *xTaskToQuery,* BaseType_t *xIndex*)

7.17.1.6 void* MPU_pvTimerGetTimerID (const TimerHandle_t *xTimer*)

7.17.1.7 uint8_t MPU_ucQueueGetQueueType (QueueHandle_t *xQueue*)

7.17.1.8 uint32_t MPU_ulTaskNotifyTake (BaseType_t *xClearCountOnExit,* TickType_t *xTicksToWait*)

7.17.1.9 UBaseType_t MPU_uxEventGroupGetNumber (void * *xEventGroup*)

7.17.1.10 UBaseType_t MPU_uxQueueGetQueueNumber (QueueHandle_t *xQueue*)

7.17.1.11 UBaseType_t MPU_uxQueueMessagesWaiting (const QueueHandle_t *xQueue*)

7.17.1.12 UBaseType_t MPU_uxQueueSpacesAvailable (const QueueHandle_t *xQueue*)

7.17.1.13 UBaseType_t MPU_uxTaskGetNumberOfTasks (void)

7.17.1.14 UBaseType_t MPU_uxTaskGetStackHighWaterMark (TaskHandle_t *xTask*)

7.17.1.15 UBaseType_t MPU_uxTaskGetSystemState (TaskStatus_t *const *pxTaskStatusArray,* const UBaseType_t *uxArraySize,* uint32_t *const *pulTotalRunTime*)

7.17.1.16 UBaseType_t MPU_uxTaskPriorityGet (TaskHandle_t *xTask*)

7.17.1.17 void MPU_vEventGroupDelete (EventGroupHandle_t *xEventGroup*)

7.17.1.18 void MPU_vQueueAddToRegistry (QueueHandle_t *xQueue,* const char * *pcName*)

7.17.1.19 void MPU_vQueueDelete (QueueHandle_t *xQueue*)

7.17.1.20 void MPU_vQueueSetQueueNumber (QueueHandle_t *xQueue,* UBaseType_t *uxQueueNumber*)

7.17.1.21 void MPU_vQueueUnregisterQueue (QueueHandle_t *xQueue*)

7.17.1.22 void MPU_vTaskAllocateMPURegions (TaskHandle_t *xTask,* const MemoryRegion_t *const *pxRegions*)

7.17.1.23 void MPU_vTaskDelay (const TickType_t *xTicksToDelay*)

7.17.1.24 void MPU_vTaskDelayUntil (TickType_t *const *pxPreviousWakeTime,* const TickType_t *xTimeIncrement*)

7.17.1.25 void MPU_vTaskDelete (TaskHandle_t *xTaskToDelete*)

7.17.1.26 void MPU_vTaskGetInfo (TaskHandle_t *xTask,* TaskStatus_t * *pxTaskStatus,* BaseType_t *xGetFreeStackSpace,* eTaskState *eState*)

7.17.1.27 void MPU_vTaskGetRunTimeStats (char * *pcWriteBuffer*)

7.17.1.28 void MPU_vTaskList (char * *pcWriteBuffer*)

7.17.1.29 void MPU_vTaskMissedYield (void)

7.17.1.30 void MPU_vTaskPrioritySet (TaskHandle_t *xTask,* UBaseType_t *uxNewPriority*)

7.17.1.31 void MPU_vTaskResume (TaskHandle_t *xTaskToResume*)

7.17.1.32 void MPU_vTaskSetApplicationTaskTag (TaskHandle_t *xTask,* TaskHookFunction_t *pxHookFunction*)

7.17.1.33 void MPU_vTaskSetThreadLocalStoragePointer (TaskHandle_t *xTaskToSet,* BaseType_t *xIndex,* void * *pvValue*)

7.17.1.34 void MPU_vTaskSetTimeOutState (TimeOut_t *const *pxTimeOut*)

7.17.1.35 void MPU_vTaskStartScheduler (void)

7.17.1.36 void MPU_vTaskSuspend (TaskHandle_t *xTaskToSuspend*)

7.17.1.37 void MPU_vTaskSuspendAll (void)

7.17.1.38 void MPU_vTimerSetTimerID (TimerHandle_t *xTimer,* void * *pvNewID*)

7.17.1.39 EventBits_t MPU_xEventGroupClearBits (EventGroupHandle_t *xEventGroup,* const EventBits_t *uxBitsToClear*)

7.17.1.40 EventGroupHandle_t MPU_xEventGroupCreate (void)

7.17.1.41 EventGroupHandle_t MPU_xEventGroupCreateStatic (StaticEventGroup_t * *pxEventGroupBuffer*)

7.17.1.42 EventBits_t MPU_xEventGroupSetBits (EventGroupHandle_t *xEventGroup,* const EventBits_t *uxBitsToSet*)

7.17.1.43 EventBits_t MPU_xEventGroupSync (EventGroupHandle_t *xEventGroup,* const EventBits_t *uxBitsToSet,* const EventBits_t *uxBitsToWaitFor,* TickType_t *xTicksToWait*)

7.17.1.44 EventBits_t MPU_xEventGroupWaitBits (EventGroupHandle_t *xEventGroup,* const EventBits_t *uxBitsToWaitFor,* const BaseType_t *xClearOnExit,* const BaseType_t *xWaitForAllBits,* TickType_t *xTicksToWait*)

7.17.1.45 BaseType_t MPU_xQueueAddToSet (QueueSetMemberHandle_t *xQueueOrSemaphore,* QueueSetHandle_t *xQueueSet*)

7.17.1.46 QueueHandle_t MPU_xQueueCreateCountingSemaphore (const UBaseType_t *uxMaxCount,* const UBaseType_t *uxInitialCount*)

7.17.1.47 **QueueHandle_t** MPU_xQueueCreateCountingSemaphoreStatic (const **UBaseType_t** *uxMaxCount,* const **UBaseType_t** *uxInitialCount,* **StaticQueue_t** * *pxStaticQueue*)

7.17.1.48 **QueueHandle_t** MPU_xQueueCreateMutex (const uint8_t *ucQueueType*)

7.17.1.49 **QueueHandle_t** MPU_xQueueCreateMutexStatic (const uint8_t *ucQueueType,* **StaticQueue_t** * *pxStaticQueue*)

7.17.1.50 **QueueSetHandle_t** MPU_xQueueCreateSet (const **UBaseType_t** *uxEventQueueLength*)

7.17.1.51 **QueueHandle_t** MPU_xQueueGenericCreate (const **UBaseType_t** *uxQueueLength,* const **UBaseType_t** *uxItemSize,* const uint8_t *ucQueueType*)

7.17.1.52 **QueueHandle_t** MPU_xQueueGenericCreateStatic (const **UBaseType_t** *uxQueueLength,* const **UBaseType_t** *uxItemSize,* uint8_t * *pucQueueStorage,* **StaticQueue_t** * *pxStaticQueue,* const uint8_t *ucQueueType*)

7.17.1.53 **BaseType_t** MPU_xQueueGenericReceive (**QueueHandle_t** *xQueue,* void *const *pvBuffer,* **TickType_t** *xTicksToWait,* const **BaseType_t** *xJustPeek*)

7.17.1.54 **BaseType_t** MPU_xQueueGenericReset (**QueueHandle_t** *xQueue,* **BaseType_t** *xNewQueue*)

7.17.1.55 **BaseType_t** MPU_xQueueGenericSend (**QueueHandle_t** *xQueue,* const void *const *pvItemToQueue,* **TickType_t** *xTicksToWait,* const **BaseType_t** *xCopyPosition*)

7.17.1.56 void* MPU_xQueueGetMutexHolder (**QueueHandle_t** *xSemaphore*)

7.17.1.57 **BaseType_t** MPU_xQueueGiveMutexRecursive (**QueueHandle_t** *pxMutex*)

7.17.1.58 **BaseType_t** MPU_xQueueRemoveFromSet (**QueueSetMemberHandle_t** *xQueueOrSemaphore,* **QueueSetHandle_t** *xQueueSet*)

7.17.1.59 **QueueSetMemberHandle_t** MPU_xQueueSelectFromSet (**QueueSetHandle_t** *xQueueSet,* const **TickType_t** *xTicksToWait*)

7.17.1.60 **BaseType_t** MPU_xQueueTakeMutexRecursive (**QueueHandle_t** *xMutex,* **TickType_t** *xTicksToWait*)

7.17.1.61 **BaseType_t** MPU_xTaskAbortDelay (**TaskHandle_t** *xTask*)

7.17.1.62 **BaseType_t** MPU_xTaskCallApplicationTaskHook (**TaskHandle_t** *xTask,* void * *pvParameter*)

7.17.1.63 **BaseType_t** MPU_xTaskCheckForTimeOut (**TimeOut_t** *const *pxTimeOut,* **TickType_t** *const *pxTicksToWait*)

7.17.1.64 **BaseType_t** MPU_xTaskCreate (**TaskFunction_t** *pxTaskCode,* const char *const *pcName,* const uint16_t *usStackDepth,* void *const *pvParameters,* **UBaseType_t** *uxPriority,* **TaskHandle_t** *const *pxCreatedTask*)

7.17.1.65 **BaseType_t** MPU_xTaskCreateRestricted (const **TaskParameters_t** *const *pxTaskDefinition,* **TaskHandle_t** * *pxCreatedTask*)

7.17.1.66 **TaskHandle_t** MPU_xTaskCreateStatic (**TaskFunction_t** *pxTaskCode,* const char ∗const *pcName,* const uint32_t *ulStackDepth,* void ∗const *pvParameters,* **UBaseType_t** *uxPriority,* **StackType_t** ∗const *puxStackBuffer,* **StaticTask_t** ∗const *pxTaskBuffer*)

7.17.1.67 **BaseType_t** MPU_xTaskGenericNotify (**TaskHandle_t** *xTaskToNotify,* uint32_t *ulValue,* **eNotifyAction** *eAction,* uint32_t ∗ *pulPreviousNotificationValue*)

7.17.1.68 **TaskHookFunction_t** MPU_xTaskGetApplicationTaskTag (**TaskHandle_t** *xTask*)

7.17.1.69 **TaskHandle_t** MPU_xTaskGetCurrentTaskHandle (void)

7.17.1.70 **TaskHandle_t** MPU_xTaskGetHandle (const char ∗ *pcNameToQuery*)

7.17.1.71 **TaskHandle_t** MPU_xTaskGetIdleTaskHandle (void)

7.17.1.72 **BaseType_t** MPU_xTaskGetSchedulerState (void)

7.17.1.73 **TickType_t** MPU_xTaskGetTickCount (void)

7.17.1.74 **BaseType_t** MPU_xTaskIncrementTick (void)

7.17.1.75 **BaseType_t** MPU_xTaskNotifyStateClear (**TaskHandle_t** *xTask*)

7.17.1.76 **BaseType_t** MPU_xTaskNotifyWait (uint32_t *ulBitsToClearOnEntry,* uint32_t *ulBitsToClearOnExit,* uint32_t ∗ *pulNotificationValue,* **TickType_t** *xTicksToWait*)

7.17.1.77 **BaseType_t** MPU_xTaskResumeAll (void)

7.17.1.78 **TimerHandle_t** MPU_xTimerCreate (const char ∗const *pcTimerName,* const **TickType_t** *xTimerPeriodInTicks,* const **UBaseType_t** *uxAutoReload,* void ∗const *pvTimerID,* **TimerCallbackFunction_t** *pxCallbackFunction*)

7.17.1.79 **TimerHandle_t** MPU_xTimerCreateStatic (const char ∗const *pcTimerName,* const **TickType_t** *xTimerPeriodInTicks,* const **UBaseType_t** *uxAutoReload,* void ∗const *pvTimerID,* **TimerCallbackFunction_t** *pxCallbackFunction,* **StaticTimer_t** ∗ *pxTimerBuffer*)

7.17.1.80 **BaseType_t** MPU_xTimerCreateTimerTask (void)

7.17.1.81 **BaseType_t** MPU_xTimerGenericCommand (**TimerHandle_t** *xTimer,* const **BaseType_t** *xCommandID,* const **TickType_t** *xOptionalValue,* **BaseType_t** ∗const *pxHigherPriorityTaskWoken,* const **TickType_t** *xTicksToWait*)

7.17.1.82 **TickType_t** MPU_xTimerGetExpiryTime (**TimerHandle_t** *xTimer*)

7.17.1.83 **TickType_t** MPU_xTimerGetPeriod (**TimerHandle_t** *xTimer*)

7.17.1.84 **TaskHandle_t** MPU_xTimerGetTimerDaemonTaskHandle (void)

7.17.1.85 **BaseType_t** MPU_xTimerIsTimerActive (**TimerHandle_t** *xTimer*)

7.17.1.86 BaseType_t MPU_xTimerPendFunctionCall (PendedFunction_t *xFunctionToPend*, void * *pvParameter1*, uint32_t *ulParameter2*, TickType_t *xTicksToWait*)

7.18 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/FreeRTOS_Source/include/mpu_wrappers.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define PRIVILEGED_FUNCTION
- #define PRIVILEGED_DATA
- #define portUSING_MPU_WRAPPERS 0

7.18.1 Macro Definition Documentation

7.18.1.1 #define portUSING_MPU_WRAPPERS 0

Definition at line 195 of file mpu_wrappers.h.

7.18.1.2 #define PRIVILEGED_DATA

Definition at line 194 of file mpu_wrappers.h.

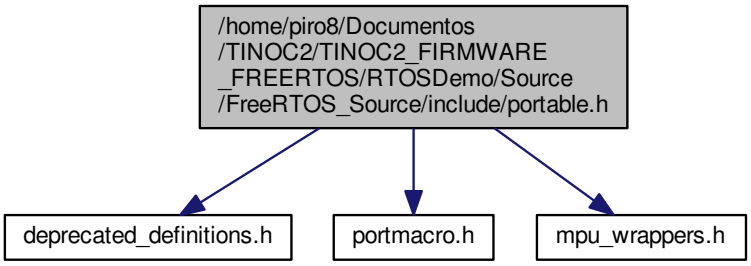
7.18.1.3 #define PRIVILEGED_FUNCTION

Definition at line 193 of file mpu_wrappers.h.

7.19 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/include/portable.h File Reference

```
#include "deprecated_definitions.h"
#include "portmacro.h"
#include "mpu_wrappers.h"
```

Include dependency graph for portable.h:



This graph shows which files directly or indirectly include this file:

Classes

- struct HeapRegion

Macros

- #define portNUM_CONFIGURABLE_REGIONS 1

Typedefs

- typedef struct HeapRegion HeapRegion_t

Functions

- StackType_t * pxPortInitialiseStack (StackType_t *pxTopOfStack, TaskFunction_t pxCode, void *pvParameters) PRIVILEGED_FUNCTION
- void vPortDefineHeapRegions (const HeapRegion_t *const pxHeapRegions) PRIVILEGED_FUNCTION
- void * pvPortMalloc (size_t xSize) PRIVILEGED_FUNCTION
- void vPortFree (void *pv) PRIVILEGED_FUNCTION
- void vPortInitialiseBlocks (void) PRIVILEGED_FUNCTION
- size_t xPortGetFreeHeapSize (void) PRIVILEGED_FUNCTION
- size_t xPortGetMinimumEverFreeHeapSize (void) PRIVILEGED_FUNCTION
- BaseType_t xPortStartScheduler (void) PRIVILEGED_FUNCTION
- void vPortEndScheduler (void) PRIVILEGED_FUNCTION

7.19.1 Macro Definition Documentation

7.19.1.1 #define portNUM_CONFIGURABLE_REGIONS 1

Definition at line 126 of file portable.h.

7.19.2 Typedef Documentation

7.19.2.1 typedef struct HeapRegion HeapRegion_t

7.19.3 Function Documentation

7.19.3.1 void∗ pvPortMalloc (size_t *xSize*)

Definition at line 111 of file heap_1.c.

7.19.3.2 StackType_t∗ pxPortInitialiseStack (StackType_t ∗ *pxTopOfStack,* TaskFunction_t *pxCode,* void ∗ *pvParameters*)

7.19.3.3 void vPortDefineHeapRegions (const HeapRegion_t ∗const *pxHeapRegions*)

7.19.3.4 void vPortEndScheduler (void)

Definition at line 230 of file port.c.

7.19.3.5 void vPortFree (void ∗ *pv*)

Definition at line 163 of file heap_1.c.

7.19.3.6 void vPortInitialiseBlocks (void)

Definition at line 175 of file heap_1.c.

7.19.3.7 size_t xPortGetFreeHeapSize (void)

Definition at line 182 of file heap_1.c.

7.19.3.8 size_t xPortGetMinimumEverFreeHeapSize (void)

7.19.3.9 BaseType_t xPortStartScheduler (void)

Definition at line 203 of file port.c.

7.20 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/include/projdefs.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define pdMS_TO_TICKS(xTimeInMs) ((TickType_t) (((TickType_t) (xTimeInMs) * (TickType_t) configTICK_RATE_HZ) / (TickType_t) 1000))
- #define pdFALSE ((BaseType_t) 0)
- #define pdTRUE ((BaseType_t) 1)
- #define pdPASS (pdTRUE)
- #define pdFAIL (pdFALSE)
- #define errQUEUE_EMPTY ((BaseType_t) 0)
- #define errQUEUE_FULL ((BaseType_t) 0)
- #define errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY (-1)
- #define errQUEUE_BLOCKED (-4)
- #define errQUEUE_YIELD (-5)
- #define configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES 0
- #define pdINTEGRITY_CHECK_VALUE 0x5a5a5a5aUL
- #define pdFREERTOS_ERRNO_NONE 0 /* No errors */
- #define pdFREERTOS_ERRNO_ENOENT 2 /* No such file or directory */
- #define pdFREERTOS_ERRNO_EINTR 4 /* Interrupted system call */
- #define pdFREERTOS_ERRNO_EIO 5 /* I/O error */
- #define pdFREERTOS_ERRNO_ENXIO 6 /* No such device or address */
- #define pdFREERTOS_ERRNO_EBADF 9 /* Bad file number */
- #define pdFREERTOS_ERRNO_EAGAIN 11 /* No more processes */
- #define pdFREERTOS_ERRNO_EWOULDBLOCK 11 /* Operation would block */
- #define pdFREERTOS_ERRNO_ENOMEM 12 /* Not enough memory */
- #define pdFREERTOS_ERRNO_EACCES 13 /* Permission denied */
- #define pdFREERTOS_ERRNO_EFAULT 14 /* Bad address */
- #define pdFREERTOS_ERRNO_EBUSY 16 /* Mount device busy */
- #define pdFREERTOS_ERRNO_EEXIST 17 /* File exists */
- #define pdFREERTOS_ERRNO_EXDEV 18 /* Cross-device link */
- #define pdFREERTOS_ERRNO_ENODEV 19 /* No such device */
- #define pdFREERTOS_ERRNO_ENOTDIR 20 /* Not a directory */
- #define pdFREERTOS_ERRNO_EISDIR 21 /* Is a directory */
- #define pdFREERTOS_ERRNO_EINVAL 22 /* Invalid argument */
- #define pdFREERTOS_ERRNO_ENOSPC 28 /* No space left on device */
- #define pdFREERTOS_ERRNO_ESPIPE 29 /* Illegal seek */
- #define pdFREERTOS_ERRNO_EROFS 30 /* Read only file system */
- #define pdFREERTOS_ERRNO_EUNATCH 42 /* Protocol driver not attached */
- #define pdFREERTOS_ERRNO_EBADE 50 /* Invalid exchange */
- #define pdFREERTOS_ERRNO_EFTYPE 79 /* Inappropriate file type or format */
- #define pdFREERTOS_ERRNO_ENMFILE 89 /* No more files */
- #define pdFREERTOS_ERRNO_ENOTEMPTY 90 /* Directory not empty */
- #define pdFREERTOS_ERRNO_ENAMETOOLONG 91 /* File or path name too long */
- #define pdFREERTOS_ERRNO_EOPNOTSUPP 95 /* Operation not supported on transport endpoint */

- #define pdFREERTOS_ERRNO_ENOBUFS 105 /* No buffer space available */
- #define pdFREERTOS_ERRNO_ENOPROTOOPT 109 /* Protocol not available */
- #define pdFREERTOS_ERRNO_EADDRINUSE 112 /* Address already in use */
- #define pdFREERTOS_ERRNO_ETIMEDOUT 116 /* Connection timed out */
- #define pdFREERTOS_ERRNO_EINPROGRESS 119 /* Connection already in progress */
- #define pdFREERTOS_ERRNO_EALREADY 120 /* Socket already connected */
- #define pdFREERTOS_ERRNO_EADDRNOTAVAIL 125 /* Address not available */
- #define pdFREERTOS_ERRNO_EISCONN 127 /* Socket is already connected */
- #define pdFREERTOS_ERRNO_ENOTCONN 128 /* Socket is not connected */
- #define pdFREERTOS_ERRNO_ENOMEDIUM 135 /* No medium inserted */
- #define pdFREERTOS_ERRNO_EILSEQ 138 /* An invalid UTF-16 sequence was encountered. */
- #define pdFREERTOS_ERRNO_ECANCELED 140 /* Operation canceled. */
- #define pdFREERTOS_LITTLE_ENDIAN 0
- #define pdFREERTOS_BIG_ENDIAN 1

Typedefs

- typedef void(* TaskFunction_t) (void *)

7.20.1 Macro Definition Documentation

7.20.1.1 #define configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES 0

Definition at line 101 of file projdefs.h.

7.20.1.2 #define errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY (-1)

Definition at line 95 of file projdefs.h.

7.20.1.3 #define errQUEUE_BLOCKED (-4)

Definition at line 96 of file projdefs.h.

7.20.1.4 #define errQUEUE_EMPTY ((BaseType_t) 0)

Definition at line 91 of file projdefs.h.

7.20.1.5 #define errQUEUE_FULL ((BaseType_t) 0)

Definition at line 92 of file projdefs.h.

7.20.1.6 #define errQUEUE_YIELD (-5)

Definition at line 97 of file projdefs.h.

7.20.1.7 #define pdFAIL (pdFALSE)

Definition at line 90 of file projdefs.h.

7.20.1.8 #define pdFALSE ((BaseType_t) 0)

Definition at line 86 of file projdefs.h.

7.20.1.9 #define pdFREERTOS_BIG_ENDIAN 1

Definition at line 156 of file projdefs.h.

7.20.1.10 #define pdFREERTOS_ERRNO_EACCES 13 /* Permission denied */

Definition at line 121 of file projdefs.h.

7.20.1.11 #define pdFREERTOS_ERRNO_EADDRINUSE 112 /* Address already in use */

Definition at line 142 of file projdefs.h.

7.20.1.12 #define pdFREERTOS_ERRNO_EADDRNOTAVAIL 125 /* Address not available */

Definition at line 146 of file projdefs.h.

7.20.1.13 #define pdFREERTOS_ERRNO_EAGAIN 11 /* No more processes */

Definition at line 118 of file projdefs.h.

7.20.1.14 #define pdFREERTOS_ERRNO_EALREADY 120 /* Socket already connected */

Definition at line 145 of file projdefs.h.

7.20.1.15 #define pdFREERTOS_ERRNO_EBADE 50 /* Invalid exchange */

Definition at line 134 of file projdefs.h.

7.20.1.16 #define pdFREERTOS_ERRNO_EBADF 9 /* Bad file number */

Definition at line 117 of file projdefs.h.

7.20.1.17 #define pdFREERTOS_ERRNO_EBUSY 16 /* Mount device busy */

Definition at line 123 of file projdefs.h.

7.20.1.18 #define pdFREERTOS_ERRNO_ECANCELED 140 /* Operation canceled. */

Definition at line 151 of file projdefs.h.

7.20.1.19 #define pdFREERTOS_ERRNO_EEXIST 17 /* File exists */

Definition at line 124 of file projdefs.h.

7.20.1.20 #define pdFREERTOS_ERRNO_EFAULT 14 /* Bad address */

Definition at line 122 of file projdefs.h.

7.20.1.21 #define pdFREERTOS_ERRNO_EFTYPE 79 /* Inappropriate file type or format */

Definition at line 135 of file projdefs.h.

7.20.1.22 #define pdFREERTOS_ERRNO_EILSEQ 138 /* An invalid UTF-16 sequence was encountered. */

Definition at line 150 of file projdefs.h.

7.20.1.23 #define pdFREERTOS_ERRNO_EINPROGRESS 119 /* Connection already in progress */

Definition at line 144 of file projdefs.h.

7.20.1.24 #define pdFREERTOS_ERRNO_EINTR 4 /* Interrupted system call */

Definition at line 114 of file projdefs.h.

7.20.1.25 #define pdFREERTOS_ERRNO_EINVAL 22 /* Invalid argument */

Definition at line 129 of file projdefs.h.

7.20.1.26 #define pdFREERTOS_ERRNO_EIO 5 /* I/O error */

Definition at line 115 of file projdefs.h.

7.20.1.27 #define pdFREERTOS_ERRNO_EISCONN 127 /* Socket is already connected */

Definition at line 147 of file projdefs.h.

7.20.1.28 #define pdFREERTOS_ERRNO_EISDIR 21 /* Is a directory */

Definition at line 128 of file projdefs.h.

7.20.1.29 #define pdFREERTOS_ERRNO_ENAMETOOLONG 91 /* File or path name too long */

Definition at line 138 of file projdefs.h.

7.20.1.30 #define pdFREERTOS_ERRNO_ENMFILE 89 /* No more files */

Definition at line 136 of file projdefs.h.

7.20.1.31 #define pdFREERTOS_ERRNO_ENOBUFS 105 /* No buffer space available */

Definition at line 140 of file projdefs.h.

7.20.1.32 #define pdFREERTOS_ERRNO_ENODEV 19 /* No such device */

Definition at line 126 of file projdefs.h.

7.20.1.33 #define pdFREERTOS_ERRNO_ENOENT 2 /* No such file or directory */

Definition at line 113 of file projdefs.h.

7.20.1.34 #define pdFREERTOS_ERRNO_ENOMEDIUM 135 /* No medium inserted */

Definition at line 149 of file projdefs.h.

7.20.1.35 #define pdFREERTOS_ERRNO_ENOMEM 12 /* Not enough memory */

Definition at line 120 of file projdefs.h.

7.20.1.36 #define pdFREERTOS_ERRNO_ENOPROTOOPT 109 /* Protocol not available */

Definition at line 141 of file projdefs.h.

7.20.1.37 #define pdFREERTOS_ERRNO_ENOSPC 28 /* No space left on device */

Definition at line 130 of file projdefs.h.

7.20.1.38 #define pdFREERTOS_ERRNO_ENOTCONN 128 /* Socket is not connected */

Definition at line 148 of file projdefs.h.

7.20.1.39 #define pdFREERTOS_ERRNO_ENOTDIR 20 /* Not a directory */

Definition at line 127 of file projdefs.h.

7.20.1.40 #define pdFREERTOS_ERRNO_ENOTEMPTY 90 /* Directory not empty */

Definition at line 137 of file projdefs.h.

7.20.1.41 #define pdFREERTOS_ERRNO_ENXIO 6 /* No such device or address */

Definition at line 116 of file projdefs.h.

7.20.1.42 #define pdFREERTOS_ERRNO_EOPNOTSUPP 95 /* Operation not supported on transport endpoint */

Definition at line 139 of file projdefs.h.

7.20.1.43 #define pdFREERTOS_ERRNO_EROFS 30 /* Read only file system */

Definition at line 132 of file projdefs.h.

7.20.1.44 #define pdFREERTOS_ERRNO_ESPIPE 29 /* Illegal seek */

Definition at line 131 of file projdefs.h.

7.20.1.45 #define pdFREERTOS_ERRNO_ETIMEDOUT 116 /* Connection timed out */

Definition at line 143 of file projdefs.h.

7.20.1.46 #define pdFREERTOS_ERRNO_EUNATCH 42 /* Protocol driver not attached */

Definition at line 133 of file projdefs.h.

7.20.1.47 #define pdFREERTOS_ERRNO_EWOULDBLOCK 11 /* Operation would block */

Definition at line 119 of file projdefs.h.

7.20.1.48 #define pdFREERTOS_ERRNO_EXDEV 18 /* Cross-device link */

Definition at line 125 of file projdefs.h.

7.20.1.49 #define pdFREERTOS_ERRNO_NONE 0 /* No errors */

Definition at line 112 of file projdefs.h.

7.20.1.50 #define pdFREERTOS_LITTLE_ENDIAN 0

Definition at line 155 of file projdefs.h.

7.20.1.51 #define pdINTEGRITY_CHECK_VALUE 0x5a5a5a5aUL

Definition at line 107 of file projdefs.h.

7.20.1.52 #define pdMS_TO_TICKS(*xTimeInMs*) ((TickType_t) (((TickType_t) (xTimeInMs) * (TickType_t) configTICK_RATE_HZ) / (TickType_t) 1000))

Definition at line 83 of file projdefs.h.

7.20.1.53 #define pdPASS (pdTRUE)

Definition at line 89 of file projdefs.h.

7.20.1.54 #define pdTRUE ((BaseType_t) 1)

Definition at line 87 of file projdefs.h.

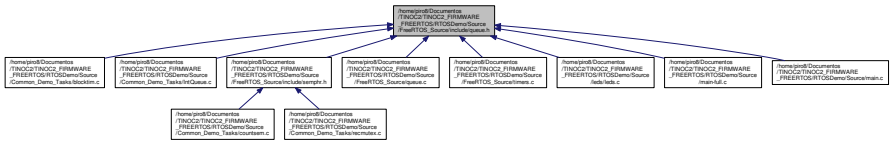
7.20.2 Typedef Documentation

7.20.2.1 typedef void(* TaskFunction_t) (void *)

Definition at line 77 of file projdefs.h.

7.21 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/FreeRTOS_Source/include/queue.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- #define queueSEND_TO_BACK ((BaseType_t) 0)
- #define queueSEND_TO_FRONT ((BaseType_t) 1)
- #define queueOVERWRITE ((BaseType_t) 2)
- #define queueQUEUE_TYPE_BASE ((uint8_t) 0U)
- #define queueQUEUE_TYPE_SET ((uint8_t) 0U)
- #define queueQUEUE_TYPE_MUTEX ((uint8_t) 1U)
- #define queueQUEUE_TYPE_COUNTING_SEMAPHORE ((uint8_t) 2U)
- #define queueQUEUE_TYPE_BINARY_SEMAPHORE ((uint8_t) 3U)
- #define queueQUEUE_TYPE_RECURSIVE_MUTEX ((uint8_t) 4U)
- #define xQueueSendToFront(xQueue, pvItemToQueue, xTicksToWait) xQueueGenericSend((xQueue), (pvItemToQueue), (xTicksToWait), queueSEND_TO_FRONT)
- #define xQueueSendToBack(xQueue, pvItemToQueue, xTicksToWait) xQueueGenericSend((xQueue), (pvItemToQueue), (xTicksToWait), queueSEND_TO_BACK)
- #define xQueueSend(xQueue, pvItemToQueue, xTicksToWait) xQueueGenericSend((xQueue), (pvItem↩ ToQueue), (xTicksToWait), queueSEND_TO_BACK)
- #define xQueueOverwrite(xQueue, pvItemToQueue) xQueueGenericSend((xQueue), (pvItemToQueue), 0, queueOVERWRITE)
- #define xQueuePeek(xQueue, pvBuffer, xTicksToWait) xQueueGenericReceive((xQueue), (pvBuffer), (xTicksToWait), pdTRUE)
- #define xQueueReceive(xQueue, pvBuffer, xTicksToWait) xQueueGenericReceive((xQueue), (pvBuffer), (xTicksToWait), pdFALSE)
- #define xQueueSendToFrontFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken) xQueue↩ GenericSendFromISR((xQueue), (pvItemToQueue), (pxHigherPriorityTaskWoken), queueSEND_TO_↩ FRONT)
- #define xQueueSendToBackFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken) xQueue↩ GenericSendFromISR((xQueue), (pvItemToQueue), (pxHigherPriorityTaskWoken), queueSEND_TO_↩ BACK)
- #define xQueueOverwriteFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken) xQueue↩ GenericSendFromISR((xQueue), (pvItemToQueue), (pxHigherPriorityTaskWoken), queueOVERWRITE)
- #define xQueueSendFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken) xQueueGeneric↩ SendFromISR((xQueue), (pvItemToQueue), (pxHigherPriorityTaskWoken), queueSEND_TO_BACK)
- #define xQueueReset(xQueue) xQueueGenericReset(xQueue, pdFALSE)

Typedefs

- typedef void * QueueHandle_t
- typedef void * QueueSetHandle_t
- typedef void * QueueSetMemberHandle_t

Functions

- BaseType_t xQueueGenericSend (QueueHandle_t xQueue, const void ∗const pvItemToQueue, TickType_t xTicksToWait, const BaseType_t xCopyPosition) PRIVILEGED_FUNCTION
- BaseType_t xQueuePeekFromISR (QueueHandle_t xQueue, void ∗const pvBuffer) PRIVILEGED_FUNCTION
- BaseType_t xQueueGenericReceive (QueueHandle_t xQueue, void ∗const pvBuffer, TickType_t xTicksToWait, const BaseType_t xJustPeek) PRIVILEGED_FUNCTION
- UBaseType_t uxQueueMessagesWaiting (const QueueHandle_t xQueue) PRIVILEGED_FUNCTION
- UBaseType_t uxQueueSpacesAvailable (const QueueHandle_t xQueue) PRIVILEGED_FUNCTION
- void vQueueDelete (QueueHandle_t xQueue) PRIVILEGED_FUNCTION
- BaseType_t xQueueGenericSendFromISR (QueueHandle_t xQueue, const void ∗const pvItemToQueue, BaseType_t ∗const pxHigherPriorityTaskWoken, const BaseType_t xCopyPosition) PRIVILEGED_FUNCTION
- BaseType_t xQueueGiveFromISR (QueueHandle_t xQueue, BaseType_t ∗const pxHigherPriorityTaskWoken) PRIVILEGED_FUNCTION
- BaseType_t xQueueReceiveFromISR (QueueHandle_t xQueue, void ∗const pvBuffer, BaseType_t ∗const pxHigherPriorityTaskWoken) PRIVILEGED_FUNCTION
- BaseType_t xQueueIsQueueEmptyFromISR (const QueueHandle_t xQueue) PRIVILEGED_FUNCTION
- BaseType_t xQueueIsQueueFullFromISR (const QueueHandle_t xQueue) PRIVILEGED_FUNCTION
- UBaseType_t uxQueueMessagesWaitingFromISR (const QueueHandle_t xQueue) PRIVILEGED_FUNCTION
- BaseType_t xQueueCRSendFromISR (QueueHandle_t xQueue, const void ∗pvItemToQueue, BaseType_t xCoRoutinePreviouslyWoken)
- BaseType_t xQueueCRReceiveFromISR (QueueHandle_t xQueue, void ∗pvBuffer, BaseType_t ∗pxTaskWoken)
- BaseType_t xQueueCRSend (QueueHandle_t xQueue, const void ∗pvItemToQueue, TickType_t xTicksToWait)
- BaseType_t xQueueCRReceive (QueueHandle_t xQueue, void ∗pvBuffer, TickType_t xTicksToWait)
- QueueHandle_t xQueueCreateMutex (const uint8_t ucQueueType) PRIVILEGED_FUNCTION
- QueueHandle_t xQueueCreateMutexStatic (const uint8_t ucQueueType, StaticQueue_t ∗pxStaticQueue) PRIVILEGED_FUNCTION
- QueueHandle_t xQueueCreateCountingSemaphore (const UBaseType_t uxMaxCount, const UBaseType_t uxInitialCount) PRIVILEGED_FUNCTION
- QueueHandle_t xQueueCreateCountingSemaphoreStatic (const UBaseType_t uxMaxCount, const UBaseType_t uxInitialCount, StaticQueue_t ∗pxStaticQueue) PRIVILEGED_FUNCTION
- void ∗ xQueueGetMutexHolder (QueueHandle_t xSemaphore) PRIVILEGED_FUNCTION
- BaseType_t xQueueTakeMutexRecursive (QueueHandle_t xMutex, TickType_t xTicksToWait) PRIVILEGED_FUNCTION
- BaseType_t xQueueGiveMutexRecursive (QueueHandle_t pxMutex) PRIVILEGED_FUNCTION
- QueueSetHandle_t xQueueCreateSet (const UBaseType_t uxEventQueueLength) PRIVILEGED_FUNCTION
- BaseType_t xQueueAddToSet (QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet) PRIVILEGED_FUNCTION
- BaseType_t xQueueRemoveFromSet (QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet) PRIVILEGED_FUNCTION
- QueueSetMemberHandle_t xQueueSelectFromSet (QueueSetHandle_t xQueueSet, const TickType_t xTicksToWait) PRIVILEGED_FUNCTION
- QueueSetMemberHandle_t xQueueSelectFromSetFromISR (QueueSetHandle_t xQueueSet) PRIVILEGED_FUNCTION
- void vQueueWaitForMessageRestricted (QueueHandle_t xQueue, TickType_t xTicksToWait, const BaseType_t xWaitIndefinitely) PRIVILEGED_FUNCTION
- BaseType_t xQueueGenericReset (QueueHandle_t xQueue, BaseType_t xNewQueue) PRIVILEGED_FUNCTION
- void vQueueSetQueueNumber (QueueHandle_t xQueue, UBaseType_t uxQueueNumber) PRIVILEGED_FUNCTION
- UBaseType_t uxQueueGetQueueNumber (QueueHandle_t xQueue) PRIVILEGED_FUNCTION
- uint8_t ucQueueGetQueueType (QueueHandle_t xQueue) PRIVILEGED_FUNCTION

7.21.1 Macro Definition Documentation

7.21.1.1 #define queueOVERWRITE ((BaseType_t) 2)

Definition at line 107 of file queue.h.

7.21.1.2 #define queueQUEUE_TYPE_BASE ((uint8_t) 0U)

Definition at line 110 of file queue.h.

7.21.1.3 #define queueQUEUE_TYPE_BINARY_SEMAPHORE ((uint8_t) 3U)

Definition at line 114 of file queue.h.

7.21.1.4 #define queueQUEUE_TYPE_COUNTING_SEMAPHORE ((uint8_t) 2U)

Definition at line 113 of file queue.h.

7.21.1.5 #define queueQUEUE_TYPE_MUTEX ((uint8_t) 1U)

Definition at line 112 of file queue.h.

7.21.1.6 #define queueQUEUE_TYPE_RECURSIVE_MUTEX ((uint8_t) 4U)

Definition at line 115 of file queue.h.

7.21.1.7 #define queueQUEUE_TYPE_SET ((uint8_t) 0U)

Definition at line 111 of file queue.h.

7.21.1.8 #define queueSEND_TO_BACK ((BaseType_t) 0)

Definition at line 105 of file queue.h.

7.21.1.9 #define queueSEND_TO_FRONT ((BaseType_t) 1)

Definition at line 106 of file queue.h.

7.21.1.10 #define xQueueOverwrite(xQueue, *pvItemToQueue*) xQueueGenericSend((xQueue), (pvItemToQueue), 0, queueOVERWRITE)

Definition at line 604 of file queue.h.

7.21.1.11 #define xQueueOverwriteFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken) xQueueGenericSendFromISR((xQueue), (pvItemToQueue), (pxHigherPriorityTaskWoken), queueOVERWRITE)

Definition at line 1287 of file queue.h.

7.21.1.12 #define xQueuePeek(xQueue, pvBuffer, xTicksToWait) xQueueGenericReceive((xQueue), (pvBuffer), (xTicksToWait), pdTRUE)

Definition at line 788 of file queue.h.

7.21.1.13 #define xQueueReceive(xQueue, pvBuffer, xTicksToWait) xQueueGenericReceive((xQueue), (pvBuffer), (xTicksToWait), pdFALSE)

Definition at line 914 of file queue.h.

7.21.1.14 #define xQueueReset(xQueue) xQueueGenericReset(xQueue, pdFALSE)

Definition at line 1576 of file queue.h.

7.21.1.15 #define xQueueSend(xQueue, pvItemToQueue, xTicksToWait) xQueueGenericSend((xQueue), (pvItemToQueue), (xTicksToWait), queueSEND_TO_BACK)

Definition at line 521 of file queue.h.

7.21.1.16 #define xQueueSendFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken) xQueueGenericSendFromISR((xQueue), (pvItemToQueue), (pxHigherPriorityTaskWoken), queueSEND_TO_BACK)

Definition at line 1361 of file queue.h.

7.21.1.17 #define xQueueSendToBack(xQueue, pvItemToQueue, xTicksToWait) xQueueGenericSend((xQueue), (pvItemToQueue), (xTicksToWait), queueSEND_TO_BACK)

Definition at line 437 of file queue.h.

7.21.1.18 #define xQueueSendToBackFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken) xQueueGenericSendFromISR((xQueue), (pvItemToQueue), (pxHigherPriorityTaskWoken), queueSEND_TO_BACK)

Definition at line 1200 of file queue.h.

7.21.1.19 #define xQueueSendToFront(xQueue, pvItemToQueue, xTicksToWait) xQueueGenericSend((xQueue), (pvItemToQueue), (xTicksToWait), queueSEND_TO_FRONT)

Definition at line 355 of file queue.h.

7.21.1.20 #define xQueueSendToFrontFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken) xQueueGenericSendFromISR((xQueue), (pvItemToQueue), (pxHigherPriorityTaskWoken), queueSEND_TO_FRONT)

Definition at line 1129 of file queue.h.

7.21.2 Typedef Documentation

7.21.2.1 typedef void* QueueHandle_t

Type by which queues are referenced. For example, a call to xQueueCreate() returns an QueueHandle_t variable that can then be used as a parameter to xQueueSend(), xQueueReceive(), etc.

Definition at line 88 of file queue.h.

7.21.2.2 typedef void* QueueSetHandle_t

Type by which queue sets are referenced. For example, a call to xQueueCreateSet() returns an xQueueSet variable that can then be used as a parameter to xQueueSelectFromSet(), xQueueAddToSet(), etc.

Definition at line 95 of file queue.h.

7.21.2.3 typedef void* QueueSetMemberHandle_t

Queue sets can contain both queues and semaphores, so the QueueSetMemberHandle_t is defined as a type to be used where a parameter or return value can be either an QueueHandle_t or an SemaphoreHandle_t.

Definition at line 102 of file queue.h.

7.21.3 Function Documentation

7.21.3.1 uint8_t ucQueueGetQueueType (QueueHandle_t *xQueue*)

7.21.3.2 UBaseType_t uxQueueGetQueueNumber (QueueHandle_t *xQueue*)

7.21.3.3 UBaseType_t uxQueueMessagesWaiting (const QueueHandle_t *xQueue*)

Definition at line 1579 of file queue.c.

7.21.3.4 **UBaseType_t** uxQueueMessagesWaitingFromISR (const **QueueHandle_t** *xQueue*)

Definition at line 1613 of file queue.c.

7.21.3.5 **UBaseType_t** uxQueueSpacesAvailable (const **QueueHandle_t** *xQueue*)

Definition at line 1595 of file queue.c.

7.21.3.6 void vQueueDelete (**QueueHandle_t** *xQueue*)

Definition at line 1625 of file queue.c.

7.21.3.7 void vQueueSetQueueNumber (**QueueHandle_t** *xQueue,* **UBaseType_t** *uxQueueNumber*)

7.21.3.8 void vQueueWaitForMessageRestricted (**QueueHandle_t** *xQueue,* **TickType_t** *xTicksToWait,* const **BaseType_t** *xWaitIndefinitely*)

7.21.3.9 **BaseType_t** xQueueAddToSet (**QueueSetMemberHandle_t** *xQueueOrSemaphore,* **QueueSetHandle_t** *xQueueSet*)

7.21.3.10 **QueueHandle_t** xQueueCreateCountingSemaphore (const **UBaseType_t** *uxMaxCount,* const **UBaseType_t** *uxInitialCount*)

7.21.3.11 **QueueHandle_t** xQueueCreateCountingSemaphoreStatic (const **UBaseType_t** *uxMaxCount,* const **UBaseType_t** *uxInitialCount,* **StaticQueue_t** ∗ *pxStaticQueue*)

7.21.3.12 **QueueHandle_t** xQueueCreateMutex (const uint8_t *ucQueueType*)

7.21.3.13 **QueueHandle_t** xQueueCreateMutexStatic (const uint8_t *ucQueueType,* **StaticQueue_t** ∗ *pxStaticQueue*)

7.21.3.14 **QueueSetHandle_t** xQueueCreateSet (const **UBaseType_t** *uxEventQueueLength*)

7.21.3.15 **BaseType_t** xQueueCRReceive (**QueueHandle_t** *xQueue,* void ∗ *pvBuffer,* **TickType_t** *xTicksToWait*)

7.21.3.16 **BaseType_t** xQueueCRReceiveFromISR (**QueueHandle_t** *xQueue,* void ∗ *pvBuffer,* **BaseType_t** ∗ *pxTaskWoken*)

7.21.3.17 **BaseType_t** xQueueCRSend (**QueueHandle_t** *xQueue,* const void ∗ *pvItemToQueue,* **TickType_t** *xTicksToWait*)

7.21.3.18 **BaseType_t** xQueueCRSendFromISR (**QueueHandle_t** *xQueue,* const void ∗ *pvItemToQueue,* **BaseType_t** *xCoRoutinePreviouslyWoken*)

7.21.3.19 **BaseType_t** xQueueGenericReceive (**QueueHandle_t** *xQueue,* void ∗const *pvBuffer,* **TickType_t** *xTicksToWait,* const **BaseType_t** *xJustPeek*)

Definition at line 1237 of file queue.c.

7.21.3.20 **BaseType_t xQueueGenericReset (QueueHandle_t *xQueue,* BaseType_t *xNewQueue*)**

Definition at line 279 of file queue.c.

7.21.3.21 **BaseType_t xQueueGenericSend (QueueHandle_t *xQueue,* const void ∗const *pvItemToQueue,* TickType_t *xTicksToWait,* const BaseType_t *xCopyPosition*)**

Definition at line 723 of file queue.c.

7.21.3.22 **BaseType_t xQueueGenericSendFromISR (QueueHandle_t *xQueue,* const void ∗const *pvItemToQueue,* BaseType_t ∗const *pxHigherPriorityTaskWoken,* const BaseType_t *xCopyPosition*)**

Definition at line 921 of file queue.c.

7.21.3.23 **void∗ xQueueGetMutexHolder (QueueHandle_t *xSemaphore*)**

7.21.3.24 **BaseType_t xQueueGiveFromISR (QueueHandle_t *xQueue,* BaseType_t ∗const *pxHigherPriorityTaskWoken*)**

Definition at line 1072 of file queue.c.

7.21.3.25 **BaseType_t xQueueGiveMutexRecursive (QueueHandle_t *pxMutex*)**

7.21.3.26 **BaseType_t xQueueIsQueueEmptyFromISR (const QueueHandle_t *xQueue*)**

Definition at line 1935 of file queue.c.

7.21.3.27 **BaseType_t xQueueIsQueueFullFromISR (const QueueHandle_t *xQueue*)**

Definition at line 1974 of file queue.c.

7.21.3.28 **BaseType_t xQueuePeekFromISR (QueueHandle_t *xQueue,* void ∗const *pvBuffer*)**

Definition at line 1525 of file queue.c.

7.21.3.29 **BaseType_t xQueueReceiveFromISR (QueueHandle_t *xQueue,* void ∗const *pvBuffer,* BaseType_t ∗const *pxHigherPriorityTaskWoken*)**

Definition at line 1434 of file queue.c.

7.21.3.30 BaseType_t xQueueRemoveFromSet (QueueSetMemberHandle_t *xQueueOrSemaphore,* QueueSetHandle_t *xQueueSet*)

7.21.3.31 QueueSetMemberHandle_t xQueueSelectFromSet (QueueSetHandle_t *xQueueSet,* const TickType_t *xTicksToWait*)

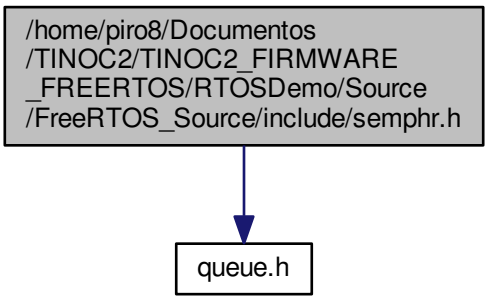
7.21.3.32 QueueSetMemberHandle_t xQueueSelectFromSetFromISR (QueueSetHandle_t *xQueueSet*)

7.21.3.33 BaseType_t xQueueTakeMutexRecursive (QueueHandle_t *xMutex,* TickType_t *xTicksToWait*)

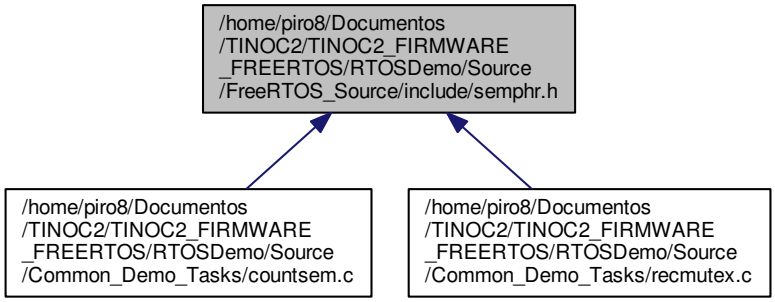
7.22 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/include/semphr.h File Reference

```
#include "queue.h"
```

Include dependency graph for semphr.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define semBINARY_SEMAPHORE_QUEUE_LENGTH ((uint8_t) 1U)
- #define semSEMAPHORE_QUEUE_ITEM_LENGTH ((uint8_t) 0U)
- #define semGIVE_BLOCK_TIME ((TickType_t) 0U)
- #define xSemaphoreTake(xSemaphore, xBlockTime) xQueueGenericReceive((QueueHandle_t) (x↩ Semaphore), NULL, (xBlockTime), pdFALSE)
- #define xSemaphoreGive(xSemaphore) xQueueGenericSend((QueueHandle_t) (xSemaphore), NULL, semGIVE_BLOCK_TIME, queueSEND_TO_BACK)
- #define xSemaphoreGiveFromISR(xSemaphore, pxHigherPriorityTaskWoken) xQueueGiveFromISR((QueueHandle_t) (xSemaphore), (pxHigherPriorityTaskWoken))
- #define xSemaphoreTakeFromISR(xSemaphore, pxHigherPriorityTaskWoken) xQueueReceiveFromISR((QueueHandle_t) (xSemaphore), NULL, (pxHigherPriorityTaskWoken))
- #define vSemaphoreDelete(xSemaphore) vQueueDelete((QueueHandle_t) (xSemaphore))
- #define xSemaphoreGetMutexHolder(xSemaphore) xQueueGetMutexHolder((xSemaphore))
- #define uxSemaphoreGetCount(xSemaphore) uxQueueMessagesWaiting((QueueHandle_t) (xSemaphore))

Typedefs

- typedef QueueHandle_t SemaphoreHandle_t

7.22.1 Macro Definition Documentation

7.22.1.1 #define semBINARY_SEMAPHORE_QUEUE_LENGTH ((uint8_t) 1U)

Definition at line 81 of file semphr.h.

7.22.1.2 #define semGIVE_BLOCK_TIME ((TickType_t) 0U)

Definition at line 83 of file semphr.h.

7.22.1.3 #define semSEMAPHORE_QUEUE_ITEM_LENGTH ((uint8_t) 0U)

Definition at line 82 of file semphr.h.

7.22.1.4 #define uxSemaphoreGetCount(*xSemaphore*) uxQueueMessagesWaiting((QueueHandle_t) (xSemaphore))

semphr.h

```
UBaseType_t uxSemaphoreGetCount( SemaphoreHandle_t xSemaphore );
```

If the semaphore is a counting semaphore then uxSemaphoreGetCount() returns its current count value. If the semaphore is a binary semaphore then uxSemaphoreGetCount() returns 1 if the semaphore is available, and 0 if the semaphore is not available.

Definition at line 1167 of file semphr.h.

7.22.1.5 #define vSemaphoreDelete(*xSemaphore*) vQueueDelete((QueueHandle_t) (xSemaphore))

Definition at line 1140 of file semphr.h.

7.22.1.6 #define xSemaphoreGetMutexHolder(*xSemaphore*) xQueueGetMutexHolder((xSemaphore))

semphr.h

```
TaskHandle_t xSemaphoreGetMutexHolder( SemaphoreHandle_t xMutex );
```

If xMutex is indeed a mutex type semaphore, return the current mutex holder. If xMutex is not a mutex type semaphore, or the mutex is available (not held by a task), return NULL.

Note: This is a good way of determining if the calling task is the mutex holder, but not a good way of determining the identity of the mutex holder as the holder may change between the function exiting and the returned value being tested.

Definition at line 1155 of file semphr.h.

7.22.1.7 #define xSemaphoreGive(*xSemaphore*) xQueueGenericSend((QueueHandle_t) (xSemaphore), NULL, semGIVE_BLOCK_TIME, queueSEND_TO_BACK)

Definition at line 489 of file semphr.h.

7.22.1.8 #define xSemaphoreGiveFromISR(*xSemaphore, pxHigherPriorityTaskWoken*) xQueueGiveFromISR((QueueHandle_t) (xSemaphore), (pxHigherPriorityTaskWoken))

Definition at line 666 of file semphr.h.

7.22.1.9 #define xSemaphoreTake(*xSemaphore, xBlockTime*) xQueueGenericReceive((QueueHandle_t) (xSemaphore), NULL, (xBlockTime), pdFALSE)

Definition at line 331 of file semphr.h.

7.22.1.10 #define xSemaphoreTakeFromISR(*xSemaphore, pxHigherPriorityTaskWoken*) xQueueReceiveFromISR((QueueHandle_t) (xSemaphore), NULL, (pxHigherPriorityTaskWoken))

semphr. h

```
xSemaphoreTakeFromISR(
                          SemaphoreHandle_t xSemaphore,
                          BaseType_t *pxHigherPriorityTaskWoken
                      )
```

Macro to take a semaphore from an ISR. The semaphore must have previously been created with a call to x↩ SemaphoreCreateBinary() or xSemaphoreCreateCounting().

Mutex type semaphores (those created using a call to xSemaphoreCreateMutex()) must not be used with this macro.

This macro can be used from an ISR, however taking a semaphore from an ISR is not a common operation. It is likely to only be useful when taking a counting semaphore when an interrupt is obtaining an object from a resource pool (when the semaphore count indicates the number of resources available).

Parameters

xSemaphore	A handle to the semaphore being taken. This is the handle returned when the semaphore was created.
pxHigherPriorityTaskWoken	xSemaphoreTakeFromISR() will set ∗pxHigherPriorityTaskWoken to pdTRUE if taking the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreTakeFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns

pdTRUE if the semaphore was successfully taken, otherwise pdFALSE

Definition at line 700 of file semphr.h.

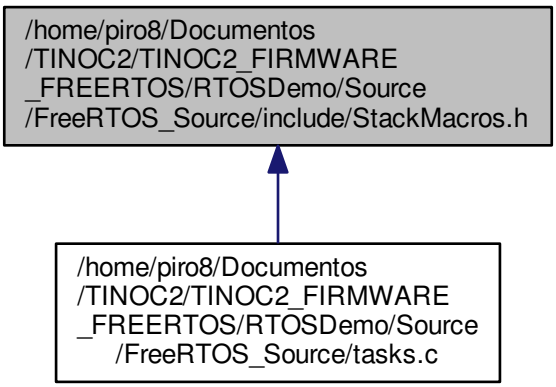
7.22.2 Typedef Documentation

7.22.2.1 typedef QueueHandle_t SemaphoreHandle_t

Definition at line 79 of file semphr.h.

7.23 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/FreeRTOS_Source/include/StackMacros.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- #define taskCHECK_FOR_STACK_OVERFLOW()

7.23.1 Macro Definition Documentation

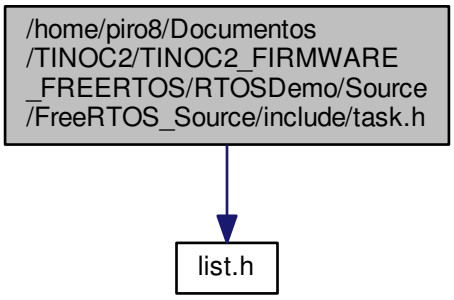
7.23.1.1 #define taskCHECK_FOR_STACK_OVERFLOW( )

Definition at line 165 of file StackMacros.h.

7.24 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/include/task.h File Reference

```
#include "list.h"
```

Include dependency graph for task.h:



This graph shows which files directly or indirectly include this file:

Classes

- struct xTIME_OUT
- struct xMEMORY_REGION
- struct xTASK_PARAMETERS
- struct xTASK_STATUS

Macros

- #define tskKERNEL_VERSION_NUMBER "V9.0.0"
- #define tskKERNEL_VERSION_MAJOR 9
- #define tskKERNEL_VERSION_MINOR 0
- #define tskKERNEL_VERSION_BUILD 0
- #define tskIDLE_PRIORITY ((UBaseType_t) 0U)
- #define taskYIELD() portYIELD()
- #define taskENTER_CRITICAL() portENTER_CRITICAL()
- #define taskENTER_CRITICAL_FROM_ISR() portSET_INTERRUPT_MASK_FROM_ISR()
- #define taskEXIT_CRITICAL() portEXIT_CRITICAL()
- #define taskEXIT_CRITICAL_FROM_ISR(x) portCLEAR_INTERRUPT_MASK_FROM_ISR(x)
- #define taskDISABLE_INTERRUPTS() portDISABLE_INTERRUPTS()
- #define taskENABLE_INTERRUPTS() portENABLE_INTERRUPTS()
- #define taskSCHEDULER_SUSPENDED ((BaseType_t) 0)
- #define taskSCHEDULER_NOT_STARTED ((BaseType_t) 1)
- #define taskSCHEDULER_RUNNING ((BaseType_t) 2)
- #define xTaskNotify(xTaskToNotify, ulValue, eAction) xTaskGenericNotify((xTaskToNotify), (ulValue), (e↩Action), NULL)
- #define xTaskNotifyAndQuery(xTaskToNotify, ulValue, eAction, pulPreviousNotifyValue) xTaskGenericNotify((xTaskToNotify), (ulValue), (eAction), (pulPreviousNotifyValue))
- #define xTaskNotifyFromISR(xTaskToNotify, ulValue, eAction, pxHigherPriorityTaskWoken) xTaskGeneric↩NotifyFromISR((xTaskToNotify), (ulValue), (eAction), NULL, (pxHigherPriorityTaskWoken))
- #define xTaskNotifyAndQueryFromISR(xTaskToNotify, ulValue, eAction, pulPreviousNotificationValue, px↩HigherPriorityTaskWoken) xTaskGenericNotifyFromISR((xTaskToNotify), (ulValue), (eAction), (pul↩PreviousNotificationValue), (pxHigherPriorityTaskWoken))
- #define xTaskNotifyGive(xTaskToNotify) xTaskGenericNotify((xTaskToNotify), (0), eIncrement, NULL)

Typedefs

- typedef void * TaskHandle_t
- typedef BaseType_t(* TaskHookFunction_t) (void *)
- typedef struct xTIME_OUT TimeOut_t
- typedef struct xMEMORY_REGION MemoryRegion_t
- typedef struct xTASK_PARAMETERS TaskParameters_t
- typedef struct xTASK_STATUS TaskStatus_t

Enumerations

- enum eTaskState {
  eRunning = 0, eReady, eBlocked, eSuspended,
  eDeleted, eInvalid }
- enum eNotifyAction {
  eNoAction = 0, eSetBits, eIncrement, eSetValueWithOverwrite,
  eSetValueWithoutOverwrite }
- enum eSleepModeStatus { eAbortSleep = 0, eStandardSleep, eNoTasksWaitingTimeout }

Functions

- void vTaskAllocateMPURegions (TaskHandle_t xTask, const MemoryRegion_t *const pxRegions) PRIVILEGED_FUNCTION
- void vTaskDelete (TaskHandle_t xTaskToDelete) PRIVILEGED_FUNCTION
- void vTaskDelay (const TickType_t xTicksToDelay) PRIVILEGED_FUNCTION
- void vTaskDelayUntil (TickType_t *const pxPreviousWakeTime, const TickType_t xTimeIncrement) PRIVILEGED_FUNCTION
- BaseType_t xTaskAbortDelay (TaskHandle_t xTask) PRIVILEGED_FUNCTION
- UBaseType_t uxTaskPriorityGet (TaskHandle_t xTask) PRIVILEGED_FUNCTION
- UBaseType_t uxTaskPriorityGetFromISR (TaskHandle_t xTask) PRIVILEGED_FUNCTION
- eTaskState eTaskGetState (TaskHandle_t xTask) PRIVILEGED_FUNCTION
- void vTaskGetInfo (TaskHandle_t xTask, TaskStatus_t *pxTaskStatus, BaseType_t xGetFreeStackSpace, eTaskState eState) PRIVILEGED_FUNCTION
- void vTaskPrioritySet (TaskHandle_t xTask, UBaseType_t uxNewPriority) PRIVILEGED_FUNCTION
- void vTaskSuspend (TaskHandle_t xTaskToSuspend) PRIVILEGED_FUNCTION
- void vTaskResume (TaskHandle_t xTaskToResume) PRIVILEGED_FUNCTION
- BaseType_t xTaskResumeFromISR (TaskHandle_t xTaskToResume) PRIVILEGED_FUNCTION
- void vTaskStartScheduler (void) PRIVILEGED_FUNCTION
- void vTaskEndScheduler (void) PRIVILEGED_FUNCTION
- void vTaskSuspendAll (void) PRIVILEGED_FUNCTION
- BaseType_t xTaskResumeAll (void) PRIVILEGED_FUNCTION
- TickType_t xTaskGetTickCount (void) PRIVILEGED_FUNCTION
- TickType_t xTaskGetTickCountFromISR (void) PRIVILEGED_FUNCTION
- UBaseType_t uxTaskGetNumberOfTasks (void) PRIVILEGED_FUNCTION
- char * pcTaskGetName (TaskHandle_t xTaskToQuery) PRIVILEGED_FUNCTION
- TaskHandle_t xTaskGetHandle (const char *pcNameToQuery) PRIVILEGED_FUNCTION
- UBaseType_t uxTaskGetStackHighWaterMark (TaskHandle_t xTask) PRIVILEGED_FUNCTION
- BaseType_t xTaskCallApplicationTaskHook (TaskHandle_t xTask, void *pvParameter) PRIVILEGED_FUNCTION
- TaskHandle_t xTaskGetIdleTaskHandle (void) PRIVILEGED_FUNCTION
- UBaseType_t uxTaskGetSystemState (TaskStatus_t *const pxTaskStatusArray, const UBaseType_t uxArraySize, uint32_t *const pulTotalRunTime) PRIVILEGED_FUNCTION
- void vTaskList (char *pcWriteBuffer) PRIVILEGED_FUNCTION
- void vTaskGetRunTimeStats (char *pcWriteBuffer) PRIVILEGED_FUNCTION
- BaseType_t xTaskGenericNotify (TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction, uint32_t *pulPreviousNotificationValue) PRIVILEGED_FUNCTION
- BaseType_t xTaskGenericNotifyFromISR (TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction, uint32_t *pulPreviousNotificationValue, BaseType_t *pxHigherPriorityTaskWoken) PRIVILEGED_FUNCTION
- BaseType_t xTaskNotifyWait (uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t *pulNotificationValue, TickType_t xTicksToWait) PRIVILEGED_FUNCTION
- void vTaskNotifyGiveFromISR (TaskHandle_t xTaskToNotify, BaseType_t *pxHigherPriorityTaskWoken) PRIVILEGED_FUNCTION
- uint32_t ulTaskNotifyTake (BaseType_t xClearCountOnExit, TickType_t xTicksToWait) PRIVILEGED_FUNCTION
- BaseType_t xTaskNotifyStateClear (TaskHandle_t xTask)
- BaseType_t xTaskIncrementTick (void) PRIVILEGED_FUNCTION
- void vTaskPlaceOnEventList (List_t *const pxEventList, const TickType_t xTicksToWait) PRIVILEGED_FUNCTION
- void vTaskPlaceOnUnorderedEventList (List_t *pxEventList, const TickType_t xItemValue, const TickType_t xTicksToWait) PRIVILEGED_FUNCTION
- void vTaskPlaceOnEventListRestricted (List_t *const pxEventList, TickType_t xTicksToWait, const BaseType_t xWaitIndefinitely) PRIVILEGED_FUNCTION
- BaseType_t xTaskRemoveFromEventList (const List_t *const pxEventList) PRIVILEGED_FUNCTION

- BaseType_t xTaskRemoveFromUnorderedEventList (ListItem_t *pxEventListItem, const TickType_t xItem↩
Value) PRIVILEGED_FUNCTION
- void vTaskSwitchContext (void) PRIVILEGED_FUNCTION
- TickType_t uxTaskResetEventItemValue (void) PRIVILEGED_FUNCTION
- TaskHandle_t xTaskGetCurrentTaskHandle (void) PRIVILEGED_FUNCTION
- void vTaskSetTimeOutState (TimeOut_t *const pxTimeOut) PRIVILEGED_FUNCTION
- BaseType_t xTaskCheckForTimeOut (TimeOut_t *const pxTimeOut, TickType_t *const pxTicksToWait) P↩
RIVILEGED_FUNCTION
- void vTaskMissedYield (void) PRIVILEGED_FUNCTION
- BaseType_t xTaskGetSchedulerState (void) PRIVILEGED_FUNCTION
- void vTaskPriorityInherit (TaskHandle_t const pxMutexHolder) PRIVILEGED_FUNCTION
- BaseType_t xTaskPriorityDisinherit (TaskHandle_t const pxMutexHolder) PRIVILEGED_FUNCTION
- UBaseType_t uxTaskGetTaskNumber (TaskHandle_t xTask) PRIVILEGED_FUNCTION
- void vTaskSetTaskNumber (TaskHandle_t xTask, const UBaseType_t uxHandle) PRIVILEGED_FUNCTION
- void vTaskStepTick (const TickType_t xTicksToJump) PRIVILEGED_FUNCTION
- eSleepModeStatus eTaskConfirmSleepModeStatus (void) PRIVILEGED_FUNCTION
- void * pvTaskIncrementMutexHeldCount (void) PRIVILEGED_FUNCTION

7.24.1 Macro Definition Documentation

7.24.1.1 #define taskDISABLE_INTERRUPTS( ) portDISABLE_INTERRUPTS()

Definition at line 242 of file task.h.

7.24.1.2 #define taskENABLE_INTERRUPTS( ) portENABLE_INTERRUPTS()

Definition at line 252 of file task.h.

7.24.1.3 #define taskENTER_CRITICAL( ) portENTER_CRITICAL()

Definition at line 217 of file task.h.

7.24.1.4 #define taskENTER_CRITICAL_FROM_ISR( ) portSET_INTERRUPT_MASK_FROM_ISR()

Definition at line 218 of file task.h.

7.24.1.5 #define taskEXIT_CRITICAL( ) portEXIT_CRITICAL()

Definition at line 232 of file task.h.

7.24.1.6 #define taskEXIT_CRITICAL_FROM_ISR(*x*) portCLEAR_INTERRUPT_MASK_FROM_ISR(x)

Definition at line 233 of file task.h.

7.24.1.7 #define taskSCHEDULER_NOT_STARTED ((BaseType_t) 1)

Definition at line 258 of file task.h.

7.24.1.8 #define taskSCHEDULER_RUNNING ((BaseType_t) 2)

Definition at line 259 of file task.h.

7.24.1.9 #define taskSCHEDULER_SUSPENDED ((BaseType_t) 0)

Definition at line 257 of file task.h.

7.24.1.10 #define taskYIELD( ) portYIELD()

Definition at line 203 of file task.h.

7.24.1.11 #define tskIDLE_PRIORITY ((UBaseType_t) 0U)

Defines the priority used by the idle task. This must not be modified.

Definition at line 193 of file task.h.

7.24.1.12 #define tskKERNEL_VERSION_BUILD 0

Definition at line 91 of file task.h.

7.24.1.13 #define tskKERNEL_VERSION_MAJOR 9

Definition at line 89 of file task.h.

7.24.1.14 #define tskKERNEL_VERSION_MINOR 0

Definition at line 90 of file task.h.

7.24.1.15 #define tskKERNEL_VERSION_NUMBER "V9.0.0"

Definition at line 88 of file task.h.

7.24.1.16 #define xTaskNotify(*xTaskToNotify, ulValue, eAction*) xTaskGenericNotify((xTaskToNotify), (ulValue), (eAction), NULL)

Definition at line 1712 of file task.h.

7.24.1.17 #define xTaskNotifyAndQuery(*xTaskToNotify, ulValue, eAction, pulPreviousNotifyValue*) xTaskGenericNotify((xTaskToNotify), (ulValue), (eAction), (pulPreviousNotifyValue))

Definition at line 1713 of file task.h.

7.24.1.18 #define xTaskNotifyAndQueryFromISR(*xTaskToNotify, ulValue, eAction, pulPreviousNotificationValue, pxHigherPriorityTaskWoken*) xTaskGenericNotifyFromISR((xTaskToNotify), (ulValue), (eAction), (pulPreviousNotificationValue), (pxHigherPriorityTaskWoken))

Definition at line 1804 of file task.h.

7.24.1.19 #define xTaskNotifyFromISR(*xTaskToNotify, ulValue, eAction, pxHigherPriorityTaskWoken*) xTaskGenericNotifyFromISR((xTaskToNotify), (ulValue), (eAction), NULL, (pxHigherPriorityTaskWoken))

Definition at line 1803 of file task.h.

7.24.1.20 #define xTaskNotifyGive(*xTaskToNotify*) xTaskGenericNotify((xTaskToNotify), (0), eIncrement, NULL)

Definition at line 1925 of file task.h.

7.24.2 Typedef Documentation

7.24.2.1 typedef struct xMEMORY_REGION MemoryRegion_t

7.24.2.2 typedef void* TaskHandle_t

Definition at line 103 of file task.h.

7.24.2.3 typedef BaseType_t(* TaskHookFunction_t) (void *)

Definition at line 109 of file task.h.

7.24.2.4 typedef struct xTASK_PARAMETERS TaskParameters_t

7.24.2.5 typedef struct xTASK_STATUS TaskStatus_t

7.24.2.6 typedef struct xTIME_OUT TimeOut_t

7.24.3 Enumeration Type Documentation

7.24.3.1 enum eNotifyAction

Enumerator

eNoAction
eSetBits
eIncrement
eSetValueWithOverwrite
eSetValueWithoutOverwrite

Definition at line 123 of file task.h.

7.24.3.2 enum eSleepModeStatus

Enumerator

eAbortSleep

eStandardSleep

eNoTasksWaitingTimeout

Definition at line 181 of file task.h.

7.24.3.3 enum eTaskState

Enumerator

eRunning

eReady

eBlocked

eSuspended

eDeleted

eInvalid

Definition at line 112 of file task.h.

7.24.4 Function Documentation

7.24.4.1 eSleepModeStatus eTaskConfirmSleepModeStatus (void)

7.24.4.2 eTaskState eTaskGetState (TaskHandle_t *xTask*)

task. h

```
eTaskState eTaskGetState( TaskHandle_t xTask );
```

INCLUDE_eTaskGetState must be defined as 1 for this function to be available. See the configuration section for more information.

Obtain the state of any task. States are encoded by the eTaskState enumerated type.

Parameters

xTask	Handle of the task to be queried.

Returns

The state of xTask at the time the function was called. Note the state of the task might change between the function being called, and the functions return value being tested by the calling task.

7.24.4.3 char* pcTaskGetName (TaskHandle_t *xTaskToQuery*)

Definition at line 2181 of file tasks.c.

7.24.4.4 void* pvTaskIncrementMutexHeldCount (void)

7.24.4.5 uint32_t ulTaskNotifyTake (BaseType_t *xClearCountOnExit,* TickType_t *xTicksToWait*)

7.24.4.6 UBaseType_t uxTaskGetNumberOfTasks (void)

Definition at line 2173 of file tasks.c.

7.24.4.7 UBaseType_t uxTaskGetStackHighWaterMark (TaskHandle_t *xTask*)

task.h

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

INCLUDE_uxTaskGetStackHighWaterMark must be set to 1 in FreeRTOSConfig.h for this function to be available.

Returns the high water mark of the stack associated with xTask. That is, the minimum free stack space there has been (in words, so on a 32 bit machine a value of 1 means 4 bytes) since the task started. The smaller the returned number the closer the task has come to overflowing its stack.

Parameters

xTask	Handle of the task associated with the stack to be checked. Set xTask to NULL to check the stack of the calling task.

Returns

The smallest amount of free stack space there has been (in words, so actual spaces on the stack rather than bytes) since the task referenced by xTask was created.

7.24.4.8 UBaseType_t uxTaskGetSystemState (TaskStatus_t *const *pxTaskStatusArray,* const UBaseType_t *uxArraySize,* uint32_t *const *pulTotalRunTime*)

configUSE_TRACE_FACILITY must be defined as 1 in FreeRTOSConfig.h for uxTaskGetSystemState() to be available.

uxTaskGetSystemState() populates an TaskStatus_t structure for each task in the system. TaskStatus_t structures contain, among other things, members for the task handle, task name, task priority, task state, and total amount of run time consumed by the task. See the TaskStatus_t structure definition in this file for the full member list.

NOTE: This function is intended for debugging use only as its use results in the scheduler remaining suspended for an extended period.

Parameters

pxTaskStatusArray	A pointer to an array of TaskStatus_t structures. The array must contain at least one TaskStatus_t structure for each task that is under the control of the RTOS. The number of tasks under the control of the RTOS can be determined using the uxTaskGetNumberOfTasks() API function.
uxArraySize	The size of the array pointed to by the pxTaskStatusArray parameter. The size is specified as the number of indexes in the array, or the number of TaskStatus_t structures contained in the array, not by the number of bytes in the array.
pulTotalRunTime	If configGENERATE_RUN_TIME_STATS is set to 1 in FreeRTOSConfig.h then ∗pulTotalRunTime is set by uxTaskGetSystemState() to the total run time (as defined by the run time stats clock, see http://www.freertos.org/rtos-run-time-stats.html) since the target booted. pulTotalRunTime can be set to NULL to omit the total run time information.

Returns

The number of TaskStatus_t structures that were populated by uxTaskGetSystemState(). This should equal the number returned by the uxTaskGetNumberOfTasks() API function, but will be zero if the value passed in the uxArraySize parameter was too small.

Example usage:

```
// This example demonstrates how a human readable table of run time stats
// information is generated from raw data provided by uxTaskGetSystemState().
// The human readable table is written to pcWriteBuffer
void vTaskGetRunTimeStats( char *pcWriteBuffer )
{
TaskStatus_t *pxTaskStatusArray;
volatile UBaseType_t uxArraySize, x;
uint32_t ulTotalRunTime, ulStatsAsPercentage;


   // Make sure the write buffer does not contain a string.
   *pcWriteBuffer = 0x00;


   // Take a snapshot of the number of tasks in case it changes while this
   // function is executing.
   uxArraySize = uxTaskGetNumberOfTasks();


   // Allocate a TaskStatus_t structure for each task.  An array could be
   // allocated statically at compile time.
   pxTaskStatusArray = pvPortMalloc( uxArraySize * sizeof( TaskStatus_t ) );


   if( pxTaskStatusArray != NULL )
   {
       // Generate raw status information about each task.
       uxArraySize = uxTaskGetSystemState( pxTaskStatusArray, uxArraySize, &ulTotalRunTime );


       // For percentage calculations.
       ulTotalRunTime /= 100UL;
```

```
        // Avoid divide by zero errors.
        if( ulTotalRunTime > 0 )
        {
            // For each populated position in the pxTaskStatusArray array,
            // format the raw data as human readable ASCII data
            for( x = 0; x < uxArraySize; x++ )
            {
                // What percentage of the total run time has the task used?
                // This will always be rounded down to the nearest integer.
                // ulTotalRunTimeDiv100 has already been divided by 100.
                ulStatsAsPercentage = pxTaskStatusArray[ x ].ulRunTimeCounter / ulTotalRunTime;


                if( ulStatsAsPercentage > 0UL )
                {
                    sprintf( pcWriteBuffer, "%s\t\t%lu\t\t%lu%%\r\n", pxTaskStatusArray[ x ].pcTas
                }
                else
                {
                    // If the percentage is zero here then the task has
                    // consumed less than 1% of the total run time.
                    sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%%\r\n", pxTaskStatusArray[ x ].pcTaskN
                }


                pcWriteBuffer += strlen( ( char * ) pcWriteBuffer );
            }
        }


        // The array is no longer needed, free the memory it consumes.
        vPortFree( pxTaskStatusArray );
    }
}
```

7.24.4.9 UBaseType_t uxTaskGetTaskNumber (TaskHandle_t *xTask*)

7.24.4.10 UBaseType_t uxTaskPriorityGet (TaskHandle_t *xTask*)

7.24.4.11 UBaseType_t uxTaskPriorityGetFromISR (TaskHandle_t *xTask*)

task. h

```
UBaseType_t uxTaskPriorityGetFromISR( TaskHandle_t xTask );
```

A version of uxTaskPriorityGet() that can be used from an ISR.

7.24.4.12 TickType_t uxTaskResetEventItemValue (void)

Definition at line 4162 of file tasks.c.

7.24.4.13 void vTaskAllocateMPURegions (TaskHandle_t *xTask,* const MemoryRegion_t ∗const *pxRegions*)

7.24.4.14 void vTaskDelay (const TickType_t *xTicksToDelay*)

7.24.4.15 void vTaskDelayUntil (TickType_t ∗const *pxPreviousWakeTime,* const TickType_t *xTimeIncrement*)

7.24.4.16 void vTaskDelete (TaskHandle_t *xTaskToDelete*)

7.24.4.17 void vTaskEndScheduler (void)

Definition at line 1933 of file tasks.c.

7.24.4.18 void vTaskGetInfo (TaskHandle_t *xTask,* TaskStatus_t ∗ *pxTaskStatus,* BaseType_t *xGetFreeStackSpace,* eTaskState *eState*)

7.24.4.19 void vTaskGetRunTimeStats (char ∗ *pcWriteBuffer*)

7.24.4.20 void vTaskList (char ∗ *pcWriteBuffer*)

7.24.4.21 void vTaskMissedYield (void)

Definition at line 3076 of file tasks.c.

7.24.4.22 void vTaskNotifyGiveFromISR (TaskHandle_t *xTaskToNotify,* BaseType_t ∗ *pxHigherPriorityTaskWoken*)

7.24.4.23 void vTaskPlaceOnEventList (List_t ∗const *pxEventList,* const TickType_t *xTicksToWait*)

Definition at line 2820 of file tasks.c.

7.24.4.24 void vTaskPlaceOnEventListRestricted (List_t ∗const *pxEventList,* TickType_t *xTicksToWait,* const BaseType_t *xWaitIndefinitely*)

7.24.4.25 void vTaskPlaceOnUnorderedEventList (List_t ∗ *pxEventList,* const TickType_t *xItemValue,* const TickType_t *xTicksToWait*)

Definition at line 2837 of file tasks.c.

7.24.4.26 void vTaskPriorityInherit (TaskHandle_t const *pxMutexHolder*)

7.24.4.27 void vTaskPrioritySet (TaskHandle_t *xTask,* UBaseType_t *uxNewPriority*)

7.24.4.28 void vTaskResume (TaskHandle_t *xTaskToResume*)

7.24.4.29 void vTaskSetTaskNumber (TaskHandle_t *xTask,* const UBaseType_t *uxHandle*)

7.24.4.30 void vTaskSetTimeOutState (TimeOut_t ∗const *pxTimeOut*)

Definition at line 3007 of file tasks.c.

7.24.4.31 void vTaskStartScheduler (void)

Definition at line 1826 of file tasks.c.

7.24.4.32 void vTaskStepTick (const TickType_t *xTicksToJump*)

7.24.4.33 void vTaskSuspend (TaskHandle_t *xTaskToSuspend*)

7.24.4.34 void vTaskSuspendAll (void)

Definition at line 1944 of file tasks.c.

7.24.4.35 void vTaskSwitchContext (void)

Definition at line 2761 of file tasks.c.

7.24.4.36 BaseType_t xTaskAbortDelay (TaskHandle_t *xTask*)

7.24.4.37 BaseType_t xTaskCallApplicationTaskHook (TaskHandle_t *xTask*, void * *pvParameter*)

task.h

```
BaseType_t xTaskCallApplicationTaskHook( TaskHandle_t xTask, void *pvParameter );
```

Calls the hook function associated with xTask. Passing xTask as NULL has the effect of calling the Running tasks (the calling task) hook function.

pvParameter is passed to the hook function for the task to interpret as it wants. The return value is the value returned by the task hook function registered by the user.

7.24.4.38 BaseType_t xTaskCheckForTimeOut (TimeOut_t *const *pxTimeOut*, TickType_t *const *pxTicksToWait*)

Definition at line 3015 of file tasks.c.

7.24.4.39 BaseType_t xTaskGenericNotify (TaskHandle_t *xTaskToNotify*, uint32_t *ulValue*, eNotifyAction *eAction*, uint32_t * *pulPreviousNotificationValue*)

7.24.4.40 BaseType_t xTaskGenericNotifyFromISR (TaskHandle_t *xTaskToNotify*, uint32_t *ulValue*, eNotifyAction *eAction*, uint32_t * *pulPreviousNotificationValue*, BaseType_t * *pxHigherPriorityTaskWoken*)

7.24.4.41 TaskHandle_t xTaskGetCurrentTaskHandle (void)

7.24.4.42 TaskHandle_t xTaskGetHandle (const char * *pcNameToQuery*)

7.24.4.43 TaskHandle_t xTaskGetIdleTaskHandle (void)

xTaskGetIdleTaskHandle() is only available if INCLUDE_xTaskGetIdleTaskHandle is set to 1 in FreeRTOSConfig.h.

Simply returns the handle of the idle task. It is not valid to call xTaskGetIdleTaskHandle() before the scheduler has been started.

7.24.4.44 BaseType_t xTaskGetSchedulerState (void)

7.24.4.45 TickType_t xTaskGetTickCount (void)

Definition at line 2127 of file tasks.c.

7.24.4.46 TickType_t xTaskGetTickCountFromISR (void)

Definition at line 2142 of file tasks.c.

7.24.4.47 BaseType_t xTaskIncrementTick (void)

Definition at line 2499 of file tasks.c.

7.24.4.48 BaseType_t xTaskNotifyStateClear (TaskHandle_t *xTask*)

7.24.4.49 BaseType_t xTaskNotifyWait (uint32_t *ulBitsToClearOnEntry,* uint32_t *ulBitsToClearOnExit,* uint32_t * *pulNotificationValue,* TickType_t *xTicksToWait*)

7.24.4.50 BaseType_t xTaskPriorityDisinherit (TaskHandle_t const *pxMutexHolder*)

7.24.4.51 BaseType_t xTaskRemoveFromEventList (const List_t *const *pxEventList*)

Definition at line 2894 of file tasks.c.

7.24.4.52 BaseType_t xTaskRemoveFromUnorderedEventList (ListItem_t * *pxEventListItem,* const TickType_t *xItemValue*)

Definition at line 2962 of file tasks.c.

7.24.4.53 BaseType_t xTaskResumeAll (void)

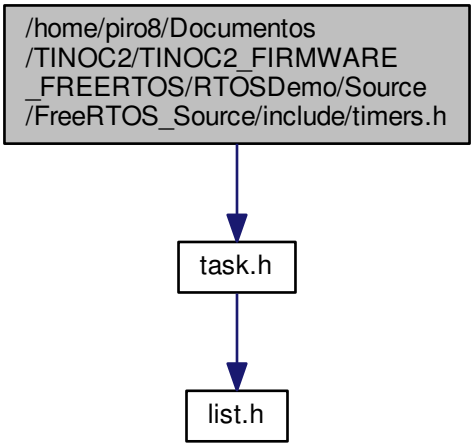
Definition at line 2017 of file tasks.c.

7.24.4.54 BaseType_t xTaskResumeFromISR (TaskHandle_t *xTaskToResume*)

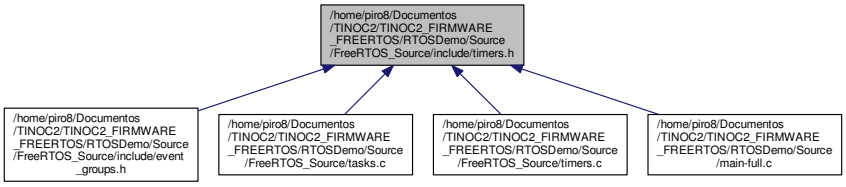
7.25 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/FreeRTOS_Source/include/timers.h File Reference

```
#include "task.h"
```

Include dependency graph for timers.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR ((BaseType_t) -2)
- #define tmrCOMMAND_EXECUTE_CALLBACK ((BaseType_t) -1)
- #define tmrCOMMAND_START_DONT_TRACE ((BaseType_t) 0)
- #define tmrCOMMAND_START ((BaseType_t) 1)
- #define tmrCOMMAND_RESET ((BaseType_t) 2)
- #define tmrCOMMAND_STOP ((BaseType_t) 3)
- #define tmrCOMMAND_CHANGE_PERIOD ((BaseType_t) 4)
- #define tmrCOMMAND_DELETE ((BaseType_t) 5)

- #define tmrFIRST_FROM_ISR_COMMAND ((BaseType_t) 6)
- #define tmrCOMMAND_START_FROM_ISR ((BaseType_t) 6)
- #define tmrCOMMAND_RESET_FROM_ISR ((BaseType_t) 7)
- #define tmrCOMMAND_STOP_FROM_ISR ((BaseType_t) 8)
- #define tmrCOMMAND_CHANGE_PERIOD_FROM_ISR ((BaseType_t) 9)
- #define xTimerStart(xTimer, xTicksToWait) xTimerGenericCommand((xTimer), tmrCOMMAND_START, (xTaskGetTickCount()), NULL, (xTicksToWait))
- #define xTimerStop(xTimer, xTicksToWait) xTimerGenericCommand((xTimer), tmrCOMMAND_STOP, 0U, NULL, (xTicksToWait))
- #define xTimerChangePeriod(xTimer, xNewPeriod, xTicksToWait) xTimerGenericCommand((xTimer), tmrCOMMAND_CHANGE_PERIOD, (xNewPeriod), NULL, (xTicksToWait))
- #define xTimerDelete(xTimer, xTicksToWait) xTimerGenericCommand((xTimer), tmrCOMMAND_DELETE, 0U, NULL, (xTicksToWait))
- #define xTimerReset(xTimer, xTicksToWait) xTimerGenericCommand((xTimer), tmrCOMMAND_RESET, (xTaskGetTickCount()), NULL, (xTicksToWait))
- #define xTimerStartFromISR(xTimer, pxHigherPriorityTaskWoken) xTimerGenericCommand((xTimer), tmrCOMMAND_START_FROM_ISR, (xTaskGetTickCountFromISR()), (pxHigherPriorityTaskWoken), 0U)
- #define xTimerStopFromISR(xTimer, pxHigherPriorityTaskWoken) xTimerGenericCommand((xTimer), tmrCOMMAND_STOP_FROM_ISR, 0, (pxHigherPriorityTaskWoken), 0U)
- #define xTimerChangePeriodFromISR(xTimer, xNewPeriod, pxHigherPriorityTaskWoken) xTimerGenericCommand((xTimer), tmrCOMMAND_CHANGE_PERIOD_FROM_ISR, (xNewPeriod), (pxHigherPriorityTaskWoken), 0U)
- #define xTimerResetFromISR(xTimer, pxHigherPriorityTaskWoken) xTimerGenericCommand((xTimer), tmrCOMMAND_RESET_FROM_ISR, (xTaskGetTickCountFromISR()), (pxHigherPriorityTaskWoken), 0U)

Typedefs

- typedef void * TimerHandle_t
- typedef void(* TimerCallbackFunction_t) (TimerHandle_t xTimer)
- typedef void(* PendedFunction_t) (void *, uint32_t)

Functions

- void * pvTimerGetTimerID (const TimerHandle_t xTimer) PRIVILEGED_FUNCTION
- void vTimerSetTimerID (TimerHandle_t xTimer, void *pvNewID) PRIVILEGED_FUNCTION
- BaseType_t xTimerIsTimerActive (TimerHandle_t xTimer) PRIVILEGED_FUNCTION
- TaskHandle_t xTimerGetTimerDaemonTaskHandle (void) PRIVILEGED_FUNCTION
- BaseType_t xTimerPendFunctionCallFromISR (PendedFunction_t xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, BaseType_t *pxHigherPriorityTaskWoken) PRIVILEGED_FUNCTION
- BaseType_t xTimerPendFunctionCall (PendedFunction_t xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, TickType_t xTicksToWait) PRIVILEGED_FUNCTION
- const char * pcTimerGetName (TimerHandle_t xTimer) PRIVILEGED_FUNCTION
- TickType_t xTimerGetPeriod (TimerHandle_t xTimer) PRIVILEGED_FUNCTION
- TickType_t xTimerGetExpiryTime (TimerHandle_t xTimer) PRIVILEGED_FUNCTION
- BaseType_t xTimerCreateTimerTask (void) PRIVILEGED_FUNCTION
- BaseType_t xTimerGenericCommand (TimerHandle_t xTimer, const BaseType_t xCommandID, const TickType_t xOptionalValue, BaseType_t *const pxHigherPriorityTaskWoken, const TickType_t xTicksToWait) PRIVILEGED_FUNCTION

7.25.1 Macro Definition Documentation

7.25.1.1 #define tmrCOMMAND_CHANGE_PERIOD ((BaseType_t) 4)

Definition at line 102 of file timers.h.

7.25.1.2 #define tmrCOMMAND_CHANGE_PERIOD_FROM_ISR ((BaseType_t) 9)

Definition at line 109 of file timers.h.

7.25.1.3 #define tmrCOMMAND_DELETE ((BaseType_t) 5)

Definition at line 103 of file timers.h.

7.25.1.4 #define tmrCOMMAND_EXECUTE_CALLBACK ((BaseType_t) -1)

Definition at line 97 of file timers.h.

7.25.1.5 #define tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR ((BaseType_t) -2)

Definition at line 96 of file timers.h.

7.25.1.6 #define tmrCOMMAND_RESET ((BaseType_t) 2)

Definition at line 100 of file timers.h.

7.25.1.7 #define tmrCOMMAND_RESET_FROM_ISR ((BaseType_t) 7)

Definition at line 107 of file timers.h.

7.25.1.8 #define tmrCOMMAND_START ((BaseType_t) 1)

Definition at line 99 of file timers.h.

7.25.1.9 #define tmrCOMMAND_START_DONT_TRACE ((BaseType_t) 0)

Definition at line 98 of file timers.h.

7.25.1.10 #define tmrCOMMAND_START_FROM_ISR ((BaseType_t) 6)

Definition at line 106 of file timers.h.

7.25.1.11 #define tmrCOMMAND_STOP ((BaseType_t) 3)

Definition at line 101 of file timers.h.

7.25.1.12 #define tmrCOMMAND_STOP_FROM_ISR ((BaseType_t) 8)

Definition at line 108 of file timers.h.

7.25.1.13 #define tmrFIRST_FROM_ISR_COMMAND ((BaseType_t) 6)

Definition at line 105 of file timers.h.

7.25.1.14 #define xTimerChangePeriod(*xTimer, xNewPeriod, xTicksToWait*) xTimerGenericCommand((xTimer), tmrCOMMAND_CHANGE_PERIOD, (xNewPeriod), NULL, (xTicksToWait))

BaseType_t xTimerChangePeriod(TimerHandle_t xTimer, TickType_t xNewPeriod, TickType_t xTicksToWait);

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerChangePeriod() changes the period of a timer that was previously created using the xTimerCreate() API function.

xTimerChangePeriod() can be called to change the period of an active or dormant state timer.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerChangePeriod() to be available.

Parameters

xTimer	The handle of the timer that is having its period changed.
xNewPeriod	The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
xTicksToWait	Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the change period command to be successfully sent to the timer command queue, should the queue already be full when xTimerChangePeriod() was called. xTicksToWait is ignored if xTimerChangePeriod() is called before the scheduler is started.

Returns

pdFAIL will be returned if the change period command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

```
* // This function assumes xTimer has already been created.  If the timer
* // referenced by xTimer is already active when it is called, then the timer
* // is deleted.  If the timer referenced by xTimer is not active when it is
* // called, then the period of the timer is set to 500ms and the timer is
* // started.
* void vAFunction( TimerHandle_t xTimer )
* {
*     if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and equivalently "if( xTimerIsTimerActi
*     {
*         // xTimer is already active - delete it.
*         xTimerDelete( xTimer );
*     }
*     else
*     {
*         // xTimer is not active, change its period to 500ms.  This will also
*         // cause the timer to start.  Block for a maximum of 100 ticks if the
*         // change period command cannot immediately be sent to the timer
*         // command queue.
*         if( xTimerChangePeriod( xTimer, 500 / portTICK_PERIOD_MS, 100 ) == pdPASS )
*         {
*             // The command was successfully sent.
*         }
*         else
*         {
*             // The command could not be sent, even after waiting for 100 ticks
*             // to pass.  Take appropriate action here.
*         }
*     }
* }
*
```

Definition at line 667 of file timers.h.

7.25.1.15 #define xTimerChangePeriodFromISR(*xTimer, xNewPeriod, pxHigherPriorityTaskWoken*) xTimerGenericCommand((xTimer), tmrCOMMAND_CHANGE_PERIOD_FROM_ISR, (xNewPeriod), (pxHigherPriorityTaskWoken), 0U)

BaseType_t xTimerChangePeriodFromISR(TimerHandle_t xTimer, TickType_t xNewPeriod, BaseType_t ∗px↩HigherPriorityTaskWoken);

A version of xTimerChangePeriod() that can be called from an interrupt service routine.

Parameters

xTimer	The handle of the timer that is having its period changed.
xNewPeriod	The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
pxHigherPriorityTaskWoken	The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerChangePeriodFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/ daemon task out of the Blocked state. If calling xTimerChangePeriodFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then ∗pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerChangePeriodFromISR() function. If xTimerChangePeriodFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Returns

pdFAIL will be returned if the command to change the timers period could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIOR↩ ITY configuration constant.

Example usage:

```
* // This scenario assumes xTimer has already been created and started.  When
* // an interrupt occurs, the period of xTimer should be changed to 500ms.
*
* // The interrupt service routine that changes the period of xTimer.
* void vAnExampleInterruptServiceRoutine( void )
* {
* BaseType_t xHigherPriorityTaskWoken = pdFALSE;
*
*     // The interrupt has occurred - change the period of xTimer to 500ms.
*     // xHigherPriorityTaskWoken was set to pdFALSE where it was defined
*     // (within this function).  As this is an interrupt service routine, only
*     // FreeRTOS API functions that end in "FromISR" can be used.
*     if( xTimerChangePeriodFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
*     {
*         // The command to change the timers period was not executed
*         // successfully.  Take appropriate action here.
*     }
*
*     // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
*     // should be performed.  The syntax required to perform a context switch
*     // from inside an ISR varies from port to port, and from compiler to
*     // compiler.  Inspect the demos for the port you are using to find the
*     // actual syntax required.
*     if( xHigherPriorityTaskWoken != pdFALSE )
*     {
*         // Call the interrupt safe yield function here (actual function
*         // depends on the FreeRTOS port being used).
*     }
* }
*
```

Definition at line 1051 of file timers.h.

7.25.1.16 #define xTimerDelete(*xTimer, xTicksToWait*) xTimerGenericCommand((xTimer), tmrCOMMAND_DELETE, 0U, NULL, (xTicksToWait))

BaseType_t xTimerDelete(TimerHandle_t xTimer, TickType_t xTicksToWait);

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerDelete() deletes a timer that was previously created using the xTimerCreate() API function.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerDelete() to be available.

Parameters

xTimer	The handle of the timer being deleted.
xTicksToWait	Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the delete command to be successfully sent to the timer command queue, should the queue already be full when xTimerDelete() was called. xTicksToWait is ignored if xTimerDelete() is called before the scheduler is started.

Returns

pdFAIL will be returned if the delete command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

See the xTimerChangePeriod() API function example usage scenario.

Definition at line 705 of file timers.h.

7.25.1.17 #define xTimerReset(*xTimer, xTicksToWait*) xTimerGenericCommand((xTimer), tmrCOMMAND_RESET, (xTaskGetTickCount()), NULL, (xTicksToWait))

BaseType_t xTimerReset(TimerHandle_t xTimer, TickType_t xTicksToWait);

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerReset() re-starts a timer that was previously created using the xTimerCreate() API function. If the timer had already been started and was already in the active state, then xTimerReset() will cause the timer to re-evaluate its expiry time so that it is relative to when xTimerReset() was called. If the timer was in the dormant state then xTimerReset() has equivalent functionality to the xTimerStart() API function.

Resetting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n' ticks after xTimerReset() was called, where 'n' is the timers defined period.

It is valid to call xTimerReset() before the scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the scheduler is started, not relative to when xTimerReset() was called.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerReset() to be available.

Parameters

xTimer	The handle of the timer being reset/started/restarted.
xTicksToWait	Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the reset command to be successfully sent to the timer command queue, should the queue already be full when xTimerReset() was called. xTicksToWait is ignored if xTimerReset() is called before the scheduler is started.

Returns

pdFAIL will be returned if the reset command could not be sent to the timer command queue even after x↩ TicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStart() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_P↩ RIORITY configuration constant.

Example usage:

```
* // When a key is pressed, an LCD back-light is switched on.  If 5 seconds pass
* // without a key being pressed, then the LCD back-light is switched off.  In
* // this case, the timer is a one-shot timer.
*
* TimerHandle_t xBacklightTimer = NULL;
*
* // The callback function assigned to the one-shot timer.  In this case the
* // parameter is not used.
* void vBacklightTimerCallback( TimerHandle_t pxTimer )
* {
*     // The timer expired, therefore 5 seconds must have passed since a key
*     // was pressed.  Switch off the LCD back-light.
*     vSetBacklightState( BACKLIGHT_OFF );
* }
*
* // The key press event handler.
* void vKeyPressEventHandler( char cKey )
* {
*     // Ensure the LCD back-light is on, then reset the timer that is
*     // responsible for turning the back-light off after 5 seconds of
*     // key inactivity.  Wait 10 ticks for the command to be successfully sent
*     // if it cannot be sent immediately.
*     vSetBacklightState( BACKLIGHT_ON );
*     if( xTimerReset( xBacklightTimer, 100 ) != pdPASS )
*     {
*         // The reset command was not executed successfully.  Take appropriate
*         // action here.
*     }
*
*     // Perform the rest of the key processing here.
* }
*
* void main( void )
* {
* int32_t x;
*
*     // Create then start the one-shot timer that is responsible for turning
*     // the back-light off if no keys are pressed within a 5 second period.
*     xBacklightTimer = xTimerCreate( "BacklightTimer",           // Just a text name, not used by the kernel.
*                                     ( 5000 / portTICK_PERIOD_MS), // The timer period in ticks.
*                                     pdFALSE,                    // The timer is a one-shot timer.
*                                     0,                          // The id is not used by the callback so can
*                                     vBacklightTimerCallback     // The callback function that switches the L
*                                   );
*
*     if( xBacklightTimer == NULL )
*     {
*         // The timer was not created.
*     }
*     else
*     {
*         // Start the timer.  No block time is specified, and even if one was
*         // it would be ignored because the scheduler has not yet been
*         // started.
*         if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )
*         {
*             // The timer could not be set into the Active state.
*         }
*     }
*
*     // ...
*     // Create tasks here.
*     // ...
*
*     // Starting the scheduler will start the timer running as it has already
*     // been set into the active state.
*     vTaskStartScheduler();
*
*     // Should not reach here.
*     for( ;; );
* }
```

*

Definition at line 829 of file timers.h.

7.25.1.18 #define xTimerResetFromISR(*xTimer, pxHigherPriorityTaskWoken*) xTimerGenericCommand((xTimer), tmrCOMMAND_RESET_FROM_ISR, (xTaskGetTickCountFromISR()), (pxHigherPriorityTaskWoken), 0U)

BaseType_t xTimerResetFromISR(TimerHandle_t xTimer, BaseType_t ∗pxHigherPriorityTaskWoken);

A version of xTimerReset() that can be called from an interrupt service routine.

Parameters

xTimer	The handle of the timer that is to be started, reset, or restarted.
pxHigherPriorityTaskWoken	The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerResetFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerResetFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then ∗pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerResetFromISR() function. If xTimerResetFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Returns

pdFAIL will be returned if the reset command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerResetFromISR() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

```
* // This scenario assumes xBacklightTimer has already been created.  When a
* // key is pressed, an LCD back-light is switched on.  If 5 seconds pass
* // without a key being pressed, then the LCD back-light is switched off.  In
* // this case, the timer is a one-shot timer, and unlike the example given for
* // the xTimerReset() function, the key press event handler is an interrupt
* // service routine.
*
* // The callback function assigned to the one-shot timer.  In this case the
* // parameter is not used.
* void vBacklightTimerCallback( TimerHandle_t pxTimer )
* {
*     // The timer expired, therefore 5 seconds must have passed since a key
*     // was pressed.  Switch off the LCD back-light.
*     vSetBacklightState( BACKLIGHT_OFF );
* }
*
* // The key press interrupt service routine.
* void vKeyPressEventInterruptHandler( void )
* {
* BaseType_t xHigherPriorityTaskWoken = pdFALSE;
*
*     // Ensure the LCD back-light is on, then reset the timer that is
```

```
*     // responsible for turning the back-light off after 5 seconds of
*     // key inactivity.  This is an interrupt service routine so can only
*     // call FreeRTOS API functions that end in "FromISR".
*     vSetBacklightState( BACKLIGHT_ON );
*
*     // xTimerStartFromISR() or xTimerResetFromISR() could be called here
*     // as both cause the timer to re-calculate its expiry time.
*     // xHigherPriorityTaskWoken was initialised to pdFALSE when it was
*     // declared (in this function).
*     if( xTimerResetFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS )
*     {
*         // The reset command was not executed successfully.  Take appropriate
*         // action here.
*     }
*
*     // Perform the rest of the key processing here.
*
*     // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
*     // should be performed.  The syntax required to perform a context switch
*     // from inside an ISR varies from port to port, and from compiler to
*     // compiler.  Inspect the demos for the port you are using to find the
*     // actual syntax required.
*     if( xHigherPriorityTaskWoken != pdFALSE )
*     {
*         // Call the interrupt safe yield function here (actual function
*         // depends on the FreeRTOS port being used).
*     }
* }
*
```

Definition at line 1137 of file timers.h.

7.25.1.19 #define xTimerStart(*xTimer, xTicksToWait*) xTimerGenericCommand((xTimer), tmrCOMMAND_START, (xTaskGetTickCount()), NULL, (xTicksToWait))

BaseType_t xTimerStart(TimerHandle_t xTimer, TickType_t xTicksToWait);

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerStart() starts a timer that was previously created using the xTimerCreate() API function. If the timer had already been started and was already in the active state, then xTimerStart() has equivalent functionality to the xTimerReset() API function.

Starting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n' ticks after xTimerStart() was called, where 'n' is the timers defined period.

It is valid to call xTimerStart() before the scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the scheduler is started, not relative to when xTimerStart() was called.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerStart() to be available.

Parameters

xTimer	The handle of the timer being started/restarted.
xTicksToWait	Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the start command to be successfully sent to the timer command queue, should the queue already be full when xTimerStart() was called. xTicksToWait is ignored if xTimerStart() is called before the scheduler is started.

Returns

pdFAIL will be returned if the start command could not be sent to the timer command queue even after x↩ TicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStart() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_P↩ RIORITY configuration constant.

Example usage:

See the xTimerCreate() API function example usage scenario.

Definition at line 545 of file timers.h.

7.25.1.20 #define xTimerStartFromISR(*xTimer, pxHigherPriorityTaskWoken*) xTimerGenericCommand((xTimer), tmrCOMMAND_START_FROM_ISR, (xTaskGetTickCountFromISR()), (pxHigherPriorityTaskWoken), 0U)

BaseType_t xTimerStartFromISR(TimerHandle_t xTimer, BaseType_t ∗pxHigherPriorityTaskWoken);

A version of xTimerStart() that can be called from an interrupt service routine.

Parameters

xTimer	The handle of the timer being started/restarted.
pxHigherPriorityTaskWoken	The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerStartFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerStartFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then ∗pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerStartFromISR() function. If xTimerStartFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Returns

pdFAIL will be returned if the start command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStartFromISR() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

```
* // This scenario assumes xBacklightTimer has already been created.  When a
* // key is pressed, an LCD back-light is switched on.  If 5 seconds pass
* // without a key being pressed, then the LCD back-light is switched off.  In
* // this case, the timer is a one-shot timer, and unlike the example given for
* // the xTimerReset() function, the key press event handler is an interrupt
* // service routine.
*
* // The callback function assigned to the one-shot timer.  In this case the
```

```
* // parameter is not used.
* void vBacklightTimerCallback( TimerHandle_t pxTimer )
* {
*     // The timer expired, therefore 5 seconds must have passed since a key
*     // was pressed.  Switch off the LCD back-light.
*     vSetBacklightState( BACKLIGHT_OFF );
* }
*
* // The key press interrupt service routine.
* void vKeyPressEventInterruptHandler( void )
* {
* BaseType_t xHigherPriorityTaskWoken = pdFALSE;
*
*     // Ensure the LCD back-light is on, then restart the timer that is
*     // responsible for turning the back-light off after 5 seconds of
*     // key inactivity.  This is an interrupt service routine so can only
*     // call FreeRTOS API functions that end in "FromISR".
*     vSetBacklightState( BACKLIGHT_ON );
*
*     // xTimerStartFromISR() or xTimerResetFromISR() could be called here
*     // as both cause the timer to re-calculate its expiry time.
*     // xHigherPriorityTaskWoken was initialised to pdFALSE when it was
*     // declared (in this function).
*     if( xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS )
*     {
*         // The start command was not executed successfully.  Take appropriate
*         // action here.
*     }
*
*     // Perform the rest of the key processing here.
*
*     // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
*     // should be performed.  The syntax required to perform a context switch
*     // from inside an ISR varies from port to port, and from compiler to
*     // compiler.  Inspect the demos for the port you are using to find the
*     // actual syntax required.
*     if( xHigherPriorityTaskWoken != pdFALSE )
*     {
*         // Call the interrupt safe yield function here (actual function
*         // depends on the FreeRTOS port being used).
*     }
* }
*
```

Definition at line 915 of file timers.h.

7.25.1.21 #define xTimerStop(*xTimer, xTicksToWait*) xTimerGenericCommand((xTimer), tmrCOMMAND_STOP, 0U, NULL, (xTicksToWait))

BaseType_t xTimerStop(TimerHandle_t xTimer, TickType_t xTicksToWait);

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerStop() stops a timer that was previously started using either of the The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() or xTimerChangePeriodFromISR() API functions.

Stopping a timer ensures the timer is not in the active state.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerStop() to be available.

Parameters

xTimer	The handle of the timer being stopped.
xTicksToWait	Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the stop command to be successfully sent to the timer command queue, should the queue already be full when xTimerStop() was called. xTicksToWait is ignored if xTimerStop() is called before the scheduler is started.

Returns

pdFAIL will be returned if the stop command could not be sent to the timer command queue even after xTicks↩ ToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TA↩ SK_PRIORITY configuration constant.

Example usage:

See the xTimerCreate() API function example usage scenario.

Definition at line 587 of file timers.h.

7.25.1.22 #define xTimerStopFromISR(*xTimer, pxHigherPriorityTaskWoken*) xTimerGenericCommand((xTimer), tmrCOMMAND_STOP_FROM_ISR, 0, (pxHigherPriorityTaskWoken), 0U)

BaseType_t xTimerStopFromISR(TimerHandle_t xTimer, BaseType_t ∗pxHigherPriorityTaskWoken);

A version of xTimerStop() that can be called from an interrupt service routine.

Parameters

xTimer	The handle of the timer being stopped.
pxHigherPriorityTaskWoken	The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerStopFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerStopFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then ∗pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerStopFromISR() function. If xTimerStopFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Returns

pdFAIL will be returned if the stop command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

```
* // This scenario assumes xTimer has already been created and started.  When
* // an interrupt occurs, the timer should be simply stopped.
*
* // The interrupt service routine that stops the timer.
* void vAnExampleInterruptServiceRoutine( void )
* {
* BaseType_t xHigherPriorityTaskWoken = pdFALSE;
*
*     // The interrupt has occurred - simply stop the timer.
*     // xHigherPriorityTaskWoken was set to pdFALSE where it was defined
*     // (within this function).  As this is an interrupt service routine, only
*     // FreeRTOS API functions that end in "FromISR" can be used.
*     if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
*     {
*         // The stop command was not executed successfully.  Take appropriate
*         // action here.
*     }
*
*     // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
*     // should be performed.  The syntax required to perform a context switch
*     // from inside an ISR varies from port to port, and from compiler to
*     // compiler.  Inspect the demos for the port you are using to find the
*     // actual syntax required.
*     if( xHigherPriorityTaskWoken != pdFALSE )
*     {
*         // Call the interrupt safe yield function here (actual function
*         // depends on the FreeRTOS port being used).
*     }
* }
*
```

Definition at line 978 of file timers.h.

7.25.2 Typedef Documentation

7.25.2.1 typedef void(* PendedFunction_t) (void *, uint32_t)

Definition at line 129 of file timers.h.

7.25.2.2 typedef void(* TimerCallbackFunction_t) (TimerHandle_t xTimer)

Definition at line 123 of file timers.h.

7.25.2.3 typedef void* TimerHandle_t

Type by which software timers are referenced. For example, a call to xTimerCreate() returns an TimerHandle↩ _t variable that can then be used to reference the subject timer in calls to other software timer API functions (for example, xTimerStart(), xTimerReset(), etc.).

Definition at line 118 of file timers.h.

7.25.3 Function Documentation

7.25.3.1 const char* pcTimerGetName (TimerHandle_t *xTimer*)

const char * const pcTimerGetName(TimerHandle_t xTimer);

Returns the name that was assigned to a timer when the timer was created.

Parameters

xTimer	The handle of the timer being queried.

Returns

The name assigned to the timer specified by the xTimer parameter.

7.25.3.2 void* pvTimerGetTimerID (const TimerHandle_t *xTimer*)

TimerHandle_t xTimerCreate(const char * const pcTimerName, TickType_t xTimerPeriodInTicks, UBaseType_t uxAutoReload, void * pvTimerID, TimerCallbackFunction_t pxCallbackFunction);

Creates a new software timer instance, and returns a handle by which the created software timer can be referenced.

Internally, within the FreeRTOS implementation, software timers use a block of memory, in which the timer data structure is stored. If a software timer is created using xTimerCreate() then the required memory is automatically dynamically allocated inside the xTimerCreate() function. (see http://www.freertos.org/a00111.↩html). If a software timer is created using xTimerCreateStatic() then the application writer must provide the memory that will get used by the software timer. xTimerCreateStatic() therefore allows a software timer to be created without using any dynamic memory allocation.

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerReset↩FromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Parameters

pcTimerName	A text name that is assigned to the timer. This is done purely to assist debugging. The kernel itself only ever references a timer by its handle, and never by its name.
xTimerPeriodInTicks	The timer period. The time is defined in tick periods so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xTimerPeriodInTicks should be set to 100. Alternatively, if the timer must expire after 500ms, then xPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
uxAutoReload	If uxAutoReload is set to pdTRUE then the timer will expire repeatedly with a frequency set by the xTimerPeriodInTicks parameter. If uxAutoReload is set to pdFALSE then the timer will be a one-shot timer and enter the dormant state after it expires.
pvTimerID	An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer.
pxCallbackFunction	The function to call when the timer expires. Callback functions must have the prototype defined by TimerCallbackFunction_t, which is "void vCallbackFunction(TimerHandle_t xTimer);".

Returns

If the timer is successfully created then a handle to the newly created timer is returned. If the timer cannot be created (because either there is insufficient FreeRTOS heap remaining to allocate the timer structures, or the timer period was set to 0) then NULL is returned.

Example usage:

```
* #define NUM_TIMERS 5
*
* // An array to hold handles to the created timers.
* TimerHandle_t xTimers[ NUM_TIMERS ];
*
* // An array to hold a count of the number of times each timer expires.
* int32_t lExpireCounters[ NUM_TIMERS ] = { 0 };
*
* // Define a callback function that will be used by multiple timer instances.
* // The callback function does nothing but count the number of times the
* // associated timer expires, and stop the timer once the timer has expired
* // 10 times.
* void vTimerCallback( TimerHandle_t pxTimer )
* {
* int32_t lArrayIndex;
* const int32_t xMaxExpiryCountBeforeStopping = 10;
*
*      // Optionally do something if the pxTimer parameter is NULL.
*      configASSERT( pxTimer );
*
*     // Which timer expired?
*     lArrayIndex = ( int32_t ) pvTimerGetTimerID( pxTimer );
*
*     // Increment the number of times that pxTimer has expired.
*     lExpireCounters[ lArrayIndex ] += 1;
*
*     // If the timer has expired 10 times then stop it from running.
*     if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping )
*     {
*         // Do not use a block time if calling a timer API function from a
*         // timer callback function, as doing so could cause a deadlock!
*         xTimerStop( pxTimer, 0 );
*     }
* }
*
* void main( void )
* {
* int32_t x;
*
*     // Create then start some timers.  Starting the timers before the scheduler
*     // has been started means the timers will start running immediately that
*     // the scheduler starts.
*     for( x = 0; x < NUM_TIMERS; x++ )
*     {
*         xTimers[ x ] = xTimerCreate(    "Timer",       // Just a text name, not used by the kernel.
*                                         ( 100 * x ),   // The timer period in ticks.
*                                         pdTRUE,        // The timers will auto-reload themselves when they e
*                                         ( void * ) x,  // Assign each timer a unique id equal to its array i
*                                         vTimerCallback // Each timer calls the same callback when it expires
*                                     );
*
*         if( xTimers[ x ] == NULL )
*         {
*             // The timer was not created.
*         }
*         else
*         {
*             // Start the timer.  No block time is specified, and even if one was
*             // it would be ignored because the scheduler has not yet been
*             // started.
*             if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
*             {
*                 // The timer could not be set into the Active state.
*             }
*         }
*     }
*
*     // ...
*     // Create tasks here.
*     // ...
*
*     // Starting the scheduler will start the timers running as they have already
*     // been set into the active state.
```

```
*     vTaskStartScheduler();
*
*     // Should not reach here.
*     for( ;; );
* }
*
```

TimerHandle_t xTimerCreateStatic(const char * const pcTimerName, TickType_t xTimerPeriodInTicks, UBase↩ Type_t uxAutoReload, void * pvTimerID, TimerCallbackFunction_t pxCallbackFunction, StaticTimer_t *pxTimer↩ Buffer);

Creates a new software timer instance, and returns a handle by which the created software timer can be referenced.

Internally, within the FreeRTOS implementation, software timers use a block of memory, in which the timer data structure is stored. If a software timer is created using xTimerCreate() then the required memory is automatically dynamically allocated inside the xTimerCreate() function. (see http://www.freertos.org/a00111.↩ html). If a software timer is created using xTimerCreateStatic() then the application writer must provide the memory that will get used by the software timer. xTimerCreateStatic() therefore allows a software timer to be created without using any dynamic memory allocation.

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerReset↩ FromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Parameters

pcTimerName	A text name that is assigned to the timer. This is done purely to assist debugging. The kernel itself only ever references a timer by its handle, and never by its name.
xTimerPeriodInTicks	The timer period. The time is defined in tick periods so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xTimerPeriodInTicks should be set to 100. Alternatively, if the timer must expire after 500ms, then xPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
uxAutoReload	If uxAutoReload is set to pdTRUE then the timer will expire repeatedly with a frequency set by the xTimerPeriodInTicks parameter. If uxAutoReload is set to pdFALSE then the timer will be a one-shot timer and enter the dormant state after it expires.
pvTimerID	An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer.
pxCallbackFunction	The function to call when the timer expires. Callback functions must have the prototype defined by TimerCallbackFunction_t, which is "void vCallbackFunction(TimerHandle_t xTimer);".
pxTimerBuffer	Must point to a variable of type StaticTimer_t, which will be then be used to hold the software timer's data structures, removing the need for the memory to be allocated dynamically.

Returns

If the timer is created then a handle to the created timer is returned. If pxTimerBuffer was NULL then NULL is returned.

Example usage:

```
*
* // The buffer used to hold the software timer's data structure.
```

```
* static StaticTimer_t xTimerBuffer;
*
* // A variable that will be incremented by the software timer's callback
* // function.
* UBaseType_t uxVariableToIncrement = 0;
*
* // A software timer callback function that increments a variable passed to
* // it when the software timer was created.  After the 5th increment the
* // callback function stops the software timer.
* static void prvTimerCallback( TimerHandle_t xExpiredTimer )
* {
* UBaseType_t *puxVariableToIncrement;
* BaseType_t xReturned;
*
*     // Obtain the address of the variable to increment from the timer ID.
*     puxVariableToIncrement = ( UBaseType_t * ) pvTimerGetTimerID( xExpiredTimer );
*
*     // Increment the variable to show the timer callback has executed.
*     ( *puxVariableToIncrement )++;
*
*     // If this callback has executed the required number of times, stop the
*     // timer.
*     if( *puxVariableToIncrement == 5 )
*     {
*         // This is called from a timer callback so must not block.
*         xTimerStop( xExpiredTimer, staticDONT_BLOCK );
*     }
* }
*
*
* void main( void )
* {
*     // Create the software time.  xTimerCreateStatic() has an extra parameter
*     // than the normal xTimerCreate() API function.  The parameter is a pointer
*     // to the StaticTimer_t structure that will hold the software timer
*     // structure.  If the parameter is passed as NULL then the structure will be
*     // allocated dynamically, just as if xTimerCreate() had been called.
*     xTimer = xTimerCreateStatic( "T1",             // Text name for the task.  Helps debugging only.  Not us
*                                  xTimerPeriod,     // The period of the timer in ticks.
*                                  pdTRUE,           // This is an auto-reload timer.
*                                  ( void * ) &uxVariableToIncrement,    // A variable incremented by the soft
*                                  prvTimerCallback, // The function to execute when the timer expires.
*                                  &xTimerBuffer );  // The buffer that will hold the software timer structure
*
*     // The scheduler has not started yet so a block time is not used.
*     xReturned = xTimerStart( xTimer, 0 );
*
*     // ...
*     // Create tasks here.
*     // ...
*
*     // Starting the scheduler will start the timers running as they have already
*     // been set into the active state.
*     vTaskStartScheduler();
*
*     // Should not reach here.
*     for( ;; );
* }
*
```

void *pvTimerGetTimerID(TimerHandle_t xTimer);

Returns the ID assigned to the timer.

IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreated() that was used to create the timer, and by calling the vTimerSetTimerID() API function.

If the same callback function is assigned to multiple timers then the timer ID can be used as time specific (timer local) storage.

Parameters

xTimer	The timer being queried.

Returns

The ID assigned to the timer being queried.

Example usage:

See the xTimerCreate() API function example usage scenario.

7.25.3.3 void vTimerSetTimerID (TimerHandle_t *xTimer,* void * *pvNewID*)

void vTimerSetTimerID(TimerHandle_t xTimer, void *pvNewID);

Sets the ID assigned to the timer.

IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreated() that was used to create the timer.

If the same callback function is assigned to multiple timers then the timer ID can be used as time specific (timer local) storage.

Parameters

xTimer	The timer being updated.
pvNewID	The ID to assign to the timer.

Example usage:

See the xTimerCreate() API function example usage scenario.

7.25.3.4 BaseType_t xTimerCreateTimerTask (void)

7.25.3.5 BaseType_t xTimerGenericCommand (TimerHandle_t *xTimer,* const BaseType_t *xCommandID,* const TickType_t *xOptionalValue,* BaseType_t *const *pxHigherPriorityTaskWoken,* const TickType_t *xTicksToWait*)

7.25.3.6 TickType_t xTimerGetExpiryTime (TimerHandle_t *xTimer*)

TickType_t xTimerGetExpiryTime(TimerHandle_t xTimer);

Returns the time in ticks at which the timer will expire. If this is less than the current tick count then the expiry time has overflowed from the current time.

Parameters

xTimer	The handle of the timer being queried.

Returns

If the timer is running then the time in ticks at which the timer will next expire is returned. If the timer is not running then the return value is undefined.

7.25.3.7 TickType_t xTimerGetPeriod (TimerHandle_t *xTimer*)

TickType_t xTimerGetPeriod(TimerHandle_t xTimer);

Returns the period of a timer.

Parameters

xTimer	The handle of the timer being queried.

Returns

The period of the timer in ticks.

7.25.3.8 TaskHandle_t xTimerGetTimerDaemonTaskHandle (void)

TaskHandle_t xTimerGetTimerDaemonTaskHandle(void);

Simply returns the handle of the timer service/daemon task. It it not valid to call xTimerGetTimerDaemonTaskHandle() before the scheduler has been started.

7.25.3.9 BaseType_t xTimerIsTimerActive (TimerHandle_t *xTimer*)

BaseType_t xTimerIsTimerActive(TimerHandle_t xTimer);

Queries a timer to see if it is active or dormant.

A timer will be dormant if: 1) It has been created but not started, or 2) It is an expired one-shot timer that has not been restarted.

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Parameters

xTimer	The timer being queried.

Returns

pdFALSE will be returned if the timer is dormant. A value other than pdFALSE will be returned if the timer is active.

Example usage:

```
* // This function assumes xTimer has already been created.
* void vAFunction( TimerHandle_t xTimer )
* {
*     if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and equivalently "if( xTimerIsTimerActi
*     {
*         // xTimer is active, do something.
*     }
*     else
*     {
*         // xTimer is not active, do something else.
*     }
* }
*
```

7.25.3.10 BaseType_t xTimerPendFunctionCall (PendedFunction_t *xFunctionToPend,* void * *pvParameter1,* uint32_t *ulParameter2,* TickType_t *xTicksToWait*)

BaseType_t xTimerPendFunctionCall(PendedFunction_t xFunctionToPend, void *pvParameter1, uint32_t ul↩Parameter2, TickType_t xTicksToWait);

Used to defer the execution of a function to the RTOS daemon task (the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

Parameters

xFunctionToPend	The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction_t prototype.
pvParameter1	The value of the callback function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, unsigned longs can be cast to a void *, or the void * can be used to point to a structure.
ulParameter2	The value of the callback function's second parameter.
xTicksToWait	Calling this function will result in a message being sent to the timer daemon task on a queue. xTicksToWait is the amount of time the calling task should remain in the Blocked state (so not using any processing time) for space to become available on the timer queue if the queue is found to be full.

Returns

pdPASS is returned if the message was successfully sent to the timer daemon task, otherwise pdFALSE is returned.

7.25.3.11 BaseType_t xTimerPendFunctionCallFromISR (PendedFunction_t *xFunctionToPend,* void * *pvParameter1,* uint32_t *ulParameter2,* BaseType_t * *pxHigherPriorityTaskWoken*)

BaseType_t xTimerPendFunctionCallFromISR(PendedFunction_t xFunctionToPend, void *pvParameter1, uint32↩_t ulParameter2, BaseType_t *pxHigherPriorityTaskWoken);

Used from application interrupt service routines to defer the execution of a function to the RTOS daemon task (the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

Ideally an interrupt service routine (ISR) is kept as short as possible, but sometimes an ISR either has a lot of processing to do, or needs to perform processing that is not deterministic. In these cases xTimerPendFunction↩CallFromISR() can be used to defer processing of a function to the RTOS daemon task.

A mechanism is provided that allows the interrupt to return directly to the task that will subsequently execute the pended callback function. This allows the callback function to execute contiguously in time with the interrupt - just as if the callback had executed in the interrupt itself.

Parameters

xFunctionToPend	The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction_t prototype.
pvParameter1	The value of the callback function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, unsigned longs can be cast to a void *, or the void * can be used to point to a structure.
ulParameter2	The value of the callback function's second parameter.
pxHigherPriorityTaskWoken	As mentioned above, calling this function will result in a message being sent to the timer daemon task. If the priority of the timer daemon task (which is set using configTIMER_TASK_PRIORITY in FreeRTOSConfig.h) is higher than the priority of the currently running task (the task the interrupt interrupted) then *pxHigherPriorityTaskWoken will be set to pdTRUE within xTimerPendFunctionCallFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason *pxHigherPriorityTaskWoken must be initialised to pdFALSE. See the example code below.

Returns

pdPASS is returned if the message was successfully sent to the timer daemon task, otherwise pdFALSE is returned.

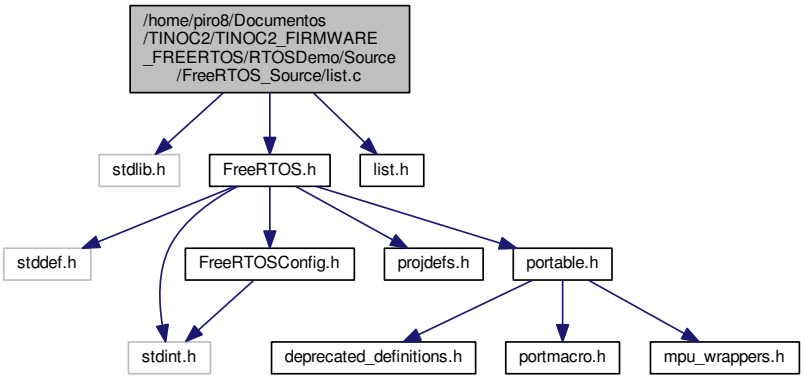
Example usage:

```
*
*   // The callback function that will execute in the context of the daemon task.
*  // Note callback functions must all use this same prototype.
*  void vProcessInterface( void *pvParameter1, uint32_t ulParameter2 )
*   {
*       BaseType_t xInterfaceToService;
*
*       // The interface that requires servicing is passed in the second
*      // parameter.  The first parameter is not used in this case.
*       xInterfaceToService = ( BaseType_t ) ulParameter2;
*
*       // ...Perform the processing here...
*   }
*
*   // An ISR that receives data packets from multiple interfaces
*  void vAnISR( void )
*   {
*       BaseType_t xInterfaceToService, xHigherPriorityTaskWoken;
*
*       // Query the hardware to determine which interface needs processing.
*       xInterfaceToService = prvCheckInterfaces();
*
*      // The actual processing is to be deferred to a task.  Request the
*      // vProcessInterface() callback function is executed, passing in the
*       // number of the interface that needs processing.  The interface to
*       // service is passed in the second parameter.  The first parameter is
*       // not used in this case.
*       xHigherPriorityTaskWoken = pdFALSE;
*       xTimerPendFunctionCallFromISR( vProcessInterface, NULL, ( uint32_t ) xInterfaceToService, &xHigherPric
*
*       // If xHigherPriorityTaskWoken is now set to pdTRUE then a context
*       // switch should be requested.  The macro used is port specific and will
*       // be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
*       // the documentation page for the port being used.
*       portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
*
*   }
*
```

7.26 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/FreeRTOS_Source/list.c File Reference

```
#include <stdlib.h>
#include "FreeRTOS.h"
#include "list.h"
```

Include dependency graph for list.c:



Functions

- void vListInitialise (List_t ∗const pxList)
- void vListInitialiseItem (ListItem_t ∗const pxItem)
- void vListInsertEnd (List_t ∗const pxList, ListItem_t ∗const pxNewListItem)
- void vListInsert (List_t ∗const pxList, ListItem_t ∗const pxNewListItem)
- UBaseType_t uxListRemove (ListItem_t ∗const pxItemToRemove)

7.26.1 Function Documentation

7.26.1.1 UBaseType_t uxListRemove (ListItem_t ∗const *pxItemToRemove*)

Definition at line 212 of file list.c.

7.26.1.2 void vListInitialise (List_t ∗const *pxList*)

Definition at line 79 of file list.c.

7.26.1.3 void vListInitialiseItem (ListItem_t ∗const *pxItem*)

Definition at line 104 of file list.c.

7.26.1.4 void vListInsert (List_t *const *pxList,* ListItem_t *const *pxNewListItem*)

Definition at line 145 of file list.c.

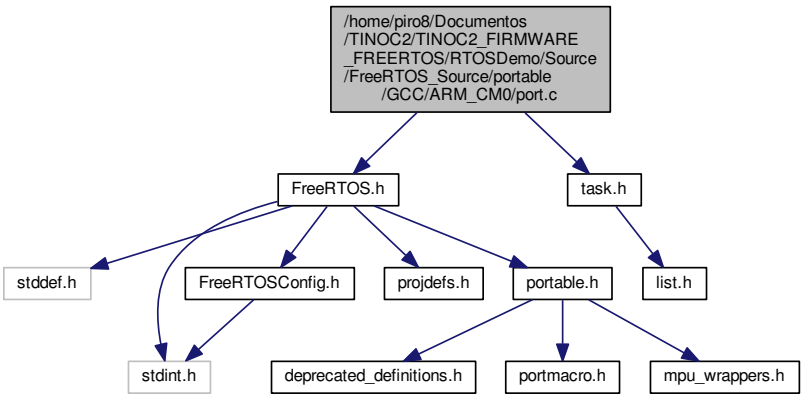
7.26.1.5 void vListInsertEnd (List_t *const *pxList,* ListItem_t *const *pxNewListItem*)

Definition at line 116 of file list.c.

7.27 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/portable/GCC/ARM_CM0/port.c File Reference

```
#include "FreeRTOS.h"
#include "task.h"
```

Include dependency graph for port.c:



Macros

- #define portNVIC_SYSTICK_CTRL ((volatile uint32_t *) 0xe000e010)
- #define portNVIC_SYSTICK_LOAD ((volatile uint32_t *) 0xe000e014)
- #define portNVIC_INT_CTRL ((volatile uint32_t *) 0xe000ed04)
- #define portNVIC_SYSPRI2 ((volatile uint32_t *) 0xe000ed20)
- #define portNVIC_SYSTICK_CLK 0x00000004
- #define portNVIC_SYSTICK_INT 0x00000002
- #define portNVIC_SYSTICK_ENABLE 0x00000001
- #define portNVIC_PENDSVSET 0x10000000
- #define portMIN_INTERRUPT_PRIORITY (255UL)
- #define portNVIC_PENDSV_PRI (portMIN_INTERRUPT_PRIORITY << 16UL)
- #define portNVIC_SYSTICK_PRI (portMIN_INTERRUPT_PRIORITY << 24UL)
- #define portINITIAL_XPSR (0x01000000)
- #define portTASK_RETURN_ADDRESS prvTaskExitError

Functions

- static void prvSetupTimerInterrupt (void)
- void xPortPendSVHandler (void xPortSysTickHandler void)
- static void prvTaskExitError (void)
- void vPortSVCHandler (void)
- void vPortStartFirstTask (void)
- BaseType_t xPortStartScheduler (void)
- void vPortEndScheduler (void)
- void vPortYield (void)
- void vPortEnterCritical (void)
- void vPortExitCritical (void)
- uint32_t ulSetInterruptMaskFromISR (void)
- void vClearInterruptMaskFromISR (uint32_t ulMask)
- void xPortPendSVHandler (void)
- void xPortSysTickHandler (void)

Variables

- static UBaseType_t uxCriticalNesting = 0xaaaaaaaa

7.27.1 Macro Definition Documentation

7.27.1.1 #define portINITIAL_XPSR (0x01000000)

Definition at line 92 of file port.c.

7.27.1.2 #define portMIN_INTERRUPT_PRIORITY (255UL)

Definition at line 87 of file port.c.

7.27.1.3 #define portNVIC_INT_CTRL ((volatile uint32_t *) 0xe000ed04)

Definition at line 81 of file port.c.

7.27.1.4 #define portNVIC_PENDSV_PRI (portMIN_INTERRUPT_PRIORITY << 16UL)

Definition at line 88 of file port.c.

7.27.1.5 #define portNVIC_PENDSVSET 0x10000000

Definition at line 86 of file port.c.

7.27.1.6 #define portNVIC_SYSPRI2 ((volatile uint32_t *) 0xe000ed20)

Definition at line 82 of file port.c.

7.27.1.7 #define portNVIC_SYSTICK_CLK 0x00000004

Definition at line 83 of file port.c.

7.27.1.8 #define portNVIC_SYSTICK_CTRL ((volatile uint32_t *) 0xe000e010)

Definition at line 79 of file port.c.

7.27.1.9 #define portNVIC_SYSTICK_ENABLE 0x00000001

Definition at line 85 of file port.c.

7.27.1.10 #define portNVIC_SYSTICK_INT 0x00000002

Definition at line 84 of file port.c.

7.27.1.11 #define portNVIC_SYSTICK_LOAD ((volatile uint32_t *) 0xe000e014)

Definition at line 80 of file port.c.

7.27.1.12 #define portNVIC_SYSTICK_PRI (portMIN_INTERRUPT_PRIORITY << 24UL)

Definition at line 89 of file port.c.

7.27.1.13 #define portTASK_RETURN_ADDRESS prvTaskExitError

Definition at line 100 of file port.c.

7.27.2 Function Documentation

7.27.2.1 void prvSetupTimerInterrupt (void) [static]

Definition at line 364 of file port.c.

7.27.2.2 static void prvTaskExitError (void) [static]

Definition at line 152 of file port.c.

7.27.2.3 uint32_t ulSetInterruptMaskFromISR (void)

Definition at line 270 of file port.c.

7.27.2.4 void vClearInterruptMaskFromISR (uint32_t *ulMask*)

Definition at line 283 of file port.c.

7.27.2.5 void vPortEndScheduler (void)

Definition at line 230 of file port.c.

7.27.2.6 void vPortEnterCritical (void)

Definition at line 250 of file port.c.

7.27.2.7 void vPortExitCritical (void)

Definition at line 259 of file port.c.

7.27.2.8 void vPortStartFirstTask (void)

Definition at line 173 of file port.c.

7.27.2.9 void vPortSVCHandler (void)

Definition at line 166 of file port.c.

7.27.2.10 void vPortYield (void)

Definition at line 238 of file port.c.

7.27.2.11 void xPortPendSVHandler (void **xPortSysTickHandler** *void*)

Definition at line 115 of file port.c.

7.27.2.12 void xPortPendSVHandler (void)

Definition at line 295 of file port.c.

7.27.2.13 BaseType_t xPortStartScheduler (void)

Definition at line 203 of file port.c.

7.27.2.14 void xPortSysTickHandler (void)

Definition at line 343 of file port.c.

7.27.3 Variable Documentation

7.27.3.1 UBaseType_t uxCriticalNesting = 0xaaaaaaaa [static]

Definition at line 105 of file port.c.

7.28 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/portable/GCC/ARM_CM0/portmacro.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define portCHAR char
- #define portFLOAT float
- #define portDOUBLE double
- #define portLONG long
- #define portSHORT short
- #define portSTACK_TYPE uint32_t
- #define portBASE_TYPE long
- #define portMAX_DELAY (TickType_t) 0xffffffffUL
- #define portTICK_TYPE_IS_ATOMIC 1
- #define portSTACK_GROWTH (-1)
- #define portTICK_PERIOD_MS ((TickType_t) 1000 / configTICK_RATE_HZ)
- #define portBYTE_ALIGNMENT 8
- #define portNVIC_INT_CTRL_REG (* ((volatile uint32_t *) 0xe000ed04))
- #define portNVIC_PENDSVSET_BIT (1UL << 28UL)
- #define portYIELD() vPortYield()
- #define portEND_SWITCHING_ISR(xSwitchRequired) if(xSwitchRequired) portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT
- #define portYIELD_FROM_ISR(x) portEND_SWITCHING_ISR(x)
- #define portSET_INTERRUPT_MASK_FROM_ISR() ulSetInterruptMaskFromISR()
- #define portCLEAR_INTERRUPT_MASK_FROM_ISR(x) vClearInterruptMaskFromISR(x)
- #define portDISABLE_INTERRUPTS() __asm volatile (" cpsid i ")
- #define portENABLE_INTERRUPTS() __asm volatile (" cpsie i ")
- #define portENTER_CRITICAL() vPortEnterCritical()
- #define portEXIT_CRITICAL() vPortExitCritical()
- #define portTASK_FUNCTION_PROTO(vFunction, pvParameters) void vFunction(void *pvParameters)
- #define portTASK_FUNCTION(vFunction, pvParameters) void vFunction(void *pvParameters)
- #define portNOP()

Typedefs

- typedef portSTACK_TYPE StackType_t
- typedef long BaseType_t
- typedef unsigned long UBaseType_t
- typedef uint32_t TickType_t

Functions

- void vPortYield (void)
- void vPortEnterCritical (void)
- void vPortExitCritical (void)
- uint32_t ulSetInterruptMaskFromISR (void) __attribute__((naked))
- void vClearInterruptMaskFromISR (uint32_t ulMask) __attribute__((naked))

7.28.1 Macro Definition Documentation

7.28.1.1 #define portBASE_TYPE long

Definition at line 95 of file portmacro.h.

7.28.1.2 #define portBYTE_ALIGNMENT 8

Definition at line 117 of file portmacro.h.

7.28.1.3 #define portCHAR char

Definition at line 89 of file portmacro.h.

7.28.1.4 #define portCLEAR_INTERRUPT_MASK_FROM_ISR(*x*) vClearInterruptMaskFromISR(x)

Definition at line 138 of file portmacro.h.

7.28.1.5 #define portDISABLE_INTERRUPTS( ) __asm volatile (" cpsid i ")

Definition at line 139 of file portmacro.h.

7.28.1.6 #define portDOUBLE double

Definition at line 91 of file portmacro.h.

7.28.1.7 #define portENABLE_INTERRUPTS( ) __asm volatile (" cpsie i ")

Definition at line 140 of file portmacro.h.

7.28.1.8 #define portEND_SWITCHING_ISR(*xSwitchRequired*) if(xSwitchRequired) portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT

Definition at line 126 of file portmacro.h.

7.28.1.9 #define portENTER_CRITICAL( ) vPortEnterCritical()

Definition at line 141 of file portmacro.h.

7.28.1.10 #define portEXIT_CRITICAL( ) vPortExitCritical()

Definition at line 142 of file portmacro.h.

7.28.1.11 #define portFLOAT float

Definition at line 90 of file portmacro.h.

7.28.1.12 #define portLONG long

Definition at line 92 of file portmacro.h.

7.28.1.13 #define portMAX_DELAY (TickType_t) 0xffffffffUL

Definition at line 106 of file portmacro.h.

7.28.1.14 #define portNOP( )

Definition at line 150 of file portmacro.h.

7.28.1.15 #define portNVIC_INT_CTRL_REG (* ((volatile uint32_t *) 0xe000ed04))

Definition at line 123 of file portmacro.h.

7.28.1.16 #define portNVIC_PENDSVSET_BIT (1UL << 28UL)

Definition at line 124 of file portmacro.h.

7.28.1.17 #define portSET_INTERRUPT_MASK_FROM_ISR( ) uISetInterruptMaskFromISR()

Definition at line 137 of file portmacro.h.

7.28.1.18 #define portSHORT short

Definition at line 93 of file portmacro.h.

7.28.1.19 #define portSTACK_GROWTH (-1)

Definition at line 115 of file portmacro.h.

7.28.1.20 #define portSTACK_TYPE uint32_t

Definition at line 94 of file portmacro.h.

7.28.1.21 #define portTASK_FUNCTION(*vFunction, pvParameters*) void vFunction(void ∗pvParameters)

Definition at line 148 of file portmacro.h.

7.28.1.22 #define portTASK_FUNCTION_PROTO(*vFunction, pvParameters*) void vFunction(void ∗pvParameters)

Definition at line 147 of file portmacro.h.

7.28.1.23 #define portTICK_PERIOD_MS ((TickType_t) 1000 / configTICK_RATE_HZ)

Definition at line 116 of file portmacro.h.

7.28.1.24 #define portTICK_TYPE_IS_ATOMIC 1

Definition at line 110 of file portmacro.h.

7.28.1.25 #define portYIELD( ) vPortYield()

Definition at line 125 of file portmacro.h.

7.28.1.26 #define portYIELD_FROM_ISR(*x*) portEND_SWITCHING_ISR(x)

Definition at line 127 of file portmacro.h.

7.28.2 Typedef Documentation

7.28.2.1 typedef long BaseType_t

Definition at line 98 of file portmacro.h.

7.28.2.2 typedef portSTACK_TYPE StackType_t

Definition at line 97 of file portmacro.h.

7.28.2.3 typedef uint32_t TickType_t

Definition at line 105 of file portmacro.h.

7.28.2.4 typedef unsigned long UBaseType_t

Definition at line 99 of file portmacro.h.

7.28.3 Function Documentation

7.28.3.1 uint32_t ulSetInterruptMaskFromISR (void)

Definition at line 270 of file port.c.

7.28.3.2 void vClearInterruptMaskFromISR (uint32_t *ulMask*)

Definition at line 283 of file port.c.

7.28.3.3 void vPortEnterCritical (void)

Definition at line 250 of file port.c.

7.28.3.4 void vPortExitCritical (void)

Definition at line 259 of file port.c.

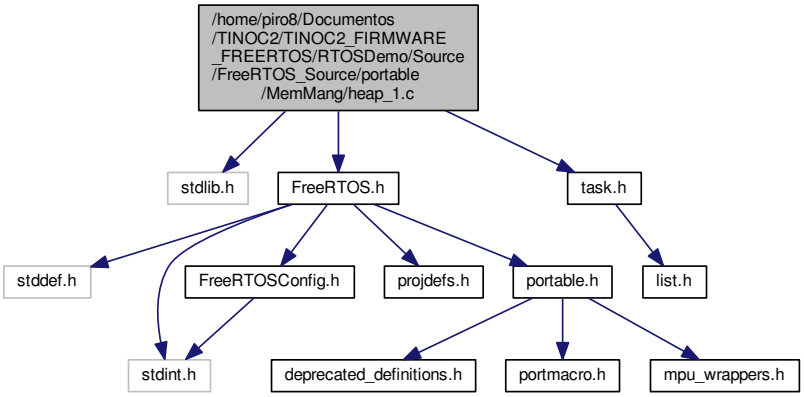
7.28.3.5 void vPortYield (void)

Definition at line 238 of file port.c.

7.29 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/FreeRTOS_Source/portable/MemMang/heap_1.c File Reference

```
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
```

Include dependency graph for heap_1.c:



Macros

- #define MPU_WRAPPERS_INCLUDED_FROM_API_FILE
- #define configADJUSTED_HEAP_SIZE (configTOTAL_HEAP_SIZE - portBYTE_ALIGNMENT)

Functions

- void ∗ pvPortMalloc (size_t xWantedSize)
- void vPortFree (void ∗pv)
- void vPortInitialiseBlocks (void)
- size_t xPortGetFreeHeapSize (void)

Variables

- static uint8_t ucHeap [configTOTAL_HEAP_SIZE]
- static size_t xNextFreeByte = (size_t) 0

7.29.1 Macro Definition Documentation

7.29.1.1 #define configADJUSTED_HEAP_SIZE (configTOTAL_HEAP_SIZE - portBYTE_ALIGNMENT)

Definition at line 95 of file heap_1.c.

7.29.1.2 #define MPU_WRAPPERS_INCLUDED_FROM_API_FILE

Definition at line 83 of file heap_1.c.

7.29.2 Function Documentation

7.29.2.1 void* pvPortMalloc (size_t *xWantedSize*)

Definition at line 111 of file heap_1.c.

7.29.2.2 void vPortFree (void * *pv*)

Definition at line 163 of file heap_1.c.

7.29.2.3 void vPortInitialiseBlocks (void)

Definition at line 175 of file heap_1.c.

7.29.2.4 size_t xPortGetFreeHeapSize (void)

Definition at line 182 of file heap_1.c.

7.29.3 Variable Documentation

7.29.3.1 uint8_t ucHeap[configTOTAL_HEAP_SIZE] [static]

Definition at line 104 of file heap_1.c.

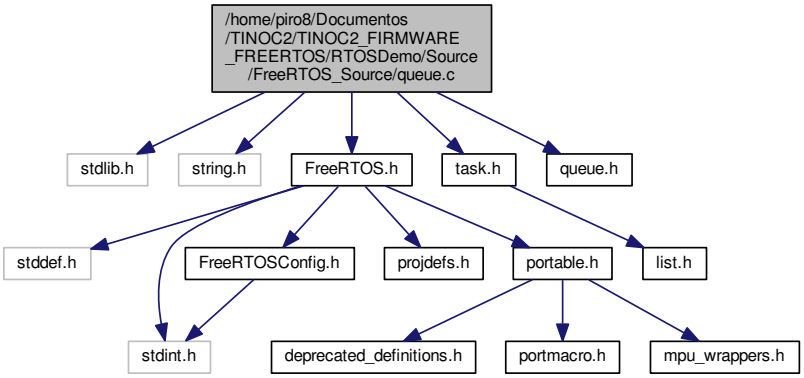
7.29.3.2 size_t xNextFreeByte = (size_t) 0 [static]

Definition at line 107 of file heap_1.c.

7.30 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/FreeRTOS_Source/queue.c File Reference

```
#include <stdlib.h>
#include <string.h>
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
```

Include dependency graph for queue.c:



Classes

- struct QueueDefinition

Macros

- #define MPU_WRAPPERS_INCLUDED_FROM_API_FILE
- #define queueUNLOCKED ((int8_t) -1)
- #define queueLOCKED_UNMODIFIED ((int8_t) 0)
- #define pxMutexHolder pcTail
- #define uxQueueType pcHead
- #define queueQUEUE_IS_MUTEX NULL
- #define queueSEMAPHORE_QUEUE_ITEM_LENGTH ((UBaseType_t) 0)
- #define queueMUTEX_GIVE_BLOCK_TIME ((TickType_t) 0U)
- #define queueYIELD_IF_USING_PREEMPTION()
- #define prvLockQueue(pxQueue)

Typedefs

- typedef struct QueueDefinition xQUEUE
- typedef xQUEUE Queue_t

Functions

- static void prvUnlockQueue (Queue_t *const pxQueue) PRIVILEGED_FUNCTION
- static BaseType_t prvIsQueueEmpty (const Queue_t *pxQueue) PRIVILEGED_FUNCTION
- static BaseType_t prvIsQueueFull (const Queue_t *pxQueue) PRIVILEGED_FUNCTION
- static BaseType_t prvCopyDataToQueue (Queue_t *const pxQueue, const void *pvItemToQueue, const BaseType_t xPosition) PRIVILEGED_FUNCTION
- static void prvCopyDataFromQueue (Queue_t *const pxQueue, void *const pvBuffer) PRIVILEGED_FUNCTION
- static void prvInitialiseNewQueue (const UBaseType_t uxQueueLength, const UBaseType_t uxItemSize, uint8_t *pucQueueStorage, const uint8_t ucQueueType, Queue_t *pxNewQueue) PRIVILEGED_FUNCTION
- BaseType_t xQueueGenericReset (QueueHandle_t xQueue, BaseType_t xNewQueue)
- BaseType_t xQueueGenericSend (QueueHandle_t xQueue, const void *const pvItemToQueue, TickType_t xTicksToWait, const BaseType_t xCopyPosition)
- BaseType_t xQueueGenericSendFromISR (QueueHandle_t xQueue, const void *const pvItemToQueue, BaseType_t *const pxHigherPriorityTaskWoken, const BaseType_t xCopyPosition)
- BaseType_t xQueueGiveFromISR (QueueHandle_t xQueue, BaseType_t *const pxHigherPriorityTaskWoken)
- BaseType_t xQueueGenericReceive (QueueHandle_t xQueue, void *const pvBuffer, TickType_t xTicksToWait, const BaseType_t xJustPeeking)
- BaseType_t xQueueReceiveFromISR (QueueHandle_t xQueue, void *const pvBuffer, BaseType_t *const pxHigherPriorityTaskWoken)
- BaseType_t xQueuePeekFromISR (QueueHandle_t xQueue, void *const pvBuffer)
- UBaseType_t uxQueueMessagesWaiting (const QueueHandle_t xQueue)
- UBaseType_t uxQueueSpacesAvailable (const QueueHandle_t xQueue)
- UBaseType_t uxQueueMessagesWaitingFromISR (const QueueHandle_t xQueue)
- void vQueueDelete (QueueHandle_t xQueue)
- BaseType_t xQueueIsQueueEmptyFromISR (const QueueHandle_t xQueue)
- BaseType_t xQueueIsQueueFullFromISR (const QueueHandle_t xQueue)

7.30.1 Macro Definition Documentation

7.30.1.1 #define MPU_WRAPPERS_INCLUDED_FROM_API_FILE

Definition at line 76 of file queue.c.

7.30.1.2 #define prvLockQueue(*pxQueue*)

Value:

```
taskENTER_CRITICAL();                                   \
    {                                                               \
        if( ( pxQueue )->cRxLock == queueUNLOCKED )                   \
        {                                                           \
            ( pxQueue )->cRxLock = queueLOCKED_UNMODIFIED;          \
        }                                                           \
        if( ( pxQueue )->cTxLock == queueUNLOCKED )                   \
        {                                                           \
            ( pxQueue )->cTxLock = queueLOCKED_UNMODIFIED;          \
        }                                                           \
    }                                                               \
    taskEXIT_CRITICAL()
```

Definition at line 264 of file queue.c.

7.30.1.3 #define pxMutexHolder pcTail

Definition at line 108 of file queue.c.

7.30.1.4 #define queueLOCKED_UNMODIFIED ((int8_t) 0)

Definition at line 95 of file queue.c.

7.30.1.5 #define queueMUTEX_GIVE_BLOCK_TIME ((TickType_t) 0U)

Definition at line 115 of file queue.c.

7.30.1.6 #define queueQUEUE_IS_MUTEX NULL

Definition at line 110 of file queue.c.

7.30.1.7 #define queueSEMAPHORE_QUEUE_ITEM_LENGTH ((UBaseType_t) 0)

Definition at line 114 of file queue.c.

7.30.1.8 #define queueUNLOCKED ((int8_t) -1)

Definition at line 94 of file queue.c.

7.30.1.9 #define queueYIELD_IF_USING_PREEMPTION( )

Definition at line 120 of file queue.c.

7.30.1.10 #define uxQueueType pcHead

Definition at line 109 of file queue.c.

7.30.2 Typedef Documentation

7.30.2.1 typedef xQUEUE Queue_t

Definition at line 169 of file queue.c.

7.30.2.2 typedef struct QueueDefinition xQUEUE

7.30.3 Function Documentation

7.30.3.1 static void prvCopyDataFromQueue (Queue_t *const *pxQueue,* void *const *pvBuffer*) [static]

Definition at line 1776 of file queue.c.

7.30.3.2 static BaseType_t prvCopyDataToQueue (Queue_t *const *pxQueue,* const void * *pvItemToQueue,* const BaseType_t *xPosition*) [static]

Definition at line 1697 of file queue.c.

7.30.3.3 static void prvInitialiseNewQueue (const UBaseType_t *uxQueueLength,* const UBaseType_t *uxItemSize,* uint8_t * *pucQueueStorage,* const uint8_t *ucQueueType,* Queue_t * *pxNewQueue*) [static]

Definition at line 432 of file queue.c.

7.30.3.4 static BaseType_t prvIsQueueEmpty (const Queue_t * *pxQueue*) [static]

Definition at line 1914 of file queue.c.

7.30.3.5 static BaseType_t prvIsQueueFull (const Queue_t * *pxQueue*) [static]

Definition at line 1953 of file queue.c.

7.30.3.6 static void prvUnlockQueue (Queue_t *const *pxQueue*) [static]

Definition at line 1794 of file queue.c.

7.30.3.7 UBaseType_t uxQueueMessagesWaiting (const QueueHandle_t *xQueue*)

Definition at line 1579 of file queue.c.

7.30.3.8 UBaseType_t uxQueueMessagesWaitingFromISR (const QueueHandle_t *xQueue*)

Definition at line 1613 of file queue.c.

7.30.3.9 UBaseType_t uxQueueSpacesAvailable (const QueueHandle_t *xQueue*)

Definition at line 1595 of file queue.c.

7.30.3.10 void vQueueDelete (QueueHandle_t *xQueue*)

Definition at line 1625 of file queue.c.

7.30.3.11 BaseType_t xQueueGenericReceive (QueueHandle_t *xQueue*, void ∗const *pvBuffer*, TickType_t *xTicksToWait*, const BaseType_t *xJustPeeking*)

Definition at line 1237 of file queue.c.

7.30.3.12 BaseType_t xQueueGenericReset (QueueHandle_t *xQueue*, BaseType_t *xNewQueue*)

Definition at line 279 of file queue.c.

7.30.3.13 BaseType_t xQueueGenericSend (QueueHandle_t *xQueue*, const void ∗const *pvItemToQueue*, TickType_t *xTicksToWait*, const BaseType_t *xCopyPosition*)

Definition at line 723 of file queue.c.

7.30.3.14 BaseType_t xQueueGenericSendFromISR (QueueHandle_t *xQueue*, const void ∗const *pvItemToQueue*, BaseType_t ∗const *pxHigherPriorityTaskWoken*, const BaseType_t *xCopyPosition*)

Definition at line 921 of file queue.c.

7.30.3.15 BaseType_t xQueueGiveFromISR (QueueHandle_t *xQueue*, BaseType_t ∗const *pxHigherPriorityTaskWoken*)

Definition at line 1072 of file queue.c.

7.30.3.16 BaseType_t xQueueIsQueueEmptyFromISR (const QueueHandle_t *xQueue*)

Definition at line 1935 of file queue.c.

7.30.3.17 BaseType_t xQueueIsQueueFullFromISR (const QueueHandle_t *xQueue*)

Definition at line 1974 of file queue.c.

7.30.3.18 BaseType_t xQueuePeekFromISR (QueueHandle_t *xQueue*, void ∗const *pvBuffer*)

Definition at line 1525 of file queue.c.

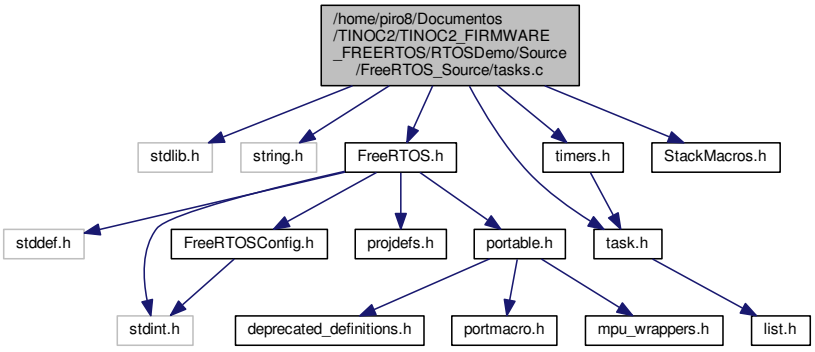
7.30.3.19 BaseType_t xQueueReceiveFromISR (QueueHandle_t *xQueue*, void ∗const *pvBuffer*, BaseType_t ∗const *pxHigherPriorityTaskWoken*)

Definition at line 1434 of file queue.c.

7.31 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/tasks.c File Reference

```
#include <stdlib.h>
#include <string.h>
#include "FreeRTOS.h"
#include "task.h"
#include "timers.h"
#include "StackMacros.h"
```

Include dependency graph for tasks.c:



Classes

- struct tskTaskControlBlock

Macros

- #define MPU_WRAPPERS_INCLUDED_FROM_API_FILE
- #define taskYIELD_IF_USING_PREEMPTION()
- #define taskNOT_WAITING_NOTIFICATION ((uint8_t) 0)
- #define taskWAITING_NOTIFICATION ((uint8_t) 1)
- #define taskNOTIFICATION_RECEIVED ((uint8_t) 2)
- #define tskSTACK_FILL_BYTE (0xa5U)
- #define tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE (((configSUPPORT_STATIC_ALLOCATION == 1) && (configSUPPORT_DYNAMIC_ALLOCATION == 1)) || (portUSING_MPU_WRAPPERS == 1))
- #define tskDYNAMICALLY_ALLOCATED_STACK_AND_TCB ((uint8_t) 0)
- #define tskSTATICALLY_ALLOCATED_STACK_ONLY ((uint8_t) 1)
- #define tskSTATICALLY_ALLOCATED_STACK_AND_TCB ((uint8_t) 2)
- #define tskBLOCKED_CHAR ('B')

- #define tskREADY_CHAR ('R')
- #define tskDELETED_CHAR ('D')
- #define tskSUSPENDED_CHAR ('S')
- #define taskRECORD_READY_PRIORITY(uxPriority)
- #define taskSELECT_HIGHEST_PRIORITY_TASK()
- #define taskRESET_READY_PRIORITY(uxPriority)
- #define portRESET_READY_PRIORITY(uxPriority, uxTopReadyPriority)
- #define taskSWITCH_DELAYED_LISTS()
- #define prvAddTaskToReadyList(pxTCB)
- #define prvGetTCBFromHandle(pxHandle) (((pxHandle) == NULL) ? (TCB_t *) pxCurrentTCB : (TCB_t *) (pxHandle))
- #define taskEVENT_LIST_ITEM_VALUE_IN_USE 0x80000000UL

Typedefs

- typedef struct tskTaskControlBlock tskTCB
- typedef tskTCB TCB_t

Functions

- static void prvInitialiseTaskLists (static void prvCheckTasksWaitingTermination void)
- static void prvAddNewTaskToReadyList (TCB_t *pxNewTCB)
- void vTaskStartScheduler (void)
- void vTaskEndScheduler (void)
- void vTaskSuspendAll (void)
- BaseType_t xTaskResumeAll (void)
- TickType_t xTaskGetTickCount (void)
- TickType_t xTaskGetTickCountFromISR (void)
- UBaseType_t uxTaskGetNumberOfTasks (void)
- char * pcTaskGetName (TaskHandle_t xTaskToQuery)
- BaseType_t xTaskIncrementTick (void)
- void vTaskSwitchContext (void)
- void vTaskPlaceOnEventList (List_t *const pxEventList, const TickType_t xTicksToWait)
- void vTaskPlaceOnUnorderedEventList (List_t *pxEventList, const TickType_t xItemValue, const TickType_t xTicksToWait)
- BaseType_t xTaskRemoveFromEventList (const List_t *const pxEventList)
- BaseType_t xTaskRemoveFromUnorderedEventList (ListItem_t *pxEventListItem, const TickType_t xItem↩Value)
- void vTaskSetTimeOutState (TimeOut_t *const pxTimeOut)
- BaseType_t xTaskCheckForTimeOut (TimeOut_t *const pxTimeOut, TickType_t *const pxTicksToWait)
- void vTaskMissedYield (void)
- static portTASK_FUNCTION (prvIdleTask, pvParameters)
- static void prvInitialiseTaskLists (void)
- static void prvCheckTasksWaitingTermination (void)
- static void prvResetNextTaskUnblockTime (void)
- TickType_t uxTaskResetEventItemValue (void)
- static void prvAddCurrentTaskToDelayedList (TickType_t xTicksToWait, const BaseType_t xCanBlock↩Indefinitely)

Variables

- PRIVILEGED_DATA TCB_t *volatile pxCurrentTCB = NULL
- static PRIVILEGED_DATA List_t pxReadyTasksLists [configMAX_PRIORITIES]
- static PRIVILEGED_DATA List_t xDelayedTaskList1
- static PRIVILEGED_DATA List_t xDelayedTaskList2
- static PRIVILEGED_DATA List_t *volatile pxDelayedTaskList
- static PRIVILEGED_DATA List_t *volatile pxOverflowDelayedTaskList
- static PRIVILEGED_DATA List_t xPendingReadyList
- static PRIVILEGED_DATA volatile UBaseType_t uxCurrentNumberOfTasks = (UBaseType_t) 0U
- static PRIVILEGED_DATA volatile TickType_t xTickCount = (TickType_t) 0U
- static PRIVILEGED_DATA volatile UBaseType_t uxTopReadyPriority = tskIDLE_PRIORITY
- static PRIVILEGED_DATA volatile BaseType_t xSchedulerRunning = pdFALSE
- static PRIVILEGED_DATA volatile UBaseType_t uxPendedTicks = (UBaseType_t) 0U
- static PRIVILEGED_DATA volatile BaseType_t xYieldPending = pdFALSE
- static PRIVILEGED_DATA volatile BaseType_t xNumOfOverflows = (BaseType_t) 0
- static PRIVILEGED_DATA UBaseType_t uxTaskNumber = (UBaseType_t) 0U
- static PRIVILEGED_DATA volatile TickType_t xNextTaskUnblockTime = (TickType_t) 0U
- static PRIVILEGED_DATA TaskHandle_t xIdleTaskHandle = NULL
- static PRIVILEGED_DATA volatile UBaseType_t uxSchedulerSuspended = (UBaseType_t) pdFALSE

7.31.1 Macro Definition Documentation

7.31.1.1 #define MPU_WRAPPERS_INCLUDED_FROM_API_FILE

Definition at line 77 of file tasks.c.

7.31.1.2 #define portRESET_READY_PRIORITY(*uxPriority,* uxTopReadyPriority)

Definition at line 197 of file tasks.c.

7.31.1.3 #define prvAddTaskToReadyList(*pxTCB*)

Value:

```
traceMOVED_TASK_TO_READY_STATE( pxTCB );
      \
    taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );
      \
    vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( (
      pxTCB )->xStateListItem ) ); \
    tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )
```

Definition at line 259 of file tasks.c.

7.31.1.4 #define prvGetTCBFromHandle(*pxHandle*) (((pxHandle) == NULL) ? (TCB_t *) pxCurrentTCB : (TCB_t *) (pxHandle))

Definition at line 272 of file tasks.c.

7.31.1.5 #define taskEVENT_LIST_ITEM_VALUE_IN_USE 0x80000000UL

Definition at line 285 of file tasks.c.

7.31.1.6 #define taskNOT_WAITING_NOTIFICATION ((uint8_t) 0)

Definition at line 110 of file tasks.c.

7.31.1.7 #define taskNOTIFICATION_RECEIVED ((uint8_t) 2)

Definition at line 112 of file tasks.c.

7.31.1.8 #define taskRECORD_READY_PRIORITY(*uxPriority*)

Value:

```
{                                                                                   \
        if( ( uxPriority ) > uxTopReadyPriority )
      \
        {
      \
            uxTopReadyPriority = ( uxPriority );
      \
        }                                                                           \
    } /* taskRECORD_READY_PRIORITY */
```

Definition at line 164 of file tasks.c.

7.31.1.9 #define taskRESET_READY_PRIORITY(*uxPriority*)

Definition at line 196 of file tasks.c.

7.31.1.10 #define taskSELECT_HIGHEST_PRIORITY_TASK( )

Value:

```
{
      \
    UBaseType_t uxTopPriority = uxTopReadyPriority;
      \
                                                                                    \
        /* Find the highest priority queue that contains ready tasks. */           \
        while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopPriority ] )
      ) )                           \
        {
      \
            configASSERT( uxTopPriority );
      \
            --uxTopPriority;                                                        \
        }                                                                           \
                                                                                    \
        /* listGET_OWNER_OF_NEXT_ENTRY indexes through the list, so the tasks of   \
        the same priority get an equal share of the processor time. */
      \
        listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &(
      pxReadyTasksLists[ uxTopPriority ] ) );           \
        uxTopReadyPriority = uxTopPriority;
      \
    } /* taskSELECT_HIGHEST_PRIORITY_TASK */
```

Definition at line 174 of file tasks.c.

7.31.1.11 #define taskSWITCH_DELAYED_LISTS( )

Value:

```
{
    \
    List_t *pxTemp;
    \
                                                                                                \
    /* The delayed tasks list should be empty when the lists are switched. */
    \
    configASSERT( ( listLIST_IS_EMPTY(
      pxDelayedTaskList ) ) );                                          \
                                                                                                \
    pxTemp = pxDelayedTaskList;
    \
    pxDelayedTaskList = pxOverflowDelayedTaskList;
    \
    pxOverflowDelayedTaskList = pxTemp;
    \
    xNumOfOverflows++;
    \
    prvResetNextTaskUnblockTime();
    \
}
```

Definition at line 239 of file tasks.c.

7.31.1.12 #define taskWAITING_NOTIFICATION ((uint8_t) 1)

Definition at line 111 of file tasks.c.

7.31.1.13 #define taskYIELD_IF_USING_PREEMPTION( )

Definition at line 104 of file tasks.c.

7.31.1.14 #define tskBLOCKED_CHAR ('B')

Definition at line 143 of file tasks.c.

7.31.1.15 #define tskDELETED_CHAR ('D')

Definition at line 145 of file tasks.c.

7.31.1.16 #define tskDYNAMICALLY_ALLOCATED_STACK_AND_TCB ((uint8_t) 0)

Definition at line 136 of file tasks.c.

7.31.1.17 #define tskREADY_CHAR ('R')

Definition at line 144 of file tasks.c.

7.31.1.18 #define tskSTACK_FILL_BYTE (0xa5U)

Definition at line 118 of file tasks.c.

7.31.1.19 #define tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE ((configSUPPORT_STATIC_ALLOCATION == 1) && (configSUPPORT_DYNAMIC_ALLOCATION == 1)) || (portUSING_MPU_WRAPPERS == 1))

Definition at line 135 of file tasks.c.

7.31.1.20 #define tskSTATICALLY_ALLOCATED_STACK_AND_TCB ((uint8_t) 2)

Definition at line 138 of file tasks.c.

7.31.1.21 #define tskSTATICALLY_ALLOCATED_STACK_ONLY ((uint8_t) 1)

Definition at line 137 of file tasks.c.

7.31.1.22 #define tskSUSPENDED_CHAR ('S')

Definition at line 146 of file tasks.c.

7.31.2 Typedef Documentation

7.31.2.1 typedef tskTCB TCB_t

Definition at line 367 of file tasks.c.

7.31.2.2 typedef struct tskTaskControlBlock tskTCB

7.31.3 Function Documentation

7.31.3.1 char* pcTaskGetName (TaskHandle_t *xTaskToQuery*)

Definition at line 2181 of file tasks.c.

7.31.3.2 static portTASK_FUNCTION (prvIdleTask , pvParameters) [static]

THIS IS THE RTOS IDLE TASK - WHICH IS CREATED AUTOMATICALLY WHEN THE SCHEDULER IS STAR↩
TED.

Definition at line 3131 of file tasks.c.

7.31.3.3 static void prvAddCurrentTaskToDelayedList (TickType_t *xTicksToWait,* const BaseType_t *xCanBlockIndefinitely*) [static]

Definition at line 4692 of file tasks.c.

7.31.3.4 static void prvAddNewTaskToReadyList (TCB_t * *pxNewTCB*) [static]

Definition at line 963 of file tasks.c.

7.31.3.5 static void prvCheckTasksWaitingTermination (void) [static]

THIS FUNCTION IS CALLED FROM THE RTOS IDLE TASK

Definition at line 3365 of file tasks.c.

7.31.3.6 static void prvInitialiseTaskLists (static void prvCheckTasksWaitingTermination *void*) [static]

Utility task that simply returns pdTRUE if the task referenced by xTask is currently in the Suspended state, or pdFALSE if the task referenced by xTask is in any other state.

Definition at line 456 of file tasks.c.

7.31.3.7 static void prvInitialiseTaskLists (void) [static]

Definition at line 3333 of file tasks.c.

7.31.3.8 static void prvResetNextTaskUnblockTime (void) [static]

Definition at line 3635 of file tasks.c.

7.31.3.9 UBaseType_t uxTaskGetNumberOfTasks (void)

Definition at line 2173 of file tasks.c.

7.31.3.10 TickType_t uxTaskResetEventItemValue (void)

Definition at line 4162 of file tasks.c.

7.31.3.11 void vTaskEndScheduler (void)

Definition at line 1933 of file tasks.c.

7.31.3.12 void vTaskMissedYield (void)

Definition at line 3076 of file tasks.c.

7.31.3.13 void vTaskPlaceOnEventList (List_t ∗const *pxEventList*, const TickType_t *xTicksToWait*)

Definition at line 2820 of file tasks.c.

7.31.3.14 void vTaskPlaceOnUnorderedEventList (List_t ∗ *pxEventList*, const TickType_t *xItemValue*, const TickType_t *xTicksToWait*)

Definition at line 2837 of file tasks.c.

7.31.3.15 void vTaskSetTimeOutState (TimeOut_t ∗const *pxTimeOut*)

Definition at line 3007 of file tasks.c.

7.31.3.16 void vTaskStartScheduler (void)

Definition at line 1826 of file tasks.c.

7.31.3.17 void vTaskSuspendAll (void)

Definition at line 1944 of file tasks.c.

7.31.3.18 void vTaskSwitchContext (void)

Definition at line 2761 of file tasks.c.

7.31.3.19 BaseType_t xTaskCheckForTimeOut (TimeOut_t ∗const *pxTimeOut*, TickType_t ∗const *pxTicksToWait*)

Definition at line 3015 of file tasks.c.

7.31.3.20 TickType_t xTaskGetTickCount (void)

Definition at line 2127 of file tasks.c.

7.31.3.21 TickType_t xTaskGetTickCountFromISR (void)

Definition at line 2142 of file tasks.c.

7.31.3.22 BaseType_t xTaskIncrementTick (void)

Definition at line 2499 of file tasks.c.

7.31.3.23 BaseType_t xTaskRemoveFromEventList (const List_t *const *pxEventList*)

Definition at line 2894 of file tasks.c.

7.31.3.24 BaseType_t xTaskRemoveFromUnorderedEventList (ListItem_t * *pxEventListItem,* const TickType_t *xItemValue*)

Definition at line 2962 of file tasks.c.

7.31.3.25 BaseType_t xTaskResumeAll (void)

Definition at line 2017 of file tasks.c.

7.31.4 Variable Documentation

7.31.4.1 PRIVILEGED_DATA TCB_t* volatile pxCurrentTCB = NULL

Definition at line 372 of file tasks.c.

7.31.4.2 PRIVILEGED_DATA List_t* volatile pxDelayedTaskList [static]

Definition at line 378 of file tasks.c.

7.31.4.3 PRIVILEGED_DATA List_t* volatile pxOverflowDelayedTaskList [static]

Definition at line 379 of file tasks.c.

7.31.4.4 PRIVILEGED_DATA List_t pxReadyTasksLists[configMAX_PRIORITIES] [static]

Definition at line 375 of file tasks.c.

7.31.4.5 PRIVILEGED_DATA volatile UBaseType_t uxCurrentNumberOfTasks = (UBaseType_t) 0U [static]

Definition at line 396 of file tasks.c.

7.31.4.6 PRIVILEGED_DATA volatile UBaseType_t uxPendedTicks = (UBaseType_t) 0U [static]

Definition at line 400 of file tasks.c.

7.31.4.7 PRIVILEGED_DATA volatile UBaseType_t uxSchedulerSuspended = (UBaseType_t) pdFALSE [static]

Definition at line 415 of file tasks.c.

7.31.4.8 PRIVILEGED_DATA UBaseType_t uxTaskNumber = (UBaseType_t) 0U [static]

Definition at line 403 of file tasks.c.

7.31.4.9 PRIVILEGED_DATA volatile UBaseType_t uxTopReadyPriority = tskIDLE_PRIORITY [static]

Definition at line 398 of file tasks.c.

7.31.4.10 PRIVILEGED_DATA List_t xDelayedTaskList1 [static]

Definition at line 376 of file tasks.c.

7.31.4.11 PRIVILEGED_DATA List_t xDelayedTaskList2 [static]

Definition at line 377 of file tasks.c.

7.31.4.12 PRIVILEGED_DATA TaskHandle_t xIdleTaskHandle = NULL [static]

Definition at line 405 of file tasks.c.

7.31.4.13 PRIVILEGED_DATA volatile TickType_t xNextTaskUnblockTime = (TickType_t) 0U [static]

Definition at line 404 of file tasks.c.

7.31.4.14 PRIVILEGED_DATA volatile BaseType_t xNumOfOverflows = (BaseType_t) 0 [static]

Definition at line 402 of file tasks.c.

7.31.4.15 PRIVILEGED_DATA List_t xPendingReadyList [static]

Definition at line 380 of file tasks.c.

7.31.4.16 PRIVILEGED_DATA volatile BaseType_t xSchedulerRunning = pdFALSE [static]

Definition at line 399 of file tasks.c.

7.31.4.17 PRIVILEGED_DATA volatile TickType_t xTickCount = (TickType_t) 0U [static]

Definition at line 397 of file tasks.c.

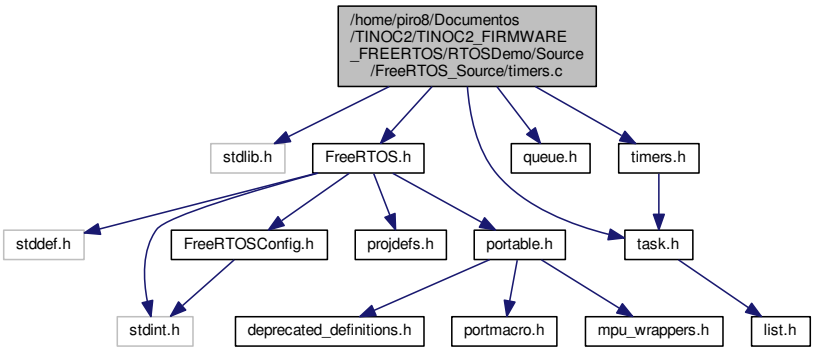
7.31.4.18 PRIVILEGED_DATA volatile BaseType_t xYieldPending = pdFALSE [static]

Definition at line 401 of file tasks.c.

7.32 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/FreeRTOS_Source/timers.c File Reference

```
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
```

Include dependency graph for timers.c:



Macros

- #define MPU_WRAPPERS_INCLUDED_FROM_API_FILE

7.32.1 Macro Definition Documentation

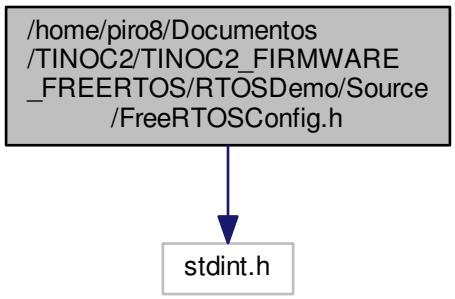
7.32.1.1 #define MPU_WRAPPERS_INCLUDED_FROM_API_FILE

Definition at line 76 of file timers.c.

7.33 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/FreeRTOSConfig.h File Reference

```
#include <stdint.h>
```

Include dependency graph for FreeRTOSConfig.h:



This graph shows which files directly or indirectly include this file:

Macros

- #define configUSE_PREEMPTION 1
- #define configUSE_IDLE_HOOK 0
- #define configUSE_TICK_HOOK 1
- #define configCPU_CLOCK_HZ (SystemCoreClock)
- #define configTICK_RATE_HZ ((TickType_t) 1000)
- #define configMAX_PRIORITIES (5)
- #define configMINIMAL_STACK_SIZE ((unsigned short) 70)
- #define configTOTAL_HEAP_SIZE ((size_t) (6500))
- #define configMAX_TASK_NAME_LEN (5)
- #define configUSE_TRACE_FACILITY 1
- #define configUSE_16_BIT_TICKS 0
- #define configIDLE_SHOULD_YIELD 1
- #define configUSE_MUTEXES 1
- #define configQUEUE_REGISTRY_SIZE 8
- #define configCHECK_FOR_STACK_OVERFLOW 2
- #define configUSE_RECURSIVE_MUTEXES 1
- #define configUSE_MALLOC_FAILED_HOOK 1
- #define configUSE_APPLICATION_TASK_TAG 0
- #define configUSE_COUNTING_SEMAPHORES 1
- #define configGENERATE_RUN_TIME_STATS 0

- #define configUSE_CO_ROUTINES 0
- #define configMAX_CO_ROUTINE_PRIORITIES (2)
- #define configUSE_TIMERS 1
- #define configTIMER_TASK_PRIORITY (2)
- #define configTIMER_QUEUE_LENGTH 2
- #define configTIMER_TASK_STACK_DEPTH (80)
- #define INCLUDE_vTaskPrioritySet 1
- #define INCLUDE_uxTaskPriorityGet 1
- #define INCLUDE_vTaskDelete 1
- #define INCLUDE_vTaskCleanUpResources 1
- #define INCLUDE_vTaskSuspend 1
- #define INCLUDE_vTaskDelayUntil 1
- #define INCLUDE_vTaskDelay 1
- #define INCLUDE_eTaskGetState 1
- #define configASSERT(x) if((x) == 0) { taskDISABLE_INTERRUPTS(); for(;;); }
- #define vPortSVCHandler SVCall_Handler
- #define xPortPendSVHandler PendSV_Handler
- #define xPortSysTickHandler SysTick_Handler

Variables

- uint32_t SystemCoreClock

7.33.1 Macro Definition Documentation

7.33.1.1 #define configASSERT(*x*) if((x) == 0) { taskDISABLE_INTERRUPTS(); for(;;); }

Definition at line 141 of file FreeRTOSConfig.h.

7.33.1.2 #define configCHECK_FOR_STACK_OVERFLOW 2

Definition at line 111 of file FreeRTOSConfig.h.

7.33.1.3 #define configCPU_CLOCK_HZ (SystemCoreClock)

Definition at line 100 of file FreeRTOSConfig.h.

7.33.1.4 #define configGENERATE_RUN_TIME_STATS 0

Definition at line 116 of file FreeRTOSConfig.h.

7.33.1.5 #define configIDLE_SHOULD_YIELD 1

Definition at line 108 of file FreeRTOSConfig.h.

7.33.1.6 #define configMAX_CO_ROUTINE_PRIORITIES (2)

Definition at line 120 of file FreeRTOSConfig.h.

7.33.1.7 #define configMAX_PRIORITIES (5)

Definition at line 102 of file FreeRTOSConfig.h.

7.33.1.8 #define configMAX_TASK_NAME_LEN (5)

Definition at line 105 of file FreeRTOSConfig.h.

7.33.1.9 #define configMINIMAL_STACK_SIZE ((unsigned short) 70)

Definition at line 103 of file FreeRTOSConfig.h.

7.33.1.10 #define configQUEUE_REGISTRY_SIZE 8

Definition at line 110 of file FreeRTOSConfig.h.

7.33.1.11 #define configTICK_RATE_HZ ((TickType_t) 1000)

Definition at line 101 of file FreeRTOSConfig.h.

7.33.1.12 #define configTIMER_QUEUE_LENGTH 2

Definition at line 125 of file FreeRTOSConfig.h.

7.33.1.13 #define configTIMER_TASK_PRIORITY (2)

Definition at line 124 of file FreeRTOSConfig.h.

7.33.1.14 #define configTIMER_TASK_STACK_DEPTH (80)

Definition at line 126 of file FreeRTOSConfig.h.

7.33.1.15 #define configTOTAL_HEAP_SIZE ((size_t) (6500))

Definition at line 104 of file FreeRTOSConfig.h.

7.33.1.16 #define configUSE_16_BIT_TICKS 0

Definition at line 107 of file FreeRTOSConfig.h.

7.33.1.17 #define configUSE_APPLICATION_TASK_TAG 0

Definition at line 114 of file FreeRTOSConfig.h.

7.33.1.18 #define configUSE_CO_ROUTINES 0

Definition at line 119 of file FreeRTOSConfig.h.

7.33.1.19 #define configUSE_COUNTING_SEMAPHORES 1

Definition at line 115 of file FreeRTOSConfig.h.

7.33.1.20 #define configUSE_IDLE_HOOK 0

Definition at line 98 of file FreeRTOSConfig.h.

7.33.1.21 #define configUSE_MALLOC_FAILED_HOOK 1

Definition at line 113 of file FreeRTOSConfig.h.

7.33.1.22 #define configUSE_MUTEXES 1

Definition at line 109 of file FreeRTOSConfig.h.

7.33.1.23 #define configUSE_PREEMPTION 1

Definition at line 97 of file FreeRTOSConfig.h.

7.33.1.24 #define configUSE_RECURSIVE_MUTEXES 1

Definition at line 112 of file FreeRTOSConfig.h.

7.33.1.25 #define configUSE_TICK_HOOK 1

Definition at line 99 of file FreeRTOSConfig.h.

7.33.1.26 #define configUSE_TIMERS 1

Definition at line 123 of file FreeRTOSConfig.h.

7.33.1.27 #define configUSE_TRACE_FACILITY 1

Definition at line 106 of file FreeRTOSConfig.h.

7.33.1.28 #define INCLUDE_eTaskGetState 1

Definition at line 137 of file FreeRTOSConfig.h.

7.33.1.29 #define INCLUDE_uxTaskPriorityGet 1

Definition at line 131 of file FreeRTOSConfig.h.

7.33.1.30 #define INCLUDE_vTaskCleanUpResources 1

Definition at line 133 of file FreeRTOSConfig.h.

7.33.1.31 #define INCLUDE_vTaskDelay 1

Definition at line 136 of file FreeRTOSConfig.h.

7.33.1.32 #define INCLUDE_vTaskDelayUntil 1

Definition at line 135 of file FreeRTOSConfig.h.

7.33.1.33 #define INCLUDE_vTaskDelete 1

Definition at line 132 of file FreeRTOSConfig.h.

7.33.1.34 #define INCLUDE_vTaskPrioritySet 1

Definition at line 130 of file FreeRTOSConfig.h.

7.33.1.35 #define INCLUDE_vTaskSuspend 1

Definition at line 134 of file FreeRTOSConfig.h.

7.33.1.36 #define vPortSVCHandler SVCall_Handler

Definition at line 145 of file FreeRTOSConfig.h.

7.33.1.37 #define xPortPendSVHandler PendSV_Handler

Definition at line 146 of file FreeRTOSConfig.h.

7.33.1.38 #define xPortSysTickHandler SysTick_Handler

Definition at line 147 of file FreeRTOSConfig.h.

7.33.2 Variable Documentation

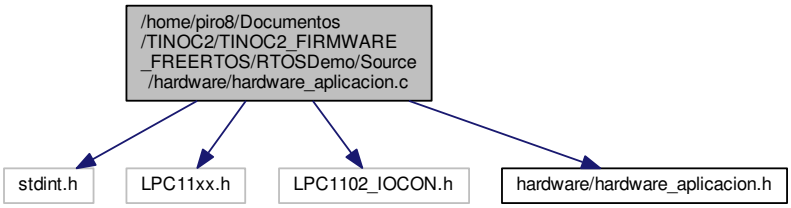
7.33.2.1 uint32_t SystemCoreClock

7.34 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/hardware/hardware_aplicacion.c File Reference

Funciones de inicialización del hardware específicas para la aplicacion.

```
#include <stdint.h>
#include <LPC11xx.h>
#include <LPC1102_IOCON.h>
#include <hardware/hardware_aplicacion.h>
```

Include dependency graph for hardware_aplicacion.c:



7.34.1 Detailed Description

Funciones de inicialización del hardware específicas para la aplicacion.

Date

Jan 17, 2018

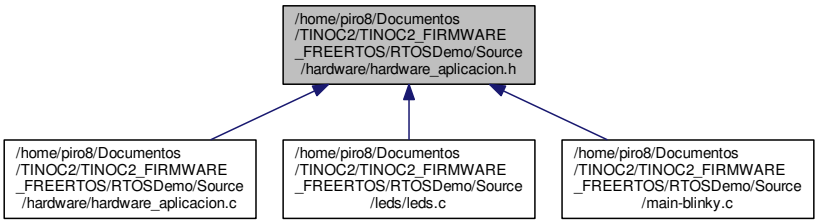
Author

Roux, Federico G. (froux@citedef.gob.ar)

7.35 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/hardware/hardware_aplicacion.h File Reference

header del archivo hardware_aplicacion.h

This graph shows which files directly or indirectly include this file:



7.35.1 Detailed Description

header del archivo hardware_aplicacion.h

Date

Jan 17, 2018

Author

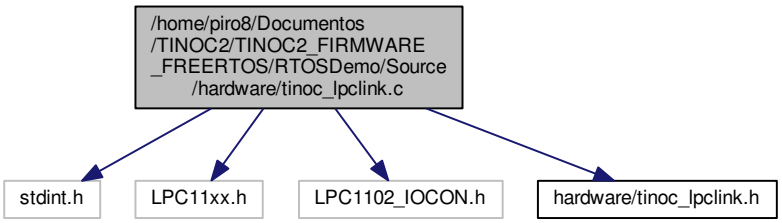
Roux, Federico G. (froux@citedef.gob.ar)

7.36 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/hardware/tinoc_lpclink.c File Reference

Funciones de inicialización del hardware específicas para la placa TINOC_LPCLINK.

```
#include <stdint.h>
#include <LPC11xx.h>
#include <LPC1102_IOCON.h>
#include <hardware/tinoc_lpclink.h>
```

Include dependency graph for tinoc_lpclink.c:



Functions

- uint32_t Inicializar_Pines_GPIO (void)

  Inicialización de pines destinados a GPIO utilizando las macros de LPC1102_IOCON.h.
- uint32_t Activar_Pin_ISP (void)
- void ReinvokeISP (void)

  Envia comando IAP para inicializar el modo ISP.

Variables

- IAP iap_entry

  variable para acceso a flash

7.36.1 Detailed Description

Funciones de inicialización del hardware específicas para la placa TINOC_LPCLINK.

Date

1 de set. de 2017

Author

Roux, Federico G. (froux@citedef.gob.ar)

7.36.2 Function Documentation

7.36.2.1 uint32_t Activar_Pin_ISP (void)

Definition at line 88 of file tinoc_lpclink.c.

7.36.2.2 uint32_t Inicializar_Pines_GPIO (void)

Inicialización de pines destinados a GPIO utilizando las macros de LPC1102_IOCON.h.

Parameters

in	*None*	
out	*0*	OK
in	*nada*	
out	*0*	OK

Definition at line 58 of file tinoc_lpclink.c.

7.36.2.3 void ReinvokeISP (void)

Envia comando IAP para inicializar el modo ISP.

Parameters

in	*nada*	
out	*nada*	

Definition at line 105 of file tinoc_lpclink.c.

7.36.3 Variable Documentation

7.36.3.1 IAP iap_entry

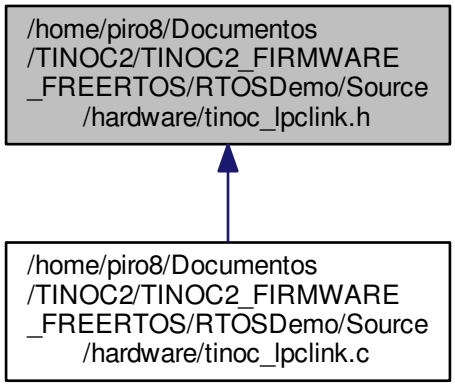
variable para acceso a flash

Definition at line 41 of file tinoc_lpclink.c.

7.37 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/hardware/tinoc_lpclink.h File Reference

header del archivo tinoc_lpclink.c

This graph shows which files directly or indirectly include this file:



Macros

- #define LPC1102_C1_BIT (1UL << 0UL)

  Defino el bit del puerto ocupado por C1.
- #define LPC1102_C1_PORT (0)

  Defino el port del puerto ocupado por C1.
- #define LPC1102_C1_GPIO LPC_GPIO0

  Defino el registro GPIO ocupado por C1.
- #define LPC1102_A2_BIT (1UL << 8UL)

  Defino el port y bit del puerto ocupado por A2.
- #define LPC1102_A2_PORT (0)

  Defino el port del puerto ocupado por A2.
- #define LPC1102_A2_GPIO LPC_GPIO0

  Defino el registro GPIO ocupado por A2.
- #define LPC1102_A3_BIT (1UL << 9UL)

  Defino el port y bit del puerto ocupado por A3.
- #define LPC1102_A3_PORT (0)

  Defino el port del puerto ocupado por A3.
- #define LPC1102_A3_GPIO LPC_GPIO0

  Defino el registro GPIO ocupado por A3.
- #define LPC1102_A4_BIT (1UL << 10UL)

  Defino el port y bit del puerto ocupado por A4.
- #define LPC1102_A4_PORT (0)

  Defino el port del puerto ocupado por A4.
- #define LPC1102_A4_GPIO LPC_GPIO0

  Defino el registro GPIO ocupado por A4.
- #define LPC1102_B4_BIT (1UL << 11UL)

  Defino el port y bit del puerto ocupado por B4.
- #define LPC1102_B4_PORT (0)

  Defino el port del puerto ocupado por B4.
- #define LPC1102_B4_GPIO LPC_GPIO0

  Defino el registro GPIO ocupado por B4.
- #define LPC1102_B3_BIT (1UL << 0UL)

  Defino el port y bit del puerto ocupado por B3.
- #define LPC1102_B3_PORT (1)

  Defino el port del puerto ocupado por B3.
- #define LPC1102_B3_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por B3.
- #define LPC1102_C4_BIT (1UL << 1UL)

  Defino el port y bit del puerto ocupado por C4.
- #define LPC1102_C4_PORT (1)

  Defino el port del puerto ocupado por C4.
- #define LPC1102_C4_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por C4.
- #define LPC1102_C3_BIT (1UL << 2UL)

  Defino el port y bit del puerto ocupado por C3.
- #define LPC1102_C3_PORT (1)

  Defino el port del puerto ocupado por C3.
- #define LPC1102_C3_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por C3.
- #define LPC1102_D4_BIT (1UL << 3UL)

Defino el port y bit del puerto ocupado por D4.

- #define LPC1102_D4_PORT (1)

  Defino el port del puerto ocupado por D4.
- #define LPC1102_D4_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por D4.
- #define LPC1102_C2_BIT (1UL << 6UL)

  Defino el port y bit del puerto ocupado por C2.
- #define LPC1102_C2_PORT (1)

  Defino el port del puerto ocupado por C2.
- #define LPC1102_C2_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por C2.
- #define LPC1102_D1_BIT (1UL << 7UL)

  Defino bit del puerto ocupado por D1.
- #define LPC1102_D1_PORT (1)

  Defino el port del puerto ocupado por D1.
- #define LPC1102_D1_GPIO LPC_GPIO1

  Defino el registro GPIO ocupado por D1.

Functions

- uint32_t Inicializar_Pines_GPIO (void)

  Inicialización de pines destinados a GPIO utilizando las macros de LPC1102_IOCON.h.
- uint32_t Activar_Pin_ISP (void)
- void ReinvokeISP (void)

  Envia comando IAP para inicializar el modo ISP.

7.37.1 Detailed Description

header del archivo tinoc_lpclink.c

Date

1 de set. de 2017

Author

Roux, Federico G. (froux@citedef.gob.ar)

7.37.2 Function Documentation

7.37.2.1 uint32_t Activar_Pin_ISP (void)

Definition at line 88 of file tinoc_lpclink.c.

7.37.2.2 uint32_t Inicializar_Pines_GPIO (void)

Inicialización de pines destinados a GPIO utilizando las macros de LPC1102_IOCON.h.

Parameters

in	*None*	
out	*0*	OK
in	*nada*	
out	*0*	OK

Definition at line 58 of file tinoc_lpclink.c.

7.37.2.3 void ReinvokeISP (void)

Envia comando IAP para inicializar el modo ISP.

Parameters

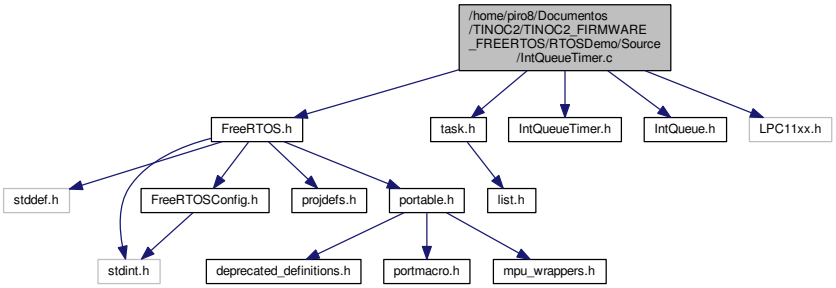
in	*nada*	
out	*nada*	

Definition at line 105 of file tinoc_lpclink.c.

7.38 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/IntQueueTimer.c File Reference

```
#include "FreeRTOS.h"
#include "task.h"
#include "IntQueueTimer.h"
#include "IntQueue.h"
#include "LPC11xx.h"
```

Include dependency graph for IntQueueTimer.c:



Macros

- #define tmrTIMER_2_FREQUENCY (2000UL)
- #define tmrTIMER_3_FREQUENCY (2001UL)
- #define tmrMAX_PRIORITY (0UL)
- #define trmSECOND_HIGHEST_PRIORITY (tmrMAX_PRIORITY + 1)

Functions

- void vInitialiseTimerForIntQueueTest (void)
- void TIMER16_0_IRQHandler (void)
- void TIMER16_1_IRQHandler (void)

7.38.1 Macro Definition Documentation

7.38.1.1 #define tmrMAX_PRIORITY (0UL)

Definition at line 87 of file IntQueueTimer.c.

7.38.1.2 #define tmrTIMER_2_FREQUENCY (2000UL)

Definition at line 82 of file IntQueueTimer.c.

7.38.1.3 #define tmrTIMER_3_FREQUENCY (2001UL)

Definition at line 83 of file IntQueueTimer.c.

7.38.1.4 #define trmSECOND_HIGHEST_PRIORITY (tmrMAX_PRIORITY + 1)

Definition at line 88 of file IntQueueTimer.c.

7.38.2 Function Documentation

7.38.2.1 void TIMER16_0_IRQHandler (void)

Definition at line 123 of file IntQueueTimer.c.

7.38.2.2 void TIMER16_1_IRQHandler (void)

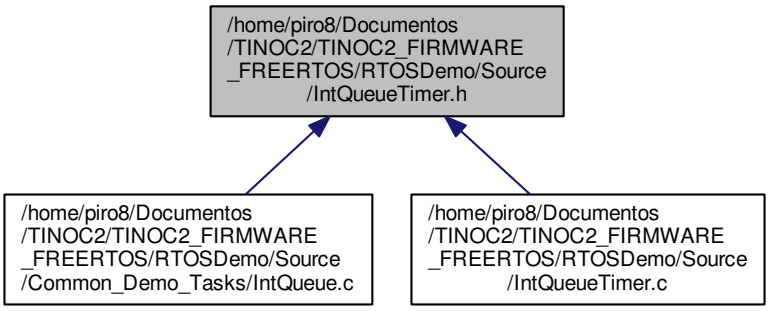
Definition at line 133 of file IntQueueTimer.c.

7.38.2.3 void vInitialiseTimerForIntQueueTest (void)

Definition at line 90 of file IntQueueTimer.c.

7.39 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/IntQueueTimer.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void vInitialiseTimerForIntQueueTest (void)
- portBASE_TYPE xTimer0Handler (void)
- portBASE_TYPE xTimer1Handler (void)

7.39.1 Function Documentation

7.39.1.1 void vInitialiseTimerForIntQueueTest (void)

Definition at line 90 of file IntQueueTimer.c.

7.39.1.2 portBASE_TYPE xTimer0Handler (void)

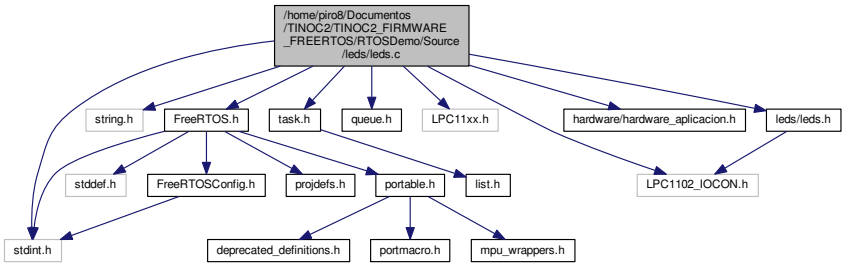
7.39.1.3 portBASE_TYPE xTimer1Handler (void)

7.40 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/leds/leds.c File Reference

Funciones de manejo de leds específicas para la aplicacion.

```
#include <stdint.h>
#include "string.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include <LPC11xx.h>
#include <LPC1102_IOCON.h>
#include <hardware/hardware_aplicacion.h>
#include <leds/leds.h>
```

Include dependency graph for leds.c:



Functions

- void Tarea_Delay (void *pvParameters)

  Tarea que inicia la cuenta del tick y envia mensaje a través de la queue.
- void Tarea_Parpadear (void *pvParameters)

  Tarea que inicia la cuenta del tick y envia mensaje a través de la queue.
- uint32_t Inicializar_Leds (void)

  Inicialización de pines destinados a leds.
- uint32_t Parpadear_Led (void)

  Cambio el estado del pin seteado como led.

Variables

- QueueHandle_t xQueue = NULL

  The queue used by both tasks.

7.40.1 Detailed Description

Funciones de manejo de leds específicas para la aplicacion.

Date

Feb 22, 2018

Author

Roux, Federico G. (froux@favaloro.edu.ar)

7.40.2 Function Documentation

7.40.2.1 uint32_t Inicializar_Leds (void)

Inicialización de pines destinados a leds.

Parameters

in	*nada*	
out	*0*	OK

Definition at line 65 of file leds.c.

7.40.2.2 uint32_t Parpadear_Led (void)

Cambio el estado del pin seteado como led.

Parameters

in	*nada*	
out	*0*	OK

Definition at line 85 of file leds.c.

7.40.2.3 void Tarea_Delay (void * *pvParameters*)

Tarea que inicia la cuenta del tick y envia mensaje a través de la queue.

Parameters

in	*nada*	
out	*0*	OK

Definition at line 115 of file leds.c.

7.40.2.4 void Tarea_Parpadear (void * *pvParameters*)

Tarea que inicia la cuenta del tick y envia mensaje a través de la queue.

Parameters

in	*nada*	
out	*0*	OK

Definition at line 154 of file leds.c.

7.40.3 Variable Documentation

7.40.3.1 xQueue = NULL

The queue used by both tasks.

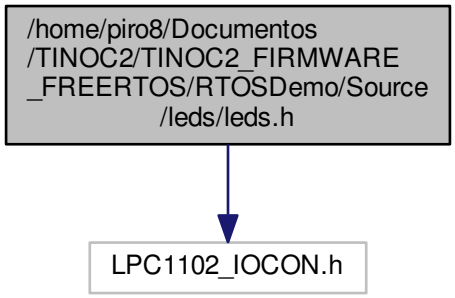
Definition at line 48 of file leds.c.

7.41 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/leds/leds.h File Reference

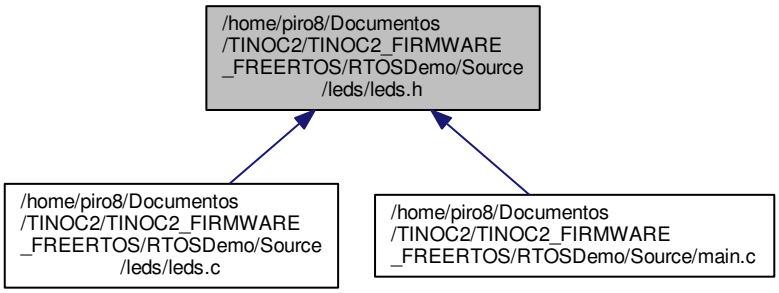
Header del archivo leds.c.

```
#include <LPC1102_IOCON.h>
```

Include dependency graph for leds.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define LED1_IOCON LPC_IOCON_B4

  Registro IOCON asociado al LED1.
- #define LED1_IOCON_FUNC LPC_IOCON_B4_FUNC_PIO0_11

  Función asociada al registro IOCON.
- #define LED1_DIR_BIT (11UL)

  Bit asociado al LED1.
- #define LED1_DIR_MAS (0x01 << LED1_DIR_BIT)

  Dirección del bit LED1.

- #define LED1_DIR_REG LPC_GPIO0->DIR

  Registro de dirección.
- #define LED1_MAS_REG LPC_GPIO0->MASKED_ACCESS

  Registro IOCON asociado al LED1.
- #define PRIORIDAD_TAREA_DELAY (tskIDLE_PRIORITY + 2)
- #define PRIORIDAD_TAREA_PARPADEAR (tskIDLE_PRIORITY + 1)
- #define mainQUEUE_SEND_PARAMETER (0x1111UL)

  Values passed to the two tasks just to check the task parameter functionality.
- #define mainQUEUE_RECEIVE_PARAMETER (0x22UL)
- #define mainQUEUE_SEND_FREQUENCY_MS (200 / portTICK_PERIOD_MS)

  The rate at which data is sent to the queue. The 200ms value is converted to ticks using the portTICK_PERIOD_MS constant.
- #define mainQUEUE_LENGTH (1)

  The number of items the queue can hold. This is 1 as the receive task will remove items as they are added, meaning the send task should always find the queue empty.

Functions

- uint32_t Inicializar_Leds (void)

  Inicialización de pines destinados a leds.
- uint32_t Parpadear_Led (void)

  Cambio el estado del pin seteado como led.
- void Tarea_Delay (void *pvParameters)

  Tarea que inicia la cuenta del tick y envia mensaje a través de la queue.
- void Tarea_Parpadear (void *pvParameters)

  Tarea que inicia la cuenta del tick y envia mensaje a través de la queue.

Variables

- QueueHandle_t xQueue

  The queue used by both tasks.

7.41.1 Detailed Description

Header del archivo leds.c.

Date

Feb 22, 2018

Author

Roux, Federico G. (froux@citedef.gob.ar)

7.41.2 Macro Definition Documentation

7.41.2.1 #define mainQUEUE_LENGTH (1)

The number of items the queue can hold. This is 1 as the receive task will remove items as they are added, meaning the send task should always find the queue empty.

Definition at line 75 of file leds.h.

7.41.2.2 #define mainQUEUE_RECEIVE_PARAMETER (0x22UL)

Definition at line 66 of file leds.h.

7.41.2.3 #define mainQUEUE_SEND_FREQUENCY_MS (200 / portTICK_PERIOD_MS)

The rate at which data is sent to the queue. The 200ms value is converted to ticks using the portTICK_PERIOD_MS constant.

Definition at line 69 of file leds.h.

7.41.2.4 #define mainQUEUE_SEND_PARAMETER (0x1111UL)

Values passed to the two tasks just to check the task parameter functionality.

Definition at line 65 of file leds.h.

7.41.3 Function Documentation

7.41.3.1 uint32_t Inicializar_Leds (void)

Inicialización de pines destinados a leds.

Parameters

in	*nada*	
out	*0*	OK

Definition at line 65 of file leds.c.

7.41.3.2 uint32_t Parpadear_Led (void)

Cambio el estado del pin seteado como led.

Parameters

in	*nada*	
out	*0*	OK

Definition at line 85 of file leds.c.

7.41.3.3 void Tarea_Delay (void * *pvParameters*)

Tarea que inicia la cuenta del tick y envia mensaje a través de la queue.

Parameters

in	*nada*	
out	*0*	OK

Definition at line 115 of file leds.c.

7.41.3.4 void Tarea_Parpadear (void * *pvParameters*)

Tarea que inicia la cuenta del tick y envia mensaje a través de la queue.

Parameters

in	*nada*	
out	*0*	OK

Definition at line 154 of file leds.c.

7.41.4 Variable Documentation

7.41.4.1 QueueHandle_t xQueue

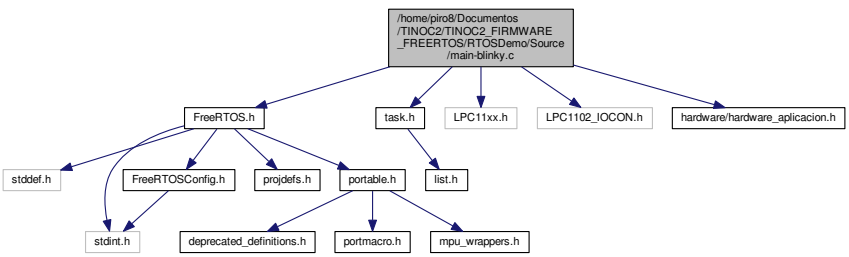
The queue used by both tasks.

Definition at line 48 of file leds.c.

7.42 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/main-blinky.c File Reference

```
#include "FreeRTOS.h"
#include "task.h"
#include "LPC11xx.h"
#include <LPC1102_IOCON.h>
#include <hardware/hardware_aplicacion.h>
```

Include dependency graph for main-blinky.c:



Functions

- void main_blinky (void)
- static void prvQueueSendTask (void *pvParameters)
- static void prvQueueReceiveTask (void *pvParameters)

7.42.1 Function Documentation

7.42.1.1 void main_blinky (void)

Definition at line 139 of file main-blinky.c.

7.42.1.2 static void prvQueueReceiveTask (void * *pvParameters*) [static]

Definition at line 150 of file main-blinky.c.

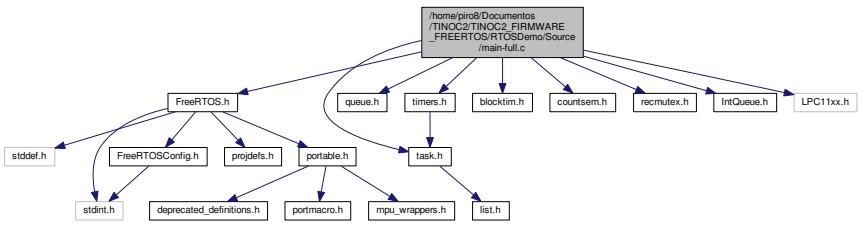
7.42.1.3 static void prvQueueSendTask (void * *pvParameters*) [static]

Definition at line 144 of file main-blinky.c.

7.43 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/main-full.c File Reference

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "blocktim.h"
#include "countsem.h"
#include "recmutex.h"
#include "IntQueue.h"
#include "LPC11xx.h"
```

Include dependency graph for main-full.c:



Macros

- #define mainCHECK_TIMER_PERIOD_MS (3000UL / portTICK_PERIOD_MS)
- #define mainERROR_CHECK_TIMER_PERIOD_MS (200UL / portTICK_PERIOD_MS)
- #define mainDONT_BLOCK (0UL)

Functions

- void vRegTest1Task (void *pvParameters)
- void vRegTest2Task (void *pvParameters)
- void vMainToggleLED (void)
- static void prvCheckTimerCallback (TimerHandle_t xTimer)
- void main_full (void)

Variables

- volatile unsigned long ulRegTest1LoopCounter = 0UL
- volatile unsigned long ulRegTest2LoopCounter = 0UL

7.43.1 Macro Definition Documentation

7.43.1.1 #define mainCHECK_TIMER_PERIOD_MS (3000UL / portTICK_PERIOD_MS)

Definition at line 136 of file main-full.c.

7.43.1.2 #define mainDONT_BLOCK (0UL)

Definition at line 144 of file main-full.c.

7.43.1.3 #define mainERROR_CHECK_TIMER_PERIOD_MS (200UL / portTICK_PERIOD_MS)

Definition at line 141 of file main-full.c.

7.43.2 Function Documentation

7.43.2.1 void main_full (void)

Definition at line 181 of file main-full.c.

7.43.2.2 static void prvCheckTimerCallback (TimerHandle_t *xTimer*) [static]

Definition at line 245 of file main-full.c.

7.43.2.3 void vMainToggleLED (void)

7.43.2.4 void vRegTest1Task (void * *pvParameters*)

7.43.2.5 void vRegTest2Task (void * *pvParameters*)

7.43.3 Variable Documentation

7.43.3.1 volatile unsigned long ulRegTest1LoopCounter = 0UL

Definition at line 177 of file main-full.c.

7.43.3.2 volatile unsigned long ulRegTest2LoopCounter = 0UL

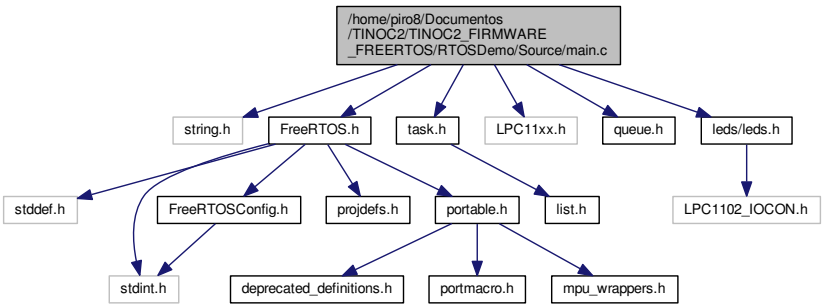
Definition at line 177 of file main-full.c.

7.44 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/main.c File Reference

Archivo principal del proyecto.

```
#include "string.h"
#include "FreeRTOS.h"
#include "task.h"
#include "LPC11xx.h"
#include "queue.h"
#include <leds/leds.h>
```

Include dependency graph for main.c:



Macros

- #define mainCHECK_INTERRUPT_STACK 1

Functions

- static void prvSetupHardware (void)

  Perform any application specific hardware configuration. The clocks,.
- int main (void)

  Programa ppal.
- void vApplicationMallocFailedHook (void)

  vApplicationMallocFailedHook() will only be called if
- void vApplicationIdleHook (void)

  vApplicationIdleHook() will only be called if configUSE_IDLE_HOOK is set
- void vApplicationStackOverflowHook (TaskHandle_t pxTask, char ∗pcTaskName)

  Run time stack overflow checking is performed if.
- void vApplicationTickHook (void)

  This function will be called by each tick interrupt if configUSE_TICK_HOOK is set to 1 in FreeRTOSConfig.h. User code can be added here, but the tick hook is called from an interrupt context, so code must not attempt to block, and only the interrupt safe FreeRTOS API functions can be used (those that end in FromISR()). Manually check the last few bytes of the interrupt stack to check they have not been overwritten. Note - the task stacks are automatically checked for overflow if configCHECK_FOR_STACK_OVERFLOW is set to 1 or 2 in FreeRTOSConifg.h, but the interrupt stack is not.

Variables

- const unsigned char ucExpectedInterruptStackValues [ ] = { 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC }

7.44.1 Detailed Description

Archivo principal del proyecto.

Version

V0.1

Date

2016

Note

FreeRTOS V9.0.0 - Copyright (C) 2016 Real Time Engineers Ltd.

ARM Limited (ARM) is supplying this software for use with Cortex-M processor based microcontrollers. This file can be freely distributed within development tools that are supporting such ARM based processors.

THIS SOFTWARE IS PROVIDED "AS IS". NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATU↩ TORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. ARM SHALL NOT, IN ANY CIRCUMSTA↩ NCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

This project provides two demo applications. A simple blinky style project, and a more comprehensive test and demo application. The mainCREATE_SIMPLE_BLINKY_DEMO_ONLY setting (defined in this file) is used to select between the two. The simply blinky demo is implemented and described in main_blinky.c. The more comprehensive test and demo application is implemented and described in main_full.c.

This file implements the code that is not demo specific, including the hardware setup and FreeRTOS hook functions. It also contains a dummy interrupt service routine called Dummy_IRQHandler() that is provided as an example of how to use interrupt safe FreeRTOS API functions (those that end in "FromISR").

VISIT http://www.FreeRTOS.org TO ENSURE YOU ARE USING THE LATEST VERSION.

This file is part of the FreeRTOS distribution.

FreeRTOS is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License (version 2) as published by the Free Software Foundation >>>> AND MODIFIED BY <<<< the Free↩ RTOS exception.

>>! NOTE: The modification to the GPL is included to allow you to !<< >>! distribute a combined work that includes FreeRTOS without being !<< >>! obliged to provide the source code for proprietary components !<< >>! outside of the FreeRTOS kernel. !<<

FreeRTOS is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. Full license text is available on the following link: http://www.freertos.org/a00114.html

```
                                                                      *
  FreeRTOS provides completely free yet professionally developed,     *
  robust, strictly quality controlled, supported, and cross           *
  platform software that is more than just the market leader, it      *
  is the industry's de facto standard.                                *
                                                                      *
  Help yourself get started quickly while simultaneously helping      *
  to support the FreeRTOS project by purchasing a FreeRTOS            *
  tutorial book, reference manual, or both:                           *
  http://www.FreeRTOS.org/Documentation                               *
                                                                      *


http://www.FreeRTOS.org/FAQHelp.html - Having a problem?  Start by reading
the FAQ page "My application does not run, what could be wrong?".  Have you
defined configASSERT()?

http://www.FreeRTOS.org/support - In return for receiving this top quality
embedded software for free we request you assist our global community by
participating in the support forum.

http://www.FreeRTOS.org/training - Investing in training allows your team to
be as productive as possible as early as possible.  Now you can receive
FreeRTOS training directly from Richard Barry, CEO of Real Time Engineers
Ltd, and the world's leading authority on the world's leading RTOS.

http://www.FreeRTOS.org/plus - A selection of FreeRTOS ecosystem products,
including FreeRTOS+Trace - an indispensable productivity tool, a DOS
compatible FAT file system, and our tiny thread aware UDP/IP stack.

http://www.FreeRTOS.org/labs - Where new FreeRTOS products go to incubate.
Come and try FreeRTOS+TCP, our new open source TCP/IP stack for FreeRTOS.

http://www.OpenRTOS.com - Real Time Engineers ltd. license FreeRTOS to High
Integrity Systems ltd. to sell under the OpenRTOS brand.  Low cost OpenRTOS
licenses offer ticketed support, indemnification and commercial middleware.

http://www.SafeRTOS.com - High Integrity Systems also provide a safety
engineered and independently SIL3 certified version for use in safety and
mission critical applications that require provable dependability.

1 tab == 4 spaces!
```

7.44.2 Macro Definition Documentation

7.44.2.1 #define mainCHECK_INTERRUPT_STACK 1

Definition at line 146 of file main.c.

7.44.3 Function Documentation

7.44.3.1 int main (void)

Programa ppal.

int main(void)

Author

NXP

Definition at line 171 of file main.c.

7.44.3.2 static void prvSetupHardware (void) [static]

Perform any application specific hardware configuration. The clocks,.

static void prvSetupHardware(void) memory, etc. are configured before main() is called.

Author

NXP

Definition at line 220 of file main.c.

7.44.4 Variable Documentation

7.44.4.1 const unsigned char ucExpectedInterruptStackValues[ ] = { 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC }

Definition at line 148 of file main.c.

7.45 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/RegTest.c File Reference

Functions

- void vRegTest1Task (void)
- void vRegTest2Task (void)

7.45.1 Function Documentation

7.45.1.1 void vRegTest1Task (void)

Definition at line 70 of file RegTest.c.

7.45.1.2 void vRegTest2Task (void)

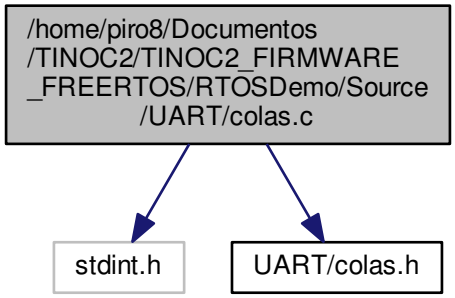
Definition at line 154 of file RegTest.c.

7.46 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/UART/colas.c File Reference

Funciones relacionadas con el manejo de colas.

```
#include <stdint.h>
#include <UART/colas.h>
```

Include dependency graph for colas.c:



Functions

- void Inicializar_Cola (t_cola *cola)

  Inicializo la estructura del tipo cola.
- uint8_t Escribir_Cola (t_cola *cola, uint8_t dato_a_escribir)

  Escribo un dato en el buffer.
- uint8_t Leer_Cola (t_cola *cola, uint8_t *dato_leido)

  Leo y extraigo un dato de la cola.

7.46.1 Detailed Description

Funciones relacionadas con el manejo de colas.

Author

Roux, Federico G. (froux@citedef.gob.ar). CITEDEF2014

7.46.2 Function Documentation

7.46.2.1 uint8_t Escribir_Cola (t_cola * *cola*, uint8_t *dato_a_escribir*)

Escribo un dato en el buffer.

Parameters

in	*t_cola*∗	canal Canal a escribir
in	*uint8*↩ *_t*	dato_a_escribir

Returns

ESCRIBIR_DATO_COLA_EXITO: Escribió el dato en la cola
ESCRIBIR_DATO_COLA_COLA_LLENA: No entran más datos

Definition at line 75 of file colas.c.

7.46.2.2 void Inicializar_Cola (t_cola ∗ *cola*)

Inicializo la estructura del tipo cola.

Parameters

in	*cola*	estructura a inicializar

Returns

Definition at line 50 of file colas.c.

7.46.2.3 uint8_t Leer_Cola (t_cola ∗ *cola,* uint8_t ∗ *dato_leido*)

Leo y extraigo un dato de la cola.

cola : Cola a leer dato_leido : puntero a la variable donde escribo

Returns

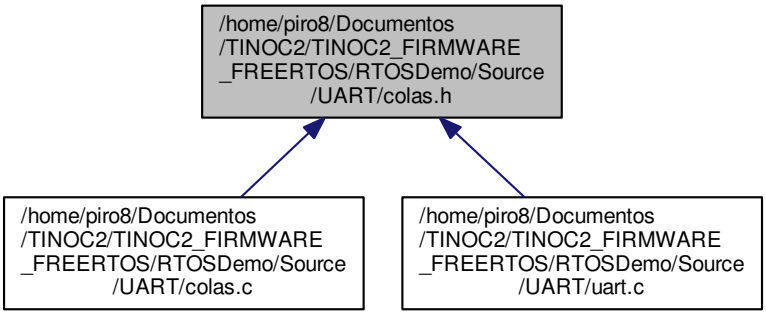
none

Definition at line 105 of file colas.c.

7.47 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/↩ Source/UART/colas.h File Reference

Header del archivo "colas.c".

This graph shows which files directly or indirectly include this file:



Classes

- struct t_cola

Macros

- #define BUFFER_N 8
- #define ESCRIBIR_COLA_COLA_LLENA -1

  No entran mas datos.
- #define ESCRIBIR_COLA_EXITO 0

  Se pudo escribir el dato sin problemas.
- #define LEER_COLA_COLA_VACIA -1
- #define LEER_COLA_EXITO 0

Enumerations

- enum t_estado_buffer { BUFFER_LLENO, BUFFER_CARGANDO, BUFFER_VACIO, BUFFER_TIMEOUT }

Functions

- void Inicializar_Cola (t_cola *cola)

  Inicializo la estructura del tipo cola.
- uint8_t Escribir_Cola (t_cola *cola, uint8_t dato_a_escribir)

  Escribo un dato en el buffer.
- uint8_t Leer_Cola (t_cola *cola, uint8_t *dato_leido)

  Leo y extraigo un dato de la cola.

7.47.1 Detailed Description

Header del archivo "colas.c".

Author

Roux, Federico G. (froux@favaloro.edu.ar)
Laboratorio de uP - UNIVERSIDAD FAVALORO 2014

7.47.2 Macro Definition Documentation

7.47.2.1 #define BUFFER_N 8

Definition at line 21 of file colas.h.

7.47.2.2 #define ESCRIBIR_COLA_COLA_LLENA -1

No entran mas datos.

Definition at line 24 of file colas.h.

7.47.2.3 #define ESCRIBIR_COLA_EXITO 0

Se pudo escribir el dato sin problemas.

Definition at line 27 of file colas.h.

7.47.2.4 #define LEER_COLA_COLA_VACIA -1

Definition at line 30 of file colas.h.

7.47.2.5 #define LEER_COLA_EXITO 0

Definition at line 31 of file colas.h.

7.47.3 Enumeration Type Documentation

7.47.3.1 enum t_estado_buffer

Enumerator

BUFFER_LLENO

BUFFER_CARGANDO

BUFFER_VACIO

BUFFER_TIMEOUT

Definition at line 38 of file colas.h.

7.47.4 Function Documentation

7.47.4.1 uint8_t Escribir_Cola (t_cola * *cola,* uint8_t *dato_a_escribir*)

Escribo un dato en el buffer.

Parameters

in	*t_cola*∗	canal Canal a escribir
in	*uint8_t*	dato_a_escribir

Returns

ESCRIBIR_DATO_COLA_EXITO: Escribió el dato en la cola
ESCRIBIR_DATO_COLA_COLA_LLENA: No entran más datos

Definition at line 75 of file colas.c.

7.47.4.2 void Inicializar_Cola (t_cola ∗ *cola*)

Inicializo la estructura del tipo cola.

Parameters

in	*cola*	estructura a inicializar

Returns

Definition at line 50 of file colas.c.

7.47.4.3 uint8_t Leer_Cola (t_cola ∗ *cola,* uint8_t ∗ *dato_leido*)

Leo y extraigo un dato de la cola.

cola : Cola a leer dato_leido : puntero a la variable donde escribo

Returns

none

Definition at line 105 of file colas.c.

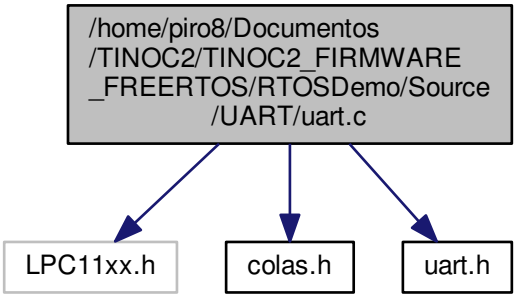
7.48 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/UART/uart.c File Reference

UART API file for NXP LPC11xx Family Microprocessors
Archivo modificado para trabajar como driver dentro del
protocolo de comunicacion PCP1

History

```
#include "LPC11xx.h"
#include <colas.h>
#include "uart.h"
```

Include dependency graph for uart.c:



Macros

- #define _USAR_COLAS_INLINE

Functions

- void UARTInit (uint32_t baudrate)

  Initialize UART0 port, setup pin select, clock, parity, stop bits, FIFO, etc.
- int Inic_UART_RS485 (void)

  Inicializo la UART para funcionar en RS485.
- int UART_RS485_Habilitar_Recepcion (void)

  Habilito la recepcion para la norma RS485 poniendo en alto el pin definido por RS485_DIR_PORT y RS485_DIR↩ _PIN.
- int UART_RS485_Deshabilitar_Recepcion (void)

  Habilito la recepcion para la norma RS485 poniendo en alto el pin definido por RS485_DIR_PORT y RS485_DIR↩ _PIN.
- void UART_Reset (int baudrate)

  Reinicio el periférico.
- void UART_IRQHandler (void)

  UART interrupt handler.

Variables

- volatile uint32_t UARTStatus
- volatile uint8_t tx_libre = 1
- volatile uint8_t UARTBuffer [BUFSIZE]
- volatile uint32_t UARTCount = 0
- volatile uint32_t contador_timeout = 0
- volatile uint32_t contador_vueltas_timeout = 0
- t_cola cola_rx
- t_cola cola_tx

7.48.1 Detailed Description

UART API file for NXP LPC11xx Family Microprocessors
Archivo modificado para trabajar como driver dentro del
protocolo de comunicacion PCP1

History

- 2008.08.21 ver 1.00 - Preliminary version, first Release
- 2014-2015 : modificacions para SATFIG
- 02/2018 : documentacion

Author

Copyright(C) 2008, NXP Semiconductor
Modificado por Roux, Federico G. (froux@citedef.gob.ar)

7.48.2 Macro Definition Documentation

7.48.2.1 #define _USAR_COLAS_INLINE

Definition at line 23 of file uart.c.

7.48.3 Function Documentation

7.48.3.1 int Inic_UART_RS485 (void)

Inicializo la UART para funcionar en RS485.

==========

Parameters

in	*None*	**None**

Definition at line 317 of file uart.c.

7.48.3.2 void UART_IRQHandler (void)

UART interrupt handler.

==========

Parameters

None	None

Definition at line 67 of file uart.c.

7.48.3.3 void UART_Reset (int *baudrate*)

Reinicio el periférico.

==========

Parameters

in	*baudrate*	None

Definition at line 159 of file uart.c.

7.48.3.4 int UART_RS485_Deshabilitar_Recepcion (void)

Habilito la recepcion para la norma RS485 poniendo en alto el pin definido por RS485_DIR_PORT y RS485_DI↩ R_PIN.

==========

Parameters

in	*None*	None

Definition at line 365 of file uart.c.

7.48.3.5 int UART_RS485_Habilitar_Recepcion (void)

Habilito la recepcion para la norma RS485 poniendo en alto el pin definido por RS485_DIR_PORT y RS485_DI↩ R_PIN.

==========

Parameters

in	*None*	**None** *

Definition at line 348 of file uart.c.

7.48.3.6 void UARTInit (uint32_t *baudrate*)

Initialize UART0 port, setup pin select, clock, parity, stop bits, FIFO, etc.

==========

Parameters

in	*uint32_t*	baudrate **None**

Definition at line 238 of file uart.c.

7.48.4 Variable Documentation

7.48.4.1 t_cola cola_rx

Definition at line 52 of file uart.c.

7.48.4.2 t_cola cola_tx

Definition at line 53 of file uart.c.

7.48.4.3 volatile uint32_t contador_timeout = 0

Definition at line 49 of file uart.c.

7.48.4.4 volatile uint32_t contador_vueltas_timeout = 0

Definition at line 50 of file uart.c.

7.48.4.5 volatile uint8_t tx_libre = 1

Definition at line 46 of file uart.c.

7.48.4.6 volatile uint8_t UARTBuffer[BUFSIZE]

Definition at line 47 of file uart.c.

7.48.4.7 volatile uint32_t UARTCount = 0

Definition at line 48 of file uart.c.

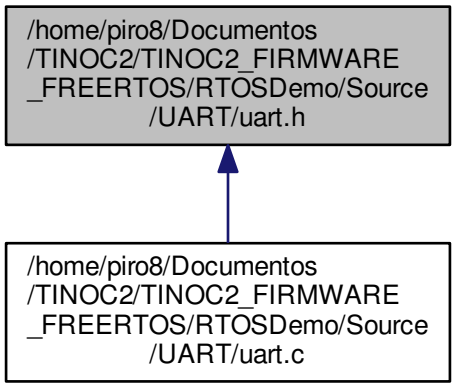
7.48.4.8 volatile uint32_t UARTStatus

Definition at line 45 of file uart.c.

7.49 /home/piro8/Documentos/TINOC2/TINOC2_FIRMWARE_FREERTOS/RTOSDemo/Source/UART/uart.h File Reference

Header del archivo uart.c.

This graph shows which files directly or indirectly include this file:



Macros

- #define RS485_ENABLED 0
- #define TX_INTERRUPT 1 /* 0 if TX uses polling, 1 interrupt driven. */
- #define MODEM_TEST 0
- #define IER_RBR 0x01
- #define IER_THRE 0x02
- #define IER_RLS 0x04
- #define IIR_PEND 0x01
- #define IIR_RLS 0x03

- #define IIR_RDA 0x02
- #define IIR_CTI 0x06
- #define IIR_THRE 0x01
- #define LSR_RDR 0x01
- #define LSR_OE 0x02
- #define LSR_PE 0x04
- #define LSR_FE 0x08
- #define LSR_BI 0x10
- #define LSR_THRE 0x20
- #define LSR_TEMT 0x40
- #define LSR_RXFE 0x80
- #define BUFSIZE 0x40
- #define RS485_NMMEN (0x1 << 0)
- #define RS485_RXDIS (0x1 << 1)
- #define RS485_AADEN (0x1 << 2)
- #define RS485_SEL (0x1 << 3)
- #define RS485_DCTRL (0x1 << 4)
- #define RS485_OINV (0x1 << 5)
- #define U0LCR_WORD_LENGTH_SELECT_5BIT (0x00 << 0)
- #define U0LCR_WORD_LENGTH_SELECT_6BIT (0x01 << 0)
- #define U0LCR_WORD_LENGTH_SELECT_7BIT (0x02 << 0)
- #define U0LCR_WORD_LENGTH_SELECT_8BIT (0x03 << 0)
- #define U0LCR_STOP_BIT_SELECT_1BIT (0x00 << 2)
- #define U0LCR_STOP_BIT_SELECT_2BIT (0x01 << 2)
- #define U0LCR_PARITY_ENABLE (0x01 << 3)
- #define U0LCR_PARITY_SELECT_ODD (0x00 << 4)
- #define U0LCR_PARITY_SELECT_EVEN (0x01 << 4)
- #define U0LCR_PARITY_SELECT_1STICK (0x02 << 4)
- #define U0LCR_PARITY_SELECT_0STICK (0x03 << 4)
- #define U0LCR_BREAK_CONTROL (0x01 << 6)
- #define U0LCR_DLAB (0x01 << 7)
- #define U0FCR_FIFO_ENABLE (0x01 << 0)
- #define U0FCR_RX_FIFO_RESET (0x01 << 1)
- #define U0FCR_TX_FIFO_RESET (0x01 << 2)
- #define U0FCR_RX_TRIGGER_LEVEL_1 (0x00 << 6)
- #define U0FCR_RX_TRIGGER_LEVEL_2 (0x01 << 6)
- #define U0FCR_RX_TRIGGER_LEVEL_3 (0x02 << 6)
- #define U0FCR_RX_TRIGGER_LEVEL_4 (0x03 << 6)
- #define BAUD_RATE_DEFAULT 9600
- #define BAUD_RATE BAUD_RATE_DEFAULT
- #define _RS485_USAR_RTS_DIR
- #define RTS_PORT 1
- #define RTS_PIN 5
- #define RS485_DIR_PORT RTS_PORT
- #define RS485_DIR_PIN RTS_PIN
- #define RS485_DEFINIR_PIN() GPIOSetDir(RS485_DIR_PORT, RS485_DIR_PIN, 1)
- #define RS485_HABILITAR_TX() GPIOSetValue(RS485_DIR_PORT, RS485_DIR_PIN, 1)
- #define RS485_HABILITAR_RX() GPIOSetValue(RS485_DIR_PORT, RS485_DIR_PIN, 0)

Functions

- void UART_IRQHandler (void)

  UART interrupt handler.
- void ModemInit (void)
- void UARTInit (uint32_t baudrate)

  Initialize UART0 port, setup pin select, clock, parity, stop bits, FIFO, etc.
- int Inic_UART_RS485 (void)

  Inicializo la UART para funcionar en RS485.
- int UART_RS485_Habilitar_Recepcion (void)

  Habilito la recepcion para la norma RS485 poniendo en alto el pin definido por RS485_DIR_PORT y RS485_DIR↩ _PIN.
- int UART_RS485_Deshabilitar_Recepcion (void)

  Habilito la recepcion para la norma RS485 poniendo en alto el pin definido por RS485_DIR_PORT y RS485_DIR↩ _PIN.
- void UARTSend (uint8_t ∗BufferPtr, uint32_t Length)

Variables

- volatile uint32_t UARTStatus
- volatile uint8_t tx_libre
- volatile uint8_t UARTBuffer [BUFSIZE]
- volatile uint32_t UARTCount
- volatile uint32_t contador_timeout
- volatile uint32_t contador_vueltas_timeout
- t_cola cola_rx
- t_cola cola_tx

7.49.1 Detailed Description

Header del archivo uart.c.

Date

Feb 22, 2018

Author

Roux, Federico G. (froux@citedef.gob.ar)

7.49.2 Macro Definition Documentation

7.49.2.1 #define _RS485_USAR_RTS_DIR

Definition at line 97 of file uart.h.

7.49.2.2 #define BAUD_RATE BAUD_RATE_DEFAULT

Definition at line 93 of file uart.h.

7.49.2.3 #define BAUD_RATE_DEFAULT 9600

Definition at line 90 of file uart.h.

7.49.2.4 #define BUFSIZE 0x40

Definition at line 43 of file uart.h.

7.49.2.5 #define IER_RBR 0x01

Definition at line 24 of file uart.h.

7.49.2.6 #define IER_RLS 0x04

Definition at line 26 of file uart.h.

7.49.2.7 #define IER_THRE 0x02

Definition at line 25 of file uart.h.

7.49.2.8 #define IIR_CTI 0x06

Definition at line 31 of file uart.h.

7.49.2.9 #define IIR_PEND 0x01

Definition at line 28 of file uart.h.

7.49.2.10 #define IIR_RDA 0x02

Definition at line 30 of file uart.h.

7.49.2.11 #define IIR_RLS 0x03

Definition at line 29 of file uart.h.

7.49.2.12 #define IIR_THRE 0x01

Definition at line 32 of file uart.h.

7.49.2.13 #define LSR_BI 0x10

Definition at line 38 of file uart.h.

7.49.2.14 #define LSR_FE 0x08

Definition at line 37 of file uart.h.

7.49.2.15 #define LSR_OE 0x02

Definition at line 35 of file uart.h.

7.49.2.16 #define LSR_PE 0x04

Definition at line 36 of file uart.h.

7.49.2.17 #define LSR_RDR 0x01

Definition at line 34 of file uart.h.

7.49.2.18 #define LSR_RXFE 0x80

Definition at line 41 of file uart.h.

7.49.2.19 #define LSR_TEMT 0x40

Definition at line 40 of file uart.h.

7.49.2.20 #define LSR_THRE 0x20

Definition at line 39 of file uart.h.

7.49.2.21 #define MODEM_TEST 0

Definition at line 21 of file uart.h.

7.49.2.22 #define RS485_AADEN (0x1 << 2)

Definition at line 49 of file uart.h.

7.49.2.23 #define RS485_DCTRL (0x1 << 4)

Definition at line 51 of file uart.h.

7.49.2.24 #define RS485_DEFINIR_PIN( ) GPIOSetDir(RS485_DIR_PORT, RS485_DIR_PIN, 1)

Definition at line 114 of file uart.h.

7.49.2.25 #define RS485_DIR_PIN RTS_PIN

Definition at line 112 of file uart.h.

7.49.2.26 #define RS485_DIR_PORT RTS_PORT

Definition at line 111 of file uart.h.

7.49.2.27 #define RS485_ENABLED 0

Definition at line 19 of file uart.h.

7.49.2.28 #define RS485_HABILITAR_RX( ) GPIOSetValue(RS485_DIR_PORT, RS485_DIR_PIN, 0)

Definition at line 116 of file uart.h.

7.49.2.29 #define RS485_HABILITAR_TX( ) GPIOSetValue(RS485_DIR_PORT, RS485_DIR_PIN, 1)

Definition at line 115 of file uart.h.

7.49.2.30 #define RS485_NMMEN (0x1 << 0)

Definition at line 47 of file uart.h.

7.49.2.31 #define RS485_OINV (0x1 << 5)

Definition at line 52 of file uart.h.

7.49.2.32 #define RS485_RXDIS (0x1 << 1)

Definition at line 48 of file uart.h.

7.49.2.33 #define RS485_SEL (0x1 << 3)

Definition at line 50 of file uart.h.

7.49.2.34 #define RTS_PIN 5

Definition at line 100 of file uart.h.

7.49.2.35 #define RTS_PORT 1

Definition at line 99 of file uart.h.

7.49.2.36 #define TX_INTERRUPT 1 /* 0 if TX uses polling, 1 interrupt driven. */

Definition at line 20 of file uart.h.

7.49.2.37 #define U0FCR_FIFO_ENABLE (0x01 << 0)

Definition at line 78 of file uart.h.

7.49.2.38 #define U0FCR_RX_FIFO_RESET (0x01 << 1)

Definition at line 80 of file uart.h.

7.49.2.39 #define U0FCR_RX_TRIGGER_LEVEL_1 (0x00 << 6)

Definition at line 83 of file uart.h.

7.49.2.40 #define U0FCR_RX_TRIGGER_LEVEL_2 (0x01 << 6)

Definition at line 84 of file uart.h.

7.49.2.41 #define U0FCR_RX_TRIGGER_LEVEL_3 (0x02 << 6)

Definition at line 85 of file uart.h.

7.49.2.42 #define U0FCR_RX_TRIGGER_LEVEL_4 (0x03 << 6)

Definition at line 86 of file uart.h.

7.49.2.43 #define U0FCR_TX_FIFO_RESET (0x01 << 2)

Definition at line 81 of file uart.h.

7.49.2.44 #define U0LCR_BREAK_CONTROL (0x01 << 6)

Definition at line 71 of file uart.h.

7.49.2.45 #define U0LCR_DLAB (0x01 << 7)

Definition at line 74 of file uart.h.

7.49.2.46 #define U0LCR_PARITY_ENABLE (0x01 << 3)

Definition at line 64 of file uart.h.

7.49.2.47 #define U0LCR_PARITY_SELECT_0STICK (0x03 << 4)

Definition at line 69 of file uart.h.

7.49.2.48 #define U0LCR_PARITY_SELECT_1STICK (0x02 << 4)

Definition at line 68 of file uart.h.

7.49.2.49 #define U0LCR_PARITY_SELECT_EVEN (0x01 << 4)

Definition at line 67 of file uart.h.

7.49.2.50 #define U0LCR_PARITY_SELECT_ODD (0x00 << 4)

Definition at line 66 of file uart.h.

7.49.2.51 #define U0LCR_STOP_BIT_SELECT_1BIT (0x00 << 2)

Definition at line 61 of file uart.h.

7.49.2.52 #define U0LCR_STOP_BIT_SELECT_2BIT (0x01 << 2)

Definition at line 62 of file uart.h.

7.49.2.53 #define U0LCR_WORD_LENGTH_SELECT_5BIT (0x00 << 0)

Definition at line 56 of file uart.h.

7.49.2.54 #define U0LCR_WORD_LENGTH_SELECT_6BIT (0x01 << 0)

Definition at line 57 of file uart.h.

7.49.2.55 #define U0LCR_WORD_LENGTH_SELECT_7BIT (0x02 << 0)

Definition at line 58 of file uart.h.

7.49.2.56 #define U0LCR_WORD_LENGTH_SELECT_8BIT (0x03 << 0)

Definition at line 59 of file uart.h.

7.49.3 Function Documentation

7.49.3.1 int Inic_UART_RS485 (void)

Inicializo la UART para funcionar en RS485.

==========

Parameters

in	*None*	**None**

Definition at line 317 of file uart.c.

7.49.3.2 void ModemInit (void)

7.49.3.3 void UART_IRQHandler (void)

UART interrupt handler.

==========

Parameters

None	**None**

Definition at line 67 of file uart.c.

7.49.3.4 int UART_RS485_Deshabilitar_Recepcion (void)

Habilito la recepcion para la norma RS485 poniendo en alto el pin definido por RS485_DIR_PORT y RS485_DI↩ R_PIN.

==========

Parameters

in	*None*	**None**

Definition at line 365 of file uart.c.

7.49.3.5 int UART_RS485_Habilitar_Recepcion (void)

Habilito la recepcion para la norma RS485 poniendo en alto el pin definido por RS485_DIR_PORT y RS485_DI↩ R_PIN.

==========

Parameters

in	*None*	**None** *

Definition at line 348 of file uart.c.

7.49.3.6 void UARTInit (uint32_t *baudrate*)

Initialize UART0 port, setup pin select, clock, parity, stop bits, FIFO, etc.

==========

Parameters

in	*uint32_t*	baudrate **None**

Definition at line 238 of file uart.c.

7.49.3.7 void UARTSend (uint8_t * *BufferPtr,* uint32_t *Length*)

7.49.4 Variable Documentation

7.49.4.1 t_cola cola_rx

Definition at line 52 of file uart.c.

7.49.4.2 t_cola cola_tx

Definition at line 53 of file uart.c.

7.49.4.3 volatile uint32_t contador_timeout

Definition at line 49 of file uart.c.

7.49.4.4 volatile uint32_t contador_vueltas_timeout

Definition at line 50 of file uart.c.

7.49.4.5 volatile uint8_t tx_libre

Definition at line 46 of file uart.c.

7.49.4.6 volatile uint8_t UARTBuffer[BUFSIZE]

Definition at line 47 of file uart.c.

7.49.4.7 volatile uint32_t UARTCount

Definition at line 48 of file uart.c.

7.49.4.8 volatile uint32_t UARTStatus

Definition at line 45 of file uart.c.