Contents

- Queue management
- Software timer management

HAN_UNIVERSITY
OF APPLIED SCIENCES

Queue management

Barry, R. (2016). *Mastering the freertos real time kernel. Pre-release 161204 Edition*. Real Time Engineers Ltd.
Chapter 4

Inter-task communication

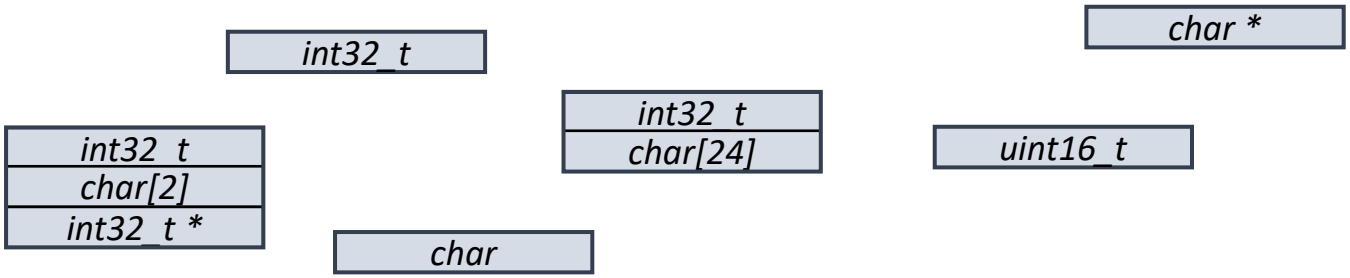
Queues provides an inter-task communicating mechanism

- Task-to-task
- Task-to-interrupt
- Interrupt-to-task

Characteristics of a Queue

Abstract Data Type

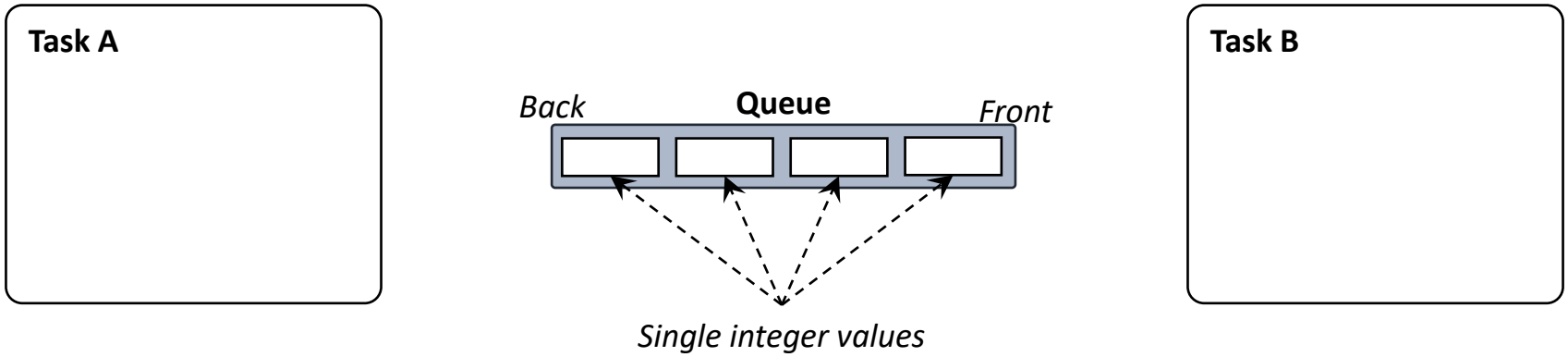
- Stores a copy of data items
- The number of data items is called a queue's **length**
- FIFO, although it is possible to overwrite the front data item in the queue
- Can be accessed by any task or ISR
- Queues are kernel objects, similar as tasks, so they require kernel resources (e.g. heap memory)



Characteristics of a Queue

Example

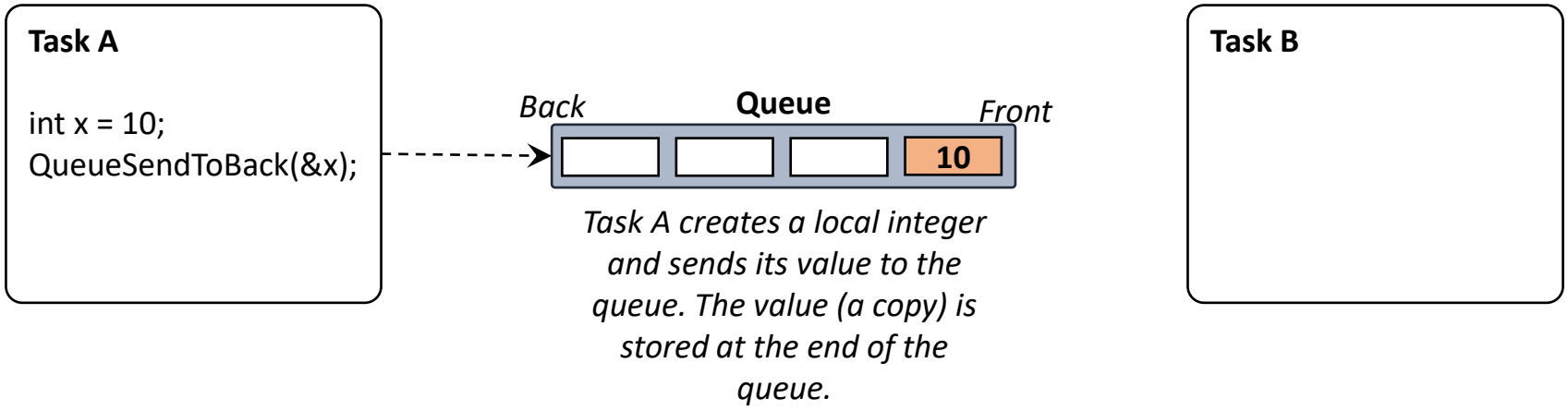
- Create two tasks A and B
- Create a queue that communicates from Task A to Task B
- The queue must be able to store 4 integer items



Characteristics of a Queue

Example

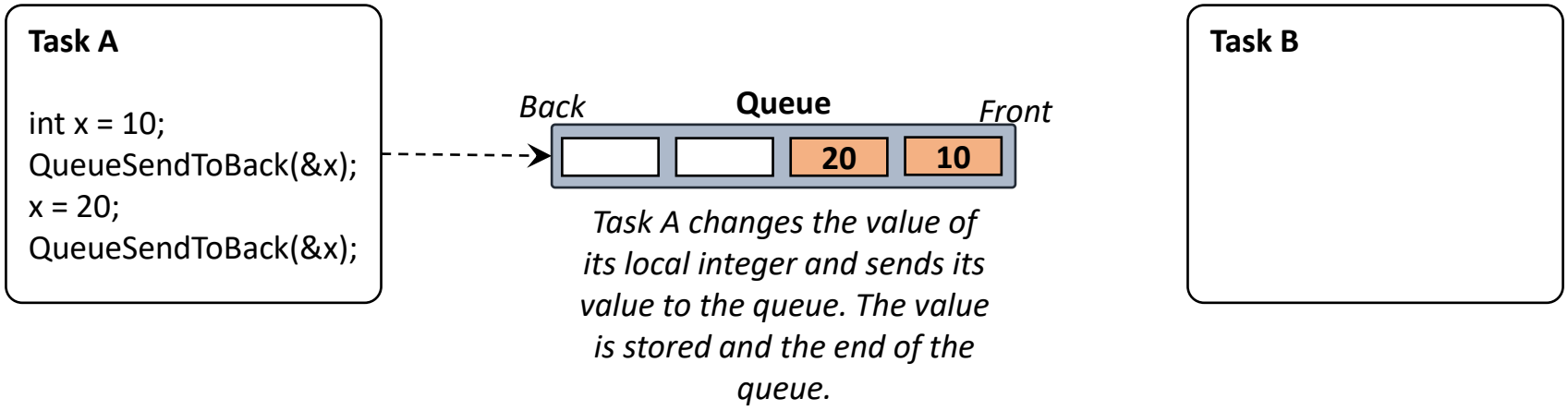
- Create two tasks A and B
- Create a queue that communicates from Task A to Task B
- The queue must be able to store 4 integer items



Characteristics of a Queue

Example

- Create two tasks A and B
- Create a queue that communicates from Task A to Task B
- The queue must be able to store 4 integer items



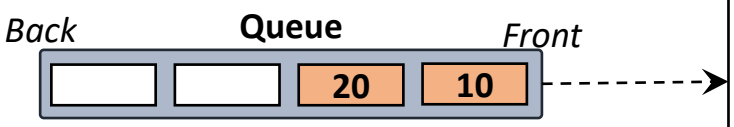
Characteristics of a Queue

Example

- Create two tasks A and B
- Create a queue that communicates from Task A to Task B
- The queue must be able to store 4 integer items

Task A

```
int x = 10;
QueueSendToBack(&x);
x = 20;
QueueSendToBack(&x);
```



Task B is at some point ready to receive data from the queue. By calling the receive function, the value at the front of the queue will be read.

Task B

```
int y;
QueueReceive(&y);
```


HAN_UNIVERSITY OF APPLIED SCIENCES

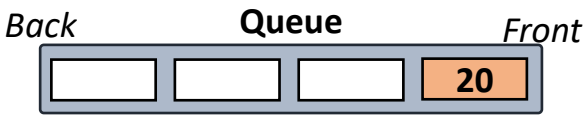
Characteristics of a Queue

Example

- Create two tasks A and B
- Create a queue that communicates from Task A to Task B
- The queue must be able to store 4 integer items

Task A

```
int x = 10;
QueueSendToBack(&x);
x = 20;
QueueSendToBack(&x);
```



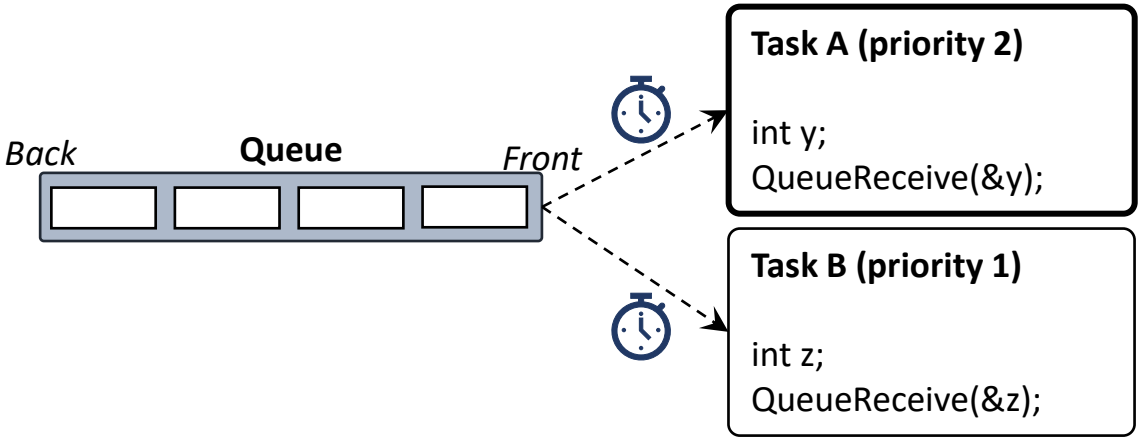
By reading from the queue, the item at the front is removed and the next item becomes the front item.

Task B

```
int y;
QueueReceive(&y);
```

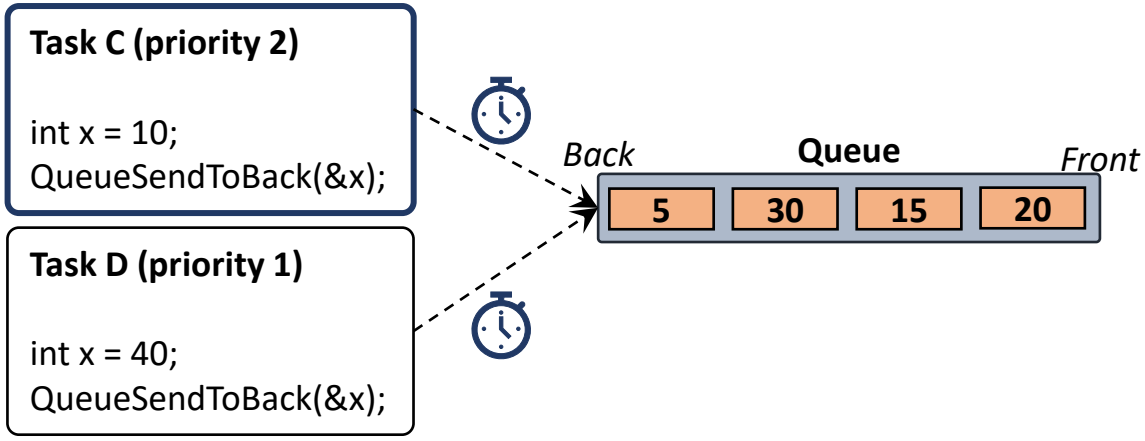
Characteristics of a Queue

- Blocking on queue reads
  - When attempting to read from a queue, a task can specify a **block** time
  - The scheduler will keep the task in Blocked state to wait for data to be available or expiration of the block time
  - Queues can have multiple *readers*, so only one task (the one with the highest priority) will leave the blocked state if new data arrives



Characteristics of a Queue

- Blocking on queue writes
  - When attempting to write to a full queue, a task can specify a **block** time
  - The scheduler will keep the task in Blocked state to wait for space to be available or expiration of the block time
  - Queues can have multiple *writers*, so only one task (the one with the highest priority) will leave the blocked state if space opens up



HAN_UNIVERSITY
OF APPLIED SCIENCES

Using a Queue

- Queues are referenced by so-called **handles**: QueueHandle_t
- The kernel allocates RAM from the FreeRTOS heap when a queue is created

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

Listing 40. The xQueueCreate() API function prototype

The maximum number of items that the queue being created can hold at any one time.

Using a Queue

- Queues are referenced by so-called **handles**: QueueHandle_t
- The kernel allocates RAM from the FreeRTOS heap when a queue is created

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

Listing 40. The xQueueCreate() API function prototype

The size in bytes of each data item that can be stored in the queue.

Using a Queue

- Queues are referenced by so-called **handles**: QueueHandle_t
- The kernel allocates RAM from the FreeRTOS heap when a queue is created

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

Listing 40. The xQueueCreate() API function prototype

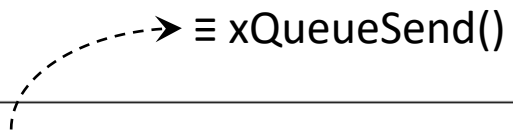
If NULL is returned, then the queue cannot be created because there is insufficient heap memory available

A non-NULL value being returned indicates that the queue has been created successfully.

Using a Queue

Three functions to send data to a queue.

Never call these functions from an ISR, because this is not safe.




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

Listing 42. The xQueueSendToBack() API function prototype

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

Listing 41. The xQueueSendToFront() API function prototype

Using a Queue

One function to read data from a queue.

Never call this function from an ISR, because this is not safe.

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

Listing 43. The xQueueReceive() API function prototype

Using a Queue

One function to query the number of items currently in the queue.

Never call this function from an ISR, because this is not safe.

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

Listing 44. The uxQueueMessagesWaiting() API function prototype

Example (see also Week 3 – Example 01)

```
/* Declare a variable of type QueueHandle_t.  This is used to store the handle
to the queue that is accessed by all three tasks. */
QueueHandle_t xQueue;

int main( void )
{
    /* The queue is created to hold a maximum of 5 values, each of which is
    large enough to hold a variable of type int32_t. */
    xQueue = xQueueCreate( 5, sizeof( int32_t ) );

    if( xQueue != NULL )
    {
        /* Create two instances of the task that will send to the queue.  The task
        parameter is used to pass the value that the task will write to the queue,
        so one task will continuously write 100 to the queue while the other task
        will continuously write 200 to the queue.  Both tasks are created at
        priority 1. */
        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

        /* Create the task that will read from the queue.  The task is created with
        priority 2, so above the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
        /* The queue could not be created. */
    }

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient FreeRTOS heap memory available for the idle task to be
    created. Chapter 2 provides more information on heap memory management. */
    for( ;; );
}
```

Listing 47. The implementation of main() in Example 10

Example (see also Week 3 – Example 01)

```
static void vSenderTask( void *pvParameters )
{
int32_t lValueToSend;
BaseType_t xStatus;

    /* Two instances of this task are created so the value that is sent to the
    queue is passed in via the task parameter - this way each instance can use
    a different value.  The queue was created to hold values of type int32_t,
    so cast the parameter to the required type. */
    lValueToSend = ( int32_t ) pvParameters;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Send the value to the queue.

        The first parameter is the queue to which data is being sent.  The
        queue was created before the scheduler was started, so before this task
        started to execute.

        The second parameter is the address of the data to be sent, in this case
        the address of lValueToSend.

        The third parameter is the Block time – the time the task should be kept
        in the Blocked state to wait for space to become available on the queue
        should the queue already be full.  In this case a block time is not
        specified because the queue should never contain more than one item, and
        therefore never be full. */
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );

        if( xStatus != pdPASS )
        {
            /* The send operation could not complete because the queue was full -
            this must be an error as the queue should never contain more than
            one item! */
            vPrintString( "Could not send to the queue.\r\n" );
        }
    }
}
```

Listing 45. Implementation of the sending task used in Example 10.

Example (see also Week 3 – Example 01)

```
static void vReceiverTask( void *pvParameters )
{
/* Declare the variable that will hold the values received from the queue. */
int32_t lReceivedValue;
BaseType_t xStatus;
const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* This call should always find the queue empty because this task will
        immediately remove any data that is written to the queue. */
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            vPrintString( "Queue should have been empty!\r\n" );
        }

        /* Receive data from the queue.

        The first parameter is the queue from which data is to be received.  The
        queue is created before the scheduler is started, and therefore before this
        task runs for the first time.

        The second parameter is the buffer into which the received data will be
        placed.  In this case the buffer is simply the address of a variable that
        has the required size to hold the received data.

        The last parameter is the block time – the maximum amount of time that the
        task will remain in the Blocked state to wait for data to be available
        should the queue already be empty. */
        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );

        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received
            value. */
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
        else
        {
            /* Data was not received from the queue even after waiting for 100ms.
            This must be an error as the sending tasks are free running and will be
            continuously writing to the queue. */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}
```

Listing 46. Implementation of the receiver task for Example 10

Example (see also Week 3 – Example 01)

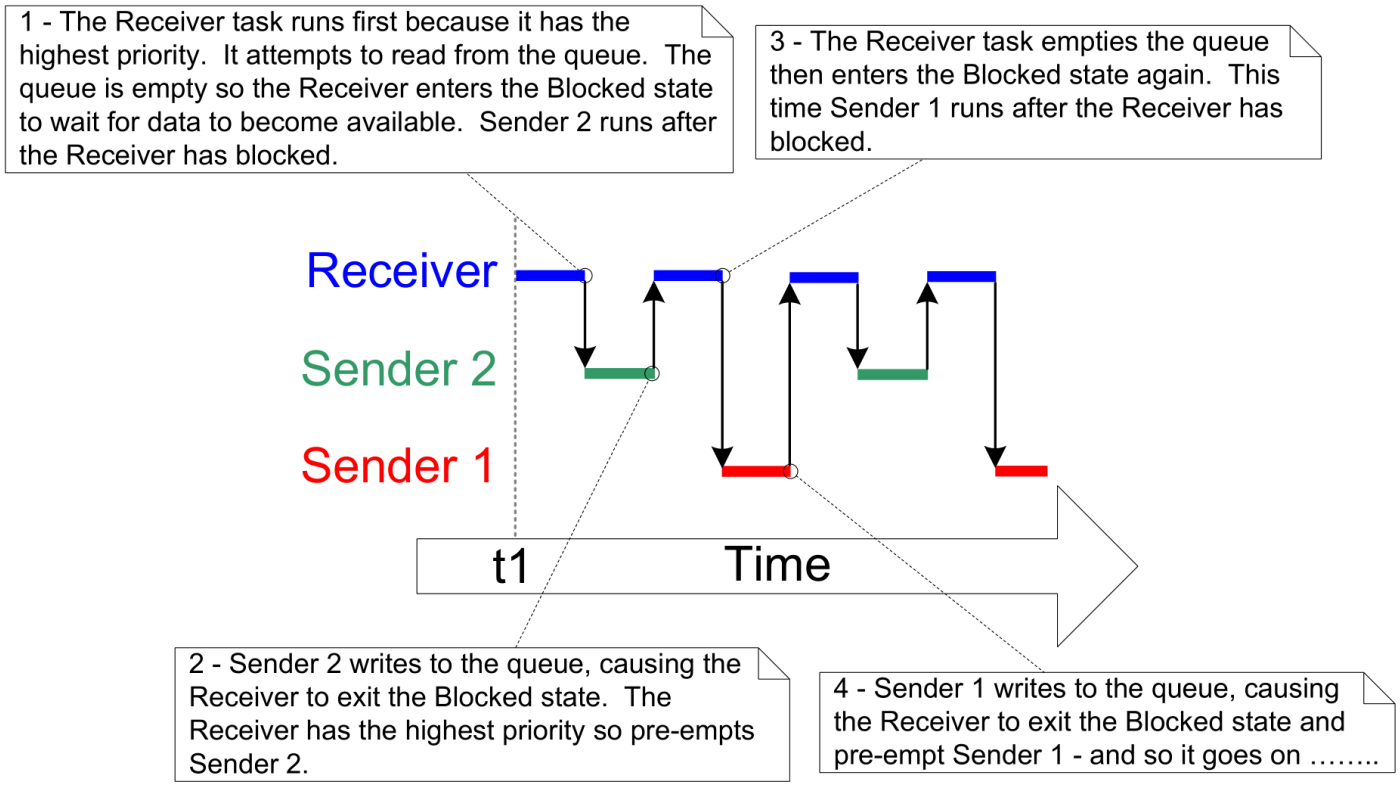


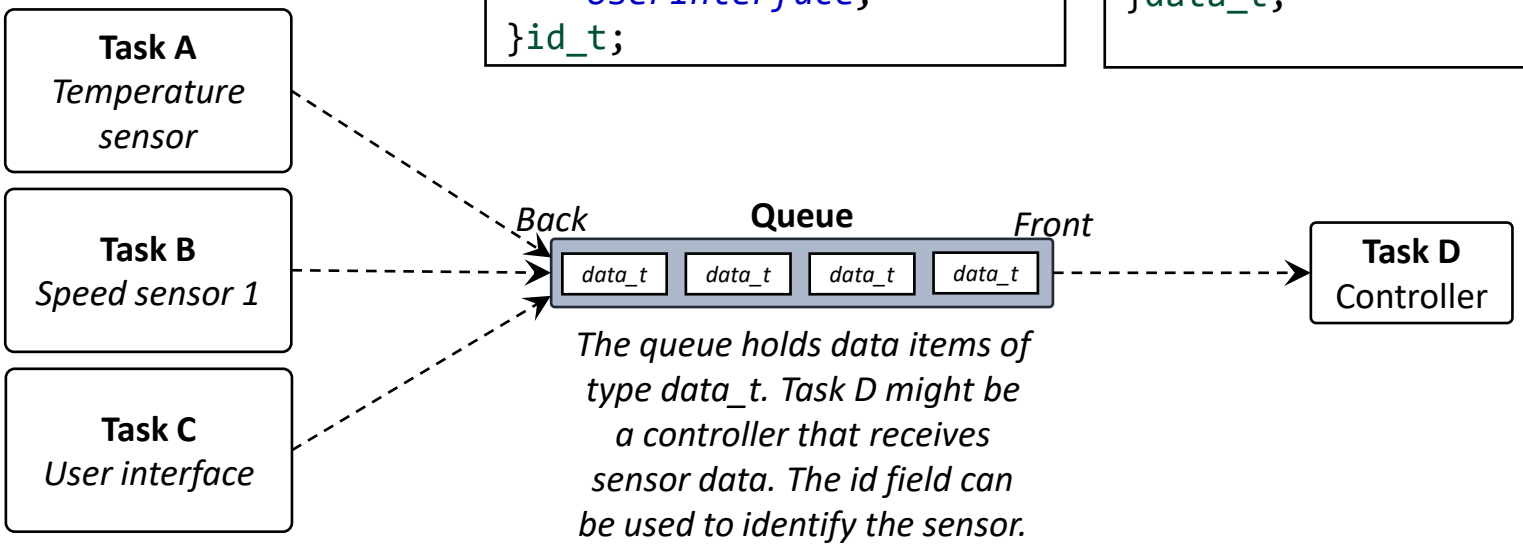
Figure 33. The sequence of execution produced by Example 10

Receiving Data from Multiple Sources

A common FreeRTOS design pattern is to use a queue to receive data from multiple sources

```
typedef enum
{
    TemperatureSensor,
    SpeedSensor,
    UserInterface,
}id_t;
```

```
typedef struct
{
    id_t id;
    int32_t value;
}data_t;
```



The queue holds data items of type data_t. Task D might be a controller that receives sensor data. The id field can be used to identify the sensor.

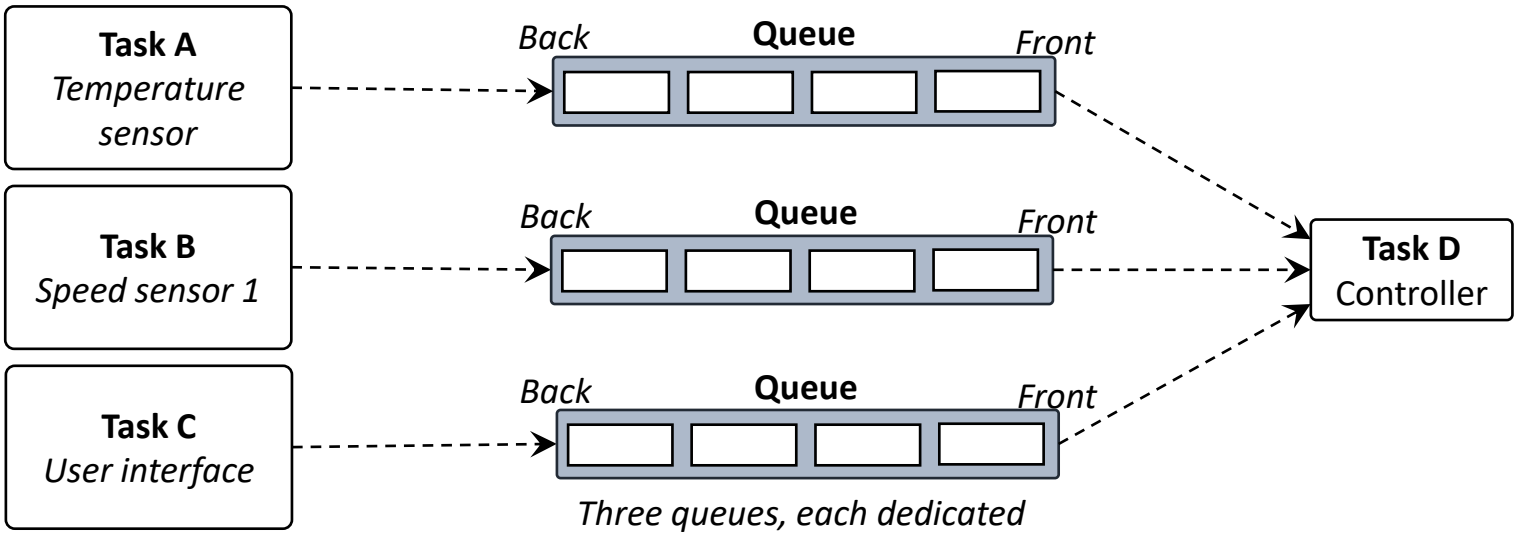
Working With Large or Variable Size Data

- So far, data values are copied into the queue
- Instead of copying all the data, we can also send pointers
- Advantages
  - Less RAM required
  - More efficient processing time
- Disadvantages
  - Must make sure the memory is not updated by more than one task at a time
  - Must make sure that the memory remains valid especially with dynamic memory allocation

- An example that uses both pointers to data and structures is the implementation of FreeRTOS+TCP TCP/IP stack

Queue Sets

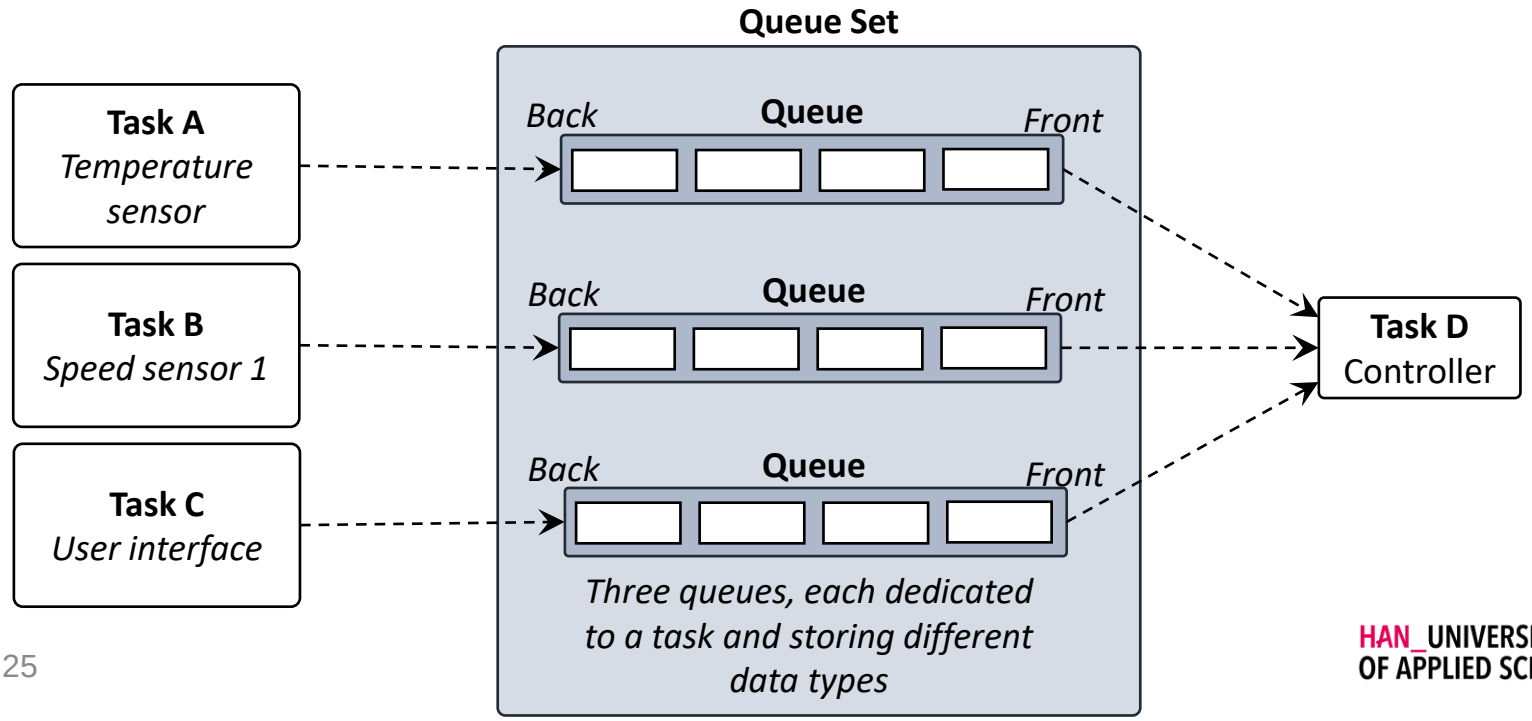
Alternative design pattern to receive data from multiple tasks



Queue Sets

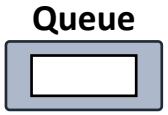
Alternative design pattern to receive data from multiple tasks

Queues can be combined into a set to allow a task to receive data from multiple queues without having to poll each queue



Mailbox

- In a mailbox, data is does not pass through, but instead remains in the mailbox until it is overwritten. In other words, the receiver reads from a mailbox, but does not remove the value.
- A mailbox in FreeRTOS actually is a queue that has a length of one.
- Using a queue as a mailbox requires two additional queue functions: xQueueOverwrite() and xQueuePeek().



Mailbox

Overwrite the data that is already in the queue.

Never call this function from an ISR, because this is not safe.

Use this function only with a queue of length one

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

Listing 68. The xQueueOverwrite() API function prototype

Mailbox

Read one item without removing it or changing the queue order.

Never call this function from an ISR, because this is not safe.

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

Listing 70. The xQueuePeek() API function prototype

Summary

- How to create a queue.
- How a queue manages the data it contains.
- How to send data to a queue.
- How to receive data from a queue.
- What it means to block on a queue.
- How to block on multiple queues.
- How to overwrite data in a queue.
- How to clear a queue.
- Mailbox implementation with a queue

Software timer management

Barry, R. (2016). *Mastering the freertos real time kernel. Pre-release 161204 Edition*. Real Time Engineers Ltd.
Chapter 5

Software Timers

Software timers are used to schedule the execution of a function

- at a set time in the future
- periodically with a fixed frequency

The function executed by the software timer is called the software timer's callback function.

Software timers are kernel objects and not related to hardware timers of a microcontroller

Software timer functionality is optional

- FreeRTOS/Source/timers.c
- Set configUSE_TIMERS to 1 in FreeRTOSConfig.h

Software Timer Callback Functions

- Implemented as a plain C function
- All callback functions are executed in the context of a special FreeRTOS task, also known as a **daemon** task

```
void ATimerCallback( TimerHandle_t xTimer );
```

Listing 72. The software timer callback function prototype

Handle to a software timer

Attributes and States of a Software Timer

Two types of software timers

1. One-shot timers
2. Auto-reload timers

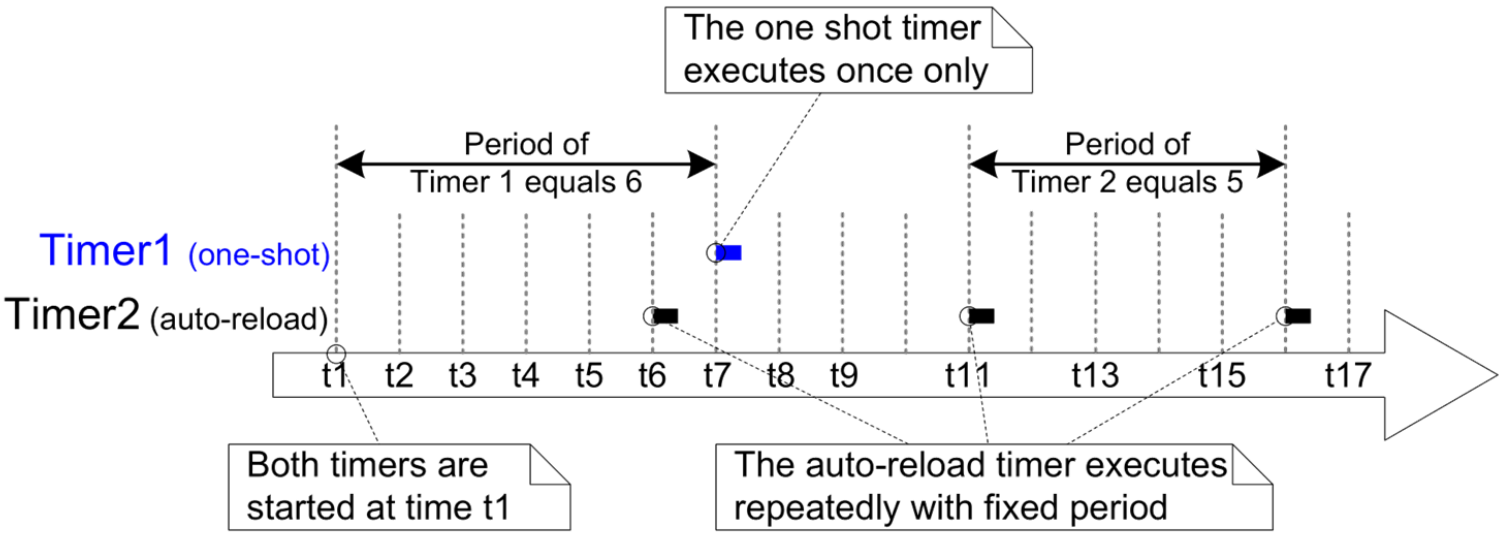


Figure 38 The difference in behavior between one-shot and auto-reload software timers

Software Timer States

A software timer can be in one of the following two states:

- Dormant
- Running

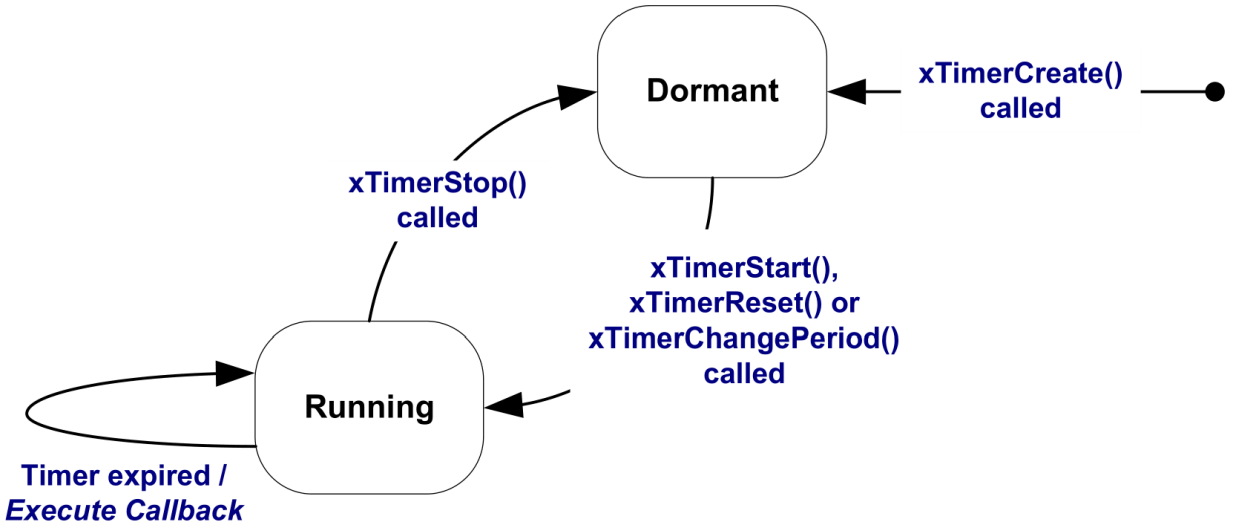


Figure 39 Auto-reload software timer states and transitions

Software Timer States

A software timer can be in one of the following two states:

- Dormant
- Running

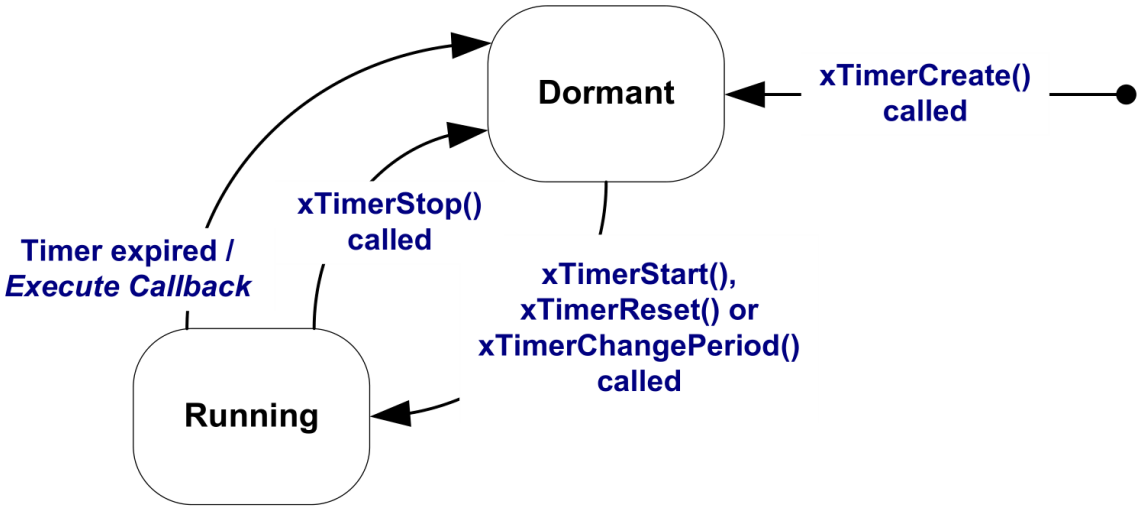


Figure 40 One-shot software timer states and transitions

The Context of a Software Timer

All software timer callback functions execute in the context of the same FreeRTOS daemon task.

The daemon task is created automatically and configured with

- configTIMER_TASK_PRIORITY
- configTIMER_TASK_STACK_DEPTH

Timer callback functions should not use functions that causes the task to enter the blocking state. Otherwise, the daemon will enter blocking state!

(For example, by using vTaskDelay() or xQueueReceive() with xTicksToWait parameter not set to zero)

Keep code in callback functions as short as possible

The Timer Command Queue

Software timer API functions send commands to the daemon task by using a queue (called *TmrQ*). The length of that queue is configured with configTIMER_QUEUE_LENGTH.

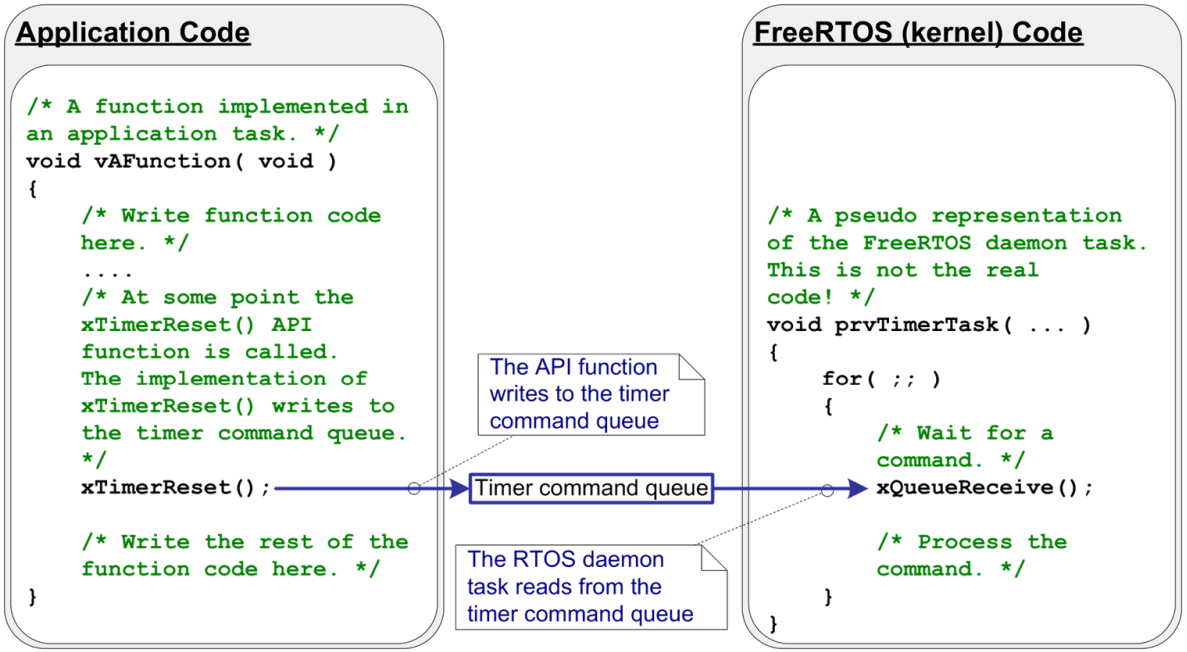


Figure 41 The timer command queue being used by a software timer API function to communicate with the RTOS daemon task

HAN_UNIVERSITY OF APPLIED SCIENCES

Daemon Task Scheduling

The daemon task is scheduled like any other FreeRTOS task. The default task name is *Tmr Svc* and can be changed by defining configTIMER_SERVICE_TASK_NAME.

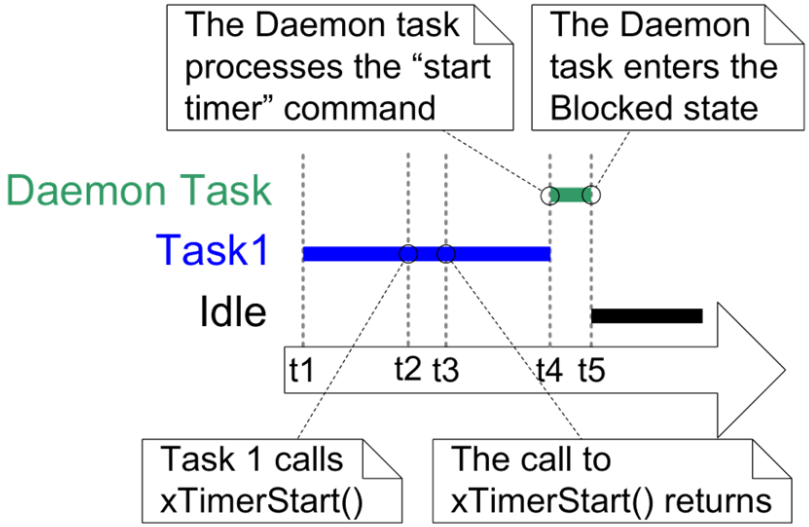


Figure 42 The execution pattern when the priority of a task calling xTimerStart() is above the priority of the daemon task

Daemon Task Scheduling

The daemon task is scheduled like any other FreeRTOS task. The default task name is *Tmr Svc* and can be changed by defining configTIMER_SERVICE_TASK_NAME.

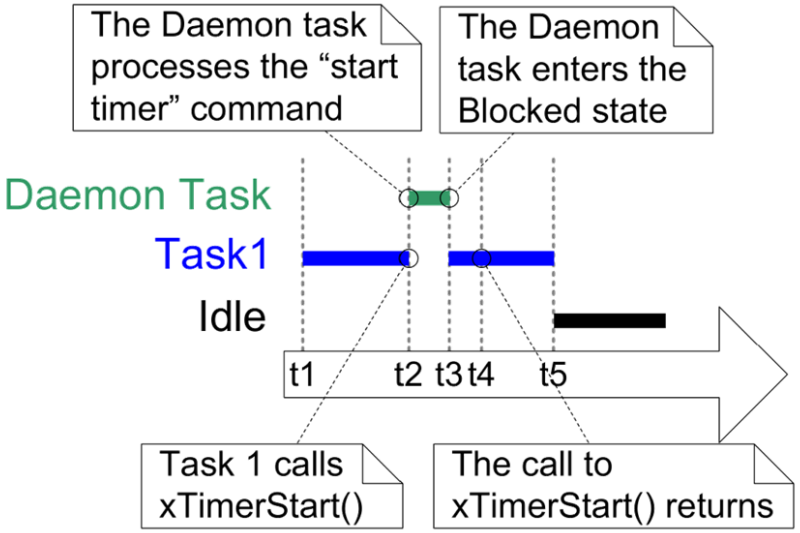


Figure 43 The execution pattern when the priority of a task calling xTimerStart() is below the priority of the daemon task

Example (see also Week 3 – Example 02)

```
/* The periods assigned to the one-shot and auto-reload timers are 3.333 second and half a
second respectively. */
#define mainONE_SHOT_TIMER_PERIOD     pdMS_TO_TICKS( 3333 )
#define mainAUTO_RELOAD_TIMER_PERIOD  pdMS_TO_TICKS( 500 )

int main( void )
{
TimerHandle_t xAutoReloadTimer, xOneShotTimer;
BaseType_t xTimer1Started, xTimer2Started;

   /* Create the one shot timer, storing the handle to the created timer in xOneShotTimer. */
   xOneShotTimer = xTimerCreate(
                   /* Text name for the software timer - not used by FreeRTOS. */
                   "OneShot",
                   /* The software timer's period in ticks. */
                   mainONE_SHOT_TIMER_PERIOD,
                   /* Setting uxAutoRealod to pdFALSE creates a one-shot software timer. */
                   pdFALSE,
                   /* This example does not use the timer id. */
                   0,
                   /* The callback function to be used by the software timer being created. */
                   prvOneShotTimerCallback );

   /* Create the auto-reload timer, storing the handle to the created timer in xAutoReloadTimer. */
   xAutoReloadTimer = xTimerCreate(
                   /* Text name for the software timer - not used by FreeRTOS. */
                   "AutoReload",
                   /* The software timer's period in ticks. */
                   mainAUTO_RELOAD_TIMER_PERIOD,
                   /* Setting uxAutoRealod to pdTRUE creates an auto-reload timer. */
                   pdTRUE,
                   /* This example does not use the timer id. */
                   0,
                   /* The callback function to be used by the software timer being created. */
                   prvAutoReloadTimerCallback );

   /* Check the software timers were created. */
   if( ( xOneShotTimer != NULL ) && ( xAutoReloadTimer != NULL ) )
   {
       /* Start the software timers, using a block time of 0 (no block time).  The scheduler has
       not been started yet so any block time specified here would be ignored anyway. */
       xTimer1Started = xTimerStart( xOneShotTimer, 0 );
       xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );

       /* The implementation of xTimerStart() uses the timer command queue, and xTimerStart()
       will fail if the timer command queue gets full.  The timer service task does not get
       created until the scheduler is started, so all commands sent to the command queue will
       stay in the queue until after the scheduler has been started.  Check both calls to
       xTimerStart() passed. */
       if( ( xTimer1Started == pdPASS ) && ( xTimer2Started == pdPASS ) )
       {
           /* Start the scheduler. */
           vTaskStartScheduler();
       }
   }

   /* As always, this line should not be reached. */
   for( ;; );
}
```

Listing 75. Creating and starting the timers used in Example 13

Example (see also Week 3 – Example 02)

```
static void prvOneShotTimerCallback( TimerHandle_t xTimer )
{
TickType_t xTimeNow;

    /* Obtain the current tick count. */
    xTimeNow = xTaskGetTickCount();

    /* Output a string to show the time at which the callback was executed. */
    vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );

    /* File scope variable. */
    ulCallCount++;
}
```

Listing 76. The callback function used by the one-shot timer in Example 13

Example (see also Week 3 – Example 02)

```
static void prvAutoReloadTimerCallback( TimerHandle_t xTimer )
{
TickType_t xTimeNow;

    /* Obtain the current tick count. */
    xTimeNow = uxTaskGetTickCount();

    /* Output a string to show the time at which the callback was executed. */
    vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow );

    ulCallCount++;
}
```

Listing 77. The callback function used by the auto-reload timer in Example 13

Changing the Period of a Timer

Change the period of a software timer.

Never call this function from an ISR, because this is not safe.

Running timer: the new expiration time is relative to when xTimerChangePeriod() was called.

Dormant timer: the new expiration time is relative to when xTimerChangePeriod() and the timer transitions into the Running state.

```
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer,
                               TickType_t xNewTimerPeriodInTicks,
                               TickType_t xTicksToWait );
```

Listing 82. The xTimerChangePeriod() API function prototype

Resetting a Software Timer

Resetting a software timer means to re-start the timer

The expiration time is recalculated to be relative to when the timer was reset

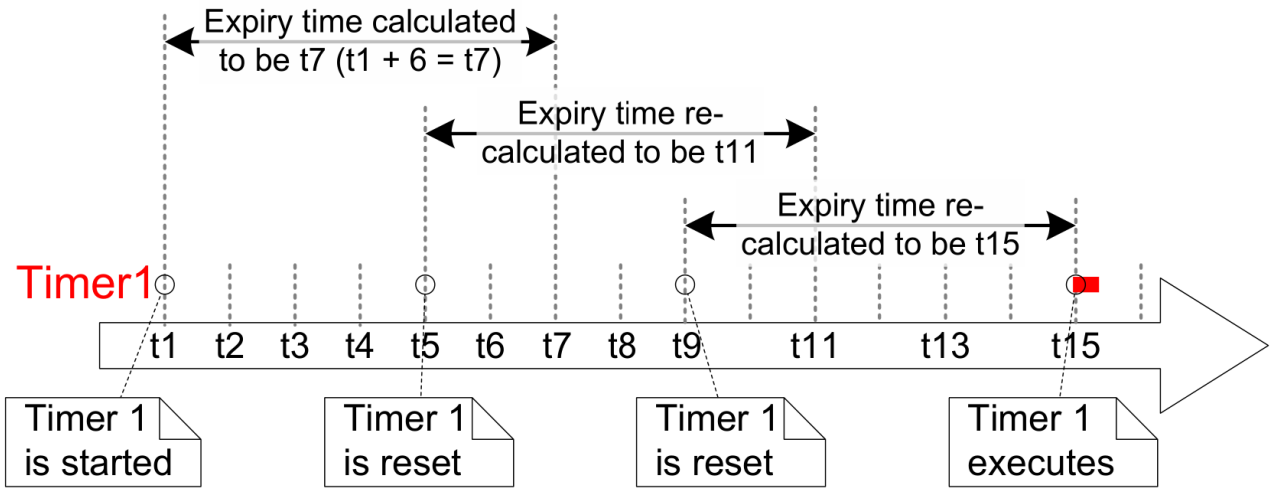


Figure 46 Starting and resetting a software timer that has a period of 6 ticks

Resetting a Software Timer (see also Week 3 – Example 03)

Demo usage

- Turn on a backlight (LED) when a key in the serial monitor is pressed
- The backlight remains on provided further keys are pressed within a certain time period
- Automatically turns off if no key presses are detected within a certain time period

Summary

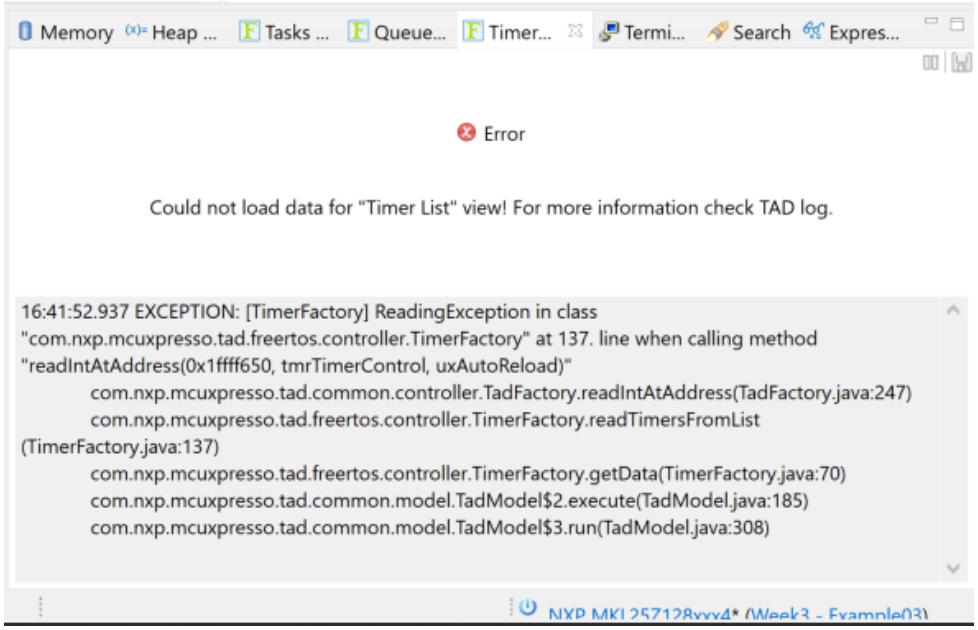
- The characteristics of a software timer compared to the characteristics of a task.
- The RTOS daemon task.
- The timer command queue.
- The difference between a one shot software timer and a periodic software timer.
- How to create, start, reset and change the period of a software timer.

Note

15/02/2022

Kernel aware debugging of software timers is not possible when using the following versions:

- Task Aware Debugger (TAD) for GDB version 11.4.0
- FreeRTOS version 10.4.4



Note

The TAD log shows (see the folder FreeRTOS_TAD_logs in the workspace):

16:31:14.485 ERROR: [VariableReader] Could not read variable expression: "**(((struct tmrTimerControl *) 0x1ffff650)->uxAutoReload)**".

16:31:14.485 EXCEPTION: [TimerFactory] ReadingException in class "com.nxp.mcuxpresso.tad.freertos.controller.TimerFactory" at 137. line when calling method "readIntAtAddress(0x1ffff650, tmrTimerControl, uxAutoReload)"

com.nxp.mcuxpresso.tad.common.controller.TadFactory.readIntAtAddress(TadFactory.java:247)

com.nxp.mcuxpresso.tad.freertos.controller.TimerFactory.readTimersFromList(TimerFactory.java:137)

com.nxp.mcuxpresso.tad.freertos.controller.TimerFactory.getData(TimerFactory.java:70)

com.nxp.mcuxpresso.tad.common.model.TadModel$2.execute(TadModel.java:185)

com.nxp.mcuxpresso.tad.common.model.TadModel$3.run(TadModel.java:308)

This message implicates that the TAD version expects an uxAutoReload field in the tmrTimerControl struct, but this is not the case in FreeRTOS version V10.4.4 (see file timers.c).

Note

06/03/2022

Refer to the following website for a solution to this problem ☺

Notice that by implementing this solution, the threads are also shown in the debugger.

https://mcuoneclipse.com/2017/07/27/troubleshooting-tips-for-freertos-thread-aware-debugging-in-eclipse/