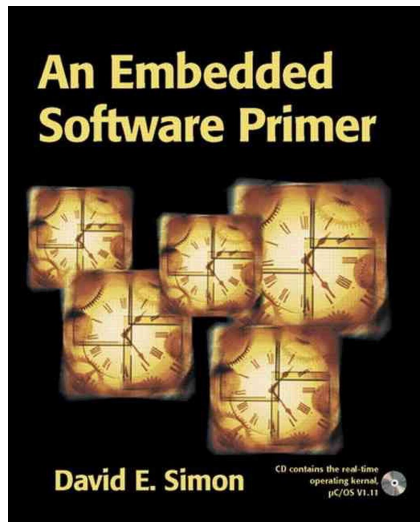


RTOS services —Part II

Kizito NKURIKIYEYU, Ph.D.

Readings

- Read Chap 7 of Simon, D. E. (1999). An Embedded Software Primer
- Read Chap 5 and 6 of Richard B. (2019). Mastering the FreeRTOS Real Time Kernel
- Topics:
 - inter-task communication
 - timer services
 - Queue, mailbox, pipes
 - memory management
 - interaction between ISR and RTOS



Message queues, Mailboxes and Pipes

- **Inter-task communication** is necessary to coordinate their activities or share data. It can be done via a global variable but this is error-prone and difficult
- **Synchronization and messaging** provides the necessary communication between tasks in one system to tasks in another system.
- Besides shared variables and semaphores, tasks can communicate with each other using queues, mailboxes and pipes. The RTOS **guarantees** that the functions provided for using these mechanisms are **reentrant**
 - **Mailbox**—data buffer that can store a fixed number of messages of a fixed size

Message queues, Mailboxes and Pipes

- **Inter-task communication** is necessary to coordinate their activities or share data. It can be done via a global variable but this is error-prone and difficult
- **Synchronization and messaging** provides the necessary communication between tasks in one system to tasks in another system.
- Besides shared variables and semaphores, tasks can communicate with each other using queues, mailboxes and pipes. The RTOS **guarantees** that the functions provided for using these mechanisms are **reentrant**
 - **Mailbox**—data buffer that can store a fixed number of messages of a fixed size
 - **Queues** —allow passing information between tasks without incurring overwrites from other tasks or entering into a race condition

Message queues, Mailboxes and Pipes

- **Inter-task communication** is necessary to coordinate their activities or share data. It can be done via a global variable but this is error-prone and difficult
- **Synchronization and messaging** provides the necessary communication between tasks in one system to tasks in another system.
- Besides shared variables and semaphores, tasks can communicate with each other using queues, mailboxes and pipes. The RTOS **guarantees** that the functions provided for using these mechanisms are **reentrant**
 - **Mailbox**—data buffer that can store a fixed number of messages of a fixed size
 - **Queues** —allow passing information between tasks without incurring overwrites from other tasks or entering into a race condition
 - **Pipe**—essentially identical to a queue, but processes byte-oriented data.

Message queues, Mailboxes and Pipes

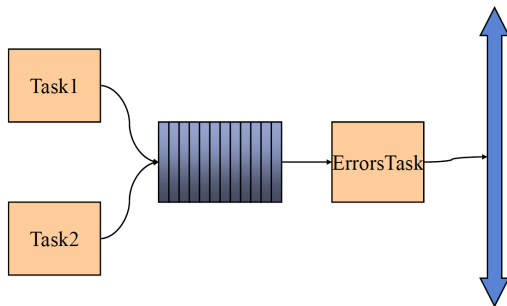
- **Inter-task communication** is necessary to coordinate their activities or share data. It can be done via a global variable but this is error-prone and difficult
- **Synchronization and messaging** provides the necessary communication between tasks in one system to tasks in another system.
- Besides shared variables and semaphores, tasks can communicate with each other using queues, mailboxes and pipes. The RTOS **guarantees** that the functions provided for using these mechanisms are **reentrant**
 - **Mailbox**—data buffer that can store a fixed number of messages of a fixed size
 - **Queues** —allow passing information between tasks without incurring overwrites from other tasks or entering into a race condition
 - **Pipe**—essentially identical to a queue, but processes byte-oriented data.
 - **Event flag**—used to synchronize internal activities while message queues and mailboxes are used to send text messages between systems

Message Queues

Message Queues

■ Simple Example

- Let's say there are two tasks, **Task1** and **Task2**, each with high priority things to do
- When an error occurs, the two tasks must report it
- However, error reporting is time consuming and might prevent these tasks to do their job properly.
- Thus, another task, **ErrorsTask**, handles error reporting
- Question: How to implement this in an RTOS?—Use an RTOS **queue**¹



¹ [https://en.wikipedia.org/wiki/Queue_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Queue_(abstract_data_type))


```

1 void Task1(void) {
2     while(true) {
3         if (system_error()) {
4             string error = get_error_message();
5             vLog_error_log_task(error);
6         }
7     }
8 }
9 void Task2(void) {
10     while(true) {
11         if (system_error()) {
12             string error = get_error_message();
13             vLog_error_log_task(error);
14         }
15     }
16 }

```

LISTING 1: Task1 and Task2 implementation snippets

```

1 void vLog_error_log_task(string error) {
2     queue_add(error);
3 }
4 void vLog_error_processing_task(string error_message) {
5     while(true) {
6         string error = queue_read_error();
7         if (error != NULL) {
8             // save error to an sd
9             // print error message
10        }
11    }
12 }

```

LISTING 2: Error logging tasks snippet

Note:

- The `queue_add()` add an error to the RTOS queue
- The `queue_read_error()` read an error from the head of the queue
- Both functions must be reentrant! —Why?

Queue in FreeRTOS

- A queue can hold a finite number of fixed size data items.
- Queues are normally used as FIFO buffers, where data is written to the end (tail) of the queue and removed from the front (head) of the queue.
- It is also possible to write to the front of a queue, and to overwrite data that is already at the front of a queue

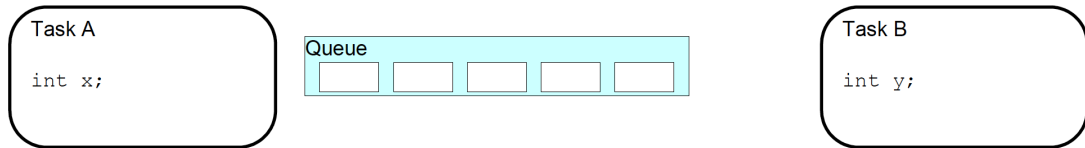


FIG 1. A queue is created to allow **Task A** and **Task B** to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.

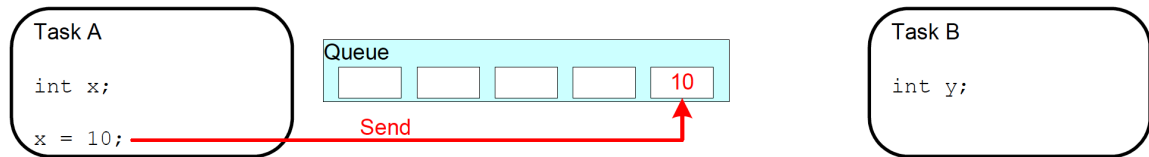


FIG 2. Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

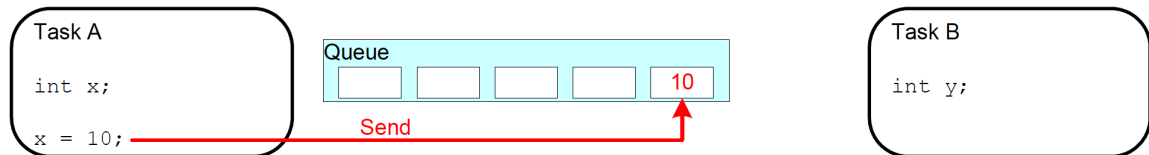


FIG 2. Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

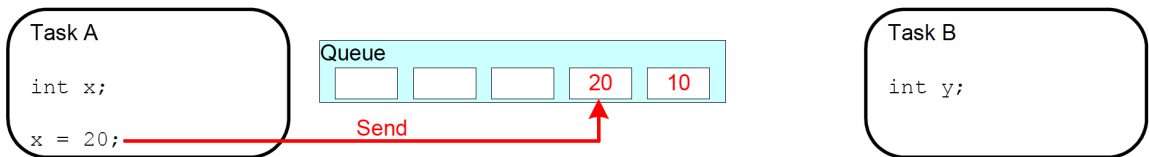


FIG 3. Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. Three empty spaces are remaining.

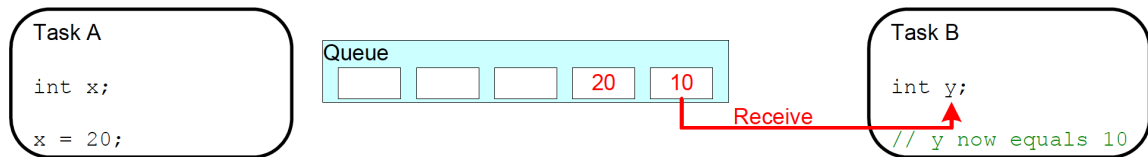


FIG 4. Task B reads from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (i.e., 10 here)

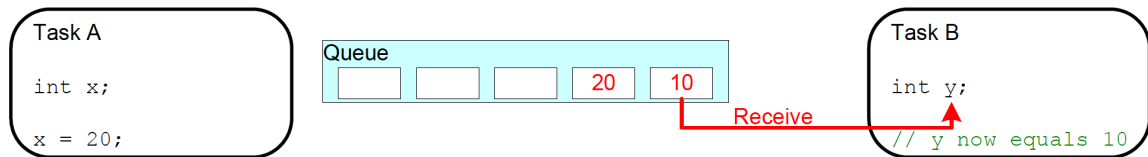


FIG 4. Task B reads from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (i.e., 10 here)

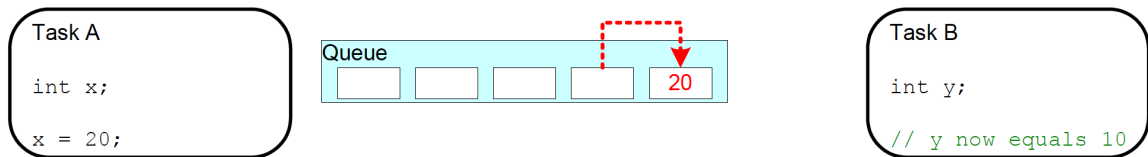


FIG 5. Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Queue in FreeRTOS

All xQUEUE have the following fields²:

- **uxLength** and **uxItemSize** indicate what is the maximum number of messages that it can hold, and the size of each message in bytes, respectively.
- **pcHead** and **pcTail** delimit the message storage zone associated with the queue. In particular, pcHead points to the base, that is, the lowest address of the memory area, and pcTail points to one byte more than the highest address of the area.
- **pcReadFrom** and **pcWriteTo** delineate the full portion of the message storage zone, and separate it from the free message storage space.
- **uxMessagesWaiting** counts how many messages are currently in the queue.
- The **xTasksWaitingToSend** field is an xList that links together all the tasks waiting to send a message into the queue when that operation cannot be performed immediately because the queue is completely full
- The **xTasksWaitingToReceive** has the same purpose as xTasksWaitingToSend but for tasks waiting to receive a message from an empty queue

²<https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/main/queue.c>

TAB 1. Contents of a FreeRTOS message queue data structure

Field	Purpose
uxLength	Maximum queue capacity (# of messages)
uxItemSize	Message size in bytes
pcHead	Lowest address of message storage zone
pcTail	Highest address of message storage zone +1
pcReadFrom	Address of oldest full element -uxItemSize
pcWriteTo	Address of next free element
uxMessagesWaiting	# of messages currently in the queue
xTasksWaitingToSend	List of tasks waiting to send a message
xTasksWaitingToReceive	List of tasks waiting to receive a message
xRxLock	Send queue lock flag and message counter
xTxLock	Receive queue lock flag and message counter

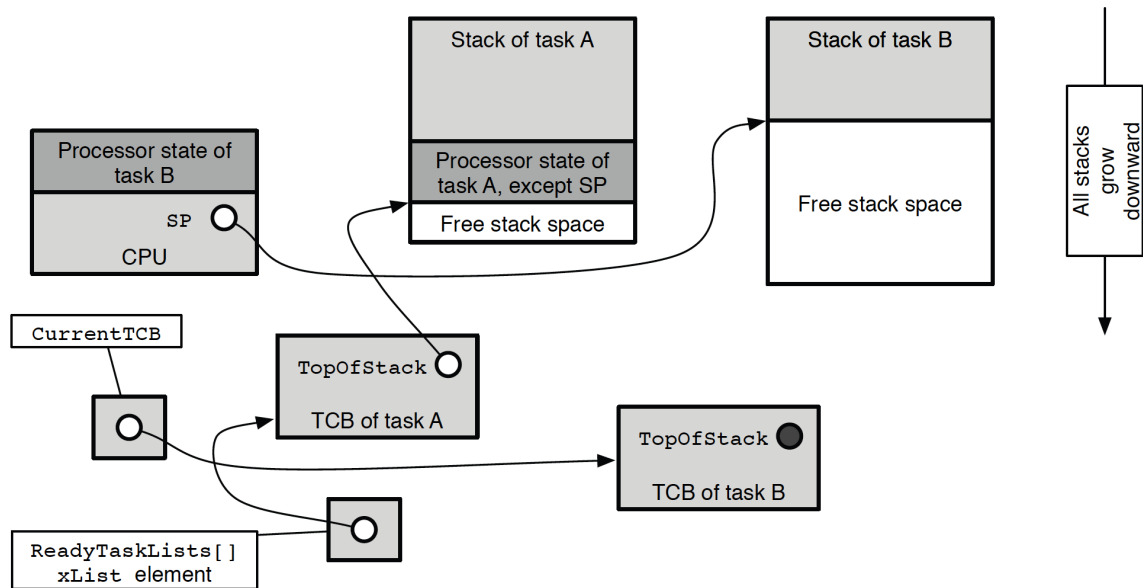


FIG 6. State of the main FreeRTOS data structures involved in a context switch after the context of task B has been restored

Using Queue in FreeRTOS

- The `xQueueCreate()` function³ creates a queue and returns a `QueueHandle_t` that references the queue it just created (Table 2).

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

TAB 2. `xQueueCreate()` parameters and return value

Parameter	Parameter description and usage note
<code>uxQueueLength</code>	The maximum number of items that the queue being created can hold at any one time.
<code>uxItemSize</code>	The size in bytes of each data item that can be stored in the queue.
Return Value	If NULL is returned, then the queue cannot be created because there is insufficient heap memory available FreeRTOS to allocate the queue data structures and storage area. A non-NULL value being returned indicates that the queue has been created successfully. The returned value should be stored as the handle to the created queue.

- After a queue has been created the `xQueueReset()`⁴ API function can be used to return the queue to its original empty state.

³<https://www.freertos.org/a00116.html>

⁴<https://www.freertos.org/a00018.html#xQueueReset>

Using Queue in FreeRTOS

- `xQueueSendToBack()` is used to send data to the back (tail) of a queue.

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

⁵These methods should never be called from an interrupt service routine. Instead, use the interrupt-safe versions `xQueueSendToFrontFromISR()` and `xQueueSendToBackFromISR()`

Using Queue in FreeRTOS

- **xQueueSendToBack()** is used to send data to the back (tail) of a queue.

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

- **xQueueSend()** is equivalent to, and exactly the same as, **xQueueSendToBack()**⁵

⁵These methods should never be called from an interrupt service routine. Instead, use the interrupt-safe versions **xQueueSendToFrontFromISR()** and **xQueueSendToBackFromISR()**

Using Queue in FreeRTOS

- **xQueueSendToBack()** is used to send data to the back (tail) of a queue.

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

- **xQueueSend()** is equivalent to, and exactly the same as, **xQueueSendToBack()**⁵
- **xQueueSendToFront()** is used to send data to the front (head) of a queue.

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

⁵These methods should never be called from an interrupt service routine. Instead, use the interrupt-safe versions **xQueueSendToFrontFromISR()** and **xQueueSendToBackFromISR()**

Using Queue in FreeRTOS

- **xQueueSendToBack()** is used to send data to the back (tail) of a queue.

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

- **xQueueSend()** is equivalent to, and exactly the same as, **xQueueSendToBack()**⁵

- **xQueueSendToFront()** is used to send data to the front (head) of a queue.

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

- **xQueueReceive()** is used to receive (read) an item from a queue. The item that is received is removed from the queue.

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

⁵These methods should never be called from an interrupt service routine. Instead, use the interrupt-safe versions **xQueueSendToFrontFromISR()** and **xQueueSendToBackFromISR()**

Using Queue in FreeRTOS

- `uxQueueMessagesWaiting()` is used to query the number of items that are currently in a queue.

```
1 BaseType_t uxQueueMessagesWaiting( QueueHandle_t xQueue );
```

- `vQueueDelete()` delete a queue when its message queue is no longer needed in order to reclaim its memory for future use

TAB 3. Summary of the main message-queue related primitives of FreeRTOS

Function	Purpose	Optional
xQueueCreate	Create a message queue	-
vQueueDelete	Delete a message queue	-
xQueueSendToBack	Send a message	-
xQueueSendToFront	Send a high-priority message	-
xQueueSendToBackFromISR	...from an interrupt handler	-
xQueueSendToFrontFromISR	...from an interrupt handler	-
xQueueReceive	Receive a message	-
xQueueReceiveFromISR	...from an interrupt handler	-
xQueuePeek	Nondestructive receive	-
uxQueueMessagesWaiting	Query current queue length	-
uxQueueMessagesWaitingFromISR	...from an interrupt handler	-
xQueueIsQueueEmptyFromISR	Check if a queue is empty	-
xQueueIsQueueFullFromISR	Check if a queue is full	-

Example

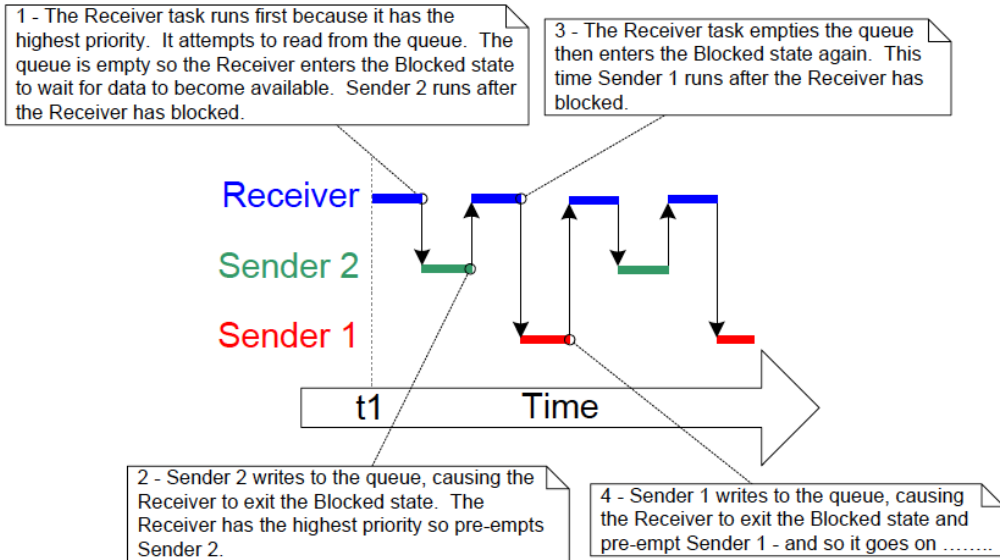


FIG 7. Expected sequence of execution

Example

```
1 static void vSenderTask( void *pvParameters ){
2     int32_t lValueToSend;
3     BaseType_t xStatus;
4     lValueToSend = ( int32_t ) pvParameters;
5     while( true ) { /*
6         -The first parameter is the queue to which data is being sent
7         -The second parameter is the address of the data to be sent
8         -The third parameter is the block time */
9         xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
10        if( xStatus != pdPASS ) {
11            /* Error because the queue was full */
12        }
13    }
14 }
```

LISTING 3: Example: Implementation of the sending task used

```

1 static void vReceiverTask( void *pvParameters ){
2     int32_t lReceivedValue;
3     BaseType_t xStatus;
4     const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );
5     while(true) { /*
6         -The first parameter is the queue the data is to be received.
7         -The second is the buffer that will receive the data into.
8         -The third parameter is the block time */
9     xStatus=xQueueReceive( xQueue,&lReceivedValue, xTicksToWait);
10        if( xStatus == pdPASS ){
11            /* Data was successfully received from the queue*/
12        }
13        else{
14            //Error: data was not received after waiting for 100ms
15        }
16    }
17 }

```

LISTING 4: Example: Implementation of the receiver task

```

1 QueueHandle_t xQueue;
2 int main( void ){
3     /* The queue is created to hold a maximum of 5 values*/
4     xQueue = xQueueCreate(5, sizeof( int32_t ) );
5     if(xQueue!=NULL )
6     {
7         xTaskCreate(vSenderTask, "Sender1", 1000, (void*)100, 1, NULL);
8         xTaskCreate(vSenderTask, "Sender2", 1000, (void*)200, 1, NULL);
9         xTaskCreate(vReceiverTask, "Receiver", 1000, NULL, 2, NULL);
10        /*Start the scheduler and let the RTOS take over*/
11        vTaskStartScheduler();
12    }
13    else
14    {
15        /* The queue could not be created. */
16    }
17 }

```

LISTING 5: Example: the implementation of main() function

Mailboxes

Mailboxes

- In general, mailbox are similar to queues⁶
- Mailbox functions:
 - Create a mailbox
 - Write to a mailbox
 - Read from a mailbox
 - Check if a mailbox has any message
 - Destroy an unused mailbox
- They exists several variations in different RTOSs⁷
- Typical use of a mailbox
 - A mailbox is used to hold data that can be read by any task
 - The data does not pass through the mailbox, but instead remains in the mailbox until it is overwritten. The sender overwrites the value in the mailbox.
 - The receiver reads the value from the mailbox, but does not remove the value from the mailbox.

⁶In FreeRTOS, a mailbox is a queue that has a length of one

⁷There is no consensus on terminology within the embedded community, and “mailbox” will mean different things in different RTOSes

Mailboxes in FreeRTOS

In FreeRTOS, a mailbox is a queue that has a length of one

■ The `xQueueOverwrite()` API⁸

- `xQueueOverwrite()` should only be used with queues that have a length of one.
- Like the `xQueueSendToBack()` API function, the `xQueueOverwrite()` API function sends data to a queue.
- Unlike `xQueueSendToBack()`, if the queue is already full, then `xQueueOverwrite()` will overwrite data that is already in the queue.

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

■ The `xQueuePeek()` API Function⁹

- Used to receive an item from a queue without removing it from the queue.
- Receives data from the head of the queue, without modifying the data stored in the queue, or the order in which data is stored in the queue.

⁸<https://www.freertos.org/xQueueOverwrite.html>

⁹<https://www.freertos.org/xQueuePeek.html>

Example

- **vUpdateMailbox()** writes a random integer value to the mailbox every 500ms
- **vReadMailbox()** reads that integer value from the mailbox after every 100ms

```
1 #include <FreeRTOS.h>
2 #include <queue.h>
3 #include <time.h>
4 #include <stdlib.h>
5 QueueHandle_t xMailbox;
6 TaskHandle_t updateTaskHandle;
7 TaskHandle_t readTaskHandle;
8 void main(void) {
9     xMailbox = xQueueCreate(1, sizeof(int32_t));
10    xTaskCreate(vUpdateMailbox, "S", 100, NULL, 1, &updateTaskHandle);
11    xTaskCreate(vReadMailbox, "R", 100, NULL, 1, &readTaskHandle);
12 }
```

LISTING 6: Implementation of main the mailbox tasks

```

1 void vUpdateMailbox(void *pvParameters ){
2     srand(time(NULL));
3     while(true) {
4         int new_mail_box_value = rand();
5         xQueueOverwrite( xMailbox, &new_mail_box_value);
6         vTaskDelay(500/portTICK_RATE_MS);
7     }
8 }
9 BaseType_t vReadMailbox(void *pvParameters ){
10     int received_value =0;
11     while(true) {
12         xQueuePeek(xMailbox, &received_value, portMAX_DELAY);
13         //The received value stored in the value_received
14         fprintf( "The received value is %d\n", received_value)
15         vTaskDelay(100/ portTICK_RATE_MS);
16     }
17 }

```

LISTING 7: Implementation of the mailbox tasks

Timer Functions

Timer Functions

- Embedded systems generally require to track time.
- A cell phone preserves battery by turning its display off after a few seconds. Network connections re-transmit data if an acknowledgement is not received within a certain period.
- Most RTOSs have a delay function that delays for a certain time period.
- Each of the tones representing a digit in a phone call must sound for 1/10th of a second followed by the same period of silence between tones.
- For example, use the function `vTaskDelay(100 / portTICK_RATE_MS)`

Questions

- How do I know that `vTaskDelay ()` works as intended? —delays based on system ticks as its parameter
- How accurate is `vTaskDelay ()`?—It is accurate to the nearest tick
- How does the RTOS know how to setup the timer hardware ?—RTOSs are microprocessor-dependent and hence the engineers that wrote the RTOS know which processor it will run on and hence can program the corresponding timer. If the timer hardware is non-standard, the user is required to write his own timer setup and interrupt routines that will be called by the RTOS.
- What is the “normal length” for a system tick?
 - There isn't one.
 - Short system times provide accurate timings with the added disadvantage of occupying the processor more and reducing throughput.
 - The designer must make a trade-off between the two.
 - In FreeRTOS, the tick is 1 ms by default¹⁰

¹⁰see mode details at <http://www.learnitmakeit.com/freertos-tick/>

Questions

- What if the system requires extremely accurate timing?¹¹
 - Use short system ticks
 - For an extremely accurate timing, one must use dedicated hardware timer for functions requiring accurate times and the RTOS for all other timings.
 - The advantage of using the OS is that one timer handles many operations simultaneously.
 - You should not create a timer that will be way too fast for the system to process.
 - In short, the faster the tick the more interrupt and the more scheduler overhead
 - FreeRTOS uses the microcontroller's TCB0 timer to generate its own tick interrupt. The FreeRTOS kernel measures the time using the tick, and every time a tick occurs, the scheduler checks if a task should be woken up or unblocked.
 - The `configCPU_CLOCK_HZ` define must be configured for the FreeRTOS timings to be correct.
 - If more than one task has the same priority, the FreeRTOS scheduler will switch between these tasks at every tick if `configUSE_TIME_SLICING` is set to 1.
 - Thus, using a higher tick rate causes the CPU time for each task to be smaller.

¹¹ You can find more FreeRTOS utilities at <https://www.freertos.org/a00021.html>

FreeRTOS software timer

What and why use software timer¹⁵

- We saw that a task can create a non-blocking timer with:
 - **vTaskDelay**—block the currently running task for a given time
 - **xTaskGetTickCount()**—non-blocking delay based on a known timestamp
 - **hardware timer**—but this is tedious and not portable
- **Software timers**—like tasks—allow to trigger actions at a given frequency
- Unlike tasks, software timers require little overhead¹²
- Software timers do not rely on the underlying hardware timers of the microcontroller, instead, they use the **FreeRTOS tick counter**.
- **Timer Accuracy**—affected by the FreeRTOS's scheduling algorithm
- **Timer Resolution**¹³—low and depends on FreeRTOS's tick frequency¹⁴

¹² Miranda, B. D., de Oliveira, R. S., & Carminati, A. (2021, July). Analysis of FreeRTOS Overheads on Periodic Tasks. <https://doi.org/10.5753/wperformance.2021.15728>

¹³ The timer resolution is the smallest unit of time that the timer can measure

¹⁴ This makes the software timers, not a good fit for scenarios requiring time resolution higher than 1 ms.

¹⁵ For more details read chap 5 of Richard Berry's "Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide", which focuses on software timer management in FreeRTOS

How to use software timer?¹⁷

- Turn them on with the following entry in `FreeRTOSConfig.h`

```
1 #define configUSE_TIMERS 1
```

- Similarly, you can configure the timer task name, priority and stack

```
1 #define configTIMER_SERVICE_TASK_NAME "Tmr Svc"  
2 #define configTIMER_TASK_PRIORITY (configMAX_PRIORITIES - 1)  
3 #define configTIMER_TASK_STACK_DEPTH (configMINIMAL_STACK_SIZE)
```

Note:

- It is a good idea to give the timer highest task priority in the system, otherwise, there will be some latency in the timer hook execution.
- The timer stack size really depends on what you are doing in the timer hooks called from the timer task¹⁶

¹⁶ To find out what your tasks are using on the stack, see [Understanding FreeRTOS Task Stack Usage](#)

¹⁷ API details at <https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html>

FreeRTOS time working principles

- They do not use the CPU unless their callback function is executing
- When a timer is created, it is assigned a **callback function** that is called whenever the timer expires
- The **timer service** or Daemon keeps an ordered list of software timers—with the timer to expire next in front of the list.
- The Timer Service task is not continuously running
 - Each timer's waking time is known
 - When a timer has expired, the Timer Service task calls its callback

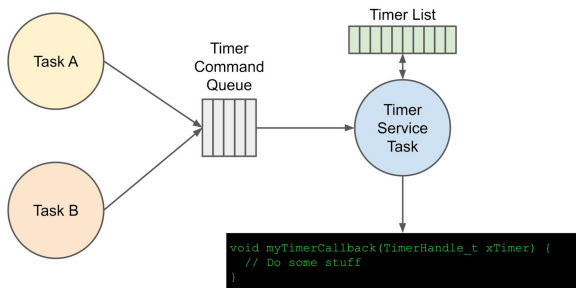


FIG 8. FreeRTOS software timer

Software Timer Callback Functions

- Regular C function
- They must have the following function prototype

```
1 void ATimerFunctionCallback ( TimerHandle_t xTimer );
```

- The callback functions execute from start to finish, and exit in the normal way.
- The callback functions should be kept short
- The callback functions **must not enter the blocked state**

Types software timers

- **One-shot timers**—Once started, it will execute its callback function once only. A one-shot timer can be restarted manually, but will not restart itself.
- **Auto-reload timers**—Once started, it will re-start itself each time it expires, resulting in periodic execution of its callback function.

Types software timers

- **One-shot timers**—Once started, it will execute its callback function once only. A one-shot timer can be restarted manually, but will not restart itself.
- **Auto-reload timers**—Once started, it will re-start itself each time it expires, resulting in periodic execution of its callback function.

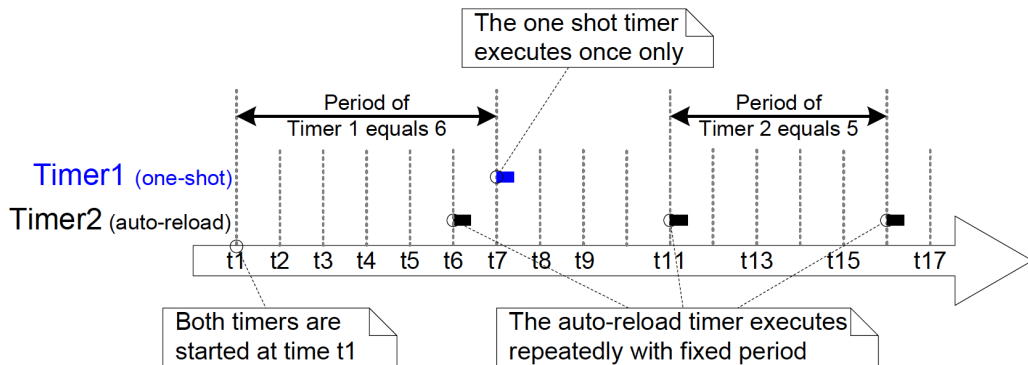


FIG 9. The difference in behavior between one-shot and auto-reload software timers

Software timer states

- **Dormant**—exists, and can be referenced by its handle, but is not running, so its callback functions will not execute.
- **Running** —execute its callback function after a time equal to its period has elapsed since the software timer entered the Running state, or since the software timer was last reset.

Software timer states

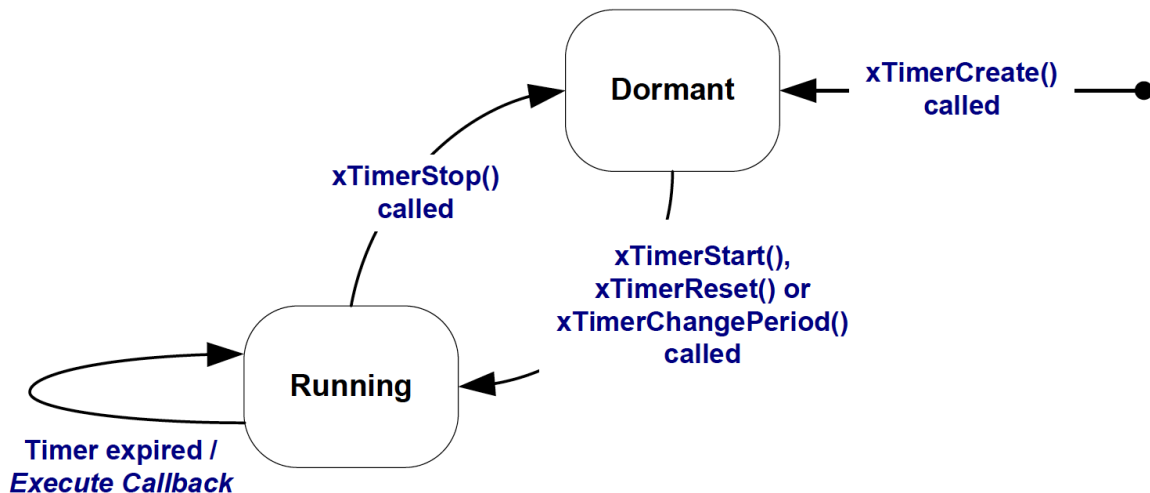


FIG 10. Auto-reload software timer states and transitions

Software timer states

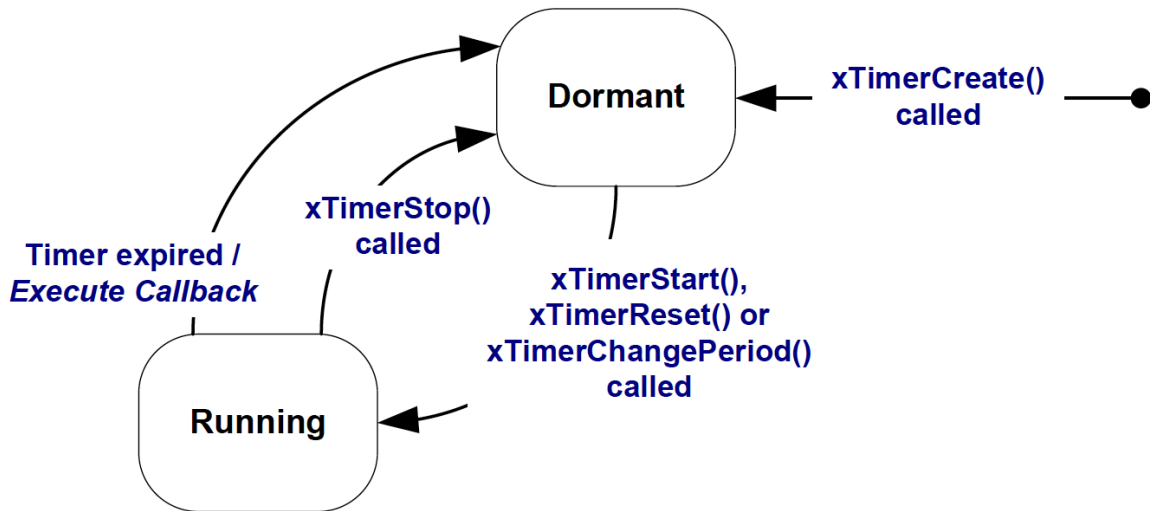


FIG 11. One-shot software timer states and transitions

The Context of a Software Timer

- The FreeRTOS daemon task is standard FreeRTOS task that is created automatically when the scheduler is started.
- Its priority and stack size should be set in the `FreeRTOSConfig.h` file
- The FreeRTOS daemon should not enter the blocked state—thus, the software timer callback functions must not call FreeRTOS API functions that will result in the calling task entering the blocked state
- The daemon task is scheduled like any other FreeRTOS task depending on its priority

Creating a software timer

Software timers can be created before the scheduler is running, or from a task after the scheduler has been started¹⁸

```
1 TimerHandle_t xTimerCreate( const char * const pcTimerName,  
2   TickType_t xTimerPeriodInTicks, UBaseType_t uxAutoReload,  
3   void * pvTimerID, TimerCallbackFunction_t pxCallbackFunction);
```

¹⁸<https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html>

Creating a software timer

Software timers can be created before the scheduler is running, or from a task after the scheduler has been started¹⁸

```
1 TimerHandle_t xTimerCreate( const char * const pcTimerName,  
2   TickType_t xTimerPeriodInTicks, UBaseType_t uxAutoReload,  
3   void * pvTimerID, TimerCallbackFunction_t pxCallbackFunction);
```

TAB 4. xTimerCreate() parameters and return value

Parameter	Explanation and significance
pcTimerName	Name assigned for purely debugging purposes
xTimerPeriod	The period of the timer. The period is specified in ticks thus the macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds to a time specified in ticks.
uxAutoReload	If uxAutoReload is set to pdTRUE, then the timer will expire repeatedly with a frequency set by the xTimerPeriod parameter. If it is set to pdFALSE, then the timer will be a one-shot and enter the dormant state after it expires.
pvTimerID	An identifier that is assigned to the timer being created and 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:

¹⁸<https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html>

Starting a software timer

A timer previously created with `xTimerCreate()` gets started with¹⁹

```
1 BaseType_t xTimerStart(TimerHandle_t xTimer,  
2                        TickType_t xTicksToWait);
```

TAB 5. `xTimerStart` parameters and return values

Parameter	Explanation and significance
<code>xTimer</code>	The handle of the timer being started/restarted.
<code>xBlockTime</code>	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
Returns	<ul style="list-style-type: none">■ The handle of the software timer being started or reset.■ <code>pdFAIL</code> will be returned if the start command could not be sent to the timer command queue even after <code>xBlockTime</code> ticks had passed.■ <code>pdPASS</code> will be returned if the command was successfully sent to the timer command queue.

¹⁹<https://www.freertos.org/FreeRTOS-timers-xTimerStart.html>

Interrupt Routines in an RTOS

Interrupts and Tasks

- Similarities between tasks and ISRs
 - Both provide a way of achieving parallel code execution.
 - Both only run when required.
 - Both can be written with C/C++ (ISRs generally no longer need to be written in assembly code).
- Differences between tasks and ISRs:
 - ISRs are brought into context by hardware while tasks gain context by the RTOS kernel
 - ISRs must exit as quickly as possible while tasks are more forgiving. For example, FreeRTOS tasks are often set up to run in an infinite while loop
 - ISR functions do not take input parameters while tasks can
 - ISRs may only access a limited ISR-specific subset of the FreeRTOS API
 - ISRs may operate completely independently of all RTOS code
 - All ISRs share the same system stack while each task has a dedicated stack
- An interrupt service routine is a hardware feature. Tasks will only run when there are no ISRs running, **so the lowest priority interrupt will interrupt the highest priority task,**

Interrupt Routines in an RTOS

In an RTOS, interrupts follows two rules that do not apply to task code

- **Rule #1**—ISR must not call any RTOS function that might block the caller²⁰
 - An RTOS interrupt must not get a semaphore
 - An RTOS interrupt must not read from an empty queue or mailbox
 - An RTOS interrupt must not wait for an event
 - An RTOS interrupt must not wait for mutex else it has to wait for other critical section code to finish before the critical codes in the ISR can run.
 - It must run to completion to reset hardware to be ready for next interrupt

²⁰FreeRTOS interrupt provides an interrupt safe version. They always end with “FromISR” and are exempt from this rule. See for example [xQueueReceiveFromISR API](#)

Interrupt Routines in an RTOS

In an RTOS, interrupts follow two rules that do not apply to task code

- **Rule #1**—ISR must not call any RTOS function that might block the caller²⁰
 - An RTOS interrupt must not get a semaphore
 - An RTOS interrupt must not read from an empty queue or mailbox
 - An RTOS interrupt must not wait for an event
 - An RTOS interrupt must not wait for mutex else it has to wait for other critical section code to finish before the critical codes in the ISR can run.
 - It must run to completion to reset hardware to be ready for next interrupt
- **Rule #2**—ISR may not call any RTOS function that cause task switching, unless RTOS knows that it is an ISR —thus will not switch task
 - An RTOS interrupt must not release semaphores
 - It must not write to mailboxes, queues on which tasks may be waiting

If an interrupt breaks these rules, the RTOS might switch control away from the interrupt routine to run another task, and the interrupt may not complete for a long time and would block all lower-priority interrupts —or even all interrupts

²⁰FreeRTOS interrupt provides an interrupt safe version. They always end with “FromISR” and are exempt from this rule. See for example [xQueueReceiveFromISR API](#)

Using the FreeRTOS API from interrupts

- Most of the FreeRTOS primitives have ISR-safe versions of their APIs.
- For example, `xQueueSend()` has an equivalent ISR-safe version, `xQueueSendFromISR()`.
- One should never call a FreeRTOS non ISR-safe function from an ISR.
- Notable peculiarities of the the ISR-safe version:
 - The FromISR variants won't block—For example, if `xQueueSendFromISR` encounters a full queue, it will immediately return.

Using the FreeRTOS API from interrupts

- Most of the FreeRTOS primitives have ISR-safe versions of their APIs.
- For example, `xQueueSend()` has an equivalent ISR-safe version, `xQueueSendFromISR()`.
- One should never call a FreeRTOS non ISR-safe function from an ISR.
- Notable peculiarities of the the ISR-safe version:
 - The `FromISR` variants won't block—For example, if `xQueueSendFromISR` encounters a full queue, it will immediately return.
 - The `FromISR` variants require an extra parameter, `BaseType_t *pxHigherPriorityTaskWoken`, which will indicate whether or not a higher priority task needs to be switched into context immediately following the interrupt.

Using the FreeRTOS API from interrupts

- Most of the FreeRTOS primitives have ISR-safe versions of their APIs.
- For example, `xQueueSend()` has an equivalent ISR-safe version, `xQueueSendFromISR()`.
- One should never call a FreeRTOS non ISR-safe function from an ISR.
- Notable peculiarities of the the ISR-safe version:
 - The `FromISR` variants won't block—For example, if `xQueueSendFromISR` encounters a full queue, it will immediately return.
 - The `FromISR` variants require an extra parameter, `BaseType_t *pxHigherPriorityTaskWoken`, which will indicate whether or not a higher priority task needs to be switched into context immediately following the interrupt.
 - Only interrupts that have a logically lower priority than what is defined by `configMAX_API_CALL_INTERRUPT_PRIORITY` in `FreeRTOSConfig.h` are permitted to call FreeRTOS API functions

The End