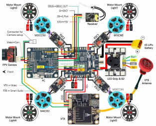
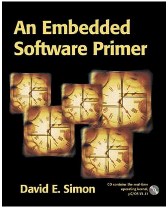


RTOS services —Part II

Kizito NKURIKIYEYEZU, 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,



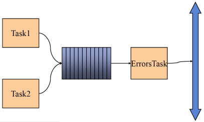
Message queues, Mailboxes and Pipes

- 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

Message Queues

- Simple Example
  - Let's say there are two tasks, Task1 and Task2, each with high priority things to do
  - When an error accurs, 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)


```
void Task1(void){
  while(true){
    if (system_error()){
      string error = get_error_message();
      vLog_error_log_task(error);
    }
  }
}
void Task2(void){
  while(true){
    if (system_error()){
      string error = get_error_message();
      vLog_error_log_task(error);
    }
  }
}
```

LISTING 1: Task1 and Task2 implementation snippets


```
void vLog_error_log_task(string error){
  queue_add(error);
}
void vLog_error_processing_task(string
  error_message){
  while(true){
    string error = queue_read_error();
    if (error != NULL){
      // save error to an sd
      // print error message
    }
  }
}
```

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 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

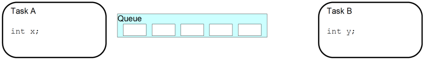


FIG 1. A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.

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.

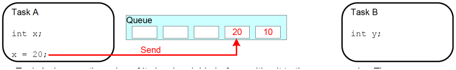


FIG 3. Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. Three empty spaces are remaining.

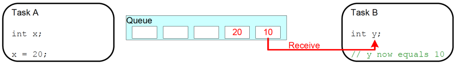


FIG 4. Task B reads from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (i.e., 10 here)

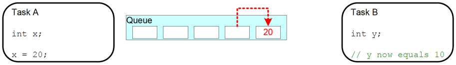


FIG 5. Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Queue in FreeRTOS

All xQUEUE have the following fields[2]:

- 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

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

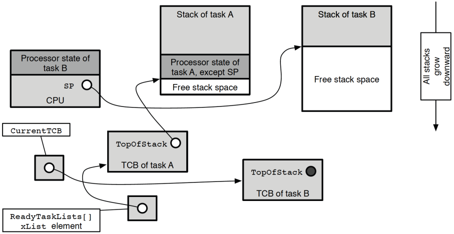


FIG 6. State of the main FreeRTOS data structures involved in a context switch after the context of task B has been restored

Using Queue in FreeRTOS

- The xQueueCreate() function[3] creates a queue and returns a QueueHandle_t that references the queue it just created (Table 2).

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

TAB 2. xQueueCreate() parameters and return value

Parameter	Parameter description and usage note
uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	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 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

Using Queue in FreeRTOS

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

```
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.

```
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.

Using Queue in FreeRTOS

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

```
UBaseType_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

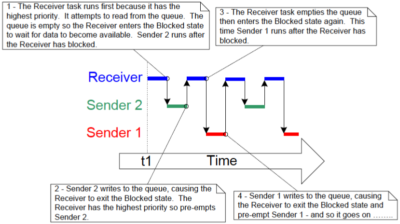


FIG 7. Expected sequence of execution

Example

```
static void vSenderTask( void *pvParameters ){
  int32_t lValueToSend;
  BaseType_t xStatus;
  lValueToSend = ( int32_t ) pvParameters;
  while(true){/*
    -The first parameter is the queue to which data
       is being sent
    -The second parameter is the address of the data
        to be sent
    -The third parameter is the block time */
    xStatus = xQueueSendToBack( xQueue, &
       lValueToSend, 0 );
    if( xStatus != pdPASS ){
      /* Error because the queue was full */
    }
  }
}
```



```
static void vReceiverTask( void *pvParameters ){
  int32_t lReceivedValue;
  BaseType_t xStatus;
  const TickType_t xTicksToWait = pdMS_TO_TICKS( 100
     );
  while(true){/*
  -The first parameter is the queue the data is to
     be received.
  -The second is the buffer that will receive the
     data into.
  -The third parameter is the block time */
  xStatus=xQueueReceive( xQueue,&lReceivedValue,
     xTicksToWait);
    if( xStatus == pdPASS ){
      /* Data was successfully received from the
         queue*/
    }
    else{
```

```c
QueueHandle_t xQueue;
int main( void ){
  /* The queue is created to hold a maximum of 5 values*/
  xQueue = xQueueCreate(5,sizeof( int32_t ) );
  if(xQueue!=NULL )
  {
    xTaskCreate(vSenderTask, "Sender1", 1000,(void*)100,1,NULL);
    xTaskCreate(vSenderTask,"Sender2",1000,(void*)200,1,NULL);
    xTaskCreate(vReceiverTask,"Receiver",1000,NULL,2,NULL);
    /*Start the scheduler and let the RTOS take over */
    vTaskStartScheduler();
  }
  else
```


Mailboxes

Mailboxes

- In general,mailbox are similar to queues[6]
- 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 exits several variations in different RTOSs[7]
- 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.

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

[7] There is no consensus on terminology within the embedded community, and

Mailboxes in FreeRTOS

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

- The xQueueOverwrite() API[8]
  - 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.

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

- The xQueuePeek() API Function[9]
  - Used to receive an item from a queue without removing it from

Example

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

```c
#include <FreeRTOS.h>
#include <queue.h>
#include <time.h>
#include <stdlib.h>
QueueHandle_t xMailbox;
TaskHandle_t updateTaskHandle;
TaskHandle_t readTaskHandle;
void main(void) {
  xMailbox = xQueueCreate(1,sizeof(int32_t));
  xTaskCreate(vUpdateMailbox,"S", 100,NULL,1,&
     updateTaskHandle);
  xTaskCreate(vReadMailbox,"R", 100,NULL,1,&
```



```c
void vUpdateMailbox(void *pvParameters ){
  srand(time(NULL));
  while(true){
    int new_mail_box_value = rand();
    xQueueOverwrite( xMailbox, &new_mail_box_value);
    vTaskDelay(500/portTICK_RATE_MS);
  }
}
BaseType_t vReadMailbox(void *pvParameters ){
  int received_value =0;
  while(true){
    xQueuePeek(xMailbox, &received_value,
       portMAX_DELAY);
    //The received value stored in the
       value_received
    fprintf( "The received value is %d\n",
       received_value)
    vTaskDelay(100/ portTICK_RATE_MS);
```


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.

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.

FreeRTOS software timer

What and why use software timer[15]

- 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 overhead[12]
- 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 frequency[14]

[12] Miranda, B. D., de Oliveira, R. S., & Carminati, A. (2021, July). Analysis of

How to use software timer?[17]

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

```c
#define configUSE_TIMERS 1
```

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

```c
#define configTIMER_SERVICE_TASK_NAME "Tmr Svc"
#define configTIMER_TASK_PRIORITY (
    configMAX_PRIORITIES - 1)
#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 task[16]

[16] To find out what your tasks are using on the stack, see Understanding

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 Deamon 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

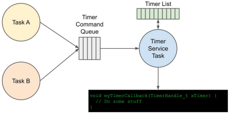


FIG 8. FreeRTOS software timer

Software Timer Callback Functions

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

```c
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.

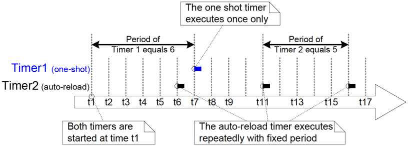


FIG 9. The difference in behavior between one-shot and auto-reload software timers

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

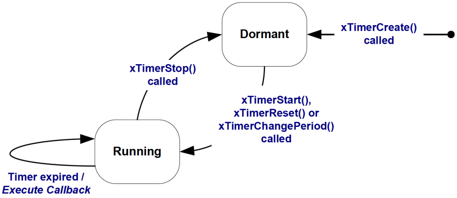


FIG 10. Auto-reload software timer states and transitions

Software timer states

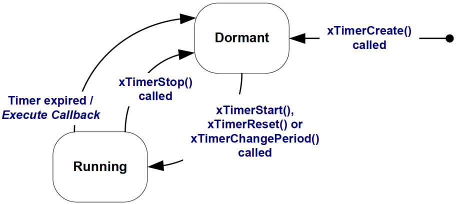


FIG 11. One-shot software timer states and transitions

The Context of a Software Timer

- The FreeRTOS daemon task is standard FreeRTOS task that is created automatically when the scheduler is started.
- Its 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 started[18]

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

TAB 4. xTimerCreate() parameters and return value

Parameter	Explaination 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 qnd 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 TimerCallback-Function_t, which is:

Starting a software timer

A timer previously created with xTimerCreate() gets started with[19]

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

TAB 5. xTimerStart parameters and return values

Parameter	Explaination and significance
xTimer	The handle of the timer being started/restarted.
xBlockTime	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	■ The handle of the software timer being started or reset. ■ pdFAIL will be returned if the start command could not be sent to the timer command queue even after xBlockTime ticks had passed. ■ pdPASS 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

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 caller[20]
  - 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

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