

Prof. L. Thiele

Embedded Systems - FS 19

Lab 3

Date : 27.3.2019

Tasks in a real-time operating system

Goals of this Lab

- Introduce a real-time operating system
- Learn what tasks are and how to create them
- A look at task states and priorities
- Intertask communication using queues
- Critical sections

1 Introduction

This lab introduces the concepts of tasks in a real-time operating system. For this purpose, FreeRTOS is used, which is designed to run applications with soft and hard real-time requirements on embedded platforms. It allows the application to be split into independent tasks, which is important when dealing with different real-time requirements. Conceptually, FreeRTOS is a real-time kernel taking care of task handling and allowing embedded applications to use multitasking. In contrast to multi-processor platforms, embedded platforms often contain only one processor, which can run only one single task at a time. In FreeRTOS this task is defined as being in the **Running** state, illustrated in Figure 1. All other states are in the **Not Running** state. The kernel decides when and which task needs to be moved into the **Running** state and consequently determines a schedule for the application.

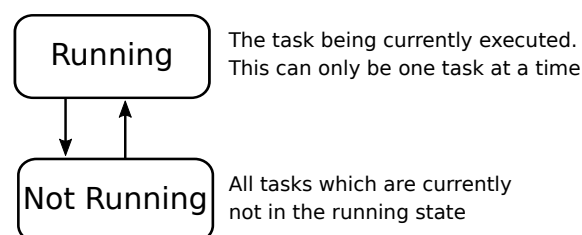


Figure 1: Simplified task states diagram in FreeRTOS

In the following, a basic application with multiple tasks will be created and the concepts of critical sections, priorities and queues will be introduced. We provide two FreeRTOS documents for further reading.¹ The book "Mastering the FreeRTOS Real Time Kernel" on which the lab is loosely based and the "FreeRTOS Reference Manual". The documents can also be downloaded for free from the FreeRTOS website (<https://www.freertos.org/>). The lab as such is self-contained, but we suggest that you have a look at the FreeRTOS Reference Manual to get a more in-depth understanding about the FreeRTOS functions. For each function example in this lab we provide a pointer to the chapter in the FreeRTOS

¹https://lectures.tik.ee.ethz.ch/es/labs/lab_documents.zip

Reference Manual v9.0.0. Additionally, the provided book provides a good start for further reading on FreeRTOS.

FreeRTOS task functions In FreeRTOS tasks can be created with the `xTaskCreate()` function. The declaration is shown in Snippet 1. The function takes multiple arguments which are explained throughout this lab.

```
0 BaseType_t xTaskCreate( TaskFunction_t pvTaskCode, // Pointer to the task function
1                         const char * const pcName, // Descriptive name of the task
2                         uint16_t usStackDepth,    // Stack size allocated for the task
3                         void *pvParameters,       // Task specific parameters
4                         UBaseType_t uxPriority,    // Priority of the task
5                         TaskHandle_t *pxCreatedTask // Handle to the task
6                         );
```

Snippet 1: Declaration of `xTaskCreate()`. (c.f. FreeRTOS Reference Chapter 2.6)

pvTaskCode: This argument points to a function which contains the logic of the task. This function must never exit. Thus, the task functionality should be encapsulated in an infinite loop. Snippet 2 shows an example prototype of a task function.

pcName: This argument should be chosen to be a descriptive name for the task. This name is not used by FreeRTOS but it facilitates the debugging process.

usStackDepth: For each task, a stack will be preallocated. The size of the stack must be specified by the developer depending on the task's memory requirements. In this lab, we set `usStackDepth` to a default value of 1000. In practice, the required stack size can be traced with FreeRTOS tools and can then be set to a reasonable small value.

pvParameters: This argument allows users to pass any information to the task function. For a generic use, this has been accomplished by a pointer. Thus, any arguments must be passed by using a pointer to a void object. A recap on how to do this can be found in Recap 1.

uxPriority: This argument can be used to assign a priority to the task. This will be explained in Task 2.

pxCreatedTask: A handle to the task. Some FreeRTOS functions require a handle to perform operations on a specific task during application runtime, for example to change the priority of a task or delete the task.

```
0 void vTaskFunction( void *pvParameters )
1 {
2     // Initialization can be done here...
3     while (1)
4     {
5         // Task functionality comes here ...
6     }
7 }
```

Snippet 2: A task function prototype

The scheduler can be started with the `vTaskStartScheduler()`, which is called from within `main()`. Usually the function will never return and the application will run forever. Consequently, all FreeRTOS configuration and task creation needs to be finished before calling `vTaskStartScheduler()`.

```
0 void vTaskStartScheduler( void )
```

Snippet 3: Declaration of `vTaskStartScheduler()`. (c.f. FreeRTOS Reference Chapter 2.31)

Recap 1: Type conversion and pointers

Sometimes, it is required to do a temporary type conversion in C, for example when a function's argument type should not be predefined to allow a generic use of the function. In C, a string is realized with a pointer to a char array. The following shows how we cast the char pointer to a void pointer and back.

```
// Type conversion with a string
char *string = "One line of characters.";
void *pvPointer = (void*)string;
char *pcPointer = (char*)pvPointer;
```

Note that we did not convert anything in the sense of manipulating the content of the string. We changed the data type, which only determines how the system should interpret the memory array, but we did not change the values in memory. As a result, after we changed the data type back from `(void*)` to `(char*)` we can work with `pcPointer` the same as with `string`.

If we want to do a temporary type conversion of multiple variables and potentially of multiple data types at once, we need to align them in memory. We can for example introduce a struct containing all our variables. This struct can then be casted similarly to the above example, except that we convert to/from `(task_param_t*)` instead of `(char*)`. Again, we do not change the struct's values in memory.

```
// Declaring and defining the struct
typedef struct {
    uint32_t value;
    int8_t anotherValue;
} task_param_t;

// Initialize the variable
task_param_t task_param = {
    .value = 0,
    .anotherValue = 42,
};

// Type conversion
void *pvStructPointer = (void*)&task_param; // Create a void pointer directing to
// the memory address of the variable
task_param_t *psStructPointer = (task_param_t*)pvStructPointer;
task_param_t param = *psStructPointer; // assign the value of psStructPointer
// to a new variable param
```

Project Structure For this lab, we provide you with a template project named `lab3_task1` and `lab3_task3` as a zip file (`lab3.zip`). Import the projects directly into your workspace using Code Composer Studio's import from archive feature. While the projects contain numerous files to separate different hardware modules, you will edit only the `main_freertos.c` file. A more detailed summary about the functionality implemented by the individual files is given in Table 1.

Task 1: Task Creation

If not already done, import the `lab3_task1` project into Code Composer Studio and open it.

Table 1: Project source files and folders of the lab3 projects. System startup code and linker files are omitted.

File/Folder Name	Description
FreeRTOS	FreeRTOS source and header files
FreeRTOSConfig.h	FreeRTOS configuration header file
main_freertos.c	Main program which needs to be modified
uart.{c,h}	UART interface initialization and communication

Task 1.1: Creating a printing task

Given the template, write the code which creates a task continuously printing out a string over UART. The task should be created in `main()` with the name "TaskA" and a corresponding task function `vTaskAFunction()`. From within the task function, the string "TaskA is running and running" should be printed and after printing, you should wait for a given period using the `vSimpleDelay()` function. Instead of hard coding the string into the task function, it should be given at the creation of the task by passing it as a task specific parameter. Set the `usStackDepth` to 1000, the `uxPriority` to 1 and the `pxCreatedTask` to NULL. Set Code Composer Studio's UART terminal to a baudrate of 115200 and check the UART output. What can you see?

Task 1.2: Creating a second concurrent task

Create a second task which prints out the string "TaskB is running and running" . Use the same settings as for the first task. What can you observe? Can you explain why it might not work properly? An improved version will be worked out in Task 2.1.

Task 1.3: Handles and deleting a task

A task can be deleted with the `vTaskDelete()` function (c.f. FreeRTOS Reference Chapter 2.11), which requires a task handle as argument in order to delete a task. Each task can be assigned a handle with the `pxCreatedTask` argument at task creation. If we want to delete a task from within its task function, we can simply use NULL as the argument to `vTaskDelete()`.

Change your program to do the following: Introduce a global counter in your code which is incremented in TaskB after printing to UART. When this counter reaches 10 the TaskB should delete TaskA using TaskA's handle. You need to create the required handles yourself. Finally, when it reaches 20, it should delete itself. Verify the correctness of your program by printing the value of the global counter to UART at the beginning of each iteration and checking the UART output.

Task 2: Critical Sections and Priorities

FreeRTOS uses a scheduling algorithm called Fixed Priority Pre-emptive Scheduling with Time Slicing. The FreeRTOS book explains the terms as follows:

Fixed Priority: Scheduling algorithms described as Fixed Priority do not change the priority assigned to the tasks being scheduled, but also do not prevent the tasks themselves from changing their own priority, or that of other tasks.

Pre-emptive: Pre-emptive scheduling algorithms will immediately "pre-empt" the Running state task if a task that has a priority higher than the Running state task enters the Ready state. Being pre-empted means being involuntarily (without explicitly yielding or blocking) moved out of the Running state and into the Ready state to allow a different task to enter the Running state.

Time Slicing: Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state. Scheduling algorithms described as using "Time Slicing" will select a new task to enter the Running state at the end of each time slice if there are other Ready state tasks that have the same priority as the Running task. A time slice is equal to the time between two RTOS tick interrupts.

Each task can be assigned a priority with the `uxPriority` argument of the `xTaskCreate()` function. The FreeRTOS scheduler uses these priorities to schedule tasks. The task with the highest priority which is ready to run will be executed. If two tasks have the same priority and are ready to run, time slicing is used to share processing time between these tasks.

Mutex Sometimes an operation on a shared resources should not be pre-empted as the correct functionality of the program is otherwise not guaranteed. In our case, the tasks created in Task 1 have the same priority and consequently time slicing pre-empts one task to allow the other task to run. When this happens during an UART transmission, the output will be corrupted.

A mutex can be used for exclusive access to a shared resource. This can be thought of as a single token attributed to the shared resource. Only if a task can successfully take the token it is allowed to use the resource. Then, the token is not available for any other task. When the task has finished using the resource it must give the token back, thus enabling other tasks to use the resource. In FreeRTOS three functions are used for mutual exclusion. The creation of a mutex is done with `xSemaphoreCreateMutex()`, a specific token can be taken with `xSemaphoreTake()` and returned with `xSemaphoreGive()` (c.f. FreeRTOS Reference Chapters 4.6, 4.16, 4.13).

```
0 // Function to create a new mutex and get its handle
1 SemaphoreHandle_t xSemaphoreCreateMutex( void );
2 // Function to take the mutex
3 xSemaphoreTake( SemaphoreHandle_t xSemaphore,          // Handle to the mutex
4               TickType_t xTicksToWait );              // Number of ticks to wait before
5                                                       // returning it automatically
6 // Function to return the mutex
7 xSemaphoreGive( SemaphoreHandle_t xSemaphore );        // Handle to the mutex
```

Snippet 4: Methods declared in the header file "semphr.h" to create and use a mutex.

Task 2.1: Mutex

Rewrite the `uart_println_mutex()` function using a mutex such that the task cannot be interrupted during UART transmission. First, declare a global `SemaphoreHandle_t` handle which you initialize in `main()` with the `xSemaphoreCreateMutex()` function. Then, use `xSemaphoreTake` and `xSemaphoreGive` to modify `uart_println_mutex()`. You can use FreeRTOS's global definition `portMAX_DELAY` as input for the `xTicksToWait` argument of `xSemaphoreTake`. Verify that your UART output is not corrupted anymore. In which order are the tasks executed?

Hint: Do not forget to include "semphr.h".

Task 2.2: Priorities

Change the priorities of your tasks such that the first task has a priority of 1 and the second task a priority of 2. What can you observe?

Non-blocking delay Our task function has a major drawback. After the string is printed via UART, the task waits for a while. In principle, the task is idle and during this time the second task could print

its string. However, the `vSimpleDelay()` function is implemented with a for-loop. From the scheduler perspective, the task is still in the running state because it is processing the for-loop. FreeRTOS offers a better solution to wait for a given time: the `vTaskDelay()` function, which is based on timers.

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

Snippet 5: A non-blocking delay function from FreeRTOS. (c.f. FreeRTOS Reference Chapter 2.9)

To fully understand how the function works, we first need to have a look at the more detailed task state machine in Figure 2, which is an extension of the state machine in Figure 1.

`vTaskDelay()` changes the state of the calling task from **Running** state to **Blocked** state for a given time period. Thus, the task will be in the **Not Running** superstate and another task, which is in the **Ready** state, can be executed. How long the delayed task will be in the **Blocked** state is specified in ticks by the `xTicksToDelay` argument. Ticks is a timing unit used by FreeRTOS internally. For convenience you can use the `pdMS_TO_TICKS()` function, which converts milliseconds to ticks. When the specified delay time has passed, an event occurs which moves the task into the **Ready** state.

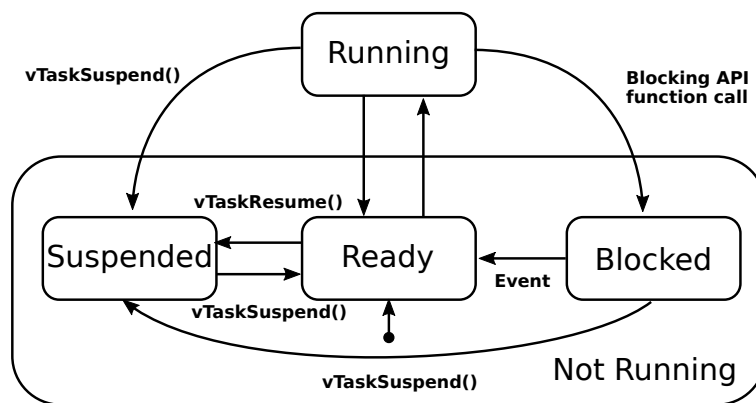


Figure 2: The task state machine of FreeRTOS

Task 2.3: Blocked state

Change the code such that both the lower priority and the higher priority task are able to execute and that the tasks print their string once every second. In which order do both tasks execute?

Task 3: Queues

Queues provide a way to communicate between tasks and are realized as First In First Out (FIFO) buffers in FreeRTOS. Each queue can maximally hold a predefined number of data items. These items can be of any size as long as the item size is fixed at the creation of the queue. In FreeRTOS, queues can be created with the `xQueueCreate()` function, where number of items (`uxQueueLength`) and item size (`uxItemSize`) must be defined. The function returns a handle to the queue which must be used to place items in the queue with `xQueueSendToBack()` and take items from the queue with `xQueueReceive()`. The handle must be available to all tasks which want to use the queue.

The `xTicksToWait` argument of `xQueueSendToBack()` and `xQueueReceive()` specify a block time. When a task tries to read from an empty queue, it will be kept in the blocked state until either another task writes to the queue or the block time expires. Similarly, when the queue is full and a task tries to write to the queue, it will be kept in the blocked state until either another task reads from the queue or

the block time expires, after which an error code will be returned. When either of these events happen, the task will be automatically moved to the ready state .

```

0 QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, // Number of items in the queue
1                           UBaseType_t uxItemSize    // Size of item in byte
2                           );
3
4 BaseType_t xQueueSendToBack( QueueHandle_t xQueue,      // Queue handle
5                             const void *pvItemToQueue, // Pointer to the item to be send
6                             TickType_t xTicksToWait    // Ticks until moved to ready state
7                             );
8
9 BaseType_t xQueueReceive( QueueHandle_t xQueue,          // Queue handle
10                          void *const pvBuffer,         // Copy of the item to be received
11                          TickType_t xTicksToWait        // Ticks until moved to ready state
12                          );

```

Snippet 6: Methods for handling queues in FreeRTOS. (c.f. FreeRTOS Reference Chapters 3.3, 3.22, 3.16)

Task 3.1: Inter-Task communication

For this task, you can use the new template project file lab3_task3.

Write a program with two tasks. Both tasks should declare one (and only one) local `uint32_t` variable. The first task (SquareTask) should initialize the local variable with the value 4. The second task (DecrementTask) should initialize it with 0. The tasks should follow a routine which is illustrated in Figure 3.

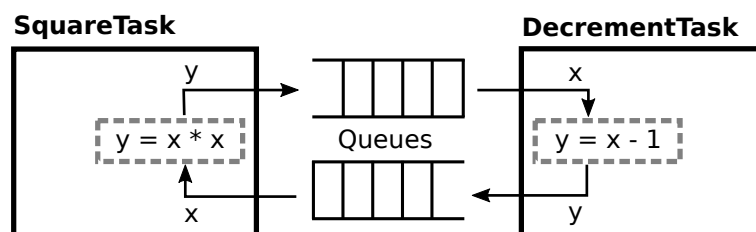


Figure 3: Routine of the application

The SquareTask routine:

- Communicate value of its local variable to the DecrementTask via a queue
- Wait for a value to be received
- Square the received value and update the local variable with the result
- Repeat from (a)

The DecrementTask routine:

- Wait for a value to be received.
- Decrement the received value by one and update the local variable with the result
- Communicate value of the local variable to the SquareTask via a queue

(d) Repeat from (a)

Each task should do the following after receiving a value

- Print its name and the current value
- Check if the value exceeds 10000. If this is the case the task should delete itself.

The overall goal of this task is to create the above mentioned functionality without using global variables. For simpler debugging you can subdivide the task into two steps:

- Start by implementing the required queue handles as global variables. Then implement the task routines and test if the UART output resembles the desired functionality.
- Avoid the global variables by passing the queue handles as function parameters. Control if your solution still works.