Embedded Systems Architectures - Laboratory Guide 2

Familiarization with FreeRTOS

Academic Year 2025/2026

Authors: Arnaldo Oliveira, Fábio Coutinho

1. Objectives

- Familiarization with FreeRTOS
- Creation of tasks and manipulation of parameters
- Communication and synchronization between tasks through Notifications and Semaphores

2. Introduction to FreeRTOS

A real-time operating system (RTOS) provides a deterministic task scheduler. Although the task scheduling rules vary by algorithm, tasks created should be completed within a specified time period. Compared to bare-metal approaches, the main advantages of using an RTOS are reduced complexity and improved software architecture, which facilitates maintenance.

The main real-time operating system supported by ESP-IDF is FreeRTOS. ESP-IDF uses its own version of FreeRTOS ported to RISC-V (or Xtensa, depending on the ESP32 variant). The fundamental difference from the basic FreeRTOS is its support for dual cores. In ESP-IDF FreeRTOS, if the processor has two cores, the application can choose which one to assign a task to, or it can let FreeRTOS choose. Other differences compared to the original FreeRTOS stem mainly from dual-core support. FreeRTOS is distributed under the MIT license. You can find the ESP-IDF FreeRTOS documentation at https://www.freertos.org/Documentation/02-Kernel/07-Books-and-manual/01-RTOS_book (it is important to read start reading this book, specifically chapters 1 – "Preface", 2 – "The FreeRTOS Kernel Distribution", 4 – "Task

Management”, 6 – “Software Timer Management”, 8 – “Resource Management” and 10 – “Task Notifications”.

2.1 The Kernel

FreeRTOS is a set of C libraries that includes a real-time kernel and modular libraries implementing complementary functionality. The FreeRTOS kernel is suitable for real-time embedded applications running on microcontrollers or small microprocessors. These types of applications typically include both hard and soft real-time requirements. Soft real-time requirements set a deadline, but failure to meet it would not render the system unusable. For example, responding very slowly to key presses can make a system seem annoyingly unresponsive, without actually making it unusable. On the other hand, strict (hard) real-time requirements set a deadline; failure to meet it would result in the system's complete failure. For example, a driver's airbag has the potential to cause more harm than good if it responds too slowly to the collision sensor signals.

The FreeRTOS kernel is a real-time operating system (RTOS) that enables applications built on FreeRTOS to meet strict real-time requirements. The kernel allows applications to be organized as a collection of independent execution threads or tasks. On a processor that has only one core, only a single execution thread can be executed at a time. The kernel decides which thread to execute by examining the priority assigned by the application developer. In the simplest case, the designer could assign higher priorities to threads that implement strict real-time requirements and lower priorities to threads that implement flexible real-time requirements. Prioritization in this way would ensure that strict real-time threads are always executed before flexible threads.

2.2 Why use a real-time operating system?

There are many well-established techniques for writing good embedded software without using a multithreading kernel. If the system to be developed is simple, these techniques may provide the most suitable solution. On the other hand, using a kernel will likely be preferable in more complex cases, but where the crossover point occurs will always be subjective.

Task prioritization can help ensure that an application meets its processing deadlines, but a kernel can also provide other, less obvious benefits, summarized below.

- **Time information abstraction -** the RTOS is responsible for runtime and provides a time-related API for the application. This makes the application code structure more straightforward and reduces overall code size.
- **Maintainability and extensibility -** abstraction of timing details results in fewer interdependencies between modules and allows the software to evolve in a controlled and predictable way. Furthermore, the kernel handles timing, so the application's performance is less susceptible to changes in the underlying hardware.
- **Modularity -** tasks are independent modules, each of which should have a well-defined purpose.
- **Team development -** tasks should also have well-defined interfaces, allowing for easier team development.
- **Easier testing -** tasks that are well-defined, independent modules with clean interfaces are easier to test in isolation.
- **Code reuse -** code designed with greater modularity and fewer interdependencies is easier to reuse.
- **Greater efficiency -** application code using an RTOS can be fully event-driven. There is no need to waste processing time polling for events that have not occurred. The efficiency gains of event-driven design are offset by the need to process RTOS interrupts and switch execution between tasks. However, applications that do not use an RTOS typically still include some form of interrupt.
- **Idle time -** the automatically created idle task runs when there are no application tasks that require processing. The idle task can measure available processing

capacity, perform background checks, or put the processor into low-power mode.

- **Power management -** the efficiency gains resulting from using an RTOS allow the processor to spend more time in low-power mode. Energy consumption can be significantly reduced by placing the processor in a low-power state each time the idle task is executed. FreeRTOS also has a special no-tick mode. Using no-tick mode allows the processor to enter a lower-power mode than it would otherwise and remain in that mode longer.
- **Flexible interrupt handling -** interrupt handlers can be kept very short by deferring processing to a task created by the application programmer or to the automatically created RTOS daemon task (also known as a timer task).
- **Mixed processing requirements -** simple design patterns can achieve a mix of periodic, continuous, and event-driven processing within an application. Furthermore, both strict and flexible real-time requirements can be met by selecting the appropriate priorities for tasks and interrupts.

3. Fundamental Aspects

FreeRTOS can be compiled with approximately twenty compilers and run on more than forty processor architectures. Each supported combination of compiler and processor corresponds to a FreeRTOS port.

3.1 Data Types

Each FreeRTOS port has a unique *portmacro.h* header file that contains (among other things) settings for two port-specific data types: **`TickType_t`** and **`BaseType_t`**.

3.1.1 TickType_t

FreeRTOS configures a periodic interrupt called a tick interrupt. The number of tick interrupts since the FreeRTOS application initialization is called the tick count. This count is used as a measure of time. The time between two tick interrupts is called the tick period, and times are typically specified as multiples of tick periods.

TickType_t is the data type used to store the tick count value and to specify times. **TickType_t** can be a 16-bit unsigned type, a 32-bit unsigned type, or a 64-bit unsigned type, depending on the **configTICK_TYPE_WIDTH_IN_BITS** setting in the *FreeRTOSConfig.h* file.

3.1.2 BaseType_t

It is always defined as the most efficient data type for the architecture. Typically, it is a 64-bit type in a 64-bit architecture, a 32-bit type in a 32-bit architecture, a 16-bit type in a 16-bit architecture, and an 8-bit type in an 8-bit architecture.

BaseType_t is generally used for return types that accept only a very limited range of values and for booleans of type **pdTRUE**/**pdFALSE**.

3.2 Programming Style

Adopting a programming style fosters more readable, consistent code, facilitates portability across family variants, reduces errors in concurrent tasks, and makes the development of embedded applications more robust and scalable.

3.2.1 Variable Names

Variables are prefixed with their type: '**c**' for **char**, '**s**' for **int16_t** (**short**), '**l**' for **int32_t** (**long**), and '**x**' for **BaseType_t** and any other non-standard types (structures, task identifiers, queue identifiers, etc.).

If a variable is unsigned, it will also have the prefix '**u**'. If a variable is a pointer, it will also have the prefix '**p**'. For example, a variable of type **uint8_t** will have the prefix '**uc**' and a variable of type pointer to char (**char ***) will have the prefix '**pc**'.

3.2.2 Function Names

Functions are prefixed with the type they return and the file in which they are defined. For example:

- **vTaskPrioritySet()** returns a void and is defined in *tasks.c*.

• **xQueueReceive()** returns a variable of type **BaseType_t** and is defined in *queue.c*.

• **pvTimerGetTimerID()** returns a pointer to void and is defined in *timers.c*.

File-scoped (private) functions are prefixed with '**prv**'.

3.2.3 Macros

Most macros are written in uppercase and preceded by lowercase letters indicating where the macro is defined. For more detailed information, you can consult the book "Mastering the FreeRTOS Real Time Kernel" or the "FreeRTOS Reference Manual", available online.

Prefix	Location of macro definition
port (for example, **portMAX_DELAY**)	*portable.h or portmacro.h*
task (for example, **taskENTER_CRITICAL()**)	*task.h*
pd (for example, **pdTRUE**)	*projdefs.h*
config (for example, **configUSE_PREEMPTION**)	*FreeRTOSConfig.h*
err (for example, **errQUEUE_FULL**)	*projdefs.h*

4. Practical Work

The practical work in this class is developed based on the book "Mastering the FreeRTOS Real Time Kernel", specifically chapters 4 – "Task Management", 8 – "Resource Management" and 10 – "Task Notifications". It is recommended to have the PDF version of the book for reference throughout the lesson plan.

4.1 Task Management

FreeRTOS tasks are the backbone of FreeRTOS. You can think of each task as an individual application running within your application as a whole. This makes it very easy to control what is happening in your application and segregate the functionality, so that individual tasks are responsible for specific sections of your application. FreeRTOS schedules tasks so that one or more can run simultaneously, depending on the number of available processor cores.

In this section, we will examine FreeRTOS tasks and how they help us structure our code to make modular programs easier to write.

Next, the “skeleton” of a program containing two functions is provided. Task 1 does something like simulate reading temperatures and task 2 simulates reading humidity, both in an infinite while loop. In this example, you want to read the temperature every second and, at the same time, read the humidity every two seconds.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void Task1()
{
   while (true)
   {
      printf("reading temperature \n");
      vTaskDelay(1000 / portTICK_PERIOD_MS);
   }
}

void Task2()
{
   while (true)
   {
      printf("reading humidity \n");
      vTaskDelay(2000 / portTICK_PERIOD_MS);
   }
}

void app_main(void)
{
   Task1();
   Task2();
}
```

Continuing with the compilation and execution of this code, you will verify that the humidity reading is never called.

Question 1: Why is the humidity reading never invoked?

FreeRTOS was designed so that when **`vTaskDelay`** is called and the task is waiting, FreeRTOS looks for other active tasks or tasks that need to be triggered and executes them as well.

Question 2: Study chapter 4, specifically the task creation functions, and modify the provided program so that the tasks are executed as expected.

Explore the different task creation options available in FreeRTOS.

Question 3: Consider that you want to identify the task being executed by passing this information as an input argument when creating the thread. Modify the program so that you obtain a result similar to the following:

```
~$ reading temperature from Task1
~$ reading temperature from Task1
~$ reading humidity from Task2
~$ reading temperature from Task1
~$ reading temperature from Task1
```

Question 4: Considering the previous example, create additional tasks and experiment with different periods and priorities. Can you find a scenario where one of the tasks is always blocked?

Question 5: The **`xTaskCreatePinnedToCore`** function allows you to select the core on which a task can be executed. Considering an ESP32-C6 target, what is the advantage over the **`xTaskCreate`** function?

4.2 Task Communication and Synchronization

Besides the modularity enabled by tasks, they often need to communicate and synchronize to coordinate their execution and safely access shared resources. For such purposes, two types of primitives are provided in FreeRTOS: Task Notifications and Semaphores, which will be addressed in the following sections.

4.2.1 Task Notifications

In this section, we will look at task notifications. Task notifications are one of the simplest ways to have a task: for example, a sender can notify a recipient of something. Task notifications require a handler for the created tasks. In this case, we will create a handler in the recipient that the sender can use to notify.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

static TaskHandle_t receiverHandler = NULL;

void Sender(void* params)
{
   while (true)
   {
      xTaskNotifyGive(receiverHandler);
      vTaskDelay(5000 / portTICK_PERIOD_MS);
   }
}

void Receiver(void* params)
{
   while (true)
   {
      ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
      printf("received notification\n");
   }
}

void app_main(void)
{
   xTaskCreate(Receiver, "receiver", 2048,
               NULL, 2, &receiverHandler);
   xTaskCreate(Sender, "sender", 2048,
               NULL, 2, NULL);
}
```

You should note that the recipient receives a notification every 5 seconds.

Question 6: Study the `ulTaskNotifyTake` function and explain why it is not necessary to delay the execution of the receiver's task with a call to the function `vTaskDelay`?

Question 7: What is the maximum time the **`ulTaskNotifyTake`** calling task should remain in the Blocked status to wait for its notification value?

4.2.2 Semaphores

Semaphores are used to synchronize access to shared resources. This section addresses the use of semaphores as mutexes to ensure mutual exclusion when accessing a shared resource (e.g., a variable or data structure, a bus, a peripheral or a file). The following code block illustrates a process for writing temperature and humidity values to a shared resource, in this case, a device or a file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"

void WriteToDevice(char* message)
{
   printf(message);
}

void Task1(void* params)
{
   while (true)
   {
      printf("reading temperature \n");
      WriteToDevice("temperature is 25C\n");
      vTaskDelay(1000 / portTICK_PERIOD_MS);
   }
}

void Task2(void* params)
{
   while (true)
   {
      printf(" reading humidity\n");
      WriteToDevice(" humidity is 50\n");
      vTaskDelay(2000 / portTICK_PERIOD_MS);
   }
}

void app_main(void)
{
   xTaskCreate(Task1, "temperature reading", 2048, NULL, 2, NULL);
   xTaskCreate(Task2, "humidity reading", 2048, NULL, 2, NULL);
}
```

Question 8: Analyze the code and explain why this code presents a problem.

Question 9: Starting from the provided code and considering the functions **`xSemaphoreCreateMutex`**, **`xSemaphoreTake`**, and **`xSemaphoreGive`**, construct a program that solves the problem identified in question 10 and, in case of an error in writing the humidity or temperature to the bus, displays the message "writing humidity timed out" / "writing temperature timed out", respectively.