Quick introduction to FreeRTOS

École d'été Temps Réel — Poitiers/Futuroscope

Jean-Luc Béchennec @ CNRS/LS2N

September 9, 2026

What is FreeRTOS ?

FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller - although its use is not limited to microcontroller applications.

A microcontroller is a small and resource constrained processor that incorporates, on a single chip, the processor itself, read only memory (ROM or Flash) to hold the program to be executed, and the random access memory (RAM) needed by the programs it executes. Typically the program is executed directly from the read only memory. (from FreeRTOS website).

- Small memory footprint so it can run on very small target.
  Runs on Arduino Uno for instance : 32kB flash, 2kB SRAM.
- Real-time fixed priority scheduler
- Periodic tasks
- Synchronization
- Critical sections
- Communication

Content

1. Setup
2. Suspending and Resuming Tasks
3. Critical Sections
4. Creating and Running Tasks
5. Implementing Periodic Tasks
6. What About the Stack Size
7. Communication

Setup

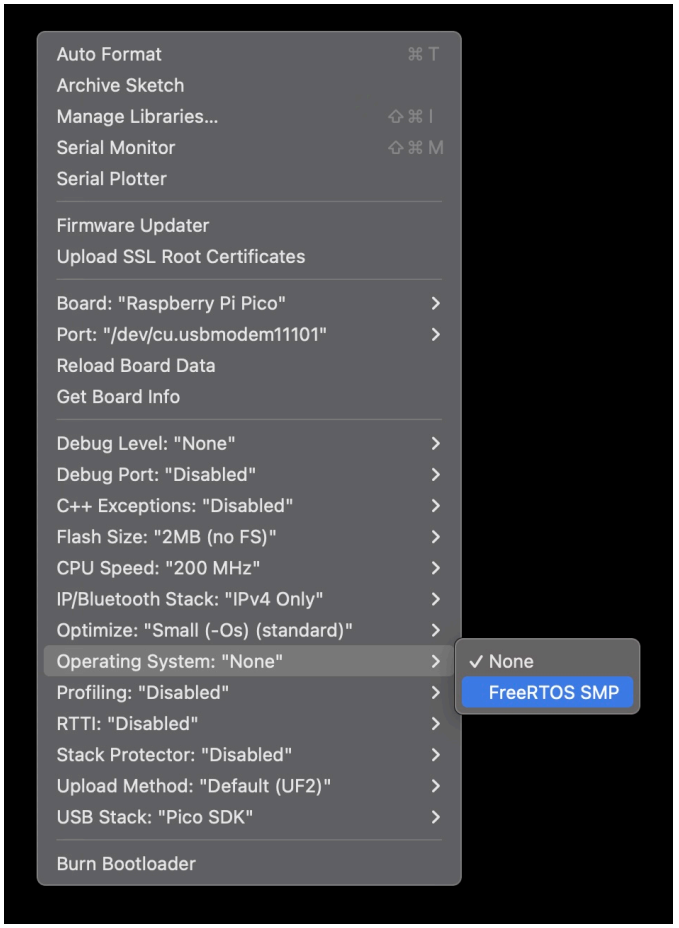
What do you need ?

This introduction to FreeRTOS includes a mix of lecture and lab sections.

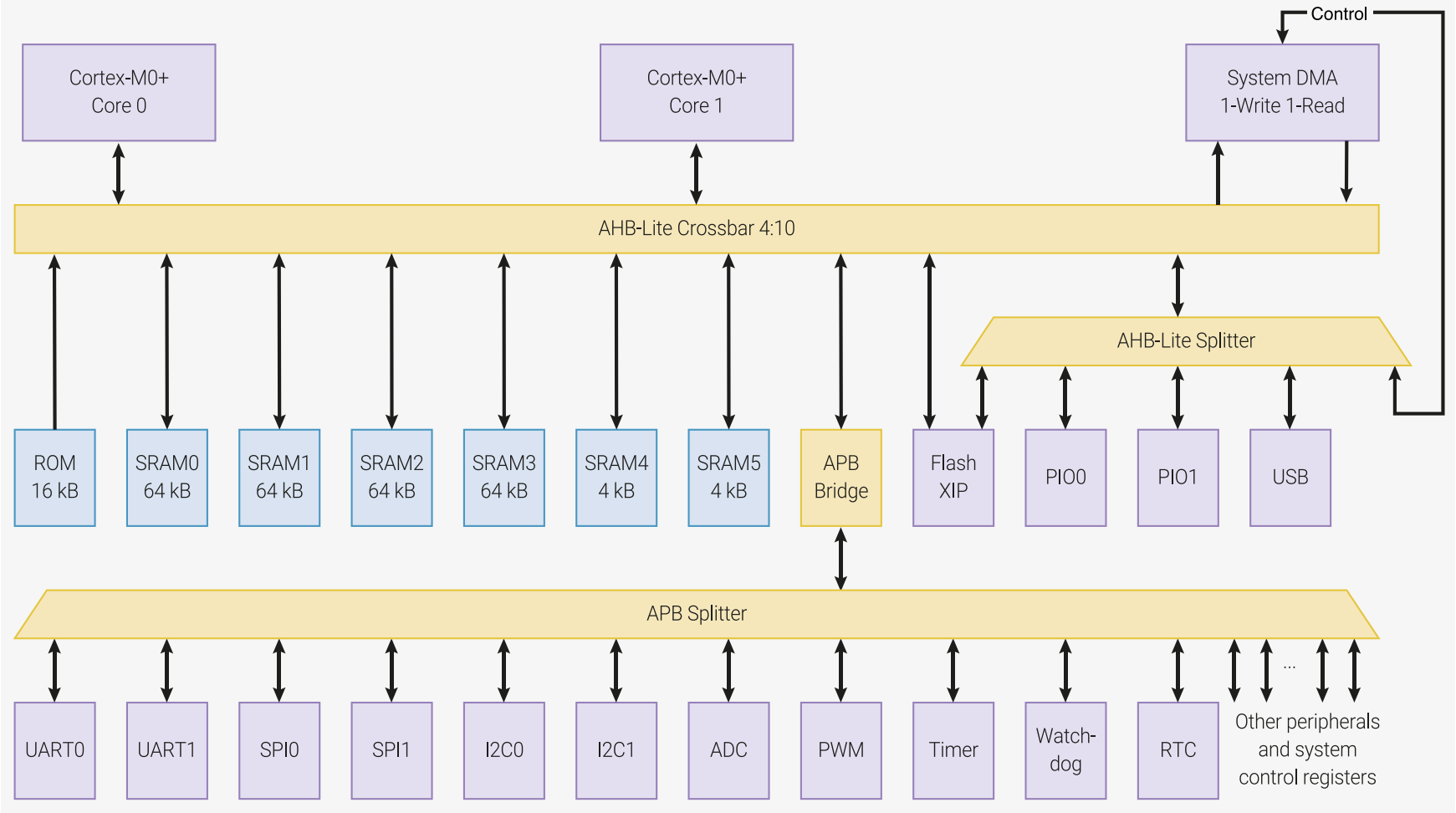
The hardware used is a Raspberry Pi Pico, a dual-core microcontroller.

To ease the deployment we use the Arduino IDE and the Arduino Pico package which allows you to develop for the Raspberry Pi Pico in the Arduino environment (compiler, configuration and upload). Normally, everything you need is already installed on the computers.

Launch the Arduino IDE and in the Tools menu, select **Board > Raspberry Pi Pico/ RP2040/RP2350 > Raspberry Pi Pico** (first item of the submenu). Then in the Tools menu select **Operating System > FreeRTOS SMP**



Architecture of the RP2040



What is the default setup ? (1)

Programming within the Arduino environment consists of providing the content of two functions: `setup` and `loop`.

- `setup` is executed once at start and
- `loop` is executed iteratively.

On the RP2040 which is a dual-core microcontroller `setup` and `loop` run on core 0.

The Arduino environment is extended to use the second core. Defining `setup1` and/or `loop1` indicates the Arduino basic software to start the second core to run `setup1` and `loop1` on core 1.

In addition the operating system FreeRTOS can be used by selecting the appropriate item in the **Tools** menu.

When **FreeRTOS SMP** is selected in the menu a few default tasks are created at start.

On Arduino Pico, FreeRTOS is configured so that tasks with the same priority are not scheduled using round-robin scheduling, unlike the configuration on the ESP32.

get starting code from https://etr26.triskell.org

What is the default setup ? (2)

To find them, open the program `what.ino`, compile it and upload it to the Pico. Then open the terminal in the IDE.

`what.ino` uses 2 FreeRTOS services:

- `UBaseType_t uxTaskGetNumberOfTasks( void )` (is it necessary to explain what it does?)
- `UBaseType_t uxTaskGetSystemState( ... )` takes 3 arguments. The first is an array of `TaskStatus_t`, the second is the size of the array and the third is a pointer to an unsigned long to store the time ealpsed since boot.

`what.ino` prints information returned by `uxTaskGetSystemState()`. Each element of the array is a `TaskStatus_t`. Here the following members of `TaskStatus_t` are printed:

pcTaskName a C string with the name of the task to ease debug;
xTaskNumber a unique identifier of the task;
eCurrentState the state of the task (eReady, eRunning, eBlocked, eSuspended, eDeleted);
uxCurrentPriority the current priority of the task;
uxBasePriority the base priority of the task;
uxCoreAffinityMask bit *i* set to 1 means the task can run on core *i*.

Work to do

Analyse and discuss the output given by `what.ino`

Suspending and Resuming Tasks

Suspending and Resuming tasks (1)

The first way to suspend and resume the execution of a task is to use the `vTaskSuspend` and `vTaskResume` services. When the argument is `NULL`, the target task is the task that called the service.

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend );
```

and

```
void vTaskResume( TaskHandle_t xTaskToResume );
```

With FreeRTOS in the Arduino environment, your application's initialization code runs after the OS starts up.

This can be problematic if you want to create new OS objects that would be used by `CORE0` and `CORE1`.

The goal here is to suspend the `CORE1` task in `setup1`, allow `setup` to run, and then resume execution of `CORE1`. To get the task handle of the current task, one can use:

```
TaskHandle_t xTaskGetCurrentTaskHandle( void );
```

In addition, task's state can be get by using

```
eTaskState eTaskGetState( TaskHandle_t xTaskToSuspend );
```

Possible values for a task's state are: `eRunning`, `eReady`, `eBlocked`, `eSuspended`, `eDeleted` and `eInvalid`

Suspending and Resuming tasks (2)

Work to do (1)

Modify `what.ino` to suspend `CORE1` in `setup1`. Note the change in the system state display.

Work to do (2)

Before suspending `CORE1`, get its handle and store it in a global variable named `core1Handle` which is initialized to NULL.

Wait for `CORE1` being `eSuspended` and resume it at the end of `setup`. This way, `setup` executes before `CORE1` code. Print again the system state to see the difference

Work to do (3)

Measure the time required to resume execution of a task using the `micros` function which returns in a `uint32_t` the number of microseconds elapsed since system startup.

Suspending and Resuming tasks (2)

Questions

1. *Is it necessary for the variable* `core1Handle` *to be* `volatile`*?*
2. *How do you synchronize the execution of the two cores to ensure that* `core1Handle` *contains the handle of* `CORE1` *before resuming execution?*

Critical Sections

Critical Sections (1)

FreeRTOS offers several mechanisms for building critical sections:

Binary Semaphore These objects can be in two states: free or taken. A task that attempts to take a free Binary Semaphore sets it to the taken state and continues execution. A task that attempts to take a Binary Semaphore that is already taken is blocked (`eBlocked` state).

Mutex The same thing as a Binary Semaphore with one small difference: priority inheritance. When a Binary Semaphore BS is held by task T1 and task T2, which has a higher priority than T1, attempts to take BS, T2 will be blocked. If a task T3 with a priority between that of T1 and T2 is now executed, T1 will be preempted by T3. The waiting time for T2 will therefore be extended by the execution time of T3, even though the priority of T3 is lower than its own. This is called priority inversion. Priority inheritance prevents this from happening. By replacing the Binary Semaphore with a Mutex, M, T1 will see its priority rise to that of T2 when T2 becomes blocked on M. T3 will therefore not be able to preempt T1 as long as the latter holds M.

Counting Semaphore This is the true semaphore as defined by *Edsger Dijkstra*. Its scope of application is broader than simply protecting a critical section. The two states of the Binary Semaphore are replaced by a counter. Taking the semaphore decrements the counter. If it is ≤ 0, the task blocks. Releasing the semaphore increments the counter.

Critical Sections (2)

Open `mutex.ino` and examine the code.

`CORE0` increments `counter` 1000 times and `CORE1` decrement it 1000 times.

Questions

What should be the final result?

But `CORE0` and `CORE1` **are in competition** to access global variable `counter`.

We need a critical section.

Critical Sections (3)

1. A Binary Semaphore/Mutex/Counting Semaphore is a variable of type `SemaphoreHandle_t`:

   ```
   SemaphoreHandle_t sem;
   ```

2. To create a mutex, use service

   ```
   SemaphoreHandle_t xSemaphoreCreateMutex()
   ```

3. To acquire a mutex, use the service

   ```
   BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore,
                              TickType_t xTicksToWait );
   ```

   First argument is the mutex, the second is the time it takes for the mutex to become available before giving up. To wait indefinitely, pass `portMAX_DELAY`. The service returns `pdTRUE` if the mutex has been acquired and `pdFALSE` otherwise. With `portMAX_DELAY`, `pdFALSE` is never returned.

4. To release a mutex, use the service

   ```
   BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
   ```

The service returns `pdTRUE` if the mutex has been released and `pdFALSE` if an error occured (usually the task try to release a mutex it had not taken).

Critical Sections (4)

Work to do

Modify `mutex.ino` to protect the accesses to `counter` with a mutex

Work to do

Verify the output is correct

Creating and Running Tasks

Creating and Running Tasks (1)

Task creation is done by calling the service

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char *const pcName,
                        const configSTACK_DEPTH_TYPE uxStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask
                      );
```

There are a lot of arguments:
pvTaskCode is the task function;
pcName is the name of the task;
uxStackDepth is the size of the task (in 32bits word on the Pico);
pvParameters is the argument of the task;
uxPriority is the priority of the task;
pxCreatedTask is a pointer to a `TaskHandle_t`.

The service returns `pdPASS` if successfull or `errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY` if not.

`configMINIMAL_STACK_SIZE` can be use for the stack size at first

The created task is added to the ready task list and, if its priority allows, is executed immediately. The task will run on any core and can migrate from one core to the other (Global Scheduler).

Creating and Running Tasks (2)

A second service allows to pin the task to a subset of the cores:

```
BaseType_t xTaskCreateAffinitySet( TaskFunction_t pxTaskCode,
                                   const char *const pcName,
                                   const configSTACK_DEPTH_TYPE uxStackDepth,
                                   void *const pvParameters,
                                   UBaseType_t uxPriority,
                                   UBaseType_t uxCoreAffinityMask,
                                   TaskHandle_t * pxCreatedTask);
```

To pin the task on core *n*, pass `1 << n` in `uxCoreAffinityMask`.

A `TaskFunction_t` is a function with the prototype:

```
void f( void * pvParameters );
```

Work to do

Open the `task.ino` file which contains the Blink1 *task : task handle, task function (an infinite loop) and creation code in setup which does not pin the task to any core.*

Run it to see on which core it runs then modify the code to pin Blink1 *on core 1.*

Implementing Periodic Tasks

Implementing Periodic Tasks (1)

Essentially, a task performs its initializations and then enters an infinite loop. It must block in order to allow lower priority tasks to execute.

To implement a periodic task with an approximate period, use the service

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

`xTicksToDelay` is a time in OS ticks. The macro `pdMS_TO_TICKS` converts milliseconds into ticks.

The period is approximate because the wake-up date is calculated by adding the delay to the date of the `vTaskDelay` call. The period is therefore the **sum of the task execution time and the delay**.

Work to do

Modify `task.ino` by setting the priority of Blink1 *to 5, Add code in `blink1Func` so that* Blink1 *blinks the LED on GP6. By using `vTaskDelay` set the LED is on during 100 milliseconds and off during 400 milliseconds*

To increase the execution time, add `for (volatile int i = 0; i < 10000; i++);` before calling one of the `vTaskDelay` functions. Otherwise, the task execution time remains negligible compared to the delays.

Implementing Periodic Tasks (2)

To have a specific period, you must be able to give a reference date in addition to the deadline.

That's what this service allows you to do.

```
void vTaskDelayUntil( TickType_t *pxPreviousWakeTime,
                      const TickType_t xTimeIncrement );
```

The first argument is a pointer to a variable where the previous wake-up date is stored. The task blocks until the date `*pxPreviousWakeTime + xTimeIncrement` and `*pxPreviousWakeTime` is updated with this date. In the task, before the infinite loop, you must initialize the variable with the current date by calling `xTaskGetTickCount()`.

Implementing Periodic Tasks (2)

Work to do

Create a second new task called Blink0*, pinned to core 0 and with priority 5, that blinks the LED on GP7. The LED is on during 100 milliseconds and off during 400 milliseconds too and uses* `vTaskDelayUntil`*.*

Question

What behaviour should you observe, explain what is happenning.

What About the Stack Size

What About the Stack Size?

Determining stack size is often overlooked. But this is one reason for software instability when the stack is not large enough.

Having an arbitrarily large stack consumes RAM for no reason.

Several methods exist:
- Offline code analysis to compute the largest stack needed;
- Verification of the stack pointer when a context switch is performed;
- Stack painting : the stack is filled with a pattern at start and how much of the stack is still painted gives the unused part.

FreeRTOS is configured to use the last one. The following function:

```
UBaseType_t uxTaskGetStackHighWaterMark ( TaskHandle_t xTask );
```

gives the number of unused words (4 bytes) at top of the stack.

Work to do

Modify your application to measure the unused stack of Blink0 *and* Blink1 *and display it.*

To do so, add a task that runs every second and print the high watermark of both tasks

Communication

Queues (1)

Queues are defined by 2 parameters:

- The size of the queue, counted in elements;
- The size of an element of the queue in bytes.

Creation

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

For example:

```
QueueHandle_t timeForLEDQueue = NULL;
const UBaseType_t tflQueueSize = 5;
const UBaseType_t tflItemSize = sizeof(uint32_t);

void setup() {
  ...
  timeForLEDQueue = xQueueCreate( tflQueueSize, tflItemSize );
  ...
}
```

Sending data

Pushing a data to a queue can be done at the end of the queue:

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

xQueue is the handle of the queue in which the data is written;

pvItemToQueue is a pointer to the data. **Note that the data itself is copied, not its pointer**;

xTicksToWait the waiting time if the queue is **full**. `portMAX_DELAY` to wait indefinitely. If `xTicksToWait`is different from 0, the calling task blocks.

`xQueueSend` returns `pdPASS` if the item was successfully pushed, `errQUEUE_FULL` otherwise.

Data can also be pushed at the front of the queue using:

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

Receiving data

Data are popped from the front of the queue using:

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

xQueue is the handle of the queue in which the data is read;
pvBuffer is a pointer to an area to store the data read from the queue;
xTicksToWait the waiting time if the queue is **empty**. `portMAX_DELAY` to wait indefinitely. If `xTicksToWait` is different from 0, the calling task blocks.

`xQueueReceive` returns `pdPASS` if the item was successfully popped, `errQUEUE_EMPTY` otherwise.

Queues (4)

Work to do

Remove Blink0 *from your application.*

Add a queue with a size of 4 and elements that are the following type:

```
typedef struct {
  uint32_t period;
  uint32_t onTime;
} BlinkParams;
```

Add a task called changeBlink *with a period varying between 4000 and 8000 milliseconds. It constructs a* `BlinkParams` *data item with a random* `period` *between 300 and 1000 milliseconds and an* `onTime` *between 50 and 200 milliseconds.*

The generated data is pushed to the queue

Modify Blink1 *to get the data from the queue and change its period and on time.*