

FreeRTOS SMP with Arduino Nano RP2040

Matteo Cenzato
Matteo Briscini
Michele Adorni

July 1, 2025

Contents

1	Project data	1
2	Project description	2
2.1	Design and implementation	2
2.1.1	Structure of the Project	2
2.1.2	Structure of the Library	2
2.2	Testing	3
2.2.1	TestSemaphores and TestSemaphoresSingleExec	3
2.2.2	TestOperations	4
3	Project outcomes	5
3.1	Concrete outcomes	5
3.2	Learning outcomes	5
3.3	Existing knowledge	6
3.4	Problems encountered	6
3.5	Future Work	6

1 Project data

- Project supervisor: Davide Baroffio

Last and first name	Person code	Email address
Adorni Michele	10739926	michele.adorni@mail.polimi.it
Briscini Matteo	10709075	matteo.briscini@mail.polimi.it
Cenzato Matteo	10744078	matteo.cenzato@mail.polimi.it

- Links to the project source code; GitHub

2 Project description

The purpose of this project is to study the usage of FreeRTOS, a lightweight open source operating system, on a Arduino Nano RP2040 Connect.

In particular, since on the board is mounted an ARM Cortex M0+, a dual core processor, we wanted to test the operating system in a Symmetric Multiprocessing (SMP) environment.

The goal of the project is to build a library which implements a consistency mechanism that is capable of deploying a generic function on both cores, collecting the results and assessing their equality.

2.1 Design and implementation

The programming environment chosen for the project is the Raspberry Pi Pico SDK. It is an open source project which contains a simple API understandable by both embedded developers and not, written in C and C++.

NB: the Arduino is based on the same silicon of the Raspberry Pi Pico 2040 (RP2040), and the SDK already contains a mapping of the ports here.

NB: some other tools such as picotool are employed in order to compile correctly the application. Every step of the setup is explained in detail on the README of the GitHub page.

2.1.1 Structure of the Project

The structure follows the official FreeRTOS community supported demos, where the main directory contains a `CMakeLists.txt` file, responsible for setting up the common dependencies of each single application, like the `pico_sdk_import.cmake`, which locates and includes the SDK dependencies.

Each directory needs to contain a `CMakeLists.txt` file responsible for creating the executable files. Especially it needs to initiate the sdk by using the dedicated function `pico_sdk_init()`.

When the whole project is compiled in the `build/` directory, one subdirectory per test chosen is created.

2.1.2 Structure of the Library

The library can be found in the file `include/LibraryFreeRTOS_RP2040.h`.

The developer needs to provide a configuration file `LibraryFreeRTOS_RP2040Config.h`, where some options of the library can be customized. Among all the options, it is worth mentioning the variable `RP2040config_testRUN_ON_CORES` which contains the number of cores active during the test procedure (in our case it is always set to 2).

In particular it offers a wrapper of all the initialization operations of both the SDK and FreeRTOS, and provides an API where the developer specifies the function name, return type of the variable, and eventual parameters passed as input to the function.

The library is responsible for automatically creating the tasks. Each test procedure comprises of three tasks, one *master* and two *slaves*.

Each *slave* is responsible for launching the function (one per core) and storing the results, both the output and the time taken to execute it. Moreover, it uses a task notification in order to notify that the execution has ended.

The *master* is responsible for creating the slaves, sleeps until all of them have finished execution, and collects all the results.

Since the tasks follow the FreeRTOS scheduler, it is **mandatory** that the slaves have less priority than the master.

Finally, as we show later in the testing section, the library supports the creation of multiple tests in the the same executable file by associating to each test a unique name.

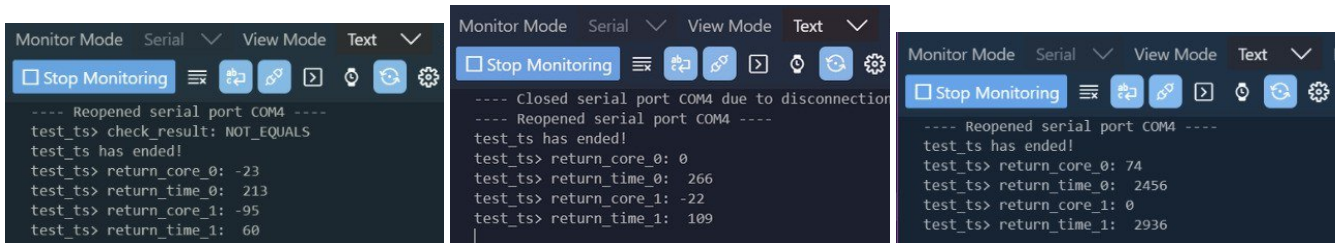


Figure 1: Outputs for each TestSemaphores test. From left to right, no locks, hardware lock, FreeRTOS lock.

2.2 Testing

The library for the moment supports two different types of tests:

- *void functions*: the library can run n different void functions (one per core) with the assumption that they all modify a shared variable, and at the end the master is responsible for printing the result contained in it.
- *non void functions*: the library can take a function which returns a primitive type (i.e. not a struct), run it on all the cores, and at the end the master prints the results obtained from the executions.

In order to understand how the library can be used, we propose three different tests.

2.2.1 TestSemaphores and TestSemaphoresSingleExec

In these procedures, we want to test the concurrent access of functions to a shared variable.

We present a classical setting where an integer variable is accessed by two functions, one responsible for incrementing it by 1 and the other decrementing it by 1, for a defined number of times. Since each addition and subtraction are not atomic, when executed concurrently the result contained in the shared variable could be different from zero.

In order to force the atomicity of the operations, we test both the usage of hardware locks provided by the SDK, and FreeRTOS locks, expecting that in both cases the final result should be equal to zero.

NB: the two testing procedures are exactly the same, except that for the first we produce three different executable files, and for the latter we produce a single executable which tests the three settings together.

NB: since the operations to perform are very basic, we force the compiler not to optimize the functions by using a preprocessor macro.

NB: this test can be performed also in a single core environment if both slaves have the same priority, as the round robin scheduler assigns the same timeslice to both the tasks. But the peculiarity of this specific test is that with multicores it works also with different priorities if the `configRUN_MULTIPLE_PRIORITIES` variable is set, as explained in detail here.

As we can see in figure 1, there is a clear interleaving between the threads running the additions and the subtractions. As expected, in the non thread safe example, we can see that the final result is not right.

The task which runs the functions, at the end of its execution, stores the value of the shared variable inside `return_core_N`. We can notice that for a thread-safe code, one task will print a value $\neq 0$, while the other one (which is also the last one to finish) will correctly print the value 0.

Moreover, for each execution we print the time elapsed (in the figure is represented in `ms`), and as expected, the fastest is the one requiring no locks to run. Interestingly, we can also see that the hardware lock is much faster than the software counterpart offered by FreeRTOS.

In figure 2 we show instead the result of the test `TestSemaphoresSingleExec`.

Here it seems that no overlap at all is happening, but it can be explained if we remember that now there are a total of 9 tasks (3 masters and 6 slaves) which need to be scheduled.

```

Monitor Mode Serial View Mode Text Port COM4 - USB Serial D
[Stop Monitoring] [Icons]
---- Closed serial port COM4 due to disconnection from the machine ----
---- Reopened serial port COM4 ----
test_nolock has ended!
test_nolock> return_core_0: 0
test_nolock> return_time_0: 23
test_nolock> return_core_1: -100
test_nolock> return_time_1: 27
test_sdk_lock has ended!
test_freertos_lock has ended!
test_sdk_lock> return_core_0: 100
test_sdk_lock> return_time_0: 65
test_sdk_lock> return_core_1: 0
test_sdk_lock> return_time_1: 65
test_freertos_lock> return_core_0: 95
test_freertos_lock> return_time_0: 1502
test_freertos_lock> return_core_1: 0
test_freertos_lock> return_time_1: 15397

```

Figure 2: Output of the TestSemaphoreSingleExec test.

```

Monitor Mode Serial View Mode Text Port COM4 - USB Serial
[Stop Monitoring] [Icons]
---- Reopened serial port COM4 ----
test_nolock> check_result: NOT_EQUALS
test_nolock has ended!
test_nolock> return_core_0: -15294
test_nolock> return_time_0: 52550
test_nolock> return_core_1: -15861
test_nolock> return_time_1: 51364
test_sdk_lock has ended!
test_sdk_lock> return_core_0: 36495
test_sdk_lock> return_time_0: 104347
test_sdk_lock> return_core_1: 0
test_sdk_lock> return_time_1: 146073
test_freertos_lock has ended!
test_freertos_lock> return_core_0: 47848
test_freertos_lock> return_time_0: 2539362
test_freertos_lock> return_core_1: 0
test_freertos_lock> return_time_1: 2796409

```

Figure 3: Output of the TestSemaphoreSingleExec test with 100000 iterations.

If we think about the execution flow after the `test_nolock` has created the two slaves, they need to be deployed one on core 0 (addition) and one on core 1 (subtraction). But on core 0 there are the other masters which are scheduled, and since they have an higher priority they will be run first. Instead, on core 1 there are no other tasks running, so the subtraction is free to run. This is why we see first the full execution of the subtraction, and after the full execution of the addition.

A similar behavior can be found in the `test_sdk_lock`, where core 0 finishes everything first; while in the `test_freertos_lock` we can see some overlapping.

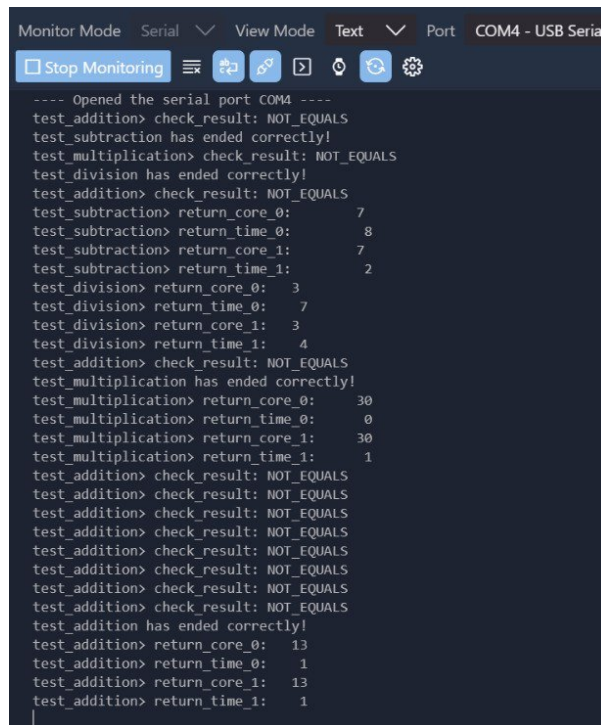
In order to force this overlapping behavior, we decided to increase the number of operations executed by each function from 100 to 100000.

In figure 3 there is a clear interleaving between the tasks, as the overhead caused by the master tasks has become negligible.

2.2.2 TestOperations

In this test we aim to study the case of *non void functions*. In particular we launch four pipelines, one per elementary operation: addition, subtraction, multiplication, division. Each pipeline deploys one *slave* per core, and at the end checks the equality of the results.

NB: we do not use floats as the Cortex M0 cpu does not have floating point units integrated.



```
Monitor Mode Serial View Mode Text Port COM4 - USB Serial
[Stop Monitoring] [Icons]
---- Opened the serial port COM4 ----
test_addition> check_result: NOT_EQUALS
test_subtraction has ended correctly!
test_multiplication> check_result: NOT_EQUALS
test_division has ended correctly!
test_addition> check_result: NOT_EQUALS
test_subtraction> return_core_0: 7
test_subtraction> return_time_0: 8
test_subtraction> return_core_1: 7
test_subtraction> return_time_1: 2
test_division> return_core_0: 3
test_division> return_time_0: 7
test_division> return_core_1: 3
test_division> return_time_1: 4
test_addition> check_result: NOT_EQUALS
test_multiplication has ended correctly!
test_multiplication> return_core_0: 30
test_multiplication> return_time_0: 0
test_multiplication> return_core_1: 30
test_multiplication> return_time_1: 1
test_addition> check_result: NOT_EQUALS
test_addition> check_result: NOT_EQUALS
test_addition> check_result: NOT_EQUALS
test_addition> check_result: NOT_EQUALS
test_addition> check_result: NOT_EQUALS
test_addition> check_result: NOT_EQUALS
test_addition> check_result: NOT_EQUALS
test_addition has ended correctly!
test_addition> return_core_0: 13
test_addition> return_time_0: 1
test_addition> return_core_1: 13
test_addition> return_time_1: 1
```

Figure 4: Output of the TestOperations test.

In figure 4 we can see the output produced by the tests, whose inputs are 10 and 3. We manually inserted some noise during the error checking phase, hence sometimes the result shown is not correct.

3 Project outcomes

3.1 Concrete outcomes

In the end, we produced a library for the execution of generic functions of the Arduino Nano RP2040 Connect using FreeRTOS SMP. We also developed an extendible testing environment where a developer can easily create new testing pipelines.

3.2 Learning outcomes

The most important skills learned by the group can be summarized as:

- *Development using an open source SDK*: the whole embedded programming process is facilitated through the usage of SDKs, and we learned how to compile and run an application for a specific device.
- *FreeRTOS SMP*: we expanded our knowledge in the field of multiprocessing at operating system level, in particular we learned how to use locks on shared variables (both hardware and software) and we understood how the scheduler operates when dealing with multiple cores.
- *Usage of macros and advanced C concepts*: macros in C are a powerful tool, but still need to be treated carefully as they can be cumbersome to understand and debug. They revealed to be very useful in our project too, and

we learned how to use them to construct data structures with generic types. Moreover, we discovered and used variadic macros and function pointers, which are powerful tools but may be not known by the standard programmer.

Nevertheless, the group also acquired some other skills which are not directly tied to the programming step. Indeed, we learned how to setup a Git repository by branching and avoiding merges whenever possible, and when some thing did not compile correctly, we learned to look on GitHub issues for discussions about the arguments and possible solutions.

3.3 Existing knowledge

Apart from the course Embedded Systems, our previous knowledge in operating systems provided by Advanced Operating Systems has been proven useful in order to understand the task structure and scheduling.

Also the course in Advanced Computer Architectures has helped us understanding some low level code and assembly (especially when debugging).

3.4 Problems encountered

The main issues encountered during the development of the project can be divided in two categories.

The first category comprises all the problems encountered while setting up the environment, which can be related to compiler version issues, wrong versions (or even missing) packages, and sometimes slightly unclear instructions in documentation.

But with some patience they were easily overcome and fixed.

The second category contains all the errors found during the development and testing of the application. For example, when running the *void functions* test initially we did not see any parallelization at all. But when we looked at the assembly code, we discovered that the operations were optimized automatically by the compiler. So we needed to manually enforce no optimization in that snippet of code.

3.5 Future Work

In order to expand the API, in the future we would like to explore different channels of communication for the output retrieval.

We already explored some possible solutions, especially for *WiFi*. The board has a dedicated embedded WiFi module, but it can be interacted only with Arduino. Since we are not using the Arduino programming environment, we would need to port the *WiFiNINA* library into our SDK.