B31DG Assignment 2 – Cyclic Executive and FreeRTOS

Embedded Software

Ignacio Chaux

H00484315

Dr. Mathini Sellathurai

April 18th 2025

Introduction

The purpose of this report is to develop and present an implementation of two different programs for a machine monitoring system using ESP32 Arduinos. The assignment is to follow strict real-time constraints and functional specifications. The first part is to develop a program using a cyclic executive approach and then the other to use a FreeRTOS framework.

Cyclic Executive

How did you design your cyclic executive?

The implementation of the cyclic executive was quite simple but came with its own struggles. The code is designed using a periodic task scheduler using a Ticker object to manage the tasks. This makes sure that the tasks are executed at their specific rate. The main function (allTasks) checks the elapsed time for each tasks using time stamps and the tasks are executed as prioritised using a time-driven manner based on their specific deadlines outlined in the assignment description. The tasks are implemented as independent functions with the monitoring process starting the tasks and ending it. The tasks are then nested in a single Ticker-driven loop that helps manage their timing. This helps get rid of the complexity of nested loops and CPU overhead. Finally, the button is implemented in the loop function to constantly ensure stable detection of the button press toggling the LED state and invoking the doWork() function. The main roadblock came to adding the frequencies and lighting up an LED. The lack of implementation was because focus was shifted more towards FreeRTOS. The second roadblock was reducing the number of violations on Task 2. Due of the way the tasks were organized and implemented it made it difficult for task 2 to catch up. The final roadblock was the displaying of task 5. Although the implementation was simple and there were no violations, it refused to display the violations message in the serial monitor. The only change made between the functional code and the faulty one was the updating of Arduino IDE.

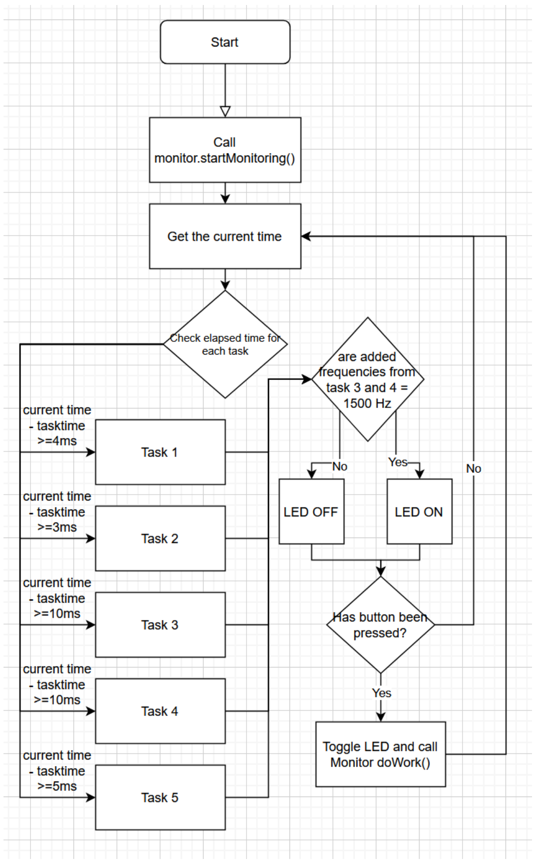
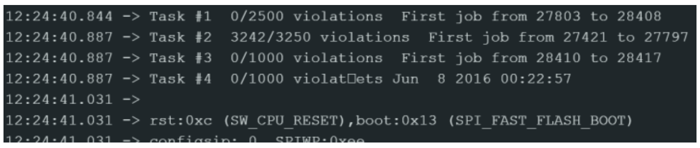

Start
Call monitor.startMonitoring()
Get the current time
Check elapsed time for each task
current time - tasktime >=4ms
Task 1
current time - tasktime >=3ms
Task 2
current time - tasktime >=10ms
Task 3
current time - tasktime >=10ms
Task 4
current time - tasktime >=5ms
Task 5
are added frequencies from task 3 and 4 = 1500 Hz
No
Yes
LED OFF
LED ON
No
Has button been pressed?
Yes
Toggle LED and call Monitor doWork()

Figure 1 - Flow chart for Cyclic executive



```
12:24:40.844 -> Task #1  0/2500 violations  First job from 27803 to 28408
12:24:40.887 -> Task #2  3242/3250 violations  First job from 27421 to 27797
12:24:40.887 -> Task #3  0/1000 violations  First job from 28410 to 28417
12:24:40.887 -> Task #4  0/1000 violat□ets Jun  8 2016 00:22:57
12:24:41.031 ->
12:24:41.031 -> rst:0xc (SW_CPU_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
12:24:41.031 -> configsip: 0, SPIWP:0xee
```

Figure 2 - Output image displaying error and violations from Cyclic Executive

How did you decide to set the priorities of your FreeRTOS tasks? Why?

Implementing proper prioritization of the FreeRTOS task is crucial for tasks to meet the proper deadline. The prioritization is straightforward, tasks with strict deadlines are the most important (task 1 and 2), thus having the highest priorities of all the tasks. The tasks with moderate deadlines were also important but were assigned medium priorities as they had more time to execute (3 and 4 ,10 ms each). Finally, the implementation of the low priorities were both input driven tasks (Task 6 and 7) and tasks with longer deadlines (Task 5) which would tolerate a lower priority as it does not need to be called excessively. This strategy ensures that the system fulfils the real-time requirements while also keeping the program stable, responsive and efficient.

How did you decide how to size the stacks of your tasks?

The stack sizes for the tasks in the FreeRTOS implementation were decided out of potential memory demands. The demands would depend on either the local variables, function calls or the semaphore operations. FreeRTOS documentation suggests larger stack sizes for tasks and tuning them based on runtime analysis to detect underflow and overflow. All the tasks were given the same stack size as it would provide a safe margin for the program to work correctly. For task 1 and 2 they need to generate signals using GPIO operations, they require low stack usage because of the simple delays and no complex data structures, therefore 4096 bytes gave a lot to work with. For task 3 and 4 they would perform frequency measurements and update variables, the stack size accounts for the timing data and the synchronization processing, 4096 bytes is plenty for the amount of work needed. The rest of the tasks involve simple logic and minimal memory usage, therefore 4096 bytes ensured enough from for any GPIO operations and debounce logic. The main concern for having a standardized stack size was to avoid stack overflow. Overflow can cause unstable behaviours and crashes to the real-time systems. Another thought was to add further expandability, providing the tasks with larger stack sizes would provide a buffer for further modification or additional features that would need to be implemented without requiring resizing.

Why and how did you use semaphores, mutexes, timers, and queues, if any?

In FreeRTOS the implementation of semaphores and mutexes are used to manage shared resources and match task execution to ensure the reliable and lag free operations in the system. Timers and queues were not used in the code because the task scheduler handled the timing and data flow satisfactorily. The first semaphore used was a binary button semaphore. It was implemented to detect button presses and ensuring that the button monitoring logic didn't miss deadlines or create lag conditions. It was implemented so that when the button was pressed, the semaphore would be triggered to allow the system to respond accordingly (toggle the LED). The mutex was used to ensure reliable access to shared variables between F1 and F2, which were updated by the frequency tasks to be read by the LED indicator. Task 3 and 4 used a semaphore take to acquire the mutex before writing F1 or F2, once the operation is complete it would release the mutex using semaphore give. The values would then go to the task 6 where they would be added, this ensured that the frequency values wouldn't have inconsistencies from the constant updates in task 3 and 4.

What is the worst-case delay (response time) between the time the push button is pressed and the time the LED is toggled (req. 7)? Will pressing the push button compromise the satisfaction of the real-time requirements (1–5)? Justify your answers.

The worst-case response time for the button on either the FreeRTOS or the cyclic executive system would be when all the other tasks are higher priority and have to run before the button LED execution. Doing simple addition of task execution time taking into account their time requirements we can determine that the worst delay would be 6.5 milliseconds. It would start with the execution of task 1, 2, 3 with a repetition of 2 and 1 before executing 4 which gives enough time for 5 to execute before the button reacts and turns on the LED. Using that pattern before executing the button would not interfere with the time requirements thus giving violations. There are many other best case scenarios where the delay between button to LED is much faster. But this is considering all the tasks before executing the button. This is because in the prioritization process of the tasks, the button was given the least important to ensure this does not overtake higher priority tasks responsible for meeting real time requirements (task 1-5).

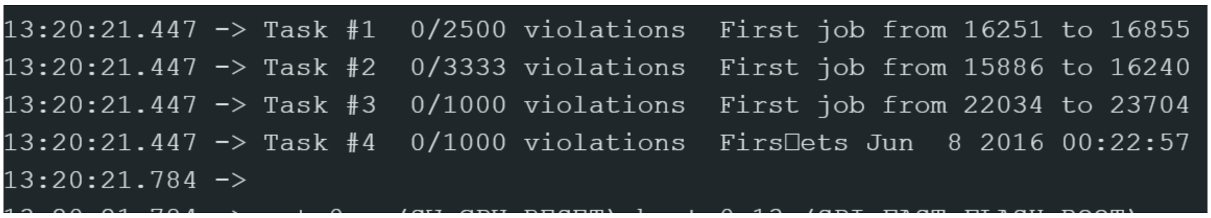
How does your FreeRTOS implementation compare with the cyclic executive implementation? What are their advantages and disadvantages?

Both FreeRTOS and the cyclic executive implementation fulfil the required real time tasks but differ in their design, flexibility and scalability. In terms of design the cyclic executive was simple as it only used a single main loop to manage all the task execution based on time intervals. The task scheduling relied on periodic checks to the elapsed time to keep up to date with the task requirements. As for FreeRTOS it works as a preemtive multitasker t manage all the tasks scheduled based on priorities and deadlines.

The advantages of cyclic executive are its simplicity, minimal overhead and minimal resources. Since the cyclic executive uses a straightforward design with a timing logic in a single loop, it makes it easy to understand for small systems. It also avoids the overhead runtime of a multitasking program such as context switching. Since it is so simple the hardware requires less memory and system resources making it perfect for smaller and more limited hardware.

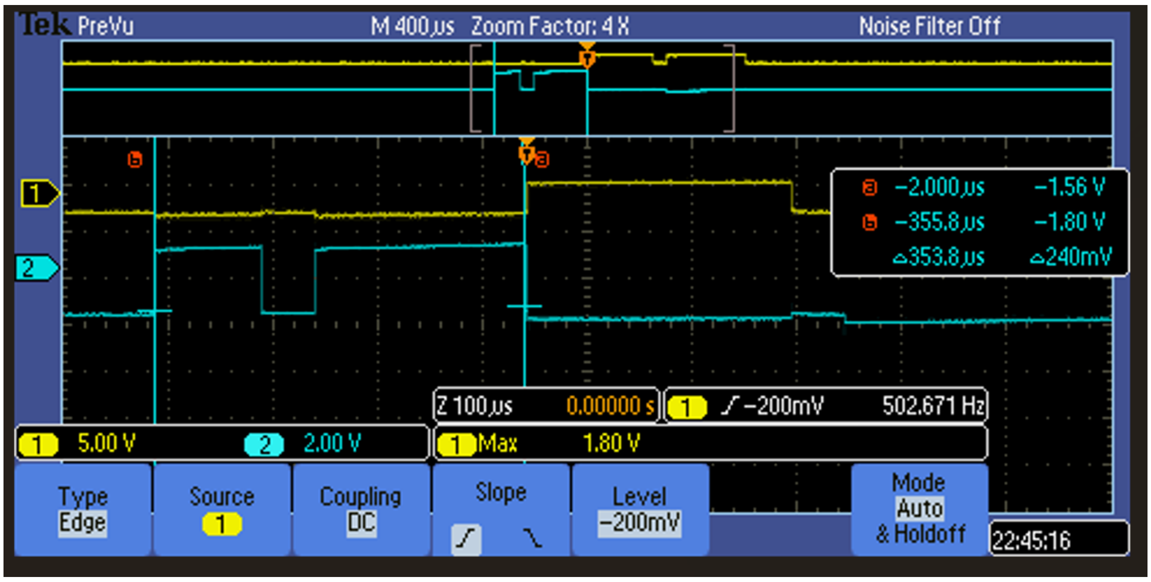
FreeRTOS has a lot more advantages, its scalability, task isolation, priority-based scheduling, ease of resource management and pre-emptive execution are only some of the benefits when working with it. FreeRTOS' can handle a larger number of tasks without adding complexity to the code. New tasks can be added easily and with their individual timing and priority. The tasks when using FreeRTOS are independent reducing the risk of interference between each task. Using priority-based scheduling means that the code can assign and schedule task with stricter deadlines to higher importance and guarantee CPU time. It also improves responsiveness for the real-time tasks compared to the cyclic executive. Semaphores, mutexes, etc. makes the sharing of resources and condition management straightforward. Finally with pre-emptive execution the higher priority tasks can be queued before the low priority tasks ensuring that they meet the deadlines even in scenarios where the timing is critical.

Overall, the FreeRTOS implementation was better executed as it had less errors to the cyclic executive while running, the main issue I encountered when doing the cyclic executive was the timing of all the tasks and reducing the violations from their timing. For FreeRTOS it became simpler as it was a matter of changing its priority and in some cases changing the size of the tasks. With this in mind, FreeRTOS still becomes a more reliable option when it comes to real-time systems, especially when the system becomes more complex.



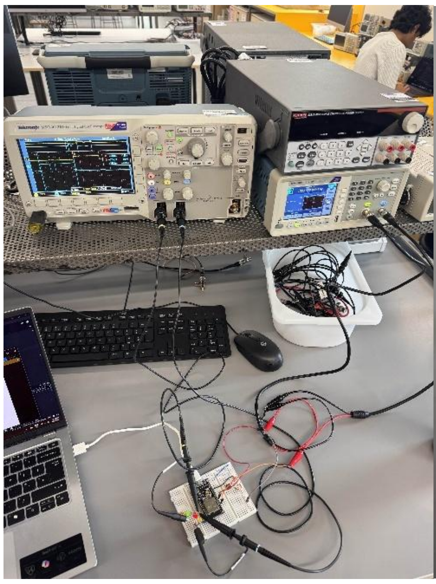
```
13:20:21.447 -> Task #1  0/2500 violations  First job from 16251 to 16855
13:20:21.447 -> Task #2  0/3333 violations  First job from 15886 to 16240
13:20:21.447 -> Task #3  0/1000 violations  First job from 22034 to 23704
13:20:21.447 -> Task #4  0/1000 violations  Firs□ets Jun  8 2016 00:22:57
13:20:21.784 ->
```

Figure 3 - FreeRTOS output



```
Tek PreVu          M 400µs  Zoom Factor: 4 X          Noise Filter Off
a  −2.000µs   −1.56 V
b  −355.8µs   −1.80 V
   △353.8µs   △240mV
Z 100µs   0.00000 s   1  ∕ −200mV   502.671 Hz
1  5.00 V    2  2.00 V
1 Max   1.80 V
Type Edge | Source 1 | Coupling DC | Slope ∕ ∖ | Level −200mV | Mode Auto & Holdoff
22:45:16
```

Figure 4 - Example output for task 2 (350 µs)



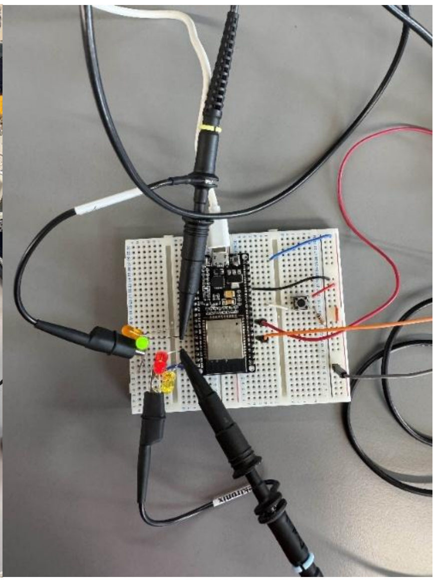


Figure 5 - Setup for running both programs

Figure 5.1 - Wiring setup for running both programs

References

Real Time Engineers Ltd. "Memory Management." *FreeRTOS*, Real Time Engineers Ltd., https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/09-Memory-management/01-Memory-management. Accessed 17 Apr. 2025.