

# The Report of B31DG-Assignment2

Wenbin Lin

github url: <https://github.com/lwb2001258/B31DG-H00484764-Assignment2>

## How did you design your cyclic executive?

The periods of the task1-5 are 4,3,10,10,5 ms respectively. So the LCM(Least Common Multiple) of the periods is 60. So in the cyclic executive part, the switch case should be splitted to 60 cases, from 0 to 59. And the frame() function can be run in one Ticker object for every 1ms. In the switch case part, if one case can finish in 1ms, and the following case can add other tasks. If one case can not finish in 1ms, such as it can only finish in 3 ms, the following two cases can not add any task, otherwise, it can not be executed. Then we can set the tasks in the cases according to the period and the execution time of the task. Such as task2, the execution time is 600us, and the period is 3 ms, so in the following case range (0-2,3-5,6-8,9-11,12-14,15-17,18-20,21-23,24-26,27-29,30-32,33-35,36-38,39-41,42-44,45-47,48-50,51-53,54-56,57-59), make sure in every case range, have one and just have one task2 in the range and make sure task2 can be finished in the range. Do it with all tasks and carefully adjust the tasks and make sure every task obey the rules.

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

Setting FreeRTOS task priorities is a design decision based on real-time requirements, task urgency, and resource contention.

Critical tasks such as responding to external events (e.g., GPIO edge detection for frequency measurement), need deterministic timing (e.g., waveform generation), interacting with hardware with strict deadlines, should be assigned higher priorities.

Non-critical tasks such as tasks which perform background operations, can tolerate delays, and have no hard deadlines, can be assigned lower priorities.

In the program of the rtos version, the jobTask1 and jobTask2 which are used to generate waves need deterministic timing, and the priority of the two tasks are set to 3. The jobTask3 and jobTask4 are used to measure the frequency, the priority of the two tasks are set to 3 too. And the jobTask5 which just delays for a short time and does not have a strict deadline, and the priority is set to 1.

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

Stack sizing in FreeRTOS is critical to avoid both stack overflow and unnecessary memory waste. There are three general strategies for stack sizing.

First, start with an estimate based on the complexity of the task (how many function calls? recursion? large local variables?), use of printf, String, std::vector, etc (which need more stack) and peripheral APIs (e.g., Serial.print() can consume significant stack).

Second, use uxTaskGetStackHighWaterMark() to monitor actual usage during runtime.

Third, adjust based on testing. Increase stack if overflow risk is detected (watchdog resets, strange behavior), or decrease if you see excessive free space.



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 pushbutton compromise the satisfaction of the RT requirements 1-5? Justify your answers.

The ISR routines run at hardware interrupt level, always preempt any FreeRTOS task, regardless of task priority (even priority 25). And they are not scheduled by the FreeRTOS kernel — they are handled directly by the CPU and the interrupt controller. So the worst-case delay for the button is less than 10us, mostly 1-7 us.

And pressing the pushbutton will affect the jobTask1-5. Because the ISR routine has higher priority than any FreeRTOS task. The function dowork() has to wait for 500us. The jobTask1 and jobTask2 in core1 will be interrupted if an ISR is triggered for the interrupt routine in core1 . So the pressing may compromise the satisfaction of the RT requirements 1-2.

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

Here's a detailed comparison between your FreeRTOS-based implementation and the cyclic executive approach.

| Feature           | Cyclic Executive                      | FreeRTOS                                      |
|-------------------|---------------------------------------|---|
| Scheduling        | Fixed time slots (static schedule)    | Dynamic, priority-based preemptive scheduling |
| Timing structure  | Repeats every frame (e.g., 20ms loop) | Tasks run independently based on logic        |
| Code architecture | Monolithic loop() with switch/cases   | Modular — each task in its own function       |

### Advantages of FreeRTOS Implementation

| Advantage         | Explanation   |
|-------------------|---|
| Flexibility       | Easier to add/remove tasks; no need to modify a static frame schedule       |
| Real preemption   | Tasks can run based on priority; urgent tasks interrupt less important ones |
| Better separation | Each task runs in its own function — better modularity and readability      |

|                          |  |
|--------------------------|--|
| Efficient CPU usage      | Idle time is released to background/low-priority tasks, programs can perform in all cores. |
| Built-in synchronization | Tools like mutexes, semaphores, task notify for coordination                               |

### **Advantages of Cyclic Executive**

| Advantage                  | Explanation   |
|----------------------------|---|
| Deterministic timing       | All execution paths known in advance; easier to analyze worst-case timing |
| No context switch overhead | Since it runs in a loop, no task switching = lower latency                |
| Low resource usage         | No task stacks, kernel scheduler, etc. — fits small MCUs better           |

### **Disadvantages of Each Approach**

FreeRTOS:

- Slightly more RAM usage due to separate stacks for each task
- Harder to analyze WCET (worst-case execution time)
- Improper task design (e.g., low-priority tasks blocking) can cause deadline misses
- It is hard to debug, and hard to analysis the running process

Cyclic Executive:

- Inflexible — adding a new task may require redesign of frame schedule
- Hard to scale — does not handle aperiodic tasks or dynamic conditions well
- Harder to maintain — everything is in one frame() function
- Unable to run multiple tasks in parallel, sometimes failing to meet the requirements.