The Report of B31DG-Assignment2

Wenbin Lin

How did you design your cyclic executive?

For this issue, first we define the following five arrays to use in the program.

- doneList: Whether each job has been completed in current period
- executeTimes:Estimated execution time of each job
- periodList:Execution period of each job
- jobCounts:Completion count of each job
- slackTimes:Slack time left for each job

For the non-rtos version, we start a job by ticker every ms to set the jobTask has not been completed and update the slack time of the job(slackTimes[i] = periodList[i] - totalTime % periodList[i] - executeTimes[i]) if ((int)(totalTime / periodList[i]) + 1 > jobCounts[i]). In the loop() function, we realize a schedule function. We traverse the five jobTasks, in the jobTasks who have not been completed in the current period, we select the jobTask who has the least slack time. And then we execute the jobTask. After finishing the jobTask, we set the jobTask finished in the current period and add one to the jobCounts of the jobTask. The detail is displayed in the following diagram(***Figure 1***).

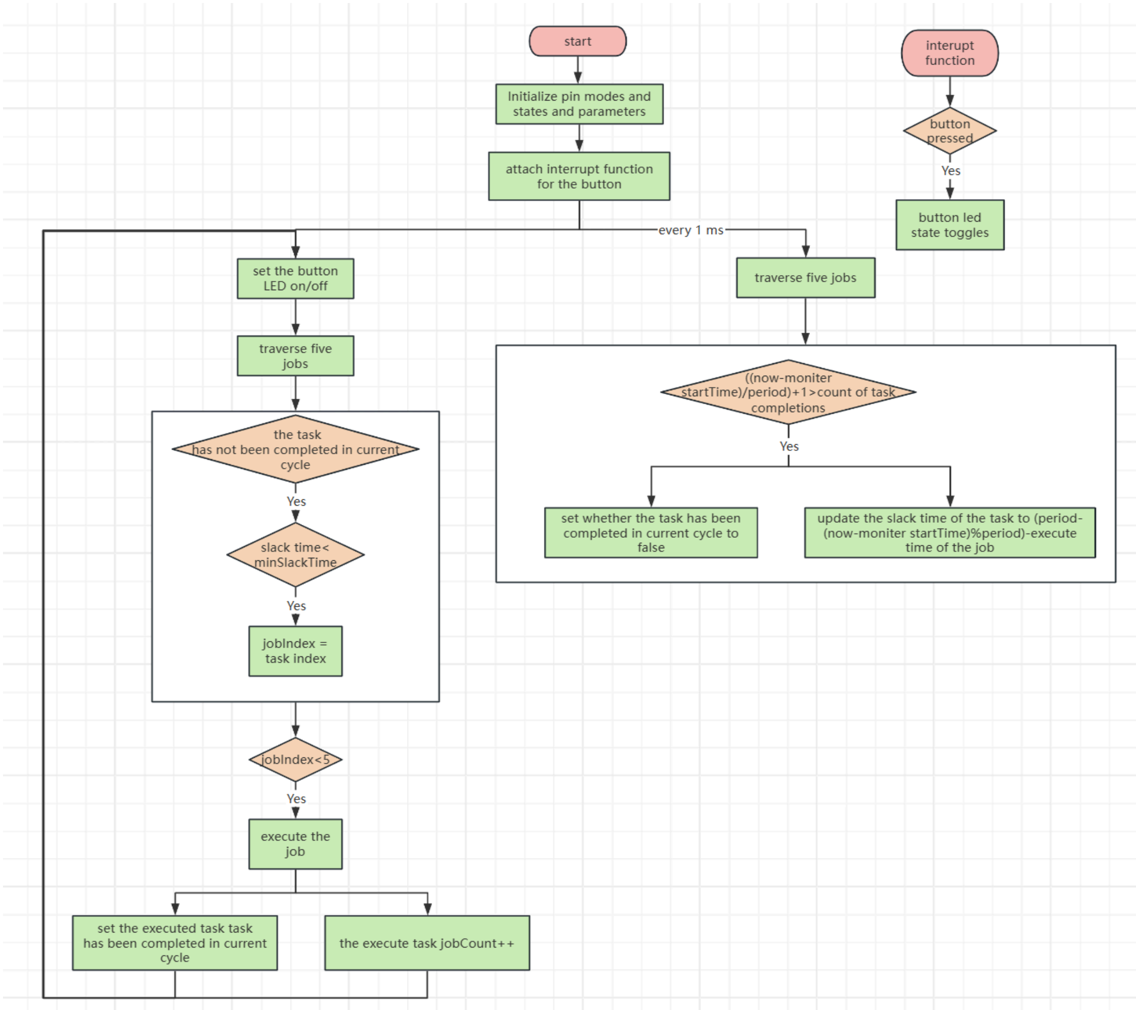

Figure 1. The flow of the program in non-rtos system

For the RTOS version, we use xTaskCreatePinnedToCore() function to create the tasks of the five jobTasks, ScheduleSlackTimeUpdateAndSetButtonLed, schedulerTaskCore1 and schedulerTaskCore0. The ScheduleSlackTimeUpdateAndSetButtonLed task is similar to the updateSlackTime() function in non-rtos version, only adding the changing led on/off by ButtonLedState. The function schedulerTaskCore1() is used to schedule the jobTask1 and jobTask2 which are spinned to core1. The function schedulerTaskCore0() is used to schedule the jobTask3, jobTask4 and jobTask5 which are spinned to core0. In the schedule function, the task which is unfinished and has the least slack time will be selected and executed by the xTaskNotifyGive() function which is used to send a notify message. And the task can receive the notify message by ulTaskNotifyTake(pdTRUE, portMAX_DELAY) function and start to execute the task. After finishing the task, it would send a notify message to trigger the schedule function and repeat the process again. The detail is displayed in the following diagram(***Figure 2***).

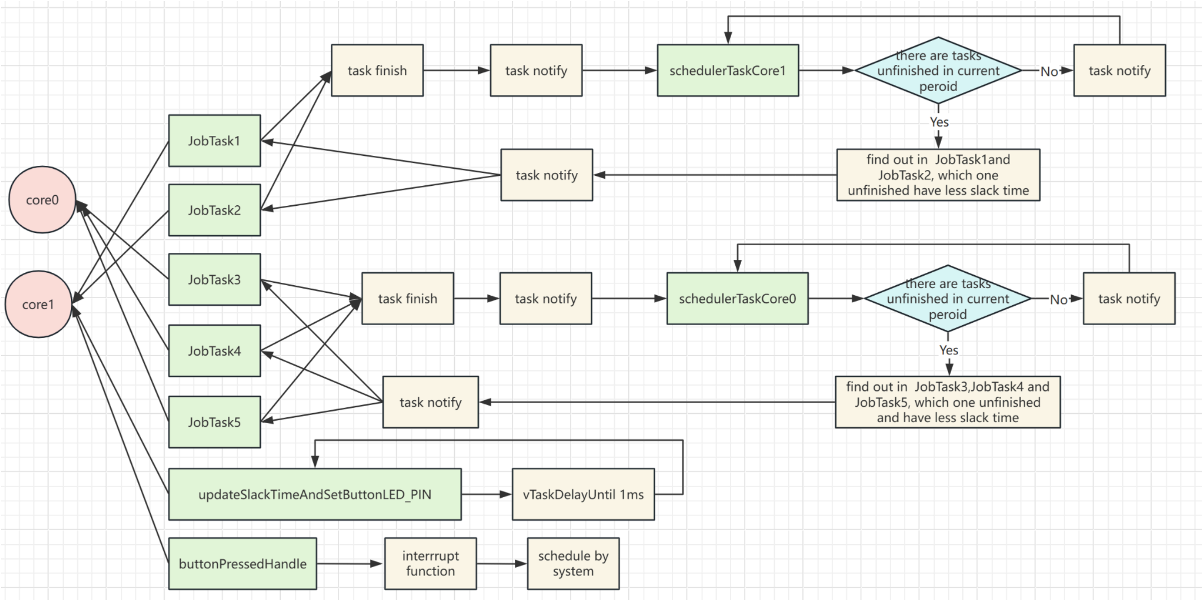

Figure 2 The flow of the program in rtos system

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 of it 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.

In my program in rtos version, for JobTask1–5 and schedulerTaskX , I set stack size to 4096. For the slackTimeUpdate task, I set stack size to 2048. It is ok for esp32, and I use uxTaskGetStackHighWaterMark() to monitor the stack size, and it shows there are still 6704 left.

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

I do not use semaphores,mutexes and timers, which seems to take more time to synchronize between give and take than notifyTask. I use xTaskNotifyGive() and ulTaskNotifyTake() to synchrone the tasks.The function xTaskNotifyGive() and ulTaskNotifyTake() are lightweight and efficient mechanisms in FreeRTOS for signaling between tasks — often used as a faster alternative to semaphores or queues. The function xTaskNotifyGive(taskHandle) is used by one task (or ISR) to "notify" another task. The function ulTaskNotifyTake() is used by the receiving task to wait for that notification. This works like a counting semaphore, but much faster and more lightweight. The detailed usage in the program I have discussed in the question how did you design your cyclic executive.

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 3-7 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 jobTask in core1 will be interrupted if an ISR is triggered. And 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
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
  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