A close-up of a form

AI-generated content may be incorrect.

**B31DG - Assignment 2**

Fraser Holman

H00368728

[fjah2000@hw.ac.uk](mailto:fjah2000@hw.ac.uk)

Due: 18th April 2025

Project GitHub Link:

<https://github.com/fholman/B31DGAssig2>

# Revision History:

|  |  |  |  |
| --- | --- | --- | --- |
| **Version Number** | **Date** | **Author** | **Notes** |
| 0.0 | 07/04 | Fraser Holman | Initial Report Layout |
| 1.0 | 11/04 | Fraser Holman | Final Report Draft |

Table of Contents

[Revision History: 2](#_Toc195107493)

[Function Setup 2](#_Toc195107494)

[Cyclic Executive 3](#_Toc195107495)

[Design 3](#_Toc195107496)

[Push Button 4](#_Toc195107497)

[RTOS 4](#_Toc195107498)

[RTOS Task Priorities 4](#_Toc195107499)

[RTOS Stack Sizes 5](#_Toc195107500)

[Semaphores / Mutexes 5](#_Toc195107501)

[Push Button 6](#_Toc195107502)

[Comparison 6](#_Toc195107503)

# Function Setup

To satisfy the 7 coursework requirements as outlined within the specifications several methods were implemented across both the cyclic executive and RTOS firmware. The only difference was the addition of a while loop within the RTOS tasks as required. The methods in Figure 1 were utilised to satisfy each requirement.



Figure 1 - RT Tasks

digitalSig1() and digitalSig2() generates a pulse as per requirement 1 and 2.

getF1() and getF2() using the measureFrequency(gpio\_num\_t signal) method to measure the frequency of a square wave signal as per requirement 3 and 4.

monitorProgram() calls the doWork() method which calls a busy wait which satisfies requirement 5.

monitorLED() toggles an LED on/off dependent on the 2 frequencies measured by getF1() and getF2() as outlined by requirement 6.

monitorButton() toggles the state of a second LED and calls the doWork() method dependent on the state of the button as required by requirement 7.

The toggleClick() ISR will change the button state and the logic is then handled within the monitorButton() method.

# Cyclic Executive

### Design

To determine the cyclic executive design it was first necessary to work out the computation time (ci) for each of the 5 methods that were to be monitored to determine the best duration for each frame. The number of frames that needed to be executed to satisfy the real-time requirements can be determined by the LCM of the period (Pi) of the 5 methods. Table 1 shows the key relevant information for each task.

Table 1 - Key Task Information

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Method** | **Pi (ms)** | **Ci (ms)** |
| Task 1 | digitalSig1() | 4 | 0.605 |
| Task 2 | digitalSig2() | 3 | 0.355 |
| Task 3 | getF1() | 10 | 1.501 |
| Task 4 | getF2() | 10 | 1.2 |
| Task 5 | monitorProgram() | 5 | 0.502 |

The LCM of the periods of the 5 tasks can be determined as 30, this is the number of frames required before all the tasks sync back up again. The length of each frame can then be determined by rounding up the task with the longest computation time. In this case that is Task 3 which takes 1.5ms to execute so therefore each frame should be 2ms long.

To determine which tasks were to run in each frame, a form of RMS scheduling was used as shown by Table 2. Initially, the tasks with the shortest periods were prioritised to run on the CPU first (in this case task 1, 2, and 5), this was then further adjusted to fit in tasks 3 and 4 with the longer periods. There were some issues with the monitor library violating if the tasks were scheduled too close to the end of its period so it was further adjusted to attempt to make each frame have as high a slack time as possible. Doing so, also allowed more opportunities to call the monitorButton() method to check on the recent state of the button. Checking the state of the button will be further discussed in the following section.

Table 2 - Cyclic Executive Design

A screenshot of a computer

AI-generated content may be incorrect.

### Push Button

Requirement 7 outlines that a push button should call the doWork() method that busy waits for 500us and toggle an LED on and off. As the program must follow the strict time constraints of the RT system it is more efficient to have the ISR change the state of a variable and the rest of the logic can be handled within a separate method. To apply this to the cyclic executive schedule the new monitorButton() method was called in every frame which had a large amount of slack time but would still have enough to spare to ensure the program runs smoothly. From testing, it was determined that if the method was called in a frame which would result in a slack time closer to 0 this occasionally caused the RT requirements to fail which was far from ideal. By following this approach the monitorButton() method was called no longer than 7 frames apart or 14ms. Therefore the worst case delay between the button press and LED toggled is 14ms which is not an observable delay by the user. By running the monitorButton() method this infrequently ensured that other tasks held priority of the CPU time but allowed the method enough time to be computed regularly enough.

# RTOS

### RTOS Task Priorities

The program was split into 7 different tasks based on the coursework requirements. The 5 main tasks with higher priorities aligned with requirements 1-5. These were the 2 signal output tasks (outlined by requirements 1 and 2), the 2 frequency measurement tasks (outlined by 3 and 4), and the do work task (outlined by 5). The 2 lower priority tasks aligned with requirements 6 and 7 to monitor an LED and button. As both tasks were not critical to the entire systems operation it was decided that both these tasks were assigned a priority of 0, on top of this to prevent precious CPU time, each task was only scheduled to run every 120ms. This is quite a large delay between method calls but as the tasks were for visual purposes (ie toggling an LED) it was not required to have the tasks update more frequently. To further save CPU time it was necessary to handle the button logic within a separate RTOS task which is signalled by the ISR with a binary semaphore to ensure that the relative ISR method is exited as quickly as possible.

The priority assigned for the 5 main tasks varied regularly during development. From the beginning, it was known that they should have a higher priority than the monitoring LED and button tasks (tasks 6 and 7) so each of these main tasks should have a priority of at least 1. Implementing higher priorities for tasks that ran more frequently was first tested to ensure they received the necessary CPU time for each cycle and was not pre-empted by the less frequent tasks. The issue found here was that the less frequent tasks (Tasks 3 and 4) were starved from CPU time and this caused RT requirements to be compromised. It was then tested to assign higher priorities to the tasks that take longer to run (Tasks 3 and 4) and it was found that this caused the shorter tasks (Task 1, 2, and 5) to violate again due to being starved of CPU time. It was then decided to assign all the tasks the same priority to ensure that neither longer duration or less frequent tasks were prioritised over the other and this resulted in a system with no violations and deadlines were met for all iterations of the software and for all edge cases.

Table 3 - Priority of RTOS Tasks

|  |  |  |
| --- | --- | --- |
| **Task** | **Related Method** | **Priority** |
| Task 1 | digitalSig1() | 1 |
| Task 2 | digitalSig2() | 1 |
| Task 3 | getF1() | 1 |
| Task 4 | getF2() | 1 |
| Task 5 | monitorProgram() | 1 |
| Task 6 | monitorButton() | 0 |
| Task 7 | monitorLED() | 0 |

### RTOS Stack Sizes

Stack size determines how much memory is assigned to each task. To determine the appropriate stack size that should be allocated to each task, when creating the RTOS task using xTaskCreate the stack size should first be initialised to much higher than is expected to ensure there is no stack overflow or memory errors. The method: *uxTaskGetStackHighWaterMark(TaskHandle\_t xTask)* can then be used to determine how much memory is unused for the corresponding task. The allocated stack size can be adjusted accordingly, although it should have an assigned stack size relatively higher than what the task uses to ensure there is no unexpected memory corruption or stack overflow. For such a small system there is plenty of memory that can be allocated to freeRTOS tasks so there is no concern of over-allocating memory between the tasks. For a much larger system it is good design to ensure each task does not waste any allocated memory by adjusting the stack size to be closer to how much memory is used to prevent allocating more memory than is available. By calling this method: *heap\_caps\_get\_free\_size(MALLOC\_CAP\_8BIT),* will return the number of available bytes that can be allocated to the program. From testing this it was determined that there was 269kb of available memory that can be allocated to the RTOS tasks, from this knowledge it was decided that a stack size can be allocated to each task to ensure that no task uses more than 50% of its allocated memory and completely prevents the risk of stack overflow or memory corruption, the total amount of memory allocated to all the RTOS tasks as determined in Table 4 was 293 bytes which is only 0.1% of the total available memory so there is no risk of running out of memory and plenty of memory is allocated to each task to ensure there are no issues.

Table 4 - Stack Size of RTOS Tasks

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task** | **Related Method** | **Allocated Stack Size (Bytes)** | **Max Stack Used (Bytes)** | **Percentage of Stack Used** |
| Task 1 | digitalSig1() | 2048 | 948 | 46% |
| Task 2 | digitalSig2() | 2048 | 948 | 46% |
| Task 3 | getF1() | 2048 | 948 | 46% |
| Task 4 | getF2() | 2048 | 948 | 46% |
| Task 5 | monitorProgram() | 2048 | 948 | 46% |
| Task 6 | monitorButton() | 2048 | 712 | 35% |
| Task 7 | monitorLED() | 2048 | 728 | 36% |

### Semaphores / Mutexes

Throughout the RTOS program, both semaphores and mutexes were utilised. A mutex can be used to protect shared resources so it was clear it would be beneficial to use for reading from or writing to the frequency values. The monitor LED task (Task 7) requires reading from the frequency variables and the measure frequency task (Tasks 3 and 4) requires writing to the frequency variables. To ensure the LED monitoring task (Task 7) can only read the frequency after the variable has been written, a mutex can be used. Initially, only 1 mutex was used to write to both frequency variables, this ensures that only 1 frequency can be written at a time and reading/writing cannot happen simultaneously. The clear issue was that both frequency tasks had their own global frequency variables, F1 and F2, using only 1 mutex prevents the tasks from writing to their own variables simultaneously so a second mutex was introduced. However, this meant the LED monitor task (Task 7) had to hold both Mutexes. Initially, both Mutexes were held simultaneously, the data was read, and then both were released, although to further improve on this it was decided to only read 1 frequency at a time to prevent the task from locking out the first mutex while waiting for the second to be free which could cause either Frequency tasks (Task 3 and 4) to have to wait on the monitor LED task (Task 7) to release a mutex. A binary semaphore was also used to signal the monitor button task (Task 6) from the ISR. It is bad design to hold too much time-consuming logic within an ISR as this can waste CPU time and cause other tasks to slow down and prevent additional interrupts. Using a binary semaphore to signal to a different task when the button has been pressed prevents the ISR from using up too much time as all the logic is delegated to another task. As discussed above this monitor button task (Task 6) was set with a priority of 0 and only checked if the semaphore was available again every 120ms to prevent the task from using up CPU time over the high priority tasks.

### Push Button

For the RTOS system, the monitorButton() method is only scheduled every 120ms although it will wait within its task for the semaphore to be signalled by the ISR. In most scenarios, the delay between pressing the button and the LED to be toggled will be instant (although dependent on when the CPU can give up time for the task) as the task will likely already be waiting for the semaphore to be signalled. In the worst-case scenario, it will take 120ms (plus time for the CPU to give up processing time for the task) but this only occurs if the button is pressed instantly a second time which in most cases is faster than a human can click. From measuring the time the button was pressed to the LED toggling using the micros() function the worst case delay observed was 5.1ms which is significantly shorter than what was first estimated above, this clearly shows that although the task is running infrequently with such a low priority the task is still given enough CPU time during operation. Running the task at such a low priority and infrequently ensures that it uses up minimal CPU time and does not cause the RT requirements to be compromised whatsoever.

# Comparison

Starting this project the initial thought was that the FreeRTOS implementation would be easier to implement over the cyclic executive design as using an RTOS system handles the scheduling for the programmer and allows multiple different tasks to appear to run in parallel. An RTOS will also allow tasks to run while other tasks are in a waiting state as compared to the cyclic executive where a delay will cause the whole system to pause. Although this is a clear benefit of the RTOS system it was found that cyclic executive allowed better control over the exact time tasks are executed and how frequently, this is an advantage for tasks such as the monitorButton() method as the programmer can decide exactly when to check the button state whereas in the RTOS system it is unsure exactly when this method would run and if it would end up using CPU time over a more important task that is constrained to the RT requirements. This can be solved by adjusting the priorities of different tasks within the RTOS system to ensure that these less constrained tasks are not pre-empting higher priority tasks. Although being able to prioritise tasks in RTOS is a clear advantage for higher priority tasks this can be a disadvantage for low priority tasks, if the RT system is already tight for time this can cause the lower priority tasks to be ignored and starved for CPU time whereas in the cyclic executive, the designer can have full control over when these less important tasks are run so therefore no tasks will be starved if designed correctly. A benefit of using an RTOS system over the cyclic executive is the scalability of the method, implementing more tasks in an RTOS system is incredibly easy as the RTOS handles the schedule itself as compared to the cyclic design where the whole system will need to be redesigned to take into account of the new task requiring CPU time as well. Although the cyclic executive has clear benefits of allowing the designer to have full control over the system, the RTOS implementation is clearly superior due to its scalability and CPU efficient design.