Electrical, Electronic & Computer Engineering

School of Engineering & Physical Sciences

**B31DG: Embedded Software 2024**

**Specifications Assignment 2**

**Released on Week 7, Demo due on Week 12, Report Week 13**

**Problem**

You need to develop two different implementations of a machine monitor system using your ESP32 kit.

You need to use this library library to test and demonstrate that each of your programs meets all its real-time requirements.

Download the library from here: <https://canvas.hw.ac.uk/files/4020953/download?download_frd=1>

See full description on how you should use the library here: <https://canvas.hw.ac.uk/files/4020955/download?download_frd=1>

Both your programs must satisfy the following 7 requirements:

1. Output a digital signal. This should be HIGH for 250μs, then LOW for 50μs, then HIGH again for 300μs, then LOW again. You should use spin wait loops (e.g. using calls to standard *delay()*) to implement the short time intervals you need. Do not use interrupts.



1. Output a second digital signal. This should be HIGH for 100μs, then LOW for 50μs, then HIGH again for 200μs, then LOW again. You should use spin wait loops (e.g. using calls to standard *delay()*) to implement the short time intervals you need. Do not use interrupts.
2. Measure the frequency of a 3.3v square wave signal. The frequency of the wave signal in input will be in the range 666Hz to 1000Hz and the signal will be a standard square wave (50% duty cycle). Let’s call this frequency F1 and measure it in Hz. You should measure F1 by polling the signal. For this exercise, do not use interrupts (which would be the more efficient method).
3. Measure the frequency of a second 3.3v square wave signal. The frequency of the wave signal in input will be in the range 833Hz to 1500Hz and the signal will be a standard square wave (50% duty cycle). Let’s call this frequency F2 and measure it in Hz. You should measure F2 by polling the signal. For this exercise, do not use interrupts (which would be the more efficient method).
4. Call the monitor’s method *doWork*().
5. Use a LED to indicate whether the sum of the two frequencies F1 and F2 is greater than 1500, i.e. when F1+F2 > 1500.
6. Monitor a pushbutton. Toggle the state of a second LED and call the monitor’s method *doWork()* whenever the pushbutton is pressed.

You should consider that requirements 1-5 must be repeated, and that they have hard real-time periodic deadlines.

Specifically, from the time you call the monitor’s method *startMonitoring()* method, this will assume:

* A deadline for (1) every 4ms [Rate = 250Hz]
* A deadline for (2) every 3ms [Rate = 333Hz]
* A deadline for (3) every 10ms [Rate = 100Hz]
* A deadline for (4) every 10ms [Rate = 100Hz]
* A deadline for (5) every 5ms. [Rate = 200Hz]

-1476 -7400

**First program**

Your first program needs to implement a cyclic executive (see slides and other material published on Unit 6 in Canvas). You cannot use FreeRTOS’s tasks or other FreeRTOS’s features, but you are allowed to use a single Ticker object (you need to include the Arduino library, i.e. Ticker.h) to implement the major cycle of your cyclic executive. Note that it will be difficult to satisfy RT requirements without accurate timing from a Ticker.

**Second program**

Your second program must be implemented using FreeRTOS.

You can use any of the FreeRTOS features (tasks, timers, queues, semaphores/mutexes…) you consider useful for your program.

It is your choice whether you implement tasks using actual FreeRTOS Tasks or callback functions called by FreeRTOS software timers, or a combination of the two approaches.

**Hints**

Try to design functions that you can re-use across the two implementations, do not implement the second program from scratch but work incrementally. For instance, you could start by first writing and testing standalone functions to address single requirements, before you combine them using a cyclic executive approach, and later adapting and re-using them with FreeRTOS. Alternatively, you could implement mockup functions, so that you can focus on the scheduling aspect, before replacing the mockup functions with the actual functions implementing the functional requirements for your program.

For a modular design, tasks should be implemented as independent as possible from each other, reducing dependencies between tasks at a minimum.

When you use FreeRTOS, use mutexes/semaphores to protect your program from race conditions, and semaphores and/or queues to synchronise independent tasks.

**3. Demo (Week 12)**

At the demo lab session, you will be asked

* to run both programs, using the provided monitoring library, to show that they meet all the specifications provided in this document. This will include showing the signals generated using the oscilloscope.
* Answer questions on your system, e.g. design, testing, performance.

**5. Submission (Week 13)**

Submit the following paperwork:

• Link to your code repository, with fully documented source code

• Signed authorship statement.

• Short (maximum 5 pages) report describing the design of your program (using one or more diagrams, as you consider necessary), and responding to the following questions:

◦ How did you design your cyclic executive?

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

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

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

◦ 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.

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

**(continues on the next page)**

**6. Marking criteria**

Work as an individual. Students **MUST** work on this project on their own. Please do not use code from another student or offer code to another student. Code may be run through a similarity checker. You will be asked questions about your code during the demo session and the assignment will not be marked if plagiarism is suspected.

The submission deadline is at the end of Week 13.

**The assignment contributes 30% towards the 50% continuous assessment mark (the first assignment is worth 20% and the exam is worth the remaining 50%).**

**Quality of the design, implementation and demo is worth 70**% of the mark for this assignment, with full marks **if Q&A is satisfactory** and:

• Functional requirements are satisfied. 5% points max for each requirement 1-7, for each of your programs. Important: Requirements will be also considered not satisfied if they are prone to race conditions.

• Your system is efficient and modular (easy to maintain and re-usable); up to 20% points will be removed if the code shows issues for each of these aspects. Specifically, your program should:

◦ Decouple tasks, with functions/FreeRTOS tasks as independent as possible from each other.

◦ Avoid wasting CPU (besides the active spin loops you have been

recommended to use for requirements 3 and 4) and memory (including by properly sizing stack sizes).

• You used good programming style, including using appropriate names for variables, using function return values/error codes, good layout/Indentation, and commenting style; up to 10% points will be removed if the code shows issues for each of these aspects.

• You delivered a clear, well-organised and smooth demonstration.

**Documentation is worth 30%,** with full marks for:

◦ Clear and complete answers to all the 6 questions highlighted in point #5. 5% points maximum for each answer.

◦ Well written, complete and well organised report, effective use of language and diagrams. Up to 10% points will be removed if your report does not satisfy one or more of these requirements.