Course code and name:	B31DG Shiyu Lu
Type of assessment:	**Individual**
Coursework Title:	Embedding Software
Student Name:	Shiyu Lu
Student ID Number:	H00364918

Declaration of authorship. By signing this form:

- **I declare** that the work I have submitted for individual assessment OR the work I have contributed to a group assessment, is entirely my own. I have NOT taken the ideas, writings or inventions of another person and used these as if they were my own. My submission or my contribution to a group submission is expressed in my own words. Any uses made within this work of the ideas, writings or inventions of others, or of any existing sources of information (books, journals, websites, etc.) are properly acknowledged and listed in the references and/or acknowledgements section.

- I confirm that I have read, understood and followed the University's Regulations on plagiarism as published on the University's website, and that I am aware of the penalties that I will face should I not adhere to the University Regulations.

- I confirm that I have read, understood and avoided the different types of plagiarism explained in the University guidance on Academic Integrity and Plagiarism

Student Signature *(type your name):* *Shiyu Lu*

Date: *07/04/2024*

Copy this page and insert it into your coursework file in front of your title page. For group assessment each group member must sign a separate form and all forms must be included with the group submission.

Your work will not be marked if a signed copy of this form is not included with your submission.

Task System Design

This project is based on the ESP32 development board, utilizing the FreeRTOS real-time operating system for multi-task management and execution. The system architecture is decomposed into several independent tasks by detailing the functional requirements, with each task responsible for a specific module.

FreeRTOS Tasks

- **Task 1 (Digital Signal Output)**: Responsible for periodically outputting digital signals with specific timing, demonstrating direct control over the hardware.
- **Task 2 & Task 3 (Measure Frequency)**: These tasks capture the rising edges of a square wave signal to compute and update frequency data. This data is later logged and output by Task 5. For these tasks, I use a method that calculates the time interval between two consecutive rising edges, then divides 1,000,000 by this interval to compute the frequency. This method provides accurate calculations with minimal fluctuation, resulting in more stable results.
- **Task 4 (Sample Analog Input & Control LED)**: Regularly samples analog signals and controls the brightness of an LED based on the sample values, demonstrating simple signal processing and analog signal reading.
- **Task 5 (Log Information)**: Periodically reads frequency data from a shared data structure and outputs it through a serial port to monitor the system's status and performance.
- **Task 7-1 & Task 7-2 (Monitor Push Button & Control LED on Button Press)**: Implements button input and LED response interaction through an event queue mechanism, showcasing typical inter-task communication. Additionally, a 50ms debouncing of the button press is implemented to ensure sensitive state transitions.
- **Task 8 (CPU Load Simulation)**: Simulates CPU load to assess the system's response time and processing capability for performance testing.

Hardware

- **Digital Signal Pin**: Uses GPIO 21, from which Task 1 outputs digital signals with specific timing.
- **Square Wave Input Pins (Square Wave Pin 1 & 2)**: GPIO 33 and 34 are used to capture external square wave signals, utilized respectively for frequency measurement in Task 2 and Task 3.
- **Analog Input Pin**: GPIO 26, where Task 4 samples analog signals and controls the on/off state of an LED (GPIO 15).

- **Button Pin**: GPIO 4, used by Task 7-1 to detect user button presses.
- **LED Pin (LED Pin 2)**: GPIO 2, where Task 7-2 controls the state of this LED based on button press events.

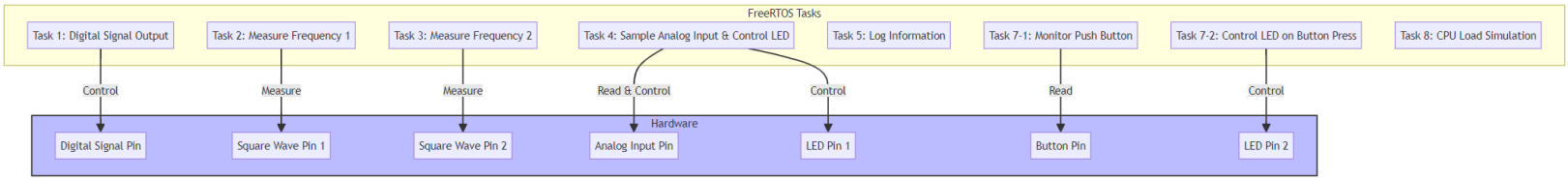

Figure 1： Show the connections between Tasks and Pins

Initialization and Configuration

At system startup, initial configurations are set through the `setup()` function, including the initialization of hardware interfaces, creation, and starting of tasks. Each task is assigned a specific priority based on its functional requirements, ensuring the responsiveness of critical tasks. After startup, FreeRTOS is responsible for the scheduling and management of tasks, achieving efficient and stable system operation through mechanisms for synchronization and mutual exclusion between tasks.

Task Scheduling and Synchronization Mechanisms

Access and Protection of Global Structures

In a multi-tasking environment, synchronizing access to global data structures is key to preventing data races and ensuring data consistency. In this project, we utilize the semaphore mechanisms provided by FreeRTOS to protect access to the frequency data structures and button event queues. **Frequency Data Structures** and **Button Press Event Queues** are crucial for sharing data between tasks, not only ensuring the safety of data transmission but also enhancing the system's flexibility and scalability. Through semaphores and mutex mechanisms, we ensure safe access to shared resources, thereby avoiding data conflicts.

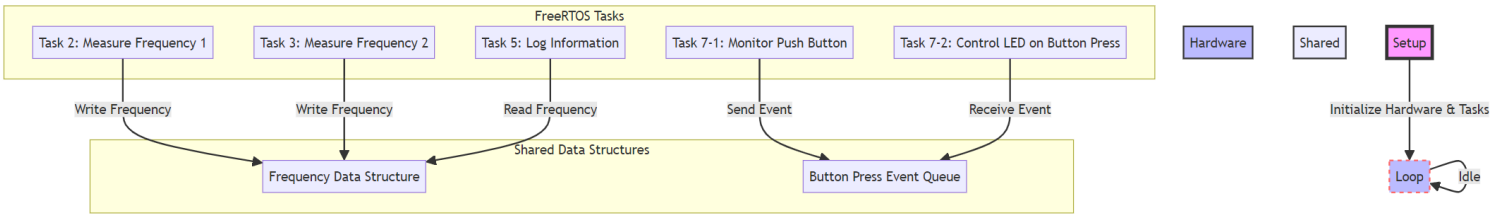

Figure 2： Share data structures and interactions between tasks

Event Triggering and Response:The interaction between Task 7-1 (Monitor Push Button) and Task 7-2 (Control LED on Button Press). When a user presses a button, Task 7-1 detects this event and sends the information to an event queue. Task 7-2 periodically checks this queue, and upon discovering a new button event, it toggles the LED's state accordingly.

Access and Synchronization of Shared Resources: Tasks 2 and 3 measure and update frequency data, which are stored in a shared frequency data structure. Task 5 (Log Information) needs to periodically read this data for recording and output. To ensure data consistency and prevent access conflicts, tasks use a mutex (Mutex) when reading and writing these shared data. They request a semaphore before accessing shared resources with `xSemaphoreTake` and release it with `xSemaphoreGive` after access is completed. This ensures that at any given moment, only one task can access the resource, thus preventing potential data race issues.

Task Priority Settings

In this project, the setting of task priorities follows these principles:

- **Task 1** and **Task 8** are assigned a priority of **1**. This reflects the fact that although these tasks are important, they do not require immediate response. Specifically, Task 1 is responsible for outputting digital signals, and Task 8 simulates CPU load; both do not respond directly to external events.

  Particularly for Task 1, if it were assigned a higher priority, the CPU would always prioritize it, leading to its long occupation of the process and causing other tasks to fail to execute normally. This also lead to watchdog issues, causing the system to restart periodically.

- **Task 2**, **Task 3**, **Task 4**, **Task 5**, **Task 7-1**, and **Task 7-2** are set with a priority of **2,** indicating that they deal with functions related to system responsiveness, requiring faster execution to ensure the system's real-time performance and user experience.

By setting priorities in this way, we ensure that functions such as frequency measurement, user input monitoring, LED control, and information logging are prioritized, while maintaining the system's flexibility and responsiveness.

Task Stack Sizing

In demonstration of Week12, We define stack size as below:

Task 1， Task 2 and **Task 3** are allocated **4096 bytes** of stack space; **Task 4 , Task 5** , **Task 7-1**, and **Task 7-2** are set with a stack size of **2048 bytes**; **Task 8** is allocated the smallest stack size of **1024 byte.**

The stack sizes above were determined through testing and experimentation. However, to ensure flawless functionality and real-time performance during our Week 12 classroom demonstration, we have been generous with the allocation of stack space. For example, assigning only 1024 bytes of memory to Tasks 7-1 and 7-2 resulted in program errors due to insufficient memory support for task execution. We can allocate 1800 bytes to make it work, but in order to ensure a smooth demonstration, we allocated 2048 bytes. Task 1、2、3 is even more allocated.

Therefore, If we want to find the most appropriate stack size, we can utilize the **`uxTaskGetStackHighWaterMark`** within our code to monitor the minimum stack space remaining during task execution. Based on our tests, the current stack allocations are as follows: Tasks 1, 2, and 3 have an actual **minimum remaining stack space of 3176 bytes**, indicating that the actual stack usage for these tasks is far below the allocated amount, allowing for potential reduction in allocated stack space. Task 4 has a minimum remaining of **1116 bytes;** Task 5**, 1268 bytes;** Task 7-1**, 1116 bytes;** Task 7-2**, 1364 bytes; and** Task 8**, 100 bytes.**

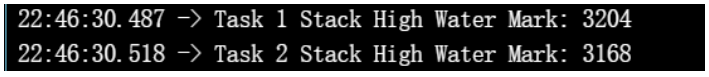

Figure 3 : Using uxTaskGetStackHighWaterMark to test the stack space remain

However, the stack sizes must be determined through testing and experimentation. Theoretical remaining figures may not fully support task execution. Continuous experimentation and testing are required to establish a stack size allocation that balances performance with efficient memory usage.

Testing and Comparison

Testing Verification and Real-Time Performance Evaluation

To verify the system's performance in meeting real-time requirements, we conducted the following two main tests: system performance under high load conditions and measurement of the response time from button press to LED reaction.

High Load Environment Test

By triggering **all tasks** simultaneously, where Task 1 outputs waveforms, Tasks 3 and 4 test frequencies and are read by Task 5, while simulating signal control of an LED's state and pressing a button to toggle another LED's state, **the system performed smoothly without any issues.**

This demonstrates the system's robust stability under high load conditions**, as showcased in Week 12 lab demonstration.**

Button Response Time Test

Measured the response time from the button press (Task 7-1) to the LED state change (Task 7-2). High precision timestamp recording was used to measure response times. A total of 50 recordings were made, calculating the average response time and identifying the longest response time from these instances.

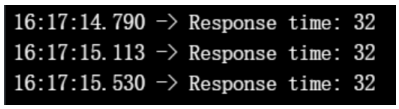

Figure4： Test output of button response time

The average response time across these fifty tests was **35.14 milliseconds**, and the maximum response time was **56 milliseconds**. I think this is a good result.

Comparison between FreeRTOS and Cyclic Executive Approaches

This project utilizes FreeRTOS, demonstrating several advantages over the traditional cyclic executive approach:

- **Task Scheduling and Priority Management**: FreeRTOS allows setting different priorities for each task, ensuring that critical tasks (such as button event handling) can respond quickly. In the cyclic executive approach, all tasks execute sequentially as per the code order, which makes it difficult to flexibly manage priorities among multiple tasks.
- **Real-Time Performance**: Through preemptive scheduling and time slicing (based on task priorities), FreeRTOS ensures high real-time performance, responding to critical events promptly. The cyclic executive approach lacks preemptive mechanisms, which may lead to response delays and is unsuitable for high real-time requirement scenarios.
- **System Resource Management and Synchronization**: FreeRTOS efficiently manages shared resources and task synchronization through semaphores, message queues, and other mechanisms, preventing resource conflicts and data inconsistencies. In the cyclic executive approach, developers need to manually handle these issues, which becomes increasingly difficult as project complexity increases.

Source Code Link: LSY-Andy/B31DG-H00364918-Assignment2 (github.com)

This github repository has always been private, and I will make it public for teachers to access after homework is due