B31DG: Embedded Software 2024 Assignment 2

Introduction

This report aims to provide a detailed description of the design and implementation of an embedded system project based on FreeRTOS. The core objective of the project is to develop a system capable of handling multiple tasks simultaneously, including the generation and measurement of digital signals, sampling of analog inputs, control of LEDs, and simulation of CPU workload.

System Design Description

The system design for this project is based on FreeRTOS, an open-source real-time operating system suitable for embedded devices.

Core Components and Tasks:

1. **Digital Signal Task**:

   This task is responsible for periodically generating a digital signal sequence. The signal cycle includes precise control of high and low levels, ensuring that the output meets the preset timing requirements.

2. **Frequency Measurement Tasks**:

   The system includes two frequency measurement tasks, each corresponding to a different analog input. Each task calculates the frequency of the input signal by detecting rising edges, which are used for subsequent processing and analysis.

3. **Analog Input Sampling Task**:

   This task regularly reads data from an analog input and calculates the average of the last ten readings. If the average exceeds a set threshold, the system provides visual feedback by lighting an LED.

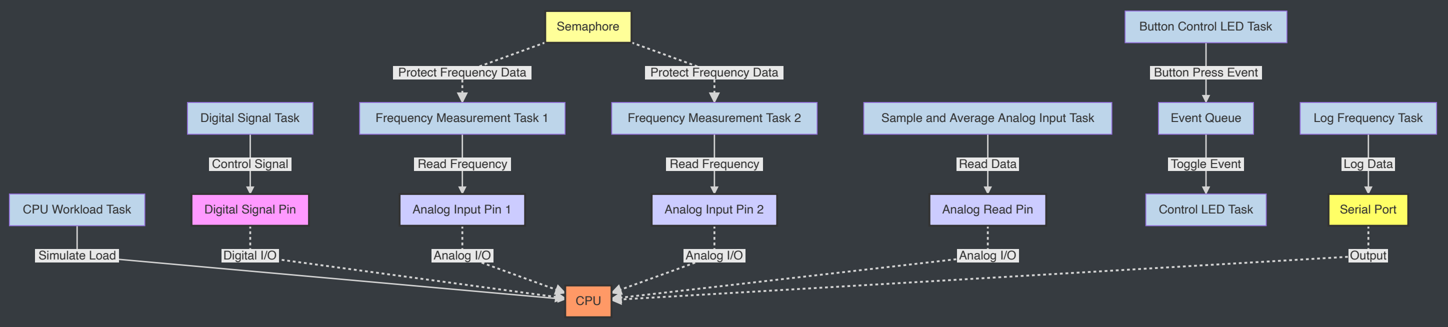
4. **Button-Controlled LED Task**:

   This task monitors the status of a physical button. When a button press is detected, it triggers a toggle of the LED state through the event queue, facilitating user interaction.

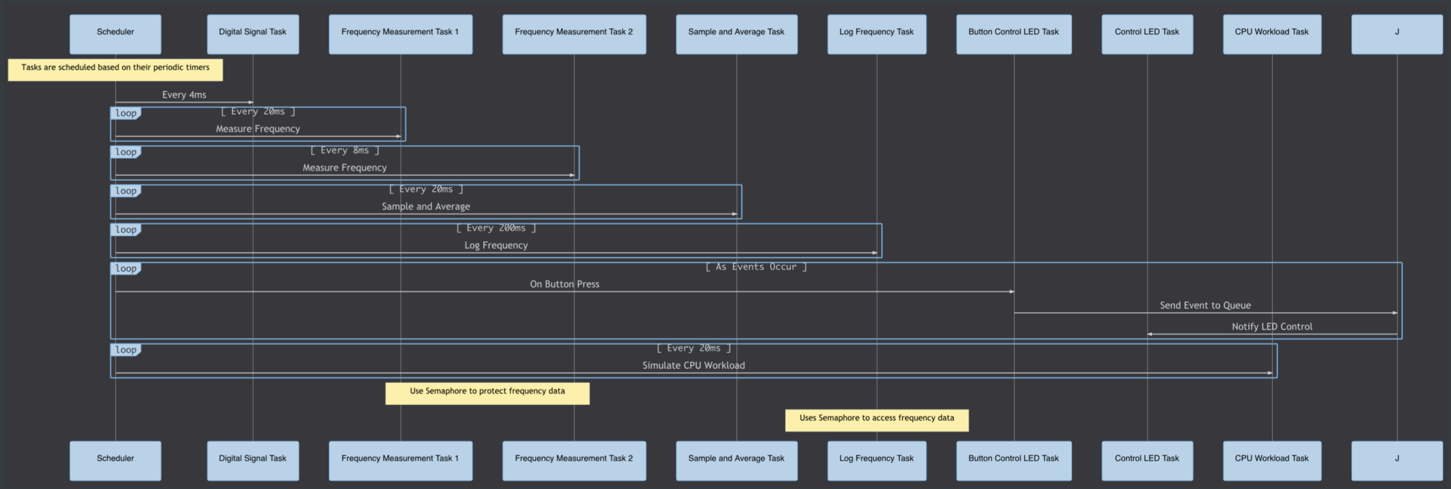
5. **CPU Workload Simulation Task**:

   To simulate CPU load, this task executes a compute-intensive loop, consuming a significant amount of processing time. This is useful for testing the system's performance under load conditions.

- **System Architecture Diagram**: Displays the modules of the system and their interactions.



Task Scheduling and Synchronization Diagram: Describes how tasks are scheduled and synchronized with each other.



Task Prioritization

High Priority:

- **Frequency Measurement Tasks**: These tasks (`taskMeasureFrequency1` and `taskMeasureFrequency2`) need to capture signal changes in real-time, therefore they are given a higher priority. Accurate and rapid frequency measurement is crucial for the overall performance and responsiveness of the system.
- **Button Control LED Task**: Responsiveness to user interactions is extremely important for user experience, thus this task also requires a higher priority to ensure low latency.

Medium Priority:

- **Analog Input Sampling and Averaging Task**: Although this task's real-time requirements are not as critical as the frequency measurement tasks, it needs regular updates to ensure the system can correctly respond to external changes.

Low Priority:

- **Logging Task**: This task has the lowest priority because its real-time requirements are minimal. Logging should not interfere with more important tasks.

Stack Size Determination

1. **Initial Setup**: Initially, each task was allocated an estimated stack size based on the complexity of the task and empirical values.
2. **Operational Observation**:

   Utilize FreeRTOS's stack monitoring features, such as the `uxTaskGetStackHighWaterMark` function, to monitor the maximum stack usage of each task during operation.

   This function returns the minimum free stack space reached by the task stack pointer since the task was created (in words). This value can be used to assess stack usage and remaining space.
3. **Adjustments and Optimization**:

   Increase Stack Size: If it is observed that a task's stack remains minimally unused, indicating that the current stack space is nearly insufficient, the stack size should be appropriately increased.

   Decrease Stack Size: If a task's stack usage rate is very low, indicating an oversized stack configuration, it can be appropriately reduced to save valuable system memory.

   Continuous Monitoring: As system operating conditions may change over time (such as code updates, feature additions, etc.), it is advisable to periodically repeat the above steps to ensure that stack configurations always adapt to the current system requirements.

Synchronization Mechanisms Used

Semaphores are the main mechanisms used in this project to synchronize and protect access to global resources. Semaphores are an advanced synchronization method that can be used not only for mutual exclusion access (mutex semaphores) but also to control the use of limited resources (counting semaphores).

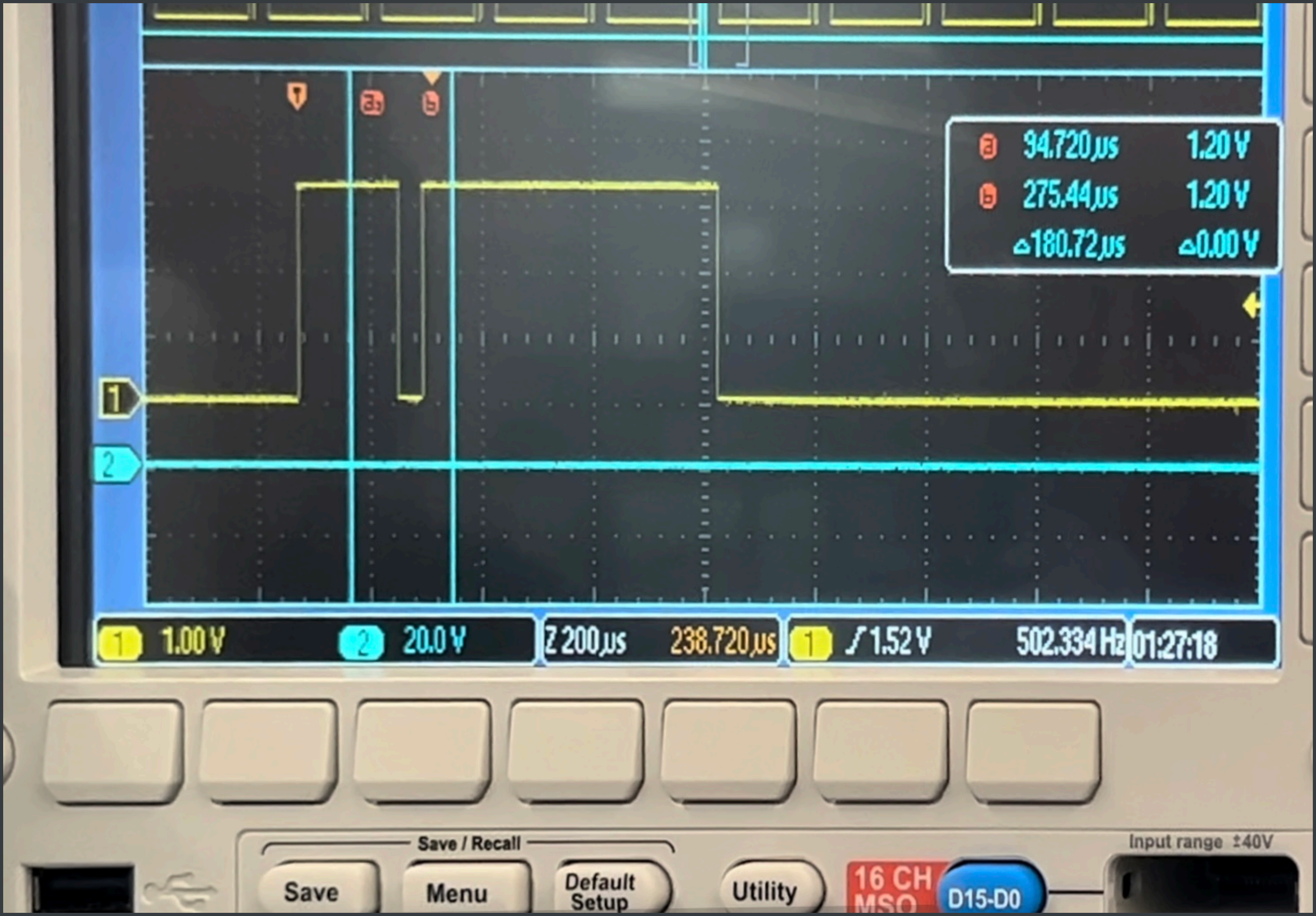
Specific Measures

1. Before a task begins processing data, it attempts to acquire the semaphore with `xSemaphoreTake`.
2. Once the semaphore is obtained, the task can safely read or modify the global data.
3. After data processing is complete, the semaphore is released with `xSemaphoreGive`, allowing other tasks to access the data.

Testing Process and Results

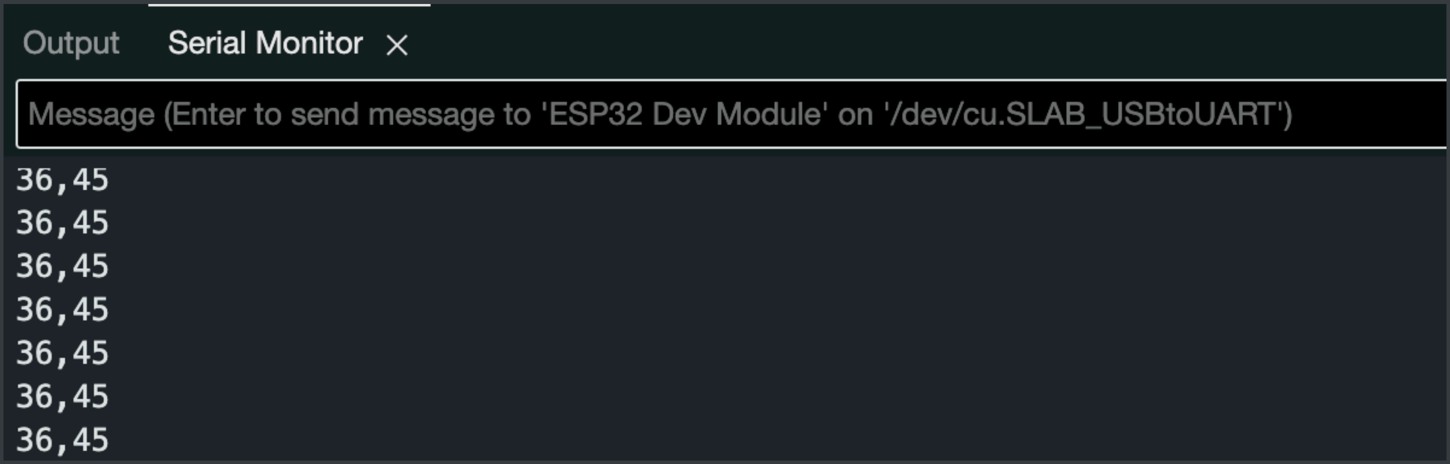
Task 1: Digital Signal Output Test

To verify the performance of the digital signal output task, we used a logic analyzer to measure the high and low states of the digital signal pin and confirmed that the signal output met design specifications

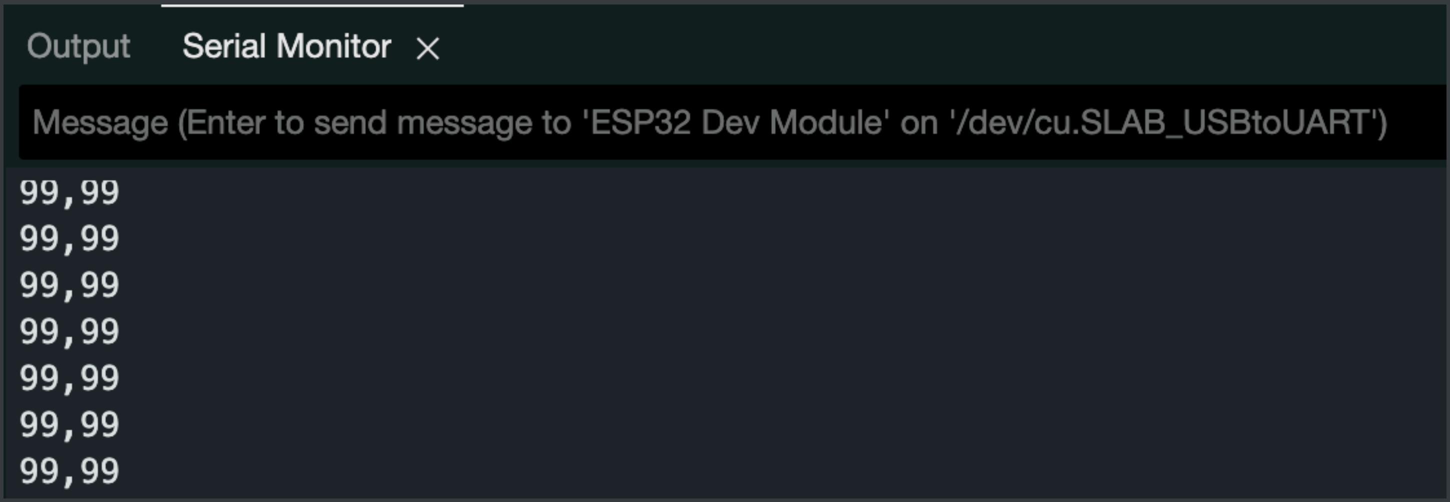


Tasks 2, 3, and 5: Frequency Measurement Testing

To ensure the accuracy and timeliness of frequency measurements, we conducted tests on Tasks 2 and 3. Task 2 aimed to measure the frequency every 20ms, with a range from 333Hz to 1000Hz, while Task 3 performed measurements every 8ms, covering a frequency range from 500Hz to 1000Hz. And eventually frequencies would be scaled and bounded between 0 to 99


Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on '/dev/cu.SLAB_USBtoUART')

```
36,45
36,45
36,45
36,45
36,45
36,45
36,45
```


Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on '/dev/cu.SLAB_USBtoUART')

```
99,99
99,99
99,99
99,99
99,99
99,99
99,99
```

Task 4: Analog Signal Sampling and Averaging Test

We tested the analog signal sampling task, which samples an analog input every 20ms and computes the average of the last 10 readings. To test its performance, we used a variable resistor to simulate different analog inputs, monitoring whether the output voltage and computed average accurately reflected input changes. Test results indicated that when the analog input exceeded a certain threshold (half of the maximum ADC reading), the corresponding LED was successfully illuminated, meeting the design requirements.

Task 7: Button-Controlled LED Toggle Test

With debouncing implemented, each button press consistently triggered a state toggle in the LED without any false triggers or delayed responses.

Button Response Time Analysis

To analyze the worst-case delay from button press to LED toggle, we need to consider the following factors:

1. **Debounce Delay**: The debounce time for the button is set to 50 milliseconds. This means that from detecting a change in button state to confirming the state is stable and processing the event, it could take up to 50 milliseconds.
2. **Task Scheduling Delay**: `buttonTask1` checks the button state every 10 milliseconds (implemented via `vTaskDelay(pdMS_TO_TICKS(10))`). This means the time from when the button event occurs to when it is captured by `buttonTask1` can be up to 10 milliseconds in the worst case.
3. **Event Queue Transmission Time**: Sending an event from `buttonTask1` to the event queue is almost instantaneous, but depending on the system load and scheduling policy, there could be a slight delay in processing the event by `controlLedTask` . Typically, this delay is minimal, but we conservatively estimate a few milliseconds.
4. **LED Control Response**: `controlLedTask` toggles the LED state immediately upon receiving the event, and this part of the delay is negligible.

In conclusion, the total worst-case delay amounts to: 65ms

Choosing FreeRTOS

Task Complexity and Diversity: The system involves a variety of tasks including real-time data handling, user interaction, and background processing. FreeRTOS's multitasking capabilities effectively manage these complexities.

Development Efficiency and Future Expansion: The rich features and community support of FreeRTOS can accelerate the development process and support future potential extensions.

Reliability: The maturity and stability of FreeRTOS minimize the risk of system failures during operation, enhancing the system's reliability.