Lab 4 Report

Authors:

Yehoshua Luna (2322458)
Carter Lee (2478429)

Date: 11/19/2025

EE 474 Autumn 2025

Table of Contents

Introduction

The purpose of this lab is to further explore various scheduling techniques and approaches using FreeRTOS with the Espressif platform. More specifically, we used preemptive scheduling algorithms, parallel computing with the dual-core ESP32-S3, and data synchronization. This is not a trivial task by any means, and the experience gained here will benefit all participants in future embedded systems projects. Our lab objectives include:

- Understanding and implementing preemptive scheduling algorithms.
- Exploring the dual-core architecture of the ESP32-S3 for parallel computing, while learning to assign and manage tasks on different cores.
- Integrating scheduling and parallel computing techniques using FreeRTOS.
- Managing shared memory between cores using semaphores.

System Design and Implementation

Part I

My system focused on implementing a Shortest Remaining Time First Scheduling (SRTFS) algorithm using FreeRTOS and the ESP32-S3. The system utilizes FreeRTOS to run four concurrent tasks: LED control (ledTask), counter display (counterTask), character output (alphabetTask), and a basic scheduler (scheduleTasks). Each task is pinned to a processor core and given its own execution stack.

Each task was assigned a specific execution time and the execution times were tracked using dedicated TickType_t variables for control over task operation timing and synchronization. Tasks decrement their "remaining time" with each time slice, and reset it when their period is finished. This was written so the SRTFS algorithm could be used. However all 3 tasks were run at the same time and different delays were added to the end of them to run the tasks similarly to the SRTFS algorithm so that some tasks could run more frequently than others.

The LCD was controlled via I2C using low-level functions (i2cW, wr) for efficient four-bit data transfer. The lcdPrint routine handled both single and double-line messages, ensuring output stays within hardware constraints. The tasks never block each other unintentionally, FreeRTOS ensures that all critical operations (display update, LED toggle, serial print) happen as scheduled.

Part II

This system is designed around the ESP32-S3's dual-core architecture, using FreeRTOS tasks to manage our four jobs according to their priority. We pinned each task to a specific core, with LCD updates and photoresistor readings running on Core 0, and anomaly detection and prime

number calculations happening on Core 1. This separation allows more important operations, like display refresh and anomaly detection, to preempt less critical tasks. More computationally heavy functions, such as prime number generation, are given a lower priority. The idea here is to balance responsiveness with parallel execution, taking advantage of our board's hardware capabilities.

Synchronization between tasks is achieved using a binary semaphore, which protects shared variables. By requiring each task to acquire the semaphore before accessing or modifying these values, our system avoids race conditions and ensures consistent data across tasks. The light sensor task maintains a circular buffer of five readings and computes a simple moving average, which smooths out environmental noise. The LCD task conditionally updates the display when our light readings change, and the anomaly detection task checks thresholds to decide whether or not to blink the LED. The prime calculator runs in the background with the lowest priority.

Overall, our system design for this section is very robust. We are effectively managing shared data and maximizing the usefulness of each core cycle using FreeRTOS.

Code Structure

Part I

The code is organized around several different types of modules, peripheral interface modules, task modules, time and execution management modules, and system setup and initialization modules. The i2cW, wr, lcdInit, and lcdPrint functions were dedicated to handling the low-level communication and initialization for the 16×2 LCD over I2C for the LCD task, enabling the task to update the display by sending straightforward string data.

The function ledTask defined the behavior for blinking a LED on a scheduled interval, tracking its own periodic execution time and toggling the LED state on and off. The function counterTask updated the LCD display with a numeric counter, cycling from 1 to 20. It displayed the current number and restricted the bounds of the range. The function alphabetTask sent the next alphabet character to the serial monitor, looping continuously through A–Z. The function scheduleTasks ran as an additional system task to monitor and compare remaining execution times of other tasks. While not strictly a preemptive scheduler, it provided a basic timing overview and could be utilized to prioritize or coordinate task execution.

The global variables (ledTaskExecutionTime) allowed each task to manage its own timing without interfering with others. Time slices and reset logic inside each task helped maintain the task execution and served as round-robin scheduling. The setup() function initialized the hardware, configured the serial port, set up I2C for LCD communication, and started all FreeRTOS tasks.

Part II

Our code is structured in three distinct sections. The first section is simply library imports, macro definitions, global variable declarations, and LCD object instantiation using the very handy liquid crystal library. Most notably, we include the FreeRTOS library and our semaphore handle in this section.

Next, our second section of code contains four task functions that perform the core operations of our program. vTaskLight() handles photoresistor readings and moving average calculations. vTaskLCD() manages conditional display updates using the LiquidCrystal_I2C library when our ambient light level reading changes. vTaskAnomaly() monitors light level thresholds and drives the LED in a three-blink pattern when an anomaly is detected. Finally, vTaskPrime() performs computational work by printing prime numbers, representing a resource-intensive background process. Each task is written as an infinite loop with delays, following FreeRTOS conventions, and uses the semaphore to safely access shared data. This modular task-based structure makes the code easier to maintain and ensures that each function has a distinct purpose.

Finally, the setup() section of our program initializes peripherals, configures pins, creates the semaphore, and launches the tasks pinned to specific cores with assigned priorities. Our loop() function is left empty because all execution is handled by FreeRTOS. This makes the program very readable and scalable.

Discussion

We decided to split the lab work evenly. Carter would take **Part I**, and Yehoshua would take **Part II**. Plans were also made to work together at least once per week.

Carter had some difficulty implementing the Shortest Remaining Time First Scheduling algorithm while writing out the tasks themselves was relatively straightforward. There was confusion on if implementing the SRTFS algorithm was possible and then how to best simulate it. It was simulated by adding different delays to the end of each task so the tasks would be run at different rates, but in the future if attempting to implement the algorithm again each task would be run in order of shortest execution time and then after a task was completed it would be suspended and only reset after all tasks were completed.

Yehoshua found that **Part II** went quite smoothly overall. The biggest challenge was giving each task an acceptable amount of delay. For example, it can be hard to read the LCD if it is updating more than 4 times per second, so the minimum delay between LCD updates was set to 250ms. Likewise, reading ambient light values any more than 10 times per second seemed extreme, so a delay time of 100ms was used. However, not all tasks needed delay. The prime finding task was given the lowest priority and could therefore run with no delay while not interfering with the other tasks on its core.

Division of Work

Carter was responsible for **Part I**, which accounted for 40% of the work. Yehoshua did **part II**, which was another 40%. Both people wrote the lab report together, which was the remaining 20% of work. Yehoshua also deoxygenated the code before submission.

Suggestions

The procedure handout for this lab did not elaborate its requirements very clearly, especially concerning **Part I**, which ended up causing us grief later on. Our primary suggestion would be fixing this oversight and perhaps organizing lab requirements in a better way. FreeRTOS simply was not designed to be a SRTF scheduler, no matter the approach. Yehoshua wishes that there were more tasks to manage for **Part II**; maybe six instead of four? Carter would definitely like the handout to be more clear overall.

Conclusions

This lab successfully demonstrates how FreeRTOS can be used to do preemptive scheduling, parallel task execution, and safe data synchronization through semaphores. By structuring our system using modular tasks with clear functions across both cores, we were able to balance the responsiveness, efficiency, and computational workload of our program. The experience gained provided valuable insight into real-world embedded systems challenges, and the skills that we used here will serve as a strong foundation for more advanced projects in both academic and professional contexts.

References

[1] Espressif Systems, "ESP32-S3 Technical Reference Manual," Espressif Systems, 2025. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/. Accessed: Nov. 18, 2025.

Total Number of hours Spent

Carter: 5
Yehoshua: 8
Total: 13