ECE 478/CIS 450 : Operating System

Final Project Report

Project Title: Enhanced Lighting Control with Voice Feedback Using ESP32-C3-LCDKit

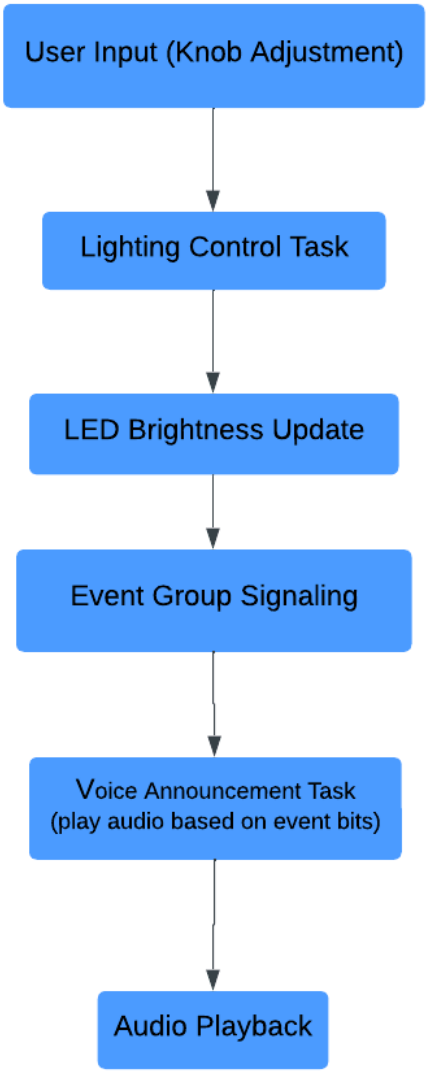
Prof: Probir Roy

Team Members: Freshta Noori, Zahraa Alhmood, Leia Sayed

Project Overview

This project demonstrates the use of the ESP32-C3-LCDKit microcontroller to test and process input from a rotary position sensor (knob). This was built upon the knob example provided. The system was developed using the ESP-IDF 5.3 framework, emphasizing key operating systems concepts such as concurrency control, task scheduling, and resource management. The goal was to improve user interaction by adding voice announcements that correspond to LED brightness levels, while maintaining system responsiveness. This report outlines the technical design, implementation details, and user guidelines for the project.

System Architecture



The system integrates hardware and software components to achieve seamless interaction between the rotary position sensor, lighting control, and user feedback. The architecture is structured around four

primary tasks: knob input processing to adjust brightness levels, lighting control task handles the brightness adjustments, the event group enables task communication, and the voice feedback

The user interacts with the system using a rotary knob, which allows for brightness adjustment of an LED panel. The knob inputs are monitored by the lighting control task, which adjusts the brightness accordingly and updates the system state. The lighting control task communicates brightness changes to the voice announcement task via an event group. The voice announcement task is responsible for playing audio feedback, stored in SPIFFS, that corresponds to the brightness level. The user input through the knob drives the core functionality. The lighting control task interprets these inputs and modifies the brightness of the LED panel. The event group acts as a signaling mechanism, encoding brightness level changes as specific event bits. These bits are monitored by the voice announcement task, which retrieves the appropriate audio file and plays it through the audio output.

```
uint32_t key = lv_event_get_key(e);
if (is_time_out(&time_500ms)) {
    if (LV_KEY_RIGHT == key) {
        if (light_set_conf.light_pwm < 100) {
            light_set_conf.light_pwm += 25;
        }
    } else if (LV_KEY_LEFT == key) {
        if (light_set_conf.light_pwm > 0) {
            light_set_conf.light_pwm -= 25;
        }
    }
}
```

The knob is the primary user interface for controlling LED brightness. This code (not something we modified, but is useful to add here for context) determines the direction of the knob's rotation and updates the brightness level accordingly. The **light_set_conf.light_pwm** variable holds the current brightness configuration and changes in this variable triggers downstream components.

```
if (xSemaphoreTake(light_config_mutex, portMAX_DELAY)) {
    switch (light_set_conf.light_pwm) {
        case 0:
            xEventGroupSetBits(lighting_event_group, BIT0);
            break;
        case 25:
            xEventGroupSetBits(lighting_event_group, BIT1);
            break;
        case 50:
            xEventGroupSetBits(lighting_event_group, BIT2);
            break;
        case 75:
            xEventGroupSetBits(lighting_event_group, BIT3);
            break;
        case 100:
            xEventGroupSetBits(lighting_event_group, BIT4);
            break;
        default:
            ESP_LOGW("Light", "Unsupported lighting level: %d", light_set_conf.light_pwm);
    }
    xSemaphoreGive(light_config_mutex);
}
```

The lighting control task is responsible for adjusting the brightness levels of the LED panel based on knob input and signaling changes to the voice announcement task via event groups. It first acquires a mutex before reading or writing shared variables to prevent race conditions to **light_set_conf**. It then sets the appropriate event bit in the **lighting_event_group** to signal the voice announcement task. The event bits correspond to different light levels and are used to trigger audio announcements. For example, BIT0 is set for brightness level 0%, BIT1 is set when brightness level is 25%, and so on.

```
esp_err_t audio_play_start()
{

    //EDIT: Create the event group
    if (lighting_event_group == NULL) {
        lighting_event_group = xEventGroupCreate();
        if (lighting_event_group == NULL) {
            ESP_LOGE(TAG, "Failed to create event group");
            return ESP_FAIL;
        }
    }

    //EDIT: Create the voice announcement task
    xTaskCreate(voice_announcement_task, "VoiceTask", 2048, NULL, 5, NULL);
```

The **lighting_event_group** is an event group that is initialized in *app_audio.c*, during system setup. Its creation is the mechanism for signaling between the two tasks. Once that's initialized, we create the new task **voice_announcement_task**.

The voice announcement task listens for event bits set by the lighting control task. When a bit is detected, it plays the corresponding audio file using the **audio_handle_info** function.

```
//EDIT:  Create voice_announcement_task to wait for events
static void voice_announcement_task(void *param) {
    while (1) {
        // Wait for any sound event bits
        EventBits_t bits = xEventGroupWaitBits(
            lighting_event_group,
            (BIT0 | BIT1 | BIT2 | BIT3 | BIT4), // for light levels 0%, 25%, 50%, 75%, 100%
            pdTRUE, // Clear bits on exit
            pdFALSE, // Wait for any bit
            portMAX_DELAY
        );

        if (bits & BIT0) audio_handle_info(SOUND_TYPE_LIGHT_OFF);
        if (bits & BIT1) audio_handle_info(SOUND_TYPE_LIGHT_LEVEL_25);
        if (bits & BIT2) audio_handle_info(SOUND_TYPE_LIGHT_LEVEL_50);
        if (bits & BIT3) audio_handle_info(SOUND_TYPE_LIGHT_LEVEL_75);
        if (bits & BIT4) audio_handle_info(SOUND_TYPE_LIGHT_LEVEL_100);
    }
}
```

The task waits indefinitely for an event bit to be set. Once a bit is detected, it triggers the playback of an audio file corresponding to the brightness level, providing real-time voice feedback.Audio playback retrieves the required file from SPIFFS storage and plays it using the ESP32 audio driver.

```
    //EDIT: Added sound for light control
    case SOUND_TYPE_LIGHT_OFF:
        sprintf(filepath, "%s/%s", CONFIG_BSP_SPIFFS_MOUNT_POINT, "light_off.mp3");
        break;
    case SOUND_TYPE_LIGHT_LEVEL_25:
        sprintf(filepath, "%s/%s", CONFIG_BSP_SPIFFS_MOUNT_POINT, "light_25.mp3");
        break;
    case SOUND_TYPE_LIGHT_LEVEL_50:
        sprintf(filepath, "%s/%s", CONFIG_BSP_SPIFFS_MOUNT_POINT, "light_50.mp3");
        break;
    case SOUND_TYPE_LIGHT_LEVEL_75:
        sprintf(filepath, "%s/%s", CONFIG_BSP_SPIFFS_MOUNT_POINT, "light_75.mp3");
        break;
    case SOUND_TYPE_LIGHT_LEVEL_100:
        sprintf(filepath, "%s/%s", CONFIG_BSP_SPIFFS_MOUNT_POINT, "light_100.mp3");
        break;
    }
```

The audio_handle_info function determines the appropriate file path based on the event type. For example, if we have triggered BIT1, which sets the audio type to SOUND_TYPE_LIGHT_LEVEL_25, then the system will play its corresponding light_25.mp3 file. The same convention applies to the rest of the lighting levels.

Concurrency Control Explanation

Concurrency is managed using FreeRTOS primitives, including mutexes and event groups, to manage the shared resources between tasks and ensure data integrity.

A **mutex** is used to lock the shared variable whenever a task is reading or updating the sensor data. This ensures that only one task can access the variable at a time, preventing race conditions and potential data corruption.Mutexes are used to protect shared resources, such as the **light_set_conf** structure in our code, which stores the current brightness configuration. A mutex ensures that only one task can access or modify this structure at a time, preventing race conditions. The mutex is created during the initialization of the lighting layer. When the lighting control task needs to update the brightness configuration, it takes the mutex using **xSemaphoreTake** and locks other tasks from accessing the data that is being used when the mutex is taken.. After completing the update, the task releases the mutex with **xSemaphoreGive.**

```
void ui_light_2color_init(lv_obj_t *parent)
{
    //EDIT: Create mutex for light configuration
    if (light_config_mutex == NULL) {
        light_config_mutex = xSemaphoreCreateMutex();
        if (light_config_mutex == NULL) {
            ESP_LOGE("Light", "Failed to create light_config_mutex");
            abort();
        }
    }
```

This mutex is initialized during system setup under the **ui_light_2color.c** file. If initialization fails, the system logs an error and stops execution to prevent undefined behavior.

Event groups serve as a signaling mechanism between the lighting control task and the voice announcement task. The lighting control task sets event bits in the event group to indicate changes in brightness. Each brightness level corresponds to a unique event bit. The voice announcement task waits for these event bits using **xEventGroupWaitBits**. When a bit is detected, the task clears the bit and plays the associated audio file.

```
// Wait for any sound event bits
EventBits_t bits = xEventGroupWaitBits(
    lighting_event_group,
    (BIT0 | BIT1 | BIT2 | BIT3 | BIT4), // for light levels 0%, 25%, 50%, 75%, 100%
    pdTRUE, // Clear bits on exit
    pdFALSE, // Wait for any bit
    portMAX_DELAY
);
```

```
switch (light_set_conf.light_pwm) {
    case 0:
        xEventGroupSetBits(lighting_event_group, BIT0);
        break;
    case 25:
        xEventGroupSetBits(lighting_event_group, BIT1);
        break;
    case 50:
        xEventGroupSetBits(lighting_event_group, BIT2);
        break;
    case 75:
        xEventGroupSetBits(lighting_event_group, BIT3);
        break;
    case 100:
        xEventGroupSetBits(lighting_event_group, BIT4);
        break;
```

The lighting control task signals brightness changes by setting event bits, and the voice announcement task responds by waiting for these bits.

The combination of mutexes and event groups ensures that the system maintains integrity while supporting concurrent operations. The lighting control task focuses on real-time brightness adjustments, while the voice announcement task handles audio playback.

User Guide

Setting Up the System: To operate the system, users must first set up the ESP-IDF development environment (version 5.3) on their computer. This includes configuring environment variables and establishing a connection to the ESP32-C3-LCDKit microcontroller. After setting up the development environment, users can navigate to the project directory and compile the program using the `idf.py build` command. The program is then flashed to the microcontroller using `idf.py flash`.

Using the Rotary Knob and understanding feedback: Once the system is running, users can rotate the knob to navigate to the lighting functionality and adjust the lighting brightness. Turning the knob clockwise increases the brightness, while rotating it counterclockwise decreases it. The brightness levels are updated in real-time and displayed on the LED panel. Simultaneously, a voice announcement informs you of the current brightness level. For example, setting the brightness to 50% triggers the announcement "Light brightness is 50%.

Troubleshooting: If the system fails to function as expected, users should verify the following:

1. The ESP32-C3-LCDKit is correctly connected to the computer.
2. The correct COM port is selected during flashing.
3. Environment variables for ESP-IDF 5.3 are properly configured. If issues persist, recompile the program and flash it again to the microcontroller.

Conclusion

The successful integration of hardware and software in this project highlights the practical application of operating systems concepts. By leveraging the ESP-IDF 5.3 framework, the project demonstrates effective task scheduling, concurrency control, and real-time system interaction. The enhanced knob panel system provides users with an intuitive interface for lighting control, supported by voice feedback and real-time data visualization.

Github Repository

Demo Video