CSE 121: Lab 4 Report

By Daniel Beltran

11/14/2023

Lab 4: Part 1

The first part of the lab required us to print onto the terminal the direction of the ESP32C3 board depending on its inclination (e.g. UP/DOWN, LEFT/RIGHT, UP RIGHT/ UP LEFT, and DOWN RIGHT / DOWN LEFT). As it was the first time working with board movement, using ChatGPT 3.5 for the conceptual overview along with reviewing the ICM-42670-P datasheet for specific addresses. Simply entered the following search query from ChatGPT:

Me: The overall objective of this task is to use the ICM-42670-P on the RUST ESP32C3 board as a Bluetooth mouse. Each time the board tilts, it should move the mouse in the X/Y direction accordingly. Write a program that prints out the board inclination for the ESP32C chip.

Here's a conceptual overview of the steps involved:

1. **Initialize Bluetooth:**
   Set up Bluetooth on the ESP32C3 to function as a HID device (Human Interface Device) for a mouse.
2. **Initialize ICM-42670-P Sensor:**
   Configure and initialize the ICM-42670-P sensor using the I2C protocol to read the accelerometer data.
3. **Data Processing:**
   Read the sensor data and process it to determine the board's inclination.
4. **Movement Simulation:**
   Map the inclination data to mouse movement directions.

As for now, bluetooth isn't necessarily useful in this part of the lab while the other three points provide essentially the next steps. Instead of using the accelerometer, the gyroscope was utilized for configuration, along with PWR_MGTM0. From this, I was able to review and define the different registers from the datasheet:

11	17	GYRO _DATA_X1	R	GYRO _DATA_X[15:8]
12	18	GYRO _DATA_X0	R	GYRO _DATA_X[7:0]
13	19	GYRO _DATA_Y1	R	GYRO_DATA_Y[15:8]
14	20	GYRO _DATA_Y0	R	GYRO_DATA_Y[7:0]

For this program, defining the X/Y registers for the gyroscope data will be utilized passing in some read function to process the data collected to indicate board inclination. Another valuable register needed is the ICM address (0x68) which we will need to pass into our I2C function calls for reading and writing. From here, the creation of a couple functions will be necessary before configuring the gyroscope and pwr_mgmt. The functions in question are the standard initializing the I2C from the previous labs along with a function reading in a register value (e.g. the defined gyroscope data) and another that writes a value with the previous data still existing.

```
void i2c_master_init(){
        i2c_config_t config = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = I2C_MASTER_SDA_IO,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_io_num = I2C_MASTER_SCL_IO,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = I2C_MASTER_FREQ_HZ,
        .clk_flags = 0,
    };

    i2c_param_config(I2C_MASTER_NUM, &config);
    i2c_driver_install(I2C_MASTER_NUM, config.mode, 0, 0, 0);
}
```

The image above shows the configuration of I2C. The following will correspond to the **readSensor()** function that reads incoming register data and follows the I2C sequence for reading/writing. Since we are passing in the gyroscope X/Y data values, this function requires one parameter to take in that value and having an unsigned 8-bit data value to store after reading what was written. This has similar formattings to previous I2C sequences and included the ICM address to configure that sensor on the chip. After writing the register value and storing it in some variable, the read value is thereby returned.

```
uint8_t readSensor(uint8_t writeVal){
    // code
    uint8_t accelData;
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (ICM_ADDR << 1) | I2C_MASTER_WRITE, true);
    i2c_master_write_byte(cmd, writeVal, true);

    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (ICM_ADDR << 1) | I2C_MASTER_READ, true);
    i2c_master_read_byte(cmd, &accelData, I2C_MASTER_NACK);

    i2c_master_stop(cmd);
    i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, pdMS_TO_TICKS(1000));
    i2c_cmd_link_delete(cmd);

    return accelData;
}
```

Essentially, the function's task is to properly read register values and configure them onto the ICM sensor for use. However, configuring the gyroscope and pwr_mgmt is essential and will be needed when calling on **app_main()**. For this part of the lab, **writeSensor()** is solely used for configuring the different gyroscope and pwr_mgmt settings.

```
int writeSensor(uint8_t writeVal, uint8_t accelData){
    // code
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (ICM_ADDR << 1) | I2C_MASTER_WRITE, true);
    i2c_master_write_byte(cmd, writeVal, true);
    i2c_master_write_byte(cmd, accelData, true);
    i2c_master_stop(cmd);
    esp_err_t ret = i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, 1000 / portTICK_PERIOD_MS);
    if(ret != ESP_OK){
      ESP_LOGE("I2C_ERROR", "Write failed: %d", ret);
    }
    i2c_cmd_link_delete(cmd);
    return ret;
}
```

Referring back to the datasheet, these are the configured values used for the gyro and pwr configuration with the setting values to be passed onto the write functions.

1F	31	PWR_MGMT0	R/W	ACCEL_LP_CL K_SEL	-	IDLE	GYRO_MODE	ACCEL_MODE
20	32	GYRO_CONFIG0	R/W	-	GYRO_UI_FS_SEL	-	GYRO_ODR	

All of the initializing occurs during app_main() by initializing the I2C and the gyroscope/pwr before reading into any values. With the help of ChatGPT, I was able to get a starting template with configuring and reading data values. The following shows the different responses that ChatGPT had given me when pressing on conceptual confusions, which ultimately helped provide a template:

```
void configureGyroscope() {
    // Write configuration data to the gyroscope's configuration register
    writeSensor(GYRO_ADDR, GYRO_CONFIG_REG, GYRO_CONFIG_VALUE);
}

void app_main() {
    initializeI2C();  // Initialize I2C communication

    // Configure gyroscope before entering the main loop
    configureGyroscope();

    // Enter the main loop to continuously read and process gyroscope data
    while (1) {
        // Read and process gyroscope data here
        // ...
        vTaskDelay(100 / portTICK_PERIOD_MS);  // Add a delay as needed
    }
}
```

This starting template allowed me to properly format my own app_main() function with initializing and configuring straight away. Since in the loop there must be the reading and processing of the raw gyroscope data, it was one of the questions that I had asked the AI. One thing to note is that the response query provides the logic within arrays but nonetheless it provides conceptual information to fill in the pieces. Instead of using an array, calling on the readSensor() function and calling on the X/Y gyroscope data values for the reading of raw data.

D

so in this, gyroX[0] = (gyroData[1] << 8) | gyroData[0]; // Combine X-axis bytes into 16-bit value
gyroY[0] = (gyroData[3] << 8) | gyroData[2]; you are talking about the raw gyro data

Yes, that's correct. In the provided code snippet, `gyroX[0]` and `gyroY[0]` represent the 16-bit assembled values for the gyroscope's X and Y axes, respectively. These values are obtained by combining the lower and higher byte data retrieved from the gyroscope sensor, assuming the raw gyro data is stored in the `gyroData` array.

The following lines of code:

c — Copy code

```c
gyroX[0] = (gyroData[1] << 8) | gyroData[0];  // Combine X-axis bytes into 1
gyroY[0] = (gyroData[3] << 8) | gyroData[2];  // Combine Y-axis bytes into 1
```

One thing to note is that the use of **ESP_LOGI**–as seen below–prints out the gyroscope data, allowing for accessing the raw data values as the chip moves. This step is crucial as it allows you to define the threshold value as board inclination will be a series of conditionals. With the use of ESP_LOGI(), will print out the direction of the board based on its inclination. ChatGPT gave a similar approach using the accelerometer that can be applied for the gyroscope, with the needed configurations. Essentially, if the gyroscope raw data is greater than or less than the threshold that indicates board inclination.

```c
// Print the coordinates
//ESP_LOGI("GYRO DATA", "Gyroscope x-axis data: %.1f", xGyro);
//ESP_LOGI("GYRO DATA", "Gyroscope y-axis data: %.1f", yGyro);
```

```
// Process accelerometer data to determine inclination
// Logic to determine UP/DOWN/LEFT/RIGHT based on accelerometer data
// This logic would depend on the orientation and thresholds for you
// For demonstration purposes, let's assume basic logic:
if (accel_x > 2000) {
    ESP_LOGI("SENSOR_DATA", "RIGHT");
} else if (accel_x < -2000) {
    ESP_LOGI("SENSOR_DATA", "LEFT");
}

if (accel_y > 2000) {
    ESP_LOGI("SENSOR_DATA", "UP");
} else if (accel_y < -2000) {
    ESP_LOGI("SENSOR_DATA", "DOWN");
}
```

ICM-42670-P Datasheet:

https://invensense.tdk.com/wp-content/uploads/2021/07/ds-000451_icm-42670-p-datasheet.pdf

Lab 4: Part 2

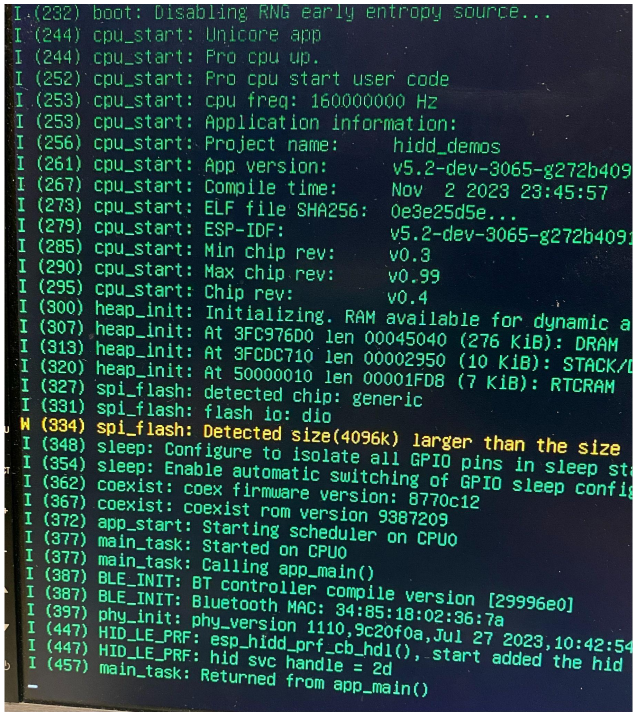
The second part of the lab was a lot simpler to complete and only required the alteration of one file, **ble_hidd_demo_main.c** given to by the lab documentation. Before editing the file, I was supposed to connect the ESP to the bluetooth, compile the file for volume UP/DOWN, and test it accordingly in the startx GUI. Since the espressif github is cloned onto our PI, it was simple to navigate to the necessary files but before flashing the code onto the ESP32C3 connection to bluetooth was necessary. By following the professor's bluetooth installation instructions for this lab and other sources, I was able to successfully activate and pair my ESP by the following commands.

```
sudo apt install pi-bluetooth bluetooth bluez blueman

Systemctl status bluetooth // To check if active

bluetoothctl

scan on

pair 34:85:18:02:36

scan off
```

The reason that I had to include other packages for installation is that I had received errors when running the above commands.

One thing to mention is that before pairing the ESP to bluetooth, I had to figure out the MAC Address for my personal device. It was super simple to achieve as it required you to run the following commands and was able to run the bluetooth commands to successfully pair the device.

```
Idf.py set-target esp32c3

Idf.py flash monitor
```



After all this trouble of configuring bluetooth onto the ESP, there was still the task of moving the mouse left to right, pause for five seconds, with bluetooth. My initial thought was to copy the folder **ble_hid_device_demo** and place it in the /esp directory while renaming it **lab4_2**. From here, I was able to just rename the **ble_hidd_demo_main.c** to **lab4_2.c** to promptly write into the file. Below shows the function that needed to be altered for the purpose of moving the mouse left and right. As you can see below, lines 180-186 are not needed as we are not configuring the volume so it can be deleted. Again, line 178 can be deleted as **esp_hidd_send_consume_value()** was not needed but prompted the idea of another function call that would prove necessary. In the **esp_hidd_prf_api.h** holds some necessary functions that will be needed**,** specifically **esp_hidd_send_mouse_value(uint16_t conn_id, uint8_t mouse_button, int8_t mickeys_x, int8_t mickeys_y)**. Here we only care about the last two parameter inputs since that will allow for the movement and mouse_button remains 0. In order to complete this, we call the function twice with two different input values for mouse_x and leaving mouse_y as 0 since it's not required to move up. Since we're working with an 8-bit integer, for the x-value I added the minimum value (-127) to the maximum value (128) with a 500 millisecond delay in between and after, so that it would move left to right then pause for 5 seconds.

```
void hid_demo_task(void *pvParameters)
{
    vTaskDelay(1000 / portTICK_PERIOD_MS);
    while(1) {
        vTaskDelay(2000 / portTICK_PERIOD_MS);
        if (sec_conn) {
            ESP_LOGI(HID_DEMO_TAG, "Send the volume");
            send_volum_up = true;
            //uint8_t key_vaule = {HID_KEY_A};
            //esp_hidd_send_keyboard_value(hid_conn_id, 0, &key_vaule, 1);
            esp_hidd_send_consumer_value(hid_conn_id, HID_CONSUMER_VOLUME_UP, true);
            vTaskDelay(3000 / portTICK_PERIOD_MS);
            if (send_volum_up) {
                send_volum_up = false;
                esp_hidd_send_consumer_value(hid_conn_id, HID_CONSUMER_VOLUME_UP, false);
                esp_hidd_send_consumer_value(hid_conn_id, HID_CONSUMER_VOLUME_DOWN, true);
                vTaskDelay(3000 / portTICK_PERIOD_MS);
                esp_hidd_send_consumer_value(hid_conn_id, HID_CONSUMER_VOLUME_DOWN, false);
            }
        }
    }
}
```

Bluetooth Github:

https://github.com/espressif/esp-idf/blob/master/examples/bluetooth/bluedroid/ble/ble_hid_device_demo/main/ble_hidd_demo_main.c

Bluetooth Installation: https://askubuntu.com/questions/776806/bluetooth-on-ubuntu-16-04

Lab 4: Part 3

Now that the gyroscope has been configured as well as bluetooth is connected on the ESP32C3. The main objective of this portion is to integrate the previous two parts to utilize the ESP as a bluetooth mouse. Like for lab4_2, copying the folder **ble_hid_device_demo** and renaming it to

lab4_3, while changing the necessary file names and altering CMakeLists.txt. While working within the same program as the previous, we mainly alter the **hide_demo_task()** while also including the **i2c_master_init(), readSensor(),** and **writeSensor()** functions to be included (above the main function as we will be calling on the function). Since we're going to be constantly changing the coordinate value of the mouse, the function **update_mouse()** is necessary to set up the threshold and sensitivity of the mouse as it has to be manually configured within the program. In the first part of the lab, a series of conditionals were used for detecting movement and outputting on the command line. The addition of this function is crucial as detecting isn't the only concern but rather updating the movements of the cursor's position (X/Y) for actual movement/functionality of a mouse. The function takes in 5 parameters for **gyro_cord** (X/Y coordinates), the **current**, **min_mouse** for minimum value, **max_mouse** for maximum value, and **steps** the mouse takes:

- Check if the gyroscope coordinates are within the range of (-2000, 2000), it is in the the point of origin so returning 0, indicating no movement.
- Check if the gyroscope coordinate is greater than 2000 indicates that the board is detecting positive movement. We also check the current value is less than the max mouse position to stay within the boundary.
- Check if the gyroscope coordinate is less than -2000 indicates that the board is detecting negative movement. We also check the current value is greater than the min mouse position to stay within the boundary.

The main structure of the function is heavily inspired by the **lab4_1.c**'s conditional that defined the threshold for board inclination. The function is very general and takes the conditions of positive/negative since we can assume we're in an X/Y plane. With this, the addition of min_mouse and max_mouse provides the boundary conditions that allow for movement in the X and Y direction. Hence the addition of the logic statement **curr > min_mouse** and **curr < max_mouse** to check if the current X/Y position is within the boundary, enabling it to move left/right/down/up. Since we're not printing on the cli, returning the steps taken and current position is necessary to properly update the values as the board detects movement.

Now that we have this, app_main() is going to be the next destination. Like part 1, much of the logic before the loop remains the same (i.e. configuring the gyroscope and pwr_mgmt settings). Reading the data for the X/Y gyroscope remains the same and is used to pass into the update_mouse() function. Initializing two X and Y values that are equivalent to the function calls to then pass into the **esp_hidd_send_mouse_value()** to initialize the functionality of moving the mouse.

```
        if (sec_conn) {
            int16_t xGyro, yGyro;
            // call on readSensor, like this
            xGyro = (readSensor(GYRO_DATA_X1) << 8) | readSensor(GYRO_DATA_X0);
            yGyro = (readSensor(GYRO_DATA_Y1) << 8) | readSensor(GYRO_DATA_Y0);

        // Update movements
        xM = update_mouse(yGyro, xM, -122, 122, 2);
        yM = update_mouse(xGyro, yM, -125, 125, -2);

        printf("xM: %d  yM: %d\n", xM, yM);
        esp_hidd_send_mouse_value(hid_conn_id, 0, xM, yM);

        vTaskDelay(5 / portTICK_PERIOD_MS);
        }
    }
```

Overall, the testing process was quite lengthy and had loads of trial/error when adjusting the sensitivity, hence the threshold.

Bluetooth Github:

https://github.com/espressif/esp-idf/blob/master/examples/bluetooth/bluedroid/ble/ble_hid_device_demo/main/ble_hidd_demo_main.c