Lab 4

by

Garett Loghry

CS466 – Embedded Systems
LABORATORY REPORT

Computer Science, Washington State University Vancouver
May 2nd, 2023

Objective:
The objective for this lab is to prepare us for lab 5, which has been noted to be difficult. This lab focuses on SPI (Serial Peripheral Interface) with the use of a GPIO expander. We also worked in experience once again using interrupts and interrupt handlers. This SPI specifically used “bit-banging” to achieve it’s goal.

Apparatus:
- Raspberry Pi Pico
- Breadboard (protoboard)
- 3 DIP buttons
- Wires
- Laptop for power and debugging
- MCP23S17 GPIO expander
- Red LED
- Logic Analyzer
- 1K resistor

Method:
There are four major sections to this lab in my opinion. First, read the documentation to get an understanding what will be coming up. Second, confirm you can run and compile using only the heartbeat task. Third, Confirm I/O to the GPIO. Lastly, Setup LED functionality, and setup interrupt.

I used a different PINOUT than Miller. Mine is as follows:

Connect 3v3 (OUT) (Pico pin 36) to the power line of your breadboard
Connect GND line (Pico pin 18) to the negative line of your breadboard
SO (Expander pin 14) connects to GPIO 19 (Pico pin 25)
SI (Expander pin 13) connects to GPIO 18 (Pico pin 24)
SCK (Expander pin 12) onnects to GPIO 17 (Pico pin 22)
CS (Expander pin 11) connects to GPIO 16 (Pico pin 21)
Connect VSS to negative line of your breadboard
Connect VDD to positive line of your breadboard
Connect A0, A1, and A2 all to negative line of your breadboard
Connect Run (Pico pin30) to RESET (Expander pin 18)

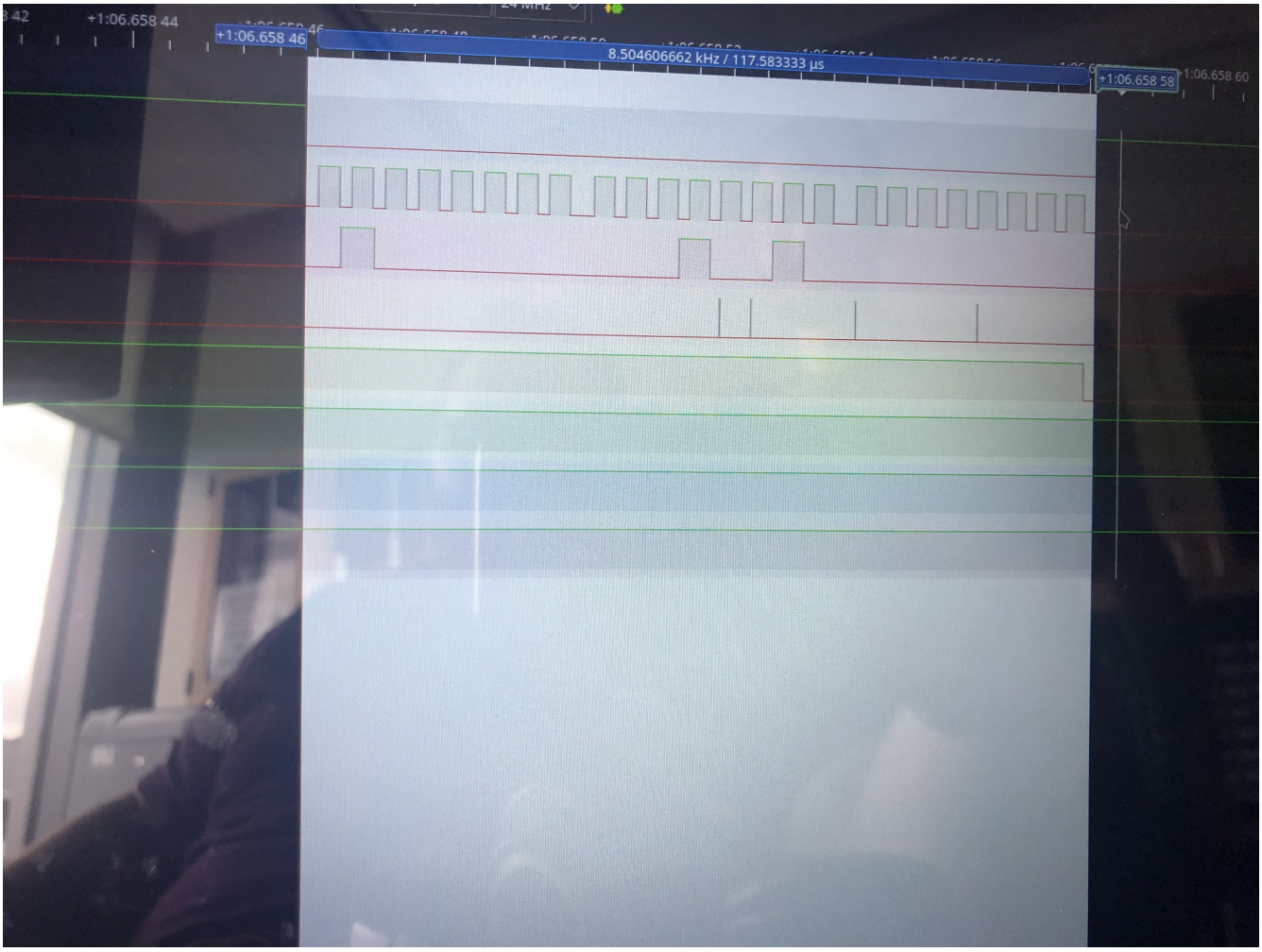
Being careful to avoid Pico pin 23 which is another GND

In later sections GB3 (Expander pin 4) connects to a button
Other side of the button should be tied low to negative line of your breadboard
INTB (Expander pin 19) should be connected to an LED
Other side of the LED should be connected to the negative line of your breadboard with a 1k resistor

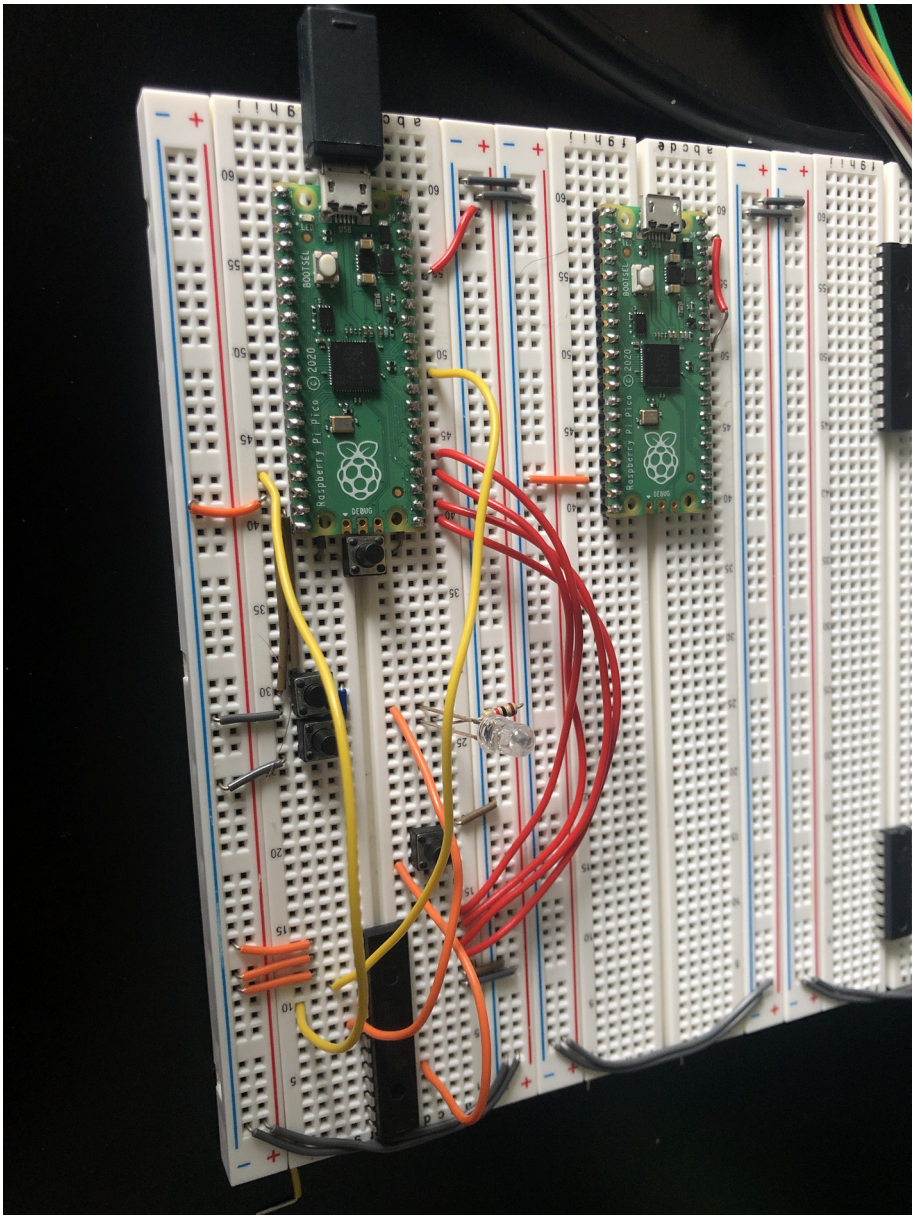
Lab calls for Switches “1” and “2”, but these are not used from what I can tell.

Data:
This was a big moment for me. We hooked up the logic analyzer and finally found that my Chip Select wasn't being set LOW indicating the end of a read cycle.

Table 1.
Here you can see the instructions being sent to and from the pico and the expander. This was the last time I was able to use the logic analyzer as Matt left for rehearsal after this.



This image sucks to work with so it gets its own page. Here’s the final result of the pico, ignore the other pico that will be used for lab 5.



Results and Analysis

Lab 4 pre lab 1

The provided SPIBang.pdf is different that the wiki page because

Lab 4 Part 1

Q1: This SPI device is different than most because of the A0, A1, and A2, along with the chip select. We can use multiple SPI devices on the same GPIO lines specifying the chip, rather than having devoted lines going to each expander.

Q2: Less lines (traces) to print on the PCB, and requires less pins in total, because the lines can be shared by different SPI devices. This means more work for the CS people, but less for EE and probably less money overall.

Lab 4 Part 9

Q3: The BANK bit gives the programmer more freedom to choose it's address values, but the lines themselves are actually the same. It's just the address values go from alternating ABAB to in a row like AABB.

Lab 4 Part 12

Q4: The limiting factor of driving this LED is the read/write time for the byte transfer. I was able to get the LED flash about 94 times per second. At this speed the LED would look visually on all the time without going fully off. Hard to tell how dim it was because the resistor was fairly high.

Lab 4 Part 12

Q5: No, I actually do the exact opposite as verifying that the other pins are unaffected by the LED change, I set the entire bank to 0xFF. In theory you could change this to 0x80 and only set the top bit, and then read afterwards to make sure you return the same value back.

Lab 4 Part 15

Q6: This resistor is required to assure that the floating value would be pulled up to high unless sunk low. With the floating value, it's possible electrostatic or other electric pulses could accidentally trigger the LED. In larger settings, this might not be an LED, but a kill switch that shuts off a large motor, and we don't want that to be a variable. The internal resistor is easy to turn on, is already verified to work, and is less susceptible to outside noise. On the downside, the 100k resistor doesn't let very much current through, so may not be suitable for all uses.

Lab 4 Part 17

I write to `INTFB` with the hex code 0x01 to clear the interrupt. Online places mention you might be able to clear by reading, but I thought writing was more explicit to clear, and it works when tested, so why monkey with anything else.

Lab 4 Part 18

See Table 1. Matt helped me set up the logic analyzer and we tracked the 4 pins to and from the expander and the pico. We tracked a period of instruction cycle for pins and interrupt handling. Only code difference was that I set the heartbeat in the microseconds (1 microsecond) and it

was hard to tell what was even going on. It's possible I don't need all those sleeps, but this is so quick for most operations, I think it would be overkill to do more. It was hard to tell the LED was even dimming because it processed so quickly.

Lab 4 Part 19

a. I tried to find a good debouncing method, but most places online said a state based debouncer would be better from what I've found. I found that my implimentation of a state based debouncer, while somewhat reliable on debouncer, wasn't anywhere near being able to handle 15 button presses a second. I tried to use a simple `vTaskDelay()`, but it really wasn't doing what I wanted, and `sleep_ms()` was worse overall. I'm not sure how to do this part, but I did the best that I could with the time and supplies given.

b. If I press and hold the button I can probably get around 10 to 15 button presses per second, but it doesn't feel worthwhile to continue to mess with this timing using a function generator or second pico, when I should probably move to lab 5. That said, I'm not unhappy with my debouncer, I don't think many people are pressing the button 15 times a second, at least humans. Visually tested was as far as was gotten on this section.

Lab 4 Part 20

Q8. There's probably a few reasons wikipedia lists SPI as the *de facto* standard. For one, it's typically seen as easier to implement than I^2C, which is the other main standard. It's been around for a good while, and only uses for pins on an IC package, which wikipedia says is less than parallel interfaces. As a whole, it means that SPI should be prioritized to learn, as it's more ubiquitous in industry. It means that there's more documentation and experience around it as well. I^2C has it's own benefits, for sure, and SPI isn't a one size fits all for every environment, but it's still the main standard for most embeded systems.

Q9. I had 117 microseconds between instruction time sent from the pico and receiving the interrupt. I don't think I counted clearing the interrupt, but I moved back to longer sleeps after 17, because it was too much information for me to handle. I used sleeps in the microseconds instead of milliseconds, and maybe could have gotten faster if I had a logic analyzer that could sample more often. I felt this was plenty fast for most given purposes.

Q10. As a whole, from the button press on the board to ISR clearing, you can view it as something like this

1. Button press causes GPIOB pin 3 to go low
2. Interrupt watching GPIO pin 3 has trigger on edge
3. Interrupt causes INTB pin to set
4. Pico GPIO pin 13 goes high
5. PICO ISR triggers on edge signaling interrupt
6. PICO finishes most recent task, pushes register onto stack, loads ISR location
7. PICO runs debouncer function
8. PICO gives semaphores
9. PICO receives semaphore
10. PICO writes to INTFB address signaling interrupt flag reset

Q11. I gave the interrupt a higher priority than the heartbeat task, so if there was a lot of interrupts (debouncing, button holds, etc) then the program would continue to service those

interrupts before giving context back. This is assumed to be the wanted behavior, since interrupts are *typically* more important, but it may need to take lower priority than some other task.

Analysis on this lab is that I made some really silly mistakes early that caused this lab to be much harder than it needed to be. The main issue I ran into was I wasn't setting the Chip Select LOW after reading, so nothing was working for me right. This is just a case of writing someone elses code and not really knowing what it was doing, but it was a hard bug to figure out. I had help from a peer, Matthew Rease, who lent me his logic analyzer and looked at my code for me. All code was still written by me, and the entire lab was finished by me as well. I didn't, and still don't feel as comfortable doing bitwise operations and I still have to spend a long time parsing what something is or how to do it, but I think I'm getting better. I'm nervous about lab 5, but we must forge ahead.

Conclusion

Ho-ly crap this lab was a dozy. I had a hell of a time getting this stuff to work, and while I'm happy with the progress I made on GPIO expanders, going back to interrupts, and SPI experience, some simple bugs made this lab tough. Because it took me so long to finish this lab, I will have critically less time to work on lab 5, but we the world continues forward. I feel moderately prepared for lab 5.

```
/* lab 4 code in its entirety below */
/**
 * @brief CS466 Lab4 SPI Bit-Bang
 *
 * Copyright (c) 2022 Washington State University.
 */

#include <stdio.h>

#include <FreeRTOS.h>
#include <task.h>
#include <semphr.h>

#include "hardware/gpio.h"
#include "pico/stdlib.h"

#include "mGpio.h"
#include "mSpi.h"
#include "myAssert.h"

const uint8_t LED_PIN = 25;
const uint8_t SW1_PIN = 14;
const uint8_t SW2_PIN = 15;

const uint8_t INT_B = 13;

#define DEBOUNCE_TIMER 1


static volatile uint32_t lastTime = 0;
static volatile bool lastState = true;


static SemaphoreHandle_t _intBtn = NULL;

/* Core functions that put/get values to/from their respective pins */
void setMOSI(bool val) { gpio_put(MOSI_PIN, val); }
void setSCK(bool val) { gpio_put(CLK_PIN, val); }
uint8_t getMISO(void) { return gpio_get(MISO_PIN); }
void setCS(bool val) { gpio_put(CS_PIN, val); }

static bool debounce(uint gpio) {
    uint32_t time = to_ms_since_boot(get_absolute_time());
    bool state = gpio_get(gpio);

    if (state != lastState) {
        lastTime = time;
        lastState = state;
        return false;
    } else if ((time - lastTime) > DEBOUNCE_TIMER) {
        lastTime = time;
        return true;
    } else return false;
}
```

```
void interrupt_from_b(uint gpio, uint32_t event) {
    if (debounce(gpio)) xSemaphoreGiveFromISR(_intBtn, NULL);
}

void int_handler(void *notUsed) {
    while (true) {
        xSemaphoreTake(_intBtn, portMAX_DELAY);
        printf("Interrupt triggered by expander...\tClearling interrupt line\n");
        mGpioWriteByte(INTFB, 0x01);
    }
}


/* INIT THE **PICO** GPIO PINS AND SET THEIR CORRECT DIR
   I KEPT GETTING THIS MIXED UP WITH EXPANDER GPIO */
void mSpiInit(void) {
    gpio_init(CS_PIN);
    gpio_set_dir(CS_PIN, GPIO_OUT);

    gpio_init(CLK_PIN);
    gpio_set_dir(CLK_PIN, GPIO_OUT);

    gpio_init(MOSI_PIN);
    gpio_set_dir(MOSI_PIN, GPIO_OUT);

    gpio_init(MISO_PIN);
    gpio_set_dir(MISO_PIN, GPIO_IN);

    gpio_init(INT_B);
    gpio_pull_up(INT_B);
    gpio_set_dir(INT_B, GPIO_IN);
    gpio_set_irq_enabled_with_callback(INT_B, GPIO_IRQ_EDGE_FALL, true,
&interrupt_from_b);

    setCS(HIGH);

}

/* INIT THE DIR OF GPIO A AND B FOR THE EXPANDER
   AND CHECK VALUE SET */
void mGpioInit(void) {
    uint8_t defvalA = mGpioReadByte(IODIRA);
    uint8_t defvalB = mGpioReadByte(IODIRB);

    uint8_t setvalA = 0x55, setvalB = 0x56;

    mGpioWriteByte(IODIRA, setvalA);
    mGpioWriteByte(IODIRB, setvalB);
    uint8_t retvalA = mGpioReadByte(IODIRA);
    uint8_t retvalB = mGpioReadByte(IODIRB);

    myAssert(defvalA == 0xff);
    myAssert(defvalB == 0xff);
```

```
    myAssert(setvalA == retvalA);
    myAssert(setvalB == retvalB);

    mGpioWriteByte(IODIRA, 0x00);
    mGpioWriteByte(IODIRB, 0x08);

    mGpioWriteByte(GPINTENB, 0x08);
    mGpioWriteByte(IOCONB, 0x02);


    mGpioWriteByte(GPPUB, 0x08);
    mGpioWriteByte(DEFVALB, 0x08);
    mGpioWriteByte(INTCONB, 0x08);


    sleep_ms(1);

//    printf("val0: 0x%02x\nval1: 0x%02x\n", val0, val1);
}

/* CODE PROVIDED BY MILLER AS PART OF
   SPI Interfacing document. Copied from there,
   small change from 'out' to outData' to match prototype */
uint8_t mSpiTransfer(uint8_t outData) {
    uint8_t count, in = 0;
/*
    gpio_put(CS_PIN, 1);
    gpio_put(CLK_PIN, 1);
    gpio_put(MOSI_PIN, 1);
    printf("%d\t%d\t%d\t%d\n", gpio_get(CS_PIN), gpio_get(CLK_PIN),
gpio_get(MOSI_PIN), gpio_get(MISO_PIN));
*/

    setSCK(LOW);
    for (count = 0; count < 8; count++) {
        in <<= 1;
        setMOSI(outData & 0x80);
        sleep_us(1);
        //printf("%d\t%d\t%d\t%d\n", gpio_get(CS_PIN), gpio_get(CLK_PIN),
gpio_get(MOSI_PIN), gpio_get(MISO_PIN));
        setSCK(HIGH);
        sleep_us(1);
        in += getMISO();
        sleep_us(1);
        setSCK(LOW);
        outData <<= 1;
    }
    setMOSI(0);
    sleep_us(1);

    return (in);
}

void mGpioWriteByte(uint8_t addr, uint8_t byte) {
```

```
    uint8_t preWrite = 0x40;        // OK THE READ/WRITE VAL
                                    // TOOK ME FOREVER TO FIND.
                                    // 010000 {0:1} 0 - write
                                    //              1 - read
    setCS(LOW);                     // indicate transfer
    mSpiTransfer(preWrite);         // indicate write - see above
    mSpiTransfer(addr);             // indicate address
    mSpiTransfer(byte);             // transfer byte
    setCS(HIGH);                    // indicate transfer complete
    sleep_ms(5);

    return;
}

uint8_t mGpioReadByte(uint8_t addr) {
    uint8_t val, preRead = 0x41;

    setCS(LOW);
    mSpiTransfer(preRead);
    mSpiTransfer(addr);
    val = mSpiTransfer(0);
    setCS(HIGH);

    return (val);
}


void hardware_init(void)
{
    const uint LED_PIN = PICO_DEFAULT_LED_PIN;
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);

    mSpiInit();
    mGpioInit();
}
//
// gpioVerifyReadWrite()
//
// This is the main function of a task that I'm using to verify that
// my GPIO and SPI functionality is working correctly.  It will be retired
// as I move on to actual GPIO-Expander Functionality.
//
void gpioVerifyReadWrite(void * notUsed)
{
    const uint32_t queryDelayMs = 500;  // ms
    uint8_t regValue;
    uint8_t count=0;

    vTaskDelay(5000);

    while (true)
    {
        mGpioWriteByte(IODIRB, count++);
```

```
        regValue = mGpioReadByte(IODIRB);
        printf("IODIRB: 0x%02x,  ", regValue);

        regValue = mGpioReadByte(IODIRA);
        printf("IODIRA: 0x%02x,  ", regValue);

        regValue = mGpioReadByte(IPOLA);
        printf("IPOLA: 0x%02x\n", regValue);

        vTaskDelay(queryDelayMs);
    }
}

// IODIRB: 0x00,  IODIRA: 0xff,  IPOLA: 0x00
// IODIRB: 0x01,  IODIRA: 0xff,  IPOLA: 0x00
// IODIRB: 0x02,  IODIRA: 0xff,  IPOLA: 0x00
// IODIRB: 0x03,  IODIRA: 0xff,  IPOLA: 0x00
// IODIRB: 0x04,  IODIRA: 0xff,  IPOLA: 0x00
// IODIRB: 0x05,  IODIRA: 0xff,  IPOLA: 0x00
// IODIRB: 0x06,  IODIRA: 0xff,  IPOLA: 0x00
// etc....

void ledCtrl(uint8_t val) {
    if((mGpioReadByte(GPIOB) & 0x08) == LOW) {
        mGpioWriteByte(GPIOA, HIGH);
    } else {
        mGpioWriteByte(GPIOA, val);
    }
}


void heartbeat(void * notUsed)
{
    const uint32_t heartbeatDelay = 200;  // ms

    while (true)
    {

        gpio_put(LED_PIN, 1);
        /*
        gpio_put(CS_PIN, 1);
        gpio_put(CLK_PIN, 1);
        gpio_put(MOSI_PIN, 1);
        */
        ledCtrl(0xFF);
        vTaskDelay(heartbeatDelay);
        gpio_put(LED_PIN, 0);
        ledCtrl(0x00);
        /*
        gpio_put(CS_PIN, 0);
        gpio_put(CLK_PIN, 0);
        gpio_put(MOSI_PIN, 0);
        */
        vTaskDelay(heartbeatDelay);
```

```
        printf("lab4 Tick\n");
    }
}

int main()
{
    stdio_init_all();
    printf("lab4 Hello!\n");

    hardware_init();

    _intBtn = xSemaphoreCreateBinary();

    xTaskCreate(heartbeat, "LED_Task", 256, NULL, 1, NULL);
    //xTaskCreate(gpioVerifyReadWrite, "GPIO_Task", 256, NULL, 2, NULL);
    xTaskCreate(int_handler, "Interrupt Handler", 256, NULL, 2, NULL);

    vTaskStartScheduler();

    while(1){};
}
```