Lab 3

by

Garett Loghry

CS466 – Embedded Systems
LABORATORY REPORT

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

Objective:
The object for this lab was to expand our knowledge and skill using different types of scheduling and task resources. The last lab used semaphores to signal, this lab used queues to signal. Both are blocked-ready-running types of schedule, but have different setups and interfaces. We also touched on asserts, which can and should be used in production code in case something fails.

Apparatus:
- Raspberry Pi Pico
- Breadboard (protoboard)
- 3 DIP buttons
- Wires
- Laptop for power and debugging

Method:
In general, I would say this lab has 4 main parts after initial setup (copying code). Listed I would say they are
1. Test and implement `assert()` and `myAssert()`
2. Create the consumer task
3. Create the single producer task
   - Expand the producer to include structure]
   - Test that it works with consumer
4. Expand to include a second producer

Data:
Lab 3 part 3 confirm that the serial port can receive a counting tick counter, and gpio Switch 1 and Switch 2

```
tick: 36,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 37,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 38,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 39,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 40,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 41,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 42,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 43,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 44,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 45,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 46,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 47,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 48,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 49,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 50,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 51,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 52,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 53,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 54,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 55,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 56,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 57,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 58,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 59,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 60,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 61,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 62,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 63,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 64,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 65,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 66,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 67,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 68,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 69,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 70,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 71,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 72,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 73,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 74,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 75,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 76,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
tick: 77,       Switch 1 is Not Pressed,        Switch 2 is Not Pressed
```

Lab 3 General showing that the assert fails and where.

```
tick: 1,        Switch 1 is NOT PRESSED,        Switch 2 is NOT PRESSED
tick: 2,        Switch 1 is NOT PRESSED,        Switch 2 is NOT PRESSED
tick: 3,        Switch 1 is NOT PRESSED,        Switch 2 is NOT PRESSED
tick: 4,        Switch 1 is NOT PRESSED,        Switch 2 is NOT PRESSED
tick: 5,        Switch 1 is NOT PRESSED,        Switch 2 is NOT PRESSED
tick: 6,        Switch 1 is NOT PRESSED,        Switch 2 is NOT PRESSED
tick: 7,        Switch 1 is NOT PRESSED,        Switch 2 is NOT PRESSED
tick: 8,        Switch 1 is PRESSED,    Switch 2 is NOT PRESSED
tick: 9,        Switch 1 is PRESSED,    Switch 2 is NOT PRESSED
Assertion Failed: uxQueueSpacesAvailable(queue) != 0 at /home/garett/Repos/WSU/CS466/cs466_23/lab3/lab3.c::151
```

Results and Analysis

Lab 3 Part 2

a. This section wasn't horribly difficult, as we copied a lot of code over. I decided to keep small things, such as separating out the `heartbeat()` task, into a smaller, core task that can get called by other functions. I also decided I liked the handler being passed, so I kept it as well.

Lab 3 Part 5

I had quite a lot of issues using `myAssert()` correctly. For starters, there was an issue with the sent out repo where the `myAssert.h` file wasn't formatted correctly (not sure what casting 0 as void does), but while I was trying to test `myAssert()` I wasn't passing an argument. This mean that the macro that was being used by the compiler was giving me a *syntax* error, rather than a function parameter error. Debugging required the help of a fellow student, Matthew Rease, as well as the instructor Miller. Once I moved the assert into a correct position, it worked fine.

Lab 3 Part 6

The linux command `make && cp lab3.uf2 /media/miller/RPI-RP2/ && sleep 2 \ && kermit ~/kermACM0` (which for me was actually `make && cp lab3.uf2 /media/garett/RPI-RP2/ && sleep 2 && screen /dev/ttyACM0`) causes the script to sleep for N (in this case 2) seconds after copying the source code to the pico. This means that we don't see the initial greeting message, or the first few ticks typically. From there, we open a serial viewer to see what `printf()` is doing after we are done sleeping.

a. Failing the assertion causes to major things to happen. First, in the screen viewer, we get the message `Assertion Failed: tickCount < 15 at /home/garett/Repos/WSU/CS466/cs466_23/lab3/lab3.c::67` indicating where the assert failed, and what file caused it. We also see the LED blink at a specified amount selected in `myAssert.c`, which is 10hz, or flashes every 100ms (50ms on, 50ms off) with a 50% duty cycle.

Lab 3 Part 13

a. Looking back, I think I interpreted the lab wrong. I thought we were supposed to keep the LED's lit, ONLY when someone was holding them, not to act a flag that can be turned on, but not turned off. This question makes me think the switches were supposed to trigger a flag that would pickup messages that can only be turned on, but that wasn't the way I configured it. Regardless, you can tell both producers have a different value, and both are sending to the

consumer because of different Hertz frequencies used by the LED's. This still uses the queue, so if a different implimentation was desired it would mostly require changing how the LED logic worked out, which isn't that much work. Hard to know which one was being asked for though.
b. I tried to use the pico assert to test, and it didn't give me a line number, or any hints as to what exactly failed the assert. This may be better for production code, but for testing purposes, it makes it better difficult. On the flip side, I ran into issues with `myAssert()` because of the use of macros. The compiler does an actual replacement, so figuring what was causing the compiler error was a bit tricky.

Conclusion

In conclusion, I learned how to use queues in the context of embedded systems, got better at programming and problem solving, and learned more about asserts, and custom assert functions. I feel prepared for lab 4.

```
/* lab 3 code in its entirety below */
/**
 * @brief CS466 Lab1 Blink proigram based on pico blink example
 *
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

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

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

// used for me internally to test w printf
#define DEBUG 0
enum task{consumer_t, producer1_t, producer2_t};  // I will forget this, probably

typedef struct {
    int priority;
    char *name;
    QueueHandle_t theQueue;
} property_t;

const uint8_t LED_PIN = 25;
const uint8_t SW1_PIN = 17;
const uint8_t SW2_PIN = 16;

/* invert GPIO switches, we pullup */
bool sw1_press(void) {
    return !gpio_get(SW1_PIN);
}

bool sw2_press(void) {
    return !gpio_get(SW2_PIN);
}

uint32_t heartbeatDelay = 500;  // ms
uint32_t tickCount = 0;

TaskHandle_t hb_task = NULL;

void hardware_init(void) {
    const uint LED_PIN = PICO_DEFAULT_LED_PIN;
    gpio_init(LED_PIN);
```

```
    gpio_set_dir(LED_PIN, GPIO_OUT);

    gpio_init(SW1_PIN);
    gpio_pull_up(SW1_PIN);
    gpio_set_dir(SW1_PIN, GPIO_IN);

    gpio_init(SW2_PIN);
    gpio_pull_up(SW2_PIN);
    gpio_set_dir(SW2_PIN, GPIO_IN);
}

/* Core Heartbeat Function.
   50% duty cycle.
   in ms
   Moved from outside of heartbeat()
   because others may want to use it */
void hb_ms(uint32_t ms) {
    gpio_put(LED_PIN, 1);
    vTaskDelay(ms);
    gpio_put(LED_PIN, 0);
    vTaskDelay(ms);
}

void hb_hz(uint32_t hz) {
    uint32_t ms = (1000/hz);
    ms /= 2;
    hb_ms(ms);
}

void heartbeat(void *notUsed) {
    while (true) {
        hb_ms(heartbeatDelay);
        tickCount++;
        printf("tick: %d,\tSwitch 1 is %s,\tSwitch 2 is %s\n", \
            tickCount, \
            gpio_get(SW1_PIN)?"NOT PRESSED":"PRESSED", \
            gpio_get(SW2_PIN)?"NOT PRESSED":"PRESSED");
        //assert(tickCount < 5); // only used in part 5, bit of a pain otherwise
        //myAssert(tickCount < 5); // only used in part 5, bit of a pain otherwise

    }
}

void consumer(void *taskPointer) {
    // task pointer is the pointer given from void *
    // needs to be recast, and this is the easiest way I know
    property_t *taskProperties = (property_t*) taskPointer;
    QueueHandle_t queue = taskProperties[consumer_t].theQueue;
    uint32_t buff;  // buff doesn't do anything other than signal
    while(1) {
        xQueueReceive(queue, &buff, portMAX_DELAY);
        if (DEBUG) printf("Message received from %d\n", buff);
        if (sw1_press() && buff == producer1_t) { /* BUTTONS HELD FOR TOO LONG WILL
CAUSE THE QUEUE
```

```
                                TO FILL AND ASSERT WILL TRIGGER.
                                THIS HAPPENS AFTER ~2s WITH CURRENT SETUP. */
            if (DEBUG) printf("SW1 pressed, message from p1 - flashing\n");
            for (int i = 0; i < 5; i++) hb_ms(25);
        }
        if (sw2_press() && buff == producer2_t) { /* BUTTONS HELD FOR TOO LONG WILL
CAUSE THE QUEUE
                                TO FILL AND ASSERT WILL TRIGGER.
                                THIS HAPPENS AFTER ~2s WITH CURRENT SETUP. */
            if (DEBUG) printf("SW1 pressed, message from p1 - flashing\n");
            for (int i = 0; i < 4; i++) hb_ms(35);
        }
    }
}

void producer1(void *taskPointer) {
    property_t *taskProperties = (property_t*) taskPointer;
    QueueHandle_t queue = taskProperties[consumer_t].theQueue;
    uint32_t buff = producer1_t, wait;
    while(1) {
        wait = (rand() % 10) * 10;
        vTaskDelay(wait);
        xQueueSend(queue, (void*) &buff, 0);
        if (DEBUG) printf("Message sent\n");
        myAssert(uxQueueSpacesAvailable(queue) != 0);

/* Section for producer flashing is for my testing
        if (sw2_press()) {      // ASSERT WON'T TRIGGER HERE,
                                // BUT THE PROGRAM WILL SLOW DOWN SENDING MESSAGES
            if (DEBUG) printf("SW2 pressed, flashing\n");
            for (int i = 0; i < 6; i++) hb_ms(35);
        }
*/
    }
}

void producer2(void *taskPointer) {
    property_t *taskProperties = (property_t*) taskPointer;
    QueueHandle_t queue = taskProperties[consumer_t].theQueue;
    uint32_t buff = producer2_t, wait;
    while(1) {
        wait = (rand() % 10) * 10;
        vTaskDelay(wait);
        xQueueSend(queue, (void*) &buff, 0);
        if (DEBUG) printf("Message sent\n");
        myAssert(uxQueueSpacesAvailable(queue) != 0);

/* Section for producer flashing is for my testing
        if (sw2_press()) {      // ASSERT WON'T TRIGGER HERE,
                                // BUT THE PROGRAM WILL SLOW DOWN SENDING MESSAGES
            if (DEBUG) printf("SW2 pressed, flashing\n");
            for (int i = 0; i < 6; i++) hb_ms(35);
        }
*/
```

```
    }
}

int main() {
    pid_t pid = getpid();   // use pid to seed program. "more" random
    srand(pid);
    stdio_init_all();
    printf("lab3 Hello!\nRandom number seed is %d", pid);
    hardware_init();

    xTaskCreate(heartbeat, "LED_Task", 256, NULL, 0, &hb_task);
    QueueHandle_t qHandle = xQueueCreate(20, sizeof( uint32_t ));
    myAssert(qHandle != NULL);

    property_t taskProperties[] = {
        {consumer_t, "Consumer", qHandle},
        {producer1_t, "Producer1", qHandle},
        {producer2_t, "Producer", qHandle}
    };

    xTaskCreate(consumer, "Consumer", 256, (void*) taskProperties, 1, NULL);
    xTaskCreate(producer1, "Producer1", 256, (void*) taskProperties, 2, NULL);
    xTaskCreate(producer2, "Producer2", 256, (void*) taskProperties, 2, NULL);

    vTaskStartScheduler();

    while(1){};
}
```