Ali Baqar and Anahit Sarao
SID#008815417
SID#008435583
July 26, 2018

Watchdog Assignment

I. Introduction

The assignment for this week involved creating a watchdog, which is basically a task that monitors other running task and sends an alert if any of the tasks are not functioning properly. The assignment for the week included creating a producer, consumer and watchdog task and then we would monitor the affects of suspending and resuming the running tasks.

II. Results

i. The first part was creating the producer task that takes a light sensor value every 1 ms and puts it on the queue. The code for the producer task can be seen in Image 1.

ii. The second part of the lab was to create the consumer that pulls data of the queue and writes it to a file on the on-board SD card.

iii. The third part of the lab was to set a bit using the XEventGroupSetBit API which holds different flags for different tasks.

iv. The watchdog task was the fourth part of the assignment which as mentioned in the Introduction just monitors the running task by using the flags and sends an alert if something is functioning properly.

v. The fifth part was to be able to suspend and resume the tasks using the terminal and Hercules.

III. Conclusion

The lab was completed without the exception of plotting the data. I do not have an SD card reader on hand to pull the data from. All the functionality has been tested and everything seems to be working fine.

Appendix

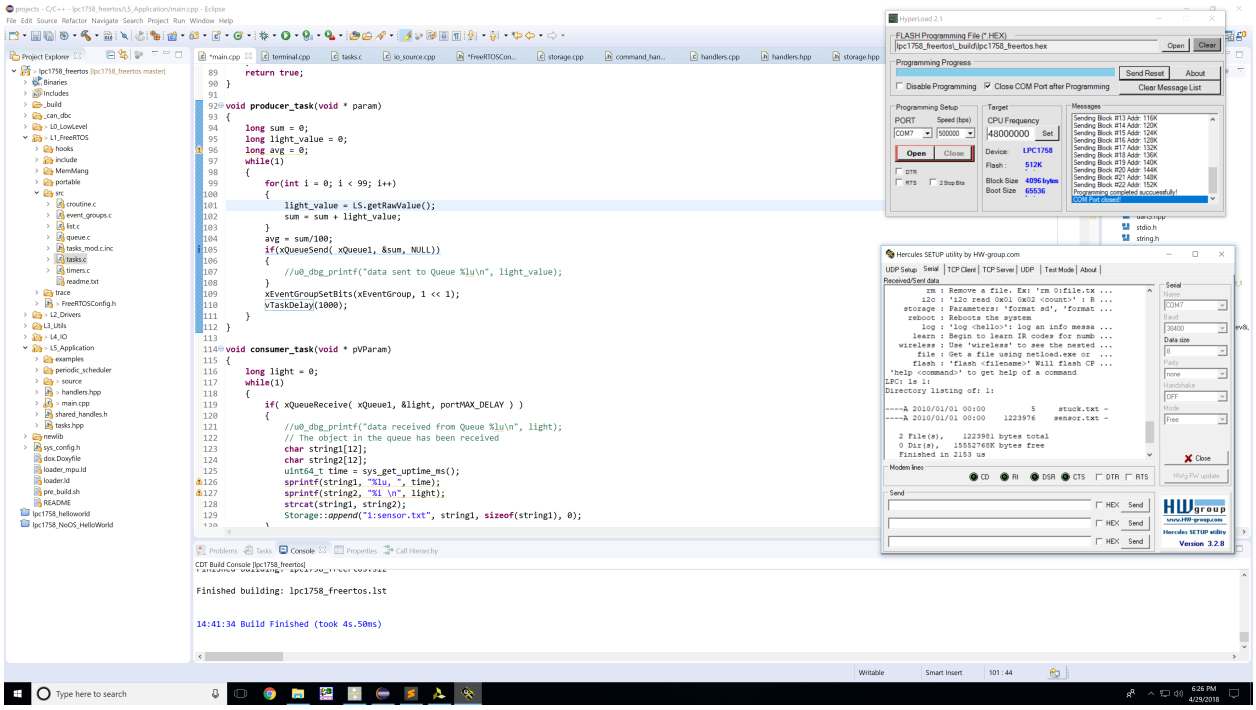

Image 1. Producer Task

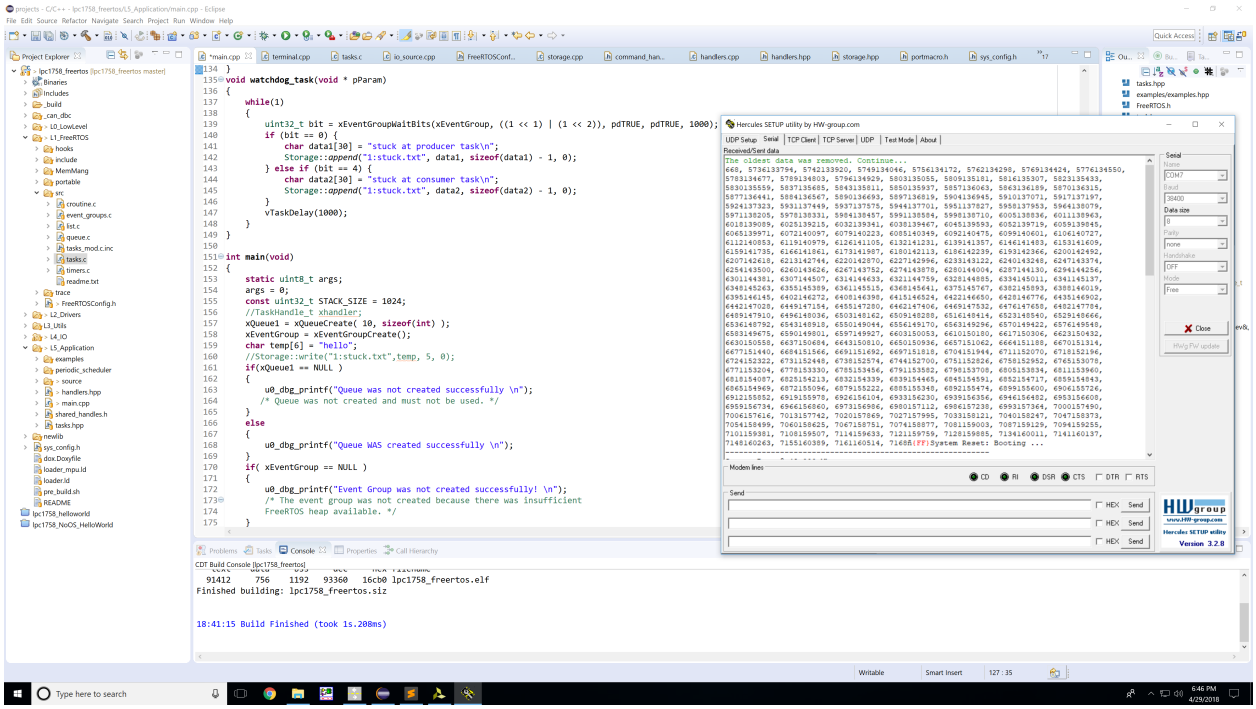

Image 2. Sensor data from *sensor.txt*

Source Code:

Main.cpp

```
/*
 *     SocialLedge.com - Copyright (C) 2013
 *
 *     This file is part of free software framework for embedded processors.
 *     You can use it and/or distribute it as long as this copyright header
 *     remains unmodified.  The code is free for personal use and requires
 *     permission to use in a commercial product.
 *
 *      THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS,
IMPLIED
 *      OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
 *      MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS
SOFTWARE.
 *      I SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL,
OR
 *      CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
 *
 *     You can reach the author of this software at :
 *          p r e e t . w i k i @ g m a i l . c o m
 */

/**
 * @file
 * @brief This is the application entry point.
 *          FreeRTOS and stdio printf is pre-configured to use uart0_min.h
before main() enters.
 *          @see L0_LowLevel/lpc_sys.h if you wish to override printf/scanf
functions.
 *
 */
#include "tasks.hpp"
#include "examples/examples.hpp"
#include "FreeRTOS.h"
#include "task.h"
#include "uart0_min.h"
#include "labGPIO_0.hpp"
#include "utilities.h"
#include "io.hpp"
#include "storage.hpp"
#include "printf_lib.h"
#include "queue.h"
#include "i2c2.hpp"
#include "uart3.hpp"
#include "stdio.h"
#include "string.h"
#include "event_groups.h"
#include "handlers.hpp"



/**
 * The main() creates tasks or "threads".  See the documentation of
scheduler_task class at scheduler_task.hpp
 * for details.  There is a very simple example towards the beginning of this
class's declaration.
 *
 * @warning SPI #1 bus usage notes (interfaced to SD & Flash):
```

```
 *      - You can read/write files from multiple tasks because it
automatically goes through SPI semaphore.
 *      - If you are going to use the SPI Bus in a FreeRTOS task, you need to
use the API at L4_IO/fat/spi_sem.h
 *
 * @warning SPI #0 usage notes (Nordic wireless)
 *      - This bus is more tricky to use because if FreeRTOS is not running,
the RIT interrupt may use the bus.
 *      - If FreeRTOS is running, then wireless task may use it.
 *        In either case, you should avoid using this bus or interfacing to
external components because
 *        there is no semaphore configured for this bus and it should be used
exclusively by nordic wireless.
 */


QueueHandle_t xQueue1;
EventGroupHandle_t xEventGroup;
TaskHandle_t consumerTask;
TaskHandle_t producerTask;
TaskHandle_t watchdogTask;


CMD_HANDLER_FUNC(taskSuspendResume)
{
    scheduler_task* pTask = scheduler_task::getTaskPtrByName("producer");
    scheduler_task* cTask =
scheduler_task::getTaskPtrByName("consumer_task");
    if(cmdParams == "resume producer task") {
        vTaskResume(producerTask);
        output.printf("resumed prodcuer task\n");
    }
    else if(cmdParams == "resume consumer task") {
        vTaskResume(consumerTask);
        output.printf("resumed consumer task \n");
    }
    else if(cmdParams == "suspend producer task") {
        vTaskSuspend(producerTask);
        output.printf("suspended producer task \n");
    }
    else if(cmdParams == "suspend consumer task"){
        vTaskSuspend(consumerTask);
        output.printf("suspended consumer task \n");
    }
    return true;
}

void producer_task(void * param)
{
    long sum = 0;
    long light_value = 0;
    int avg = 0;
    while(1)
    {
        for(int i = 0; i < 99; i++)
        {
            light_value = LS.getRawValue();
            sum = sum + light_value;
```

```
        }
        avg = (int)sum/100;
        if(xQueueSend( xQueue1, &avg, NULL))
        {
            //u0_dbg_printf("data sent to Queue %d \n", light_value);
        }
        xEventGroupSetBits(xEventGroup, 1 << 1);
        vTaskDelay(1000);
    }
}

void consumer_task(void * pVParam)
{
    int light = 0;
    while(1)
    {
        if( xQueueReceive( xQueue1, &light, portMAX_DELAY ) )
        {
            //u0_dbg_printf("data received from Queue %lu\n", light);
            // The object in the queue has been received
            char string1[12];
            char string2[12];
            uint64_t time = sys_get_uptime_ms();
            sprintf(string1, "%lu, ", time);
            sprintf(string2, "%d \n", light);
            strcat(string1, string2);
            Storage::append("1:sensor.txt", string1, sizeof(string1), 0);
        }
        xEventGroupSetBits(xEventGroup, 1 << 2);
        vTaskDelay(1000);
    }
}
void watchdog_task(void * pParam)
{
    while(1)
    {
        uint32_t bit = xEventGroupWaitBits(xEventGroup, ((1 << 1) | (1 <<
2)), pdTRUE, pdTRUE, 1000);
        if (bit == 0) {
            char data1[30] = "stuck at producer task\n";
            Storage::append("1:stuck.txt", data1, sizeof(data1) - 1, 0);
        } else if (bit == 4) {
            char data2[30] = "stuck at consumer task\n";
            Storage::append("1:stuck.txt", data2, sizeof(data2) - 1, 0);
        }
        vTaskDelay(1000);
    }
}

int main(void)
{
    static uint8_t args;
    args = 0;
    const uint32_t STACK_SIZE = 1024;
    //TaskHandle_t xhandler;
    xQueue1 = xQueueCreate( 10, sizeof(int) );
    xEventGroup = xEventGroupCreate();
    char temp[6] = "hello";
```

```
    //Storage::write("1:stuck.txt",temp, 5, 0);
    if(xQueue1 == NULL )
    {
        u0_dbg_printf("Queue was not created successfully \n");
       /* Queue was not created and must not be used. */
    }
    else
    {
        u0_dbg_printf("Queue WAS created successfully \n");
    }
    if( xEventGroup == NULL )
    {
        u0_dbg_printf("Event Group was not created successfully! \n");
        /* The event group was not created because there was insufficient
        FreeRTOS heap available. */
    }
    else
    {
        u0_dbg_printf("Event Group WAS created successfully! \n");
        /* The event group was created. */
    }
    xTaskCreate(producer_task, "producer", STACK_SIZE, &args,
PRIORITY_MEDIUM, &producerTask);
    xTaskCreate(consumer_task, "consumer", STACK_SIZE, &args,
PRIORITY_MEDIUM, &consumerTask);
    xTaskCreate(watchdog_task, "watchdog", STACK_SIZE, &args, PRIORITY_HIGH,
&watchdogTask);

    scheduler_add_task(new terminalTask(PRIORITY_HIGH));
    scheduler_start(); ///< This shouldn't return
    return -1;
}
```