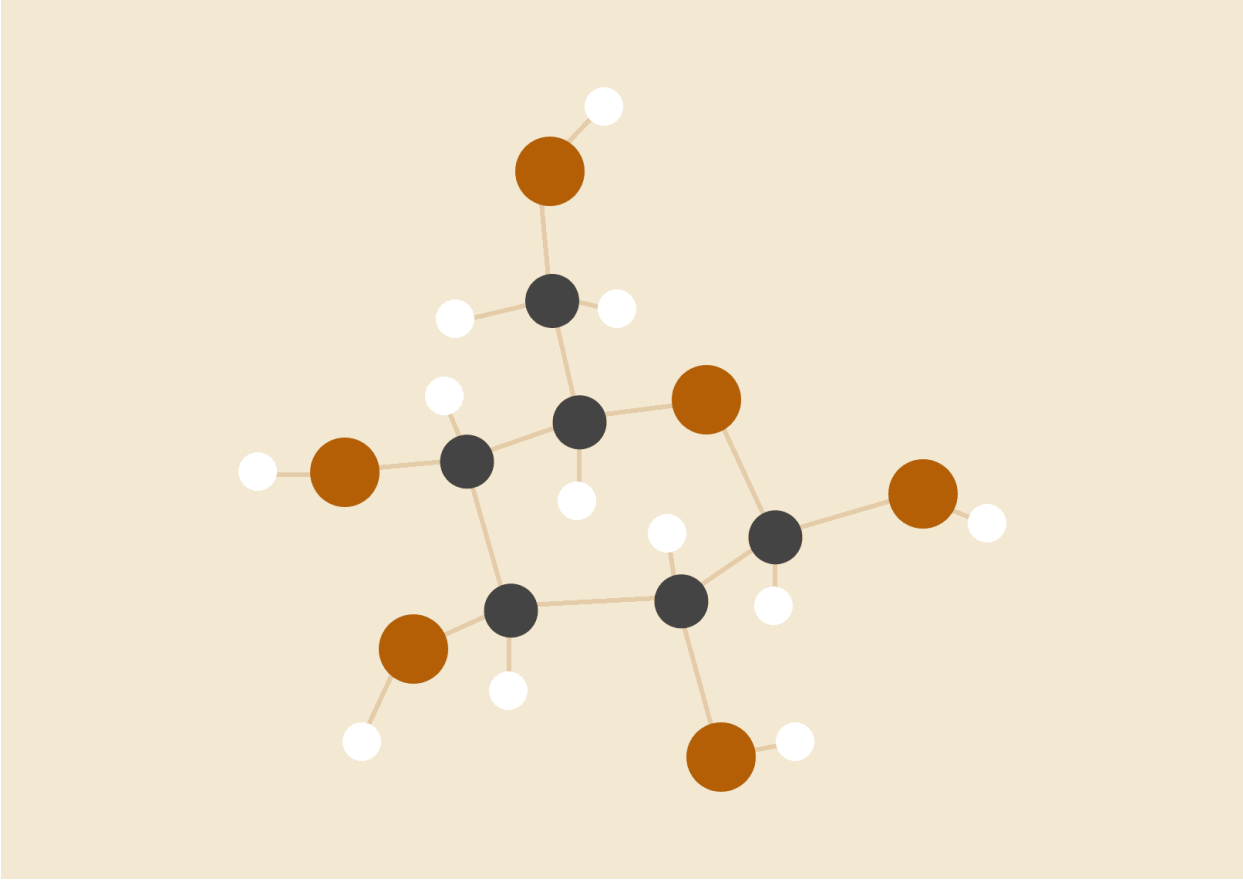
COMP2004 Coursework

Final Report



Bethany Marie Ward (#10605926)

12.05.2021

INTRODUCTION

The code I have written is merely a starting point for the coursework. It only aims to meet requirements 1, 2, 3, 5, 6, 7, 10, 11, 12 and 13 of the specification. I avoided requirements 4, 8 and 9 because of time constraints. I did some research on these requirements before starting the code, and found very few resources for them online. I believe I have completed my chosen requirements to the best of my ability, with the resources and time that I had. For the duration of the assessment, you may consider logging to be on, which in tail may satisfy requirement 12.

REQUIREMENT 1 - MEASURE SENSOR DATA

I took the code from the starter template supplied by the lecturer and used it as a baseline for this function. I added a time measurement from the 'time.h' library to show that my sampling time, 1000ms, matched the default sampling time of one second outlined in the specification.

```
115  // Measure sensor data (Req 1). I have taken in a FIFO argument to write data
     to the buffer as I measure it.
116  void measureSensorData() {
117      float temperature, pressure;
118      float ambience;
119
120      bmp280.initialize();
121
122      printf("Sampling...\n");
123      while (true) {
124          temperature = bmp280.getTemperature();
125          pressure = bmp280.getPressure();
126          ambience =  ldr;
127          auto time = std::chrono::system_clock::to_time_t
             (std::chrono::system_clock::now());
128          string s = ctime(&time);
129
130          printf("Datetime: %s", ctime(&time));
131          printf("Temperature (C): %.1f; Pressure (mBar): %.1f ; Light: %.1f\n",
             temperature, pressure, ambience);
132
133          if (f.isFull(f) == true) {  // Check for full FIFO and remove old data.
134              remove(f);
135          }
136
137          add(f, ("Datetime: " + s + " ; Temperature (C): " + to_string
             (temperature) + " ; Pressure (mBar): " + to_string(pressure) + " ;
             Light: " + to_string(ambience)));  // Then add new data...
138
139          ThisThread::sleep_for(1000ms); //...and repeat each second
140      }
```

```
Problems x  >_ NUCLEO-F429ZI x  Output x  Debug Console x  Libraries x
Datetime: Thu Jan  1 03:52:43 1970
Temperature (C): 28.8; Pressure (mBar): 1001.4 ; Light: 0.3
Datetime: Thu Jan  1 03:52:45 1970
Temperature (C): 28.8; Pressure (mBar): 1001.3 ; Light: 0.3
Datetime: Thu Jan  1 03:52:46 1970
Temperature (C): 28.8; Pressure (mBar): 1001.3 ; Light: 0.3
```

I logged the measured data to the serial terminal to show that real time sensor data was indeed being measured. I also added this data to a cell in the **FIFO** ring **buffer** I had made, which will be covered more in the section 'Requirement 3'. This buffer full of data is in an array in its own structure. There is no conceivable jitter, which one will see when running the code.

I am aware that the time recorded is not true to the actual time and date. Unfortunately, I could not find an appropriate function to measure the actual datetime.

REQUIREMENT 2 - WRITE TIME AND SENSOR DATA TO SD

```
// Write sensor data to SD card using FIFO buffer (req 2, req 3)
int writeSD() {
  t1.terminate();
  printf("\nWriting...\n");

  // Ensure SD can be initialised (req 10)
  if (0 != sd.init()) {
    redLED = 1;
    printf("Initialisation failed \n");
    return -1;
  }

  // If SD is mounted, write is attempted
  FATFileSystem fs("sd", &sd);
  FILE *fp = fopen("/sd/data.txt", "w");
  if (fp == NULL) { // Check for ability to write to file on SD
    redLED = 1;
    error("Could not open a file for write\n");
    sd.deinit();
    ThisThread::yield();
    t1.start(measureSensorData);
    return -1;
  } else {
    // Green LED flashes for duration of write (req 13)
    // Copy FIFO buffer to text file (req 2, req 3)
    for (int i = f.start; i < f.length; i++) {
        greenLED = !greenLED;
        string x = remove(f);
        int n = x.length();
        char char_array[n+1];
        strcpy(char_array, x.c_str());
        fprintf(fp, "%s\n", char_array);
    }

    // Close everything off in order
    fclose(fp);
    printf("SD Write done...\n");
    greenLED = 0;
    sd.deinit();
    ThisThread::yield();
    t1.start(measureSensorData);
    return 0;
  }
}
```

I terminate the first **thread***, t1, which measures sensor data, to enable the SD write. I must do this because otherwise, the serial terminal is occupied forever by measureSensorData().

I call a **blocking** method[=] init(), to determine whether SD is ready to be written to. If not, I report an error and **light the red LED**. Similarly, I test whether I can write to a file on the SD card. If I cannot, I **unblock**[=] with deinit(), stop the current **thread*** and resume the first **thread***.

If I can, then I commence reading through the FIFO buffer until I reach the end of it, while also flashing the **green LED** on and off.** I should note here that the **green LED** is on** prior to calling writeSD().

I had to convert the strings to a char_array in order to copy the data across. Thanks to the structure of my ring buffer, this was not too difficult.

Finally, I tidied everything up: I closed the file; I logged the success to the serial terminal; I switched the **green LED** off**; I **uninitialised**[=] the SD; I stopped the second **thread**; and I resumed the first **thread**.*

I believe that this particular function was written to a satisfactory quality. Please note that I have **logged** each step to the serial terminal.

* Requirement 6 - Program must be multithreaded

[=] Requirement 10 - Mitigate against the occurrence of deadlocks

** Requirement 13 - LED functionality of SD card interactions

Requirement 12 - Log messages to the serial terminal and light red LED if error occurs

REQUIREMENT 3 - CREATE AN INTERNAL FIFO MEMORY BUFFER

```
// FIFO structure for req 2
struct FIFO {
    public:
      int start = -1;
      int length = 0;
      int end = -1;
      string* data = new string[16];

      bool isFull(FIFO f) {
         return length >= 10;
      }

      bool isEmpty(FIFO f) {
         return length <= 0;
      }
};

// Global FIFO buffer (req 3)
FIFO f;
```

To the left is the structure of my FIFO internal memory buffer. As you can see, the buffer initially has the same beginning as it does the end. This is because at the start, there will be no data in it. The length is also measured and starts off at 0 because, again, there is no data in the array yet. I also have two functions in the structure to easily test if the buffer is full or empty.

I decided it would be best to initialise the buffer outside of all functions, so that when the FIFO is called it is using the same buffer throughout.

I also have two other functions for this buffer: add and remove.

```
// add function for FIFO buffer (req 3) adds new data unless buffer is full
static void add(FIFO f, string s) {
    if (f.isFull(f)) {
        error("Ring buffer is full - cannot add more data to be written");
        redLED = 1;
    } else {
        if (f.isEmpty(f)) { // If buffer empty, reset start and add data there
        f.start = f.end = 0;
        f.data[f.start] = s;
        } else {
            if (f.end == 15) {  // If end of buffer at end of array, loop around
                f.end = -1;
            }
            f.end = f.end +1;   // Add data to end of buffer
            f.data[f.end] = s;
        }
        f.length++; // Keep track of length
    }
}
```

```
// remove function for FIFO buffer (req 3) removes the oldest data unless buffer is empty
static string remove(FIFO f) {
    if (f.isEmpty(f)) {
        error("Ring buffer is empty - no data to write");
        redLED = 1;
        return f.data[0];
    } else {
        string data = f.data[f.start];  // Oldest data is found at start
        f.start = f.start+1;
        if (f.start == 16) {
            f.start = 0;
        }
        f.length--; // Keep track of length
        return data;
    }
}
```

Add will take a string s and add it to the supplied FIFO buffer f, provided the buffer isn't full already. The **red LED** will light if the buffer is full, and the error will be **logged** to the serial terminal . Data is either added to the start or to the next point from the end, depending on whether or not the buffer is empty. If we find that the end is at the last point in the array, we reset the end to the start of the array - 1, before we change the end pointer to the next across and add the data to the array. This resetting to the start of the array is what makes this buffer a ring buffer. Finally we increase the length by 1, since the buffer has an extra one bit of data.

The remove function is similar, except it doesn't require checking if the buffer is full. If the buffer is empty, of course the program will report and **log** the error and the **red LED** will be on . Otherwise, the data from the start of the array (i.e. the oldest data) will be copied in order to be returned. The start pointer will be altered as the end pointer was in the add function. The length will be decreased, as when the length was increased before.

There is some problem with the buffer, though. It seems to hinder writing to the SD card.

REQUIREMENT 5 - USING C++ AND ONLY MBED-OS LIBRARIES

```
#include "BMP280_SPI.h"
#include "../lib/uopmsb/uop_msb_2_0_0.h"
#include "FATFileSystem.h"
#include "SDBlockDevice.h"
#include "mbed.h"
#include <iostream>
#include "rtos.h"
#include "time.h"
#include <chrono>

using namespace std;
using namespace uop_msb_200;
```

None of these libraries required downloading or importing. They were all found in mbed-os 6.10.0 and libraries provided by the module.

I used 'BMP280_SPI.h' to measure sensor data. The 'uop_msb_200' library was used for locating certain buttons, sensors and LEDs on the board. The 'FATFileSystem' was used in my 'writeSD' method, as was the 'SDBlockDevice', which was particularly useful in summoning blocking methods to fulfil requirement 10. For writing to the serial terminal (and potentially reading inputs, which I didn't get to), 'iostream' was used. 'Rtos.h' was used for threading. Finally, 'time.h' and '<chrono>' were used for reading the system clock.

REQUIREMENT 7 - MINIMISE POWER CONSUMPTION

I believe I satisfied this requirement at the same time I satisfied Requirement 6. By making threads sleep, terminating threads, swapping threads and multithreading, I believe I minimised the power consumption of the program.

REQUIREMENT 11 - PROPERLY INDENT AND COMMENT THE SOFTWARE'S CODE

There are plenty of examples above of me indenting and commenting on the code. I will say that I wasn't sure how to properly indent the structure's 'public:' line and lines below it. Formatting the document implies that 'print:' ought to be on the same indent as the declaration of the structure, but to me it looks and feels wrong. I hope that my way of indenting it makes it more comfortable and readable.

As for writing the author's name and group at the top of the file, you will see that I have. I only have one .cpp file to submit which I have actually written myself, so have only written my name and group once.

```
// COMP2004 Coursework
// All code by Bethany Marie Ward (req 11)
// Student Reference Number : 10605926
// Group: D
// I hope that I have satisfied reqs 1, 2, 3, 5, 6, 7, 10, 11, 12, 13 [worth 72
// marks]. Reqs 5, 7, 11 and 12 are reqs that I have considered passive, and so
// have not been identified in comments. I have not attempted reqs 4, 8 and 9
// [worth 28 marks].

```

CONCLUSION

Although my code does not fully work, I do believe that my logic is sound. Most of my issues likely lie in the finer details of the code. The code I have supplied is all in one file, ‘main.cpp’, so it should be straightforward to run. I am optimistic and look forward to hearing my results.