ECE 6483 : Real Time Embedded Systems

Embedded challenge 2022: Blood pressure monitor using STM32F429ZI

Due 16th May, 2022

Utkarsh Shekhar
us450@nyu.edu

Siddharth Kandpal
sk8944@nyu.edu

1. Problem Statement:

The challenge asked students to implement a blood pressure monitor using a STM32F-series programmable microcontroller. The agenda was to have a blood pressure tubing attached to a hose which connects to a pressure sensor allowing us to accurately map the test subjects two primary attributes, namely blood pressure and heartrate using our microcontroller. Our idea behind this project revolves around simplicity, & affordability, meaning this genuine equipment can be used by anyone to help them monitor their health in the pursuit to live better lives.

2. Device Specifications:

The microcontroller in use is STM32F429ZI. These devices are based on the high-performance Arm® Cortex-M4 32-bit RISC core operating at a frequency of up to 180 MHz The Cortex-M4 core features a Floating-point unit (FPU) single precision which supports all Arm® single-precision data-processing instructions and data types. It also implements a full set of DSP instructions and a memory protection unit (MPU) which enhances application security.

These devices incorporate high-speed embedded memories (Flash memory up to 2 Mbyte, up to 256 Kbytes of SRAM), up to 4 Kbytes of backup SRAM, and an extensive range of enhanced I/O's and peripherals connected to two APB buses, two AHB buses and a 32-bit multi AHB bus matrix.

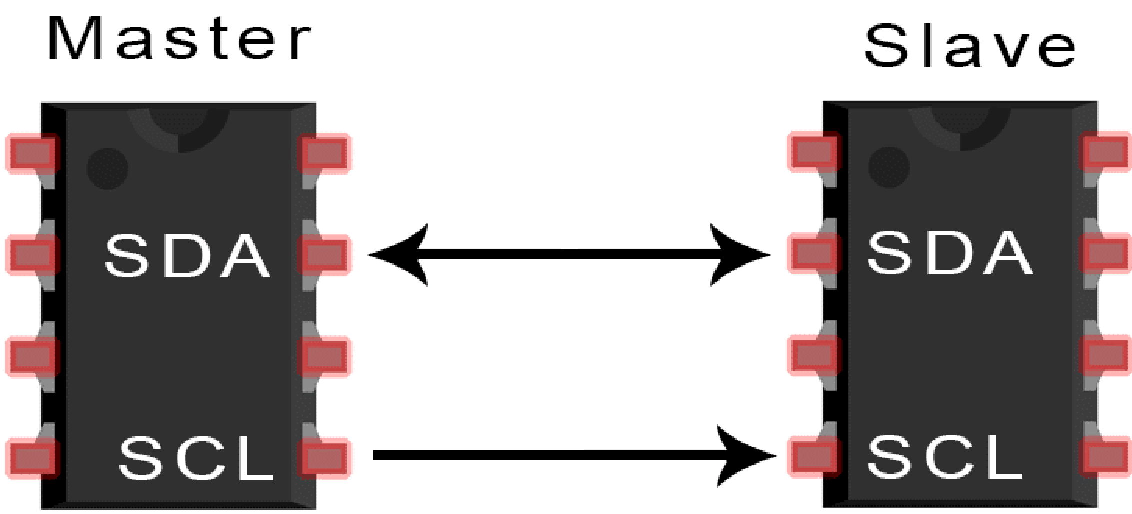
All devices offer three 12-bit ADCs, two DACs, a low-power RTC, twelve general-purpose 16-bit timers including two PWM timers for motor control, two general-purpose 32-bit timers. They also feature standard and advanced communication interfaces.

The sensor in use is Honeywell MPRLS0300YG00001BB, which can run on a voltage supply range of around1.8 ~ 3.6V.

3. Device interface logic

Our program makes use of a protocol known as I^2C , also known as Inter-Integrated circuit. The Inter-Integrated Circuit (I^2C) Protocol is a protocol intended to allow multiple "peripheral" digital integrated circuits ("chips") to communicate with one or more "controller" chips. Like the Serial Peripheral Interface (SPI), it is only intended for short distance communications within a single device. Like Asynchronous Serial

Interfaces (such as RS-232 or UARTs), it only requires two signal wires to exchange information.



I^2C protocol in a nutshell

Why I^2C ?

I^2C requires a mere two wires, like asynchronous serial, but those two wires can support up to 1008 peripheral devices. Also, unlike SPI, I^2C can support a multi-controller system, allowing more than one controller to communicate with all peripheral devices on the bus (although the controller devices can't talk to each other over the bus and must take turns using the bus lines).

Data rates fall between asynchronous serial and SPI; most I^2C devices can communicate at 100kHz or 400kHz. There is some overhead with I^2C; for every 8 bits of data to be sent, one extra bit of meta data must be transmitted.

Each I^2C bus consists of two signals: SDA and SCL. SDA (Serial Data) is the data signal and SCL (Serial Clock) is the clock signal.

4. Source code

Following are the series of snippets describing our code which drives the device. The main.cpp file will be attached along with the report for ease of validation and verification.

```
#include "mbed.h"
#include "drivers/LCD_DISCO_F429ZI.h"
#define BACKGROUND 1
#define FOREGROUND 0
#define GRAPH_PADDING 5

I2C i2c(PC_9, PA_8);

// LCD Section
LCD_DISCO_F429ZI lcd;

//Buffer for LCD texts
char display_buf[3][60];
uint32_t graph_width=lcd.GetXSize()-2*GRAPH_PADDING;
uint32_t graph_height=graph_width;

// LCD setup function borrowed from Demo 8
void setup_background_layer(){
  lcd.SelectLayer(BACKGROUND);
  lcd.Clear(LCD_COLOR_BLACK);
  lcd.SetBackColor(LCD_COLOR_BLACK);
  lcd.SetTextColor(LCD_COLOR_GREEN);
  lcd.SetLayerVisible(BACKGROUND,ENABLE);
  lcd.SetTransparency(BACKGROUND,0x7Fu);
}

// LCD setup function borrowed from Demo 8
void setup_foreground_layer(){
    lcd.SelectLayer(FOREGROUND);
    lcd.Clear(LCD_COLOR_BLACK);
    lcd.SetBackColor(LCD_COLOR_BLACK);
    lcd.SetTextColor(LCD_COLOR_LIGHTGREEN);
}

// LCD setup function borrowed from Demo 8
void draw_graph_window(uint32_t horiz_tick_spacing){
  lcd.SelectLayer(BACKGROUND);

  lcd.DrawRect(GRAPH_PADDING,GRAPH_PADDING,graph_width,graph_width);
  for (uint32_t i = 0 ; i < graph_width;i+=horiz_tick_spacing){
    lcd.DrawVLine(GRAPH_PADDING+i,graph_height,GRAPH_PADDING);
  }
}
```

```
// LCD setup function borrowed from Demo 8
uint16_t mapPixelY(float inputY,float minVal, float maxVal, int32_t minPixelY, int32_t maxPixelY){
  const float mapped_pixel_y=(float)maxPixelY-(inputY)/(maxVal-minVal)*((float)maxPixelY-(float)minPixelY);
  return mapped_pixel_y;
}

void getDiastolic(float MAP, int mapIndex, int endIndex, float *slopeArray, int *bestSlopeIndex)
{
    float diaSlopeThreshold = 0.8 * MAP;
    float slopeDiff = 0;
    float minDiffSlope = (float)INT32_MAX;

    int i = 0;

    for (i = mapIndex + 1; i < endIndex; i++){
    if ((slopeArray[i] >= 0.0) && (slopeArray[i] < diaSlopeThreshold))
    {
        slopeDiff = diaSlopeThreshold - slopeArray[i];
        if (slopeDiff < minDiffSlope)
        {
        minDiffSlope = slopeDiff;
        (*bestSlopeIndex) = i + 1;
        }
    }
    }
}

//find the index of the sytolic value
void getSystolic(float MAP, int mapIndex, float *slopeArray, int *bestSlopeIndex)
{
    float sysSlopeThreshold = 0.5 * MAP;
    float slopeDiff = 0;
    float minDiffSlope = (float)INT32_MAX;

    int i = 0;

    for (i = 0; i < mapIndex; i++)
    {
    if ((slopeArray[i] >= 0.0) && (slopeArray[i] < sysSlopeThreshold))
    {
        slopeDiff = sysSlopeThreshold - slopeArray[i];
        if (slopeDiff < minDiffSlope)
```

```c
        {
        minDiffSlope = slopeDiff;
        (*bestSlopeIndex) = i + 1;
        }
    }
    }
}

// Get the mean pressure based on the reading till now stored in the array
void getMeanPressure(float *pressureArray, int endIndex, float *timeArray, float *slopeArray, float *MAP, int *mapIndex)
{

    int i = 0;
    float pressureDiff = 0.0;
    float timeDiff = 0;

    for (i = 1; i < endIndex; i++)
    {
    pressureDiff = pressureArray[i] - pressureArray[i - 1];
    timeDiff = (timeArray[i] - timeArray[i - 1]);

    // Calculate Slope
    if (timeDiff != 0)
    {
        slopeArray[i - 1] = abs((pressureDiff / timeDiff));
    }
    // Calculate Max +ve Slope
    if (slopeArray[i - 1] > (*MAP))
    {
        (*MAP) = slopeArray[i - 1];
        (*mapIndex) = i;
    }
    }
}

//Find heart rate from the systolic and diastolic index values
void getHeartRate(int bestSSlopeIndex, int bestDSlopeIndex, float *slopeArray, float *timeArray, int *heartRate)
{
    int hr = 0;

    for (int i = bestSSlopeIndex; i <= bestDSlopeIndex; i++){
        if (slopeArray[i] >= 0.0)
            hr++;
```

```
    }
    (*heartRate) = ((hr / (timeArray[bestDSlopeIndex] - timeArray[bestSSlopeIndex])) * 60);
}

// Function to reset buffer memory and clear the line of LCD
void resetLCDBuffer()
{
    memset(display_buf[0], ' ', 60);
    memset(display_buf[1], ' ', 60);
    memset(display_buf[2], ' ', 60);
    lcd.DisplayStringAt(0, LINE(16), (uint8_t *)display_buf[0], LEFT_MODE);
    lcd.DisplayStringAt(0, LINE(17), (uint8_t *)display_buf[1], LEFT_MODE);
    lcd.DisplayStringAt(0, LINE(18), (uint8_t *)display_buf[2], LEFT_MODE);
}

//Print the buffers on the LCD
void printBufferToLCD()
{
    lcd.DisplayStringAt(0, LINE(16), (uint8_t *)display_buf[0], LEFT_MODE);
    lcd.DisplayStringAt(0, LINE(17), (uint8_t *)display_buf[1], LEFT_MODE);
    lcd.DisplayStringAt(0, LINE(18), (uint8_t *)display_buf[2], LEFT_MODE);
}

int main()
{
    thread_sleep_for(500);

    //Define Addresses and Sensor variables
    int sensorWriteAddr = (0x18 << 1);
    const char sensorOutReg[] = {0xAA, 0x00, 0x00};
    char sensorPresOut[4] ;
    float sensorOut = 0;
    sensorPresOut[0] = 0x00;

    //Define Pressure Variables
    int pressure = 0, pressureChange = 0, prevPressure = 0;
    char *pressureMessage = "";
    char optimal[50] = "Deflation correct";
    char fast[50] = "Deflation rate high";
    char slow[50] = "Deflation rate low";
    float pressureArray[150];
    float plotArray[300];
    float presSlopeArray[150] = {1.0};
```

```
    float meanPressure = 0.0;

    //Time and tick variables
    float timeArray[150];
    int counter = 0;
    int plotCounter = 0;
    Timer timerVar;

    //Define indexes for getting values from arrays
    int meanPresIndex = 0;
    int bestSysSlopeIndex = 0;
    int bestDiaSlopeIndex = 0;

    //HeartRate Variables
    int heartRate = 0;

    //Sensor limits as described in the sheet
    float oMin = 419430;
    float oMax = 3774873;
    float pMin = 0.0;
    float pMax = 300.0;

    bool increase = true, decrease = false;

    //Setup the LCD as done in Demo 8
    setup_background_layer();

    setup_foreground_layer();

    draw_graph_window(10);
    int graphTick = 0;
    int time_ms = 0;

    lcd.SelectLayer(FOREGROUND);
    timerVar.start();
    while(1)
    {
        //Decide the state of user's attempt using the pressure value
        if (pressure > 151){
            increase = false;
        }

        if (!increase && pressure < 151){
```

```
        decrease = true;
    }
    if (pressure < 30 && decrease){
        break;
    }

    //Using the Method 2 in sensor, Find the output values and generate pressue values
    i2c.write(sensorWriteAddr, sensorOutReg, 3);
    i2c.read(sensorWriteAddr, &sensorPresOut[0], 4);
    thread_sleep_for(5);
    sensorOut = ((sensorPresOut[1] << 16) | (sensorPresOut[2] << 8) | (sensorPresOut[3]));
    pressure = (((sensorOut - oMin) * (pMax - pMin)) / (oMax - oMin)) + pMin;
    pressureChange =  prevPressure - pressure;

    //Change LCD message based on pressure change
    if (pressureChange >= 2 && pressureChange <= 4)
    {
        pressureMessage = optimal;
    }
    else if (pressureChange > 4.0)
    {
        pressureMessage = fast;
    }
    else
    {
        pressureMessage = slow;
    }

    //Plot the graph of the pressure
    time_ms = timerVar.read_ms();
    plotCounter++;
    plotArray[plotCounter] = pressure;
    for(graphTick = 0; graphTick < plotCounter; graphTick++){
        const uint32_t target_x_coord=GRAPH_PADDING+(graphTick + 10);
        const uint32_t old_pixelY=mapPixelY(plotArray[graphTick],-2,200,GRAPH_PADDING,GRAPH_PADDING+graph_height);
        const uint32_t new_pixelY=mapPixelY(plotArray[graphTick],-2,200,GRAPH_PADDING,GRAPH_PADDING+graph_height);
        lcd.DrawPixel(target_x_coord,old_pixelY,LCD_COLOR_BLACK);
        lcd.DrawPixel(target_x_coord,new_pixelY,LCD_COLOR_RED);
    }

    //Start the array storage when user begins decresing pressure
    if (decrease)
    {
```

```
            //If the process takes more than 75 sec (150*.5) abandon the run as values likely to be wrong
            if(counter == 150)
            {
                resetLCDBuffer();
                snprintf(display_buf[0],60,"Abandoning reading ");
                snprintf(display_buf[1],60,"time limit is 1.5 min ");
                snprintf(display_buf[2],60,"Abandoning reading ");
                printBufferToLCD();
                break;
            }
            //Print out cue to the user to help with pressure process
            resetLCDBuffer();
            snprintf(display_buf[0],60,"Pressure is %d",(int)pressure);
            snprintf(display_buf[1],60, "Changed %d", (int)pressureChange);
            snprintf(display_buf[2],60, pressureMessage);
            printBufferToLCD();

            pressureArray[counter] = pressure;
            timeArray[counter] = time_ms / 1000;
            counter++;
        }
        else
        {
            //Print out cue to the user to not exceed pressure too much
            if(pressure > 151){
                resetLCDBuffer();
                snprintf(display_buf[0],60,"Pressure is %d",(int)pressure);
                snprintf(display_buf[1],60, "Decrease pressure");
                printBufferToLCD();
            }
            else{
                //Print out cue to the user to reach the 150 threshold
                resetLCDBuffer();
                snprintf(display_buf[0],60,"Pressure is %d",(int)pressure);
                snprintf(display_buf[1],60, "Increase till 150");
                printBufferToLCD();
            }
        }
        //Save the current pressure value as the previous one
        prevPressure = pressure;
        thread_sleep_for(1000);  //Take readings every 500ms
    }

```

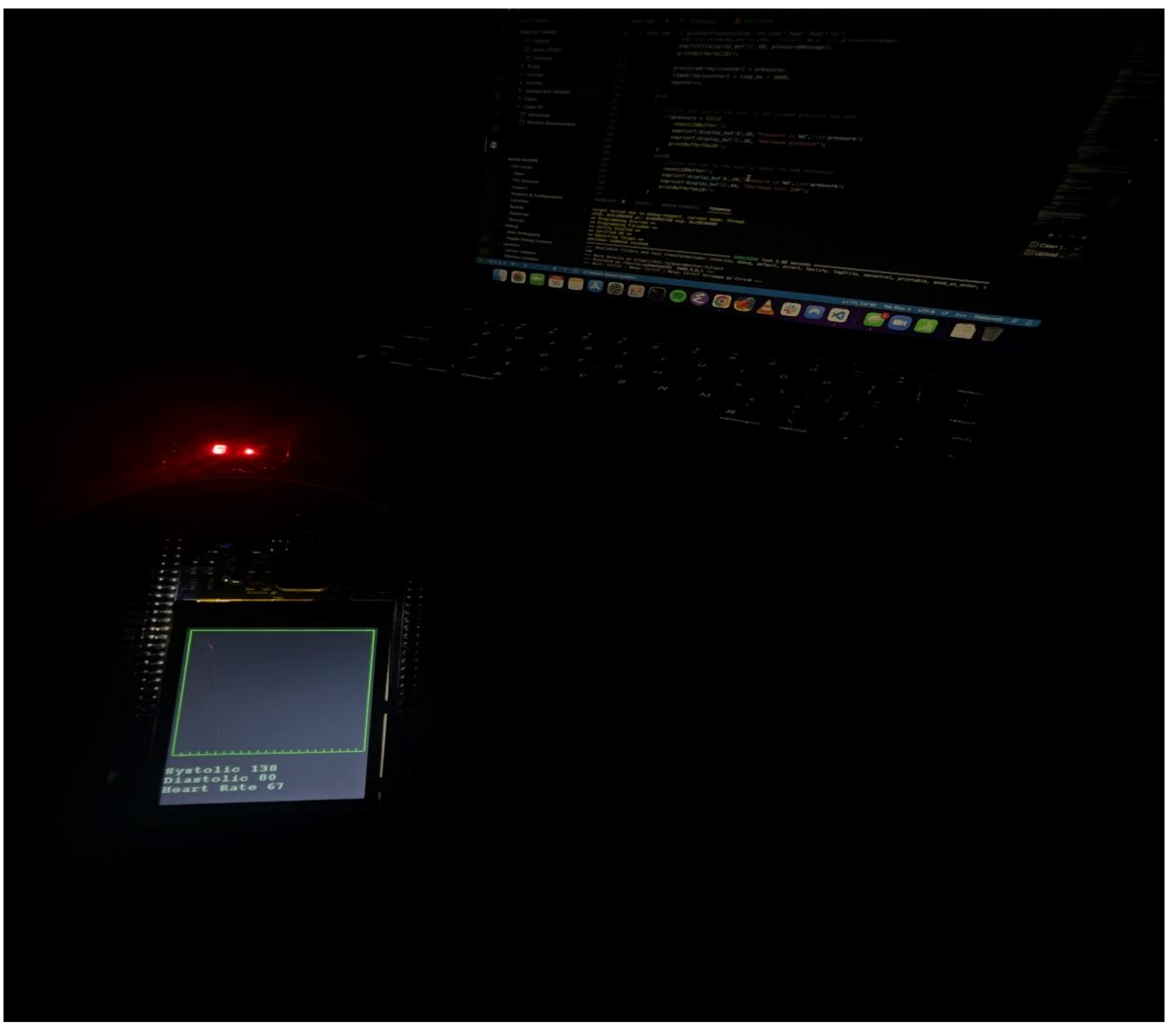
```

    //Get The values from the data recieved and print them onto the buffer
    timerVar.stop();
    getMeanPressure(pressureArray, counter, timeArray, presSlopeArray, &meanPressure, &meanPresIndex);
    getSystolic(meanPressure, meanPresIndex, presSlopeArray, &bestSysSlopeIndex);
    getDiastolic(meanPressure, meanPresIndex, counter, presSlopeArray, &bestDiaSlopeIndex);
    getHeartRate(bestSysSlopeIndex, bestDiaSlopeIndex, presSlopeArray, timeArray, &heartRate);
    resetLCDBuffer();
    snprintf(display_buf[0],60,"Systolic %d",(int)pressureArray[bestSysSlopeIndex]);
    snprintf(display_buf[1],60, "Diastolic %d", (int)pressureArray[bestDiaSlopeIndex]);
    snprintf(display_buf[2],60, "Heart Rate %d", heartRate);
    printBufferToLCD();
}
```

5. Results

After numerous trial and error instances, we were able to breakthrough the challenge which was verified through our results shown below. The funda is simple, our device asks the user to increase pressure through the cuff (in mmHg), displaying the value of current pressure, as the user continues to inflate. On hitting a pressure value of 150, we ask the user to decrease the pressure at a drop rate of about 2~4. The LCD indicates the drop rate and would display if the user has the valve too loose, in which case the pressure drops too rapidly. Remember, the BP & Heartrate readings are supposed to be taken when the user is at ease, and any form of jitter in such cases might lead to a miscalibration of results.

Here's what the outcome looks like when the user map's their reading on the device (dimmed the lights because of dim LCD resolution. P.S: drive link attached showing working) :



At first glance, we notice the LCD displays the Systole, Diastole and Heartrate readings. Also, the process of inflation and deflation is displayed through a graph-esque figure which basically resembles pixel-mapping on the LCD. This graph indicates the rise and fall in pressure by our sensor, trying to calculate the readings through pressure-formula obtained from the sensor guide. Holistically, we need the strap inflated to such an extent that it cuts off blood flow in our arm and then gradually measures the slow inflow as the cuff relaxes, calculating the numbers.

6. Conclusion

In a nutshell, we were able to obtain close to ideal values with constant precision maintained through trial & error's, which gave us insight about how a simple microcontroller can perform such tasks provided its programmed properly to do so.

7. References and Acknowledgement

We would like to thank Professor Matthew Campisi and the TA's, who provided us with the opportunity to learn about RTOS, Embedded and such devices alike. Also, a moment to thank my fellow group member who worked alongside me, helping me achieve desirable results to present on the plate for this amazing challenge.

References: Mbed official website, Honeywell sensor guide & class notes.

Code is attached along with the submission report as a "main.cpp" file.

Drive link for the video of our device ~1min appx:

https://drive.google.com/drive/folders/1IYT_w52EdR7ixzOpO0sZUndnWQ0b8Lqe?usp=sharing

YouTube link for the video ~1min appx:

https://www.youtube.com/shorts/W7dmVhTjNCA

Drive link for the zip-file containing our project code:

https://drive.google.com/drive/folders/1tgItI1HA25R_feV1yvDEd0T58d0VNGNp?usp=sharing

Thank you!