Heart Monitor

Implementation & Design Report

Name: Youssef A. Farag
ID: 900162758

1. Hardware

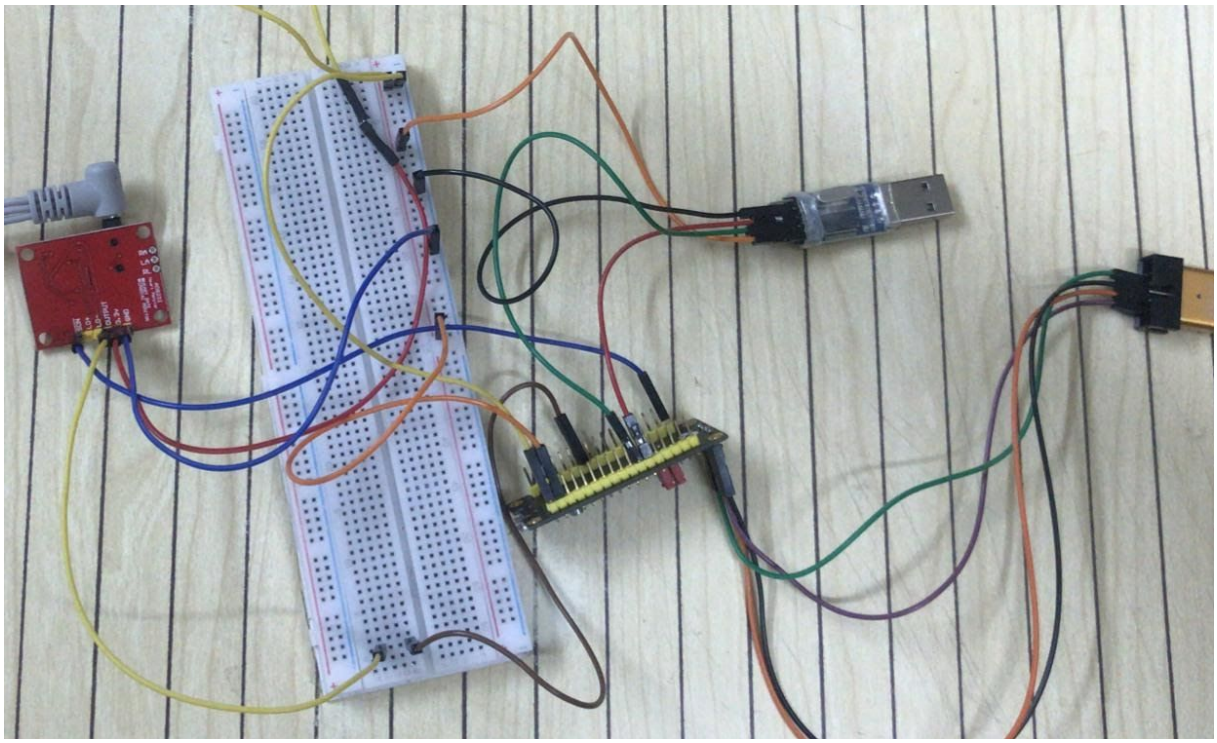
As provided, I am using the STM BlackPill (STM32F103C8) as the microcontroller, connected to it the AD82332 (Heart Rate monitor) with its connection of three ECG electrodes. As the AD82332 is providing analog signals, I am using the ADC inside the microcontroller to convert the ECG signals to digital and be able to process them. Also, I am using the St-link to load the c code; besides, the USB-to-TTL is used to facilitate the UART connection. In the next part, I will elaborate on each connection briefly.

Microcontroller - Heart Monitor connection:

- The GND pin in the Hear Monitor is connected to the main ground provided from the Microcontroller
- The VDD pin in the Hear Monitor is connected to the main VDD provided from the Microcontroller
- The output pin in the Hear Monitor is connected to the MC's ADC's input pin (A1) to allow the ADC to digitalize the analog signal coming from the monitor
- The SDN pin to turn the Heart monitor off when not needed (when there is no sampling request is issued) is connected to a GPIO (output - B11) to utilize power and increase efficiency.

Microcontroller - PC:

- ST-Link is used to loading the c code on the MC; VDD, GND, SWDIO, SWCLK are connected to their respective pins in the MC.
- USB-to-TLL is used to facilitate the UART connection; VDD, GND, RX, TX are connected to their respective pins in the MC, noting that the RX and TX are reversed in such connection.



2 Embedded C Code

As mentioned in the proposal provided earlier, the chosen software architecture design is round robin with interrupts; therefore, the c code consists of the while loop in the main.c along with the interrupt handlers that provide the main loop with the needed data to fetch the sampling rate from the user, start the sampling from the monitor, collect the data from ADC and display it to the user within the time frame of 60 seconds. In the next part, I will explain briefly the variables used in the code, the main while loop, and the interrupt handlers used for the project.

1. Variables:
    - char s[8]; Array of characters that have been sent from the user to the MC through tht UART to indicate the start and end of sampling command and sampling rate; usually in the form of 'ssXXXXss' where XXXX is the sample rate.
    - int flg = 0; Flag1 to indicate the start of conversion, turned to one once the input (s array) is filled up with the 8 characters; in other words, when the last 'ss' is received.
    - int ctt = 0; Counter that starts to count after the start of sampling (flg =1) and counts each time the timer handler is triggered aka each sample taken. It is used to count the number of samples taken to facilitate the stopping condition. For example, if the sample rate is S samples / second, then the stopping condition for sampling is ctt > S * 60 (1 minute worth of samples)
    - char out[5]; Array of characters that receive the data out of the ADC for each conversion and transmit it over the UART in a readable manner
    - int flg3 = 0; Flag3 to indicate the end of the sampling command and start of the sampling process and timer. It is triggered when the ending chars are sent ('ss').
2. Handlers:
    - UART Handler: It is basically to enable the receiving of the sampling command (start sample and sample rate) at any point of time outside the main while loop

```
void USART1_IRQHandler(void)
{
        HAL_UART_IRQHandler(&huart1);
        HAL_UART_Receive_IT(&huart1,(uint8_t*)&s,sizeof(s));
}
```

    - Systick Handler: Enables the timer to raise an interrupt any time it reaches its period. It also starts incrementing the ctt counter and starts the ADC to begin collecting data from the monitor if the start of the conversion flag is raised aka when the user has already sent the sampling command.

```
void SysTick_Handler(void)
{
  /* USER CODE BEGIN SysTick_IRQn 0 */

 if(flg == 1)
 {
    ctt++;
    HAL_ADC_Start(&hadc1);
 }

  HAL_IncTick();
  /* USER CODE BEGIN SysTick_IRQn 1 */
  /* USER CODE END SysTick_IRQn 1 */
}
```

- ADC Handler: The software enters this handler whenever the ADC_Start command is issued which is only issued in the systick handler (every time the timer reaches its period). When the timer counter reaches the specified period, it raises the ADC_Start interrupt making the software execute the ADC handler in which it collects the ADC data from the monitor and transmit it back through the UART.

```
void ADC1_2_IRQHandler(void)
{
  /* USER CODE BEGIN ADC1_2_IRQn 0 */

    res = HAL_ADC_GetValue(&hadc1);
    sprintf(out, "%d", res);
    HAL_UART_Transmit_IT(&huart1,(uint8_t*)&out,sizeof(out));

  /* USER CODE END ADC1_2_IRQn 0 */
     HAL_ADC_IRQHandler(&hadc1);
  /* USER CODE BEGIN ADC1_2_IRQn 1 */
  /* USER CODE END ADC1_2_IRQn 1 */
}
```

3. The Main While(1) loop: As in the round-robin with interrupts, the main code runs in the While loop while some higher priority functions are executed in the Handlers; therefore the while loop in the main.c is the core of the project.

   In the while loop code, we see a nested if condition that I will explain briefly.

   - **if(s[7]=='s' && flg3 == 0):** This checks if the sampling command has been sent completely (‘ssXXXXss’) to start the systick timer and the ADC collection of data. The flg3 also checks whether or not the 60 second

worth of data has been reached so that it can stop the timer after one minute.

- **if(ctt > mk*60):** mk is the sample rate (int(parsed from the char s array entered by the users; in other words it is the XXXX in the 'ssXXXXss' message. Also, remeber that ctt is the counter counting total number of samples taken. Then, this if condition simply checks if the number of samples exceeds the maximum number of samples in 60 seconds, for instance, if the sample rate is S samples/sec then ctt must reach 60*S to indicate it has collected the required amount of samples in a minute.
- **if(s[0]=='s' && flg3==0):** This condition indicates that the user has started sending data for the sampling command, basically is making the while loop to do nothing until the whole command is transmitted.
- **ELSE:** If the flg3 has been set, this means that the program has already collected the needed amount of samples in 60 seconds and therefore the while loop will return the total number of samples recorded (ctt) to 1, the sample rate (mk) to 1, clears the flg of starting the conversion (ADC) and clear the char s array to prepare it for another sampling command from the user.

```
while (1)
 {
        if(s[7]=='s' && flg3 == 0)
        {
        if(flg ==0)
        {
                strncpy(kk,s+2,4);
                sscanf(kk, "%d", &mk);
                flg = 1;
                HAL_GPIO_WritePin(GPIOB,GPIO_PIN_11,GPIO_PIN_SET);
                HAL_SYSTICK_Config((1.0/mk) * 8000000 - 1);
        }

        if(ctt > mk*60)
                flg3 = 1;
        }else if(s[0]=='s' && flg3==0)
        {
        }else
        {
                ctt = 1;
                mk = 1;
                memset(s, 0, sizeof(s));
                memset(kk, 0, sizeof(kk));
                flg3 = 0;
                flg = 0;
```

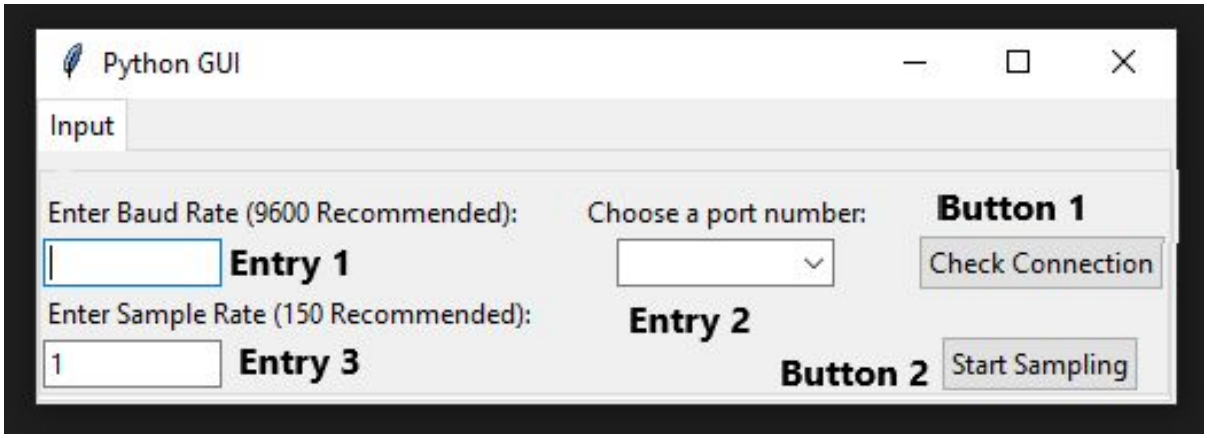
```
            HAL_GPIO_WritePin(GPIOB,GPIO_PIN_11,GPIO_PIN_RESET);
            }
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */

 }
 /* USER CODE END 3 */
}
```

3. Python serial connection

I then utilized python to establish the same UART communication used on tera term but over the python code. I also developed a simple graphical user interface to check the connection with the microcontroller, send the required sampling rate (samples/second), and view and update the sampled data dynamically over time. The next section will be divided into two parts; GUI code and serial communication code.

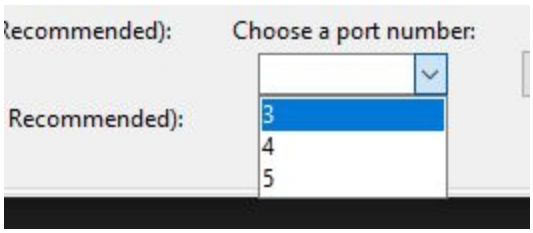
3.1 Python GUI

The GUI is simple and only contains the needed fields:



- Entry 1: A text box to allow the user to enter the required baud rate. I am using a baud rate of 9600 and that's why I have wrote it as recommended.

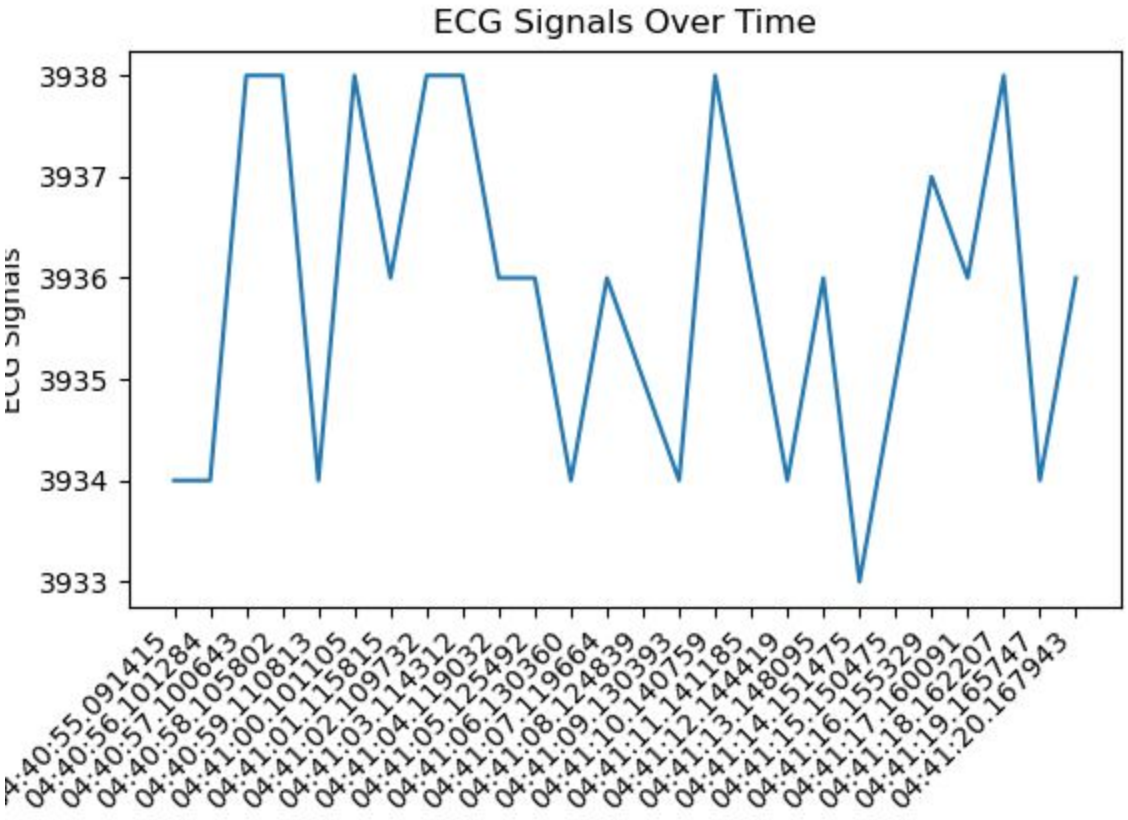
- Entry 2: A drop down list with the port available in the laptop, as I only have three USB ports, numbered from 3-5, I have listed them in a readonly drop down list where the user has to choose the used port.



- Entry 3: To enter the sampling rate ranging from 1 - 1500; I have chosen 1500 to be the maximum sampling rate as it would be unreasonable to sample with more than that. It is merely an assumption to provide some control. Also, the sampling rate can't be zero or a negative number. For both these validations, the label changes its text to alert the user.

- Button 1: Using the provided baud rate and port, a serial connection will be established once the user presses the check connection; in other words, no sampling will start without this button being pressed first. As the serial connection is started, it will be check as valid and the text in the button will offer the validation

- Button 2: This button will not function unless button 1 is valid; in other words, a connection is being established. It will use the provided sampling rate to send to the MC over the UART, signaling the start of ADC conversion. It opens another tab for the animated graph to start plotting the upcoming data from the UART.



3.2 Python Code

I will explain briefly the python code I used to do some tasks I mentioned in the above section. The GUI was created using tkinter library in python.

- Function1 - clickMe: Get the baud rate along with the port and start the serail connection using the pyserail library. It also checks if the connection is valid and modifies the Button 1 text to valid to notify the user that the connection has been established. It also raises a flag (flg) to true to notify the rest of the code that a serail communication has been established. This function is called whenever button 1 - check for connection (named action) is clicked.

```
def clickMe():
    ser.baudrate = name.get()
    ser.port = "COM"+ (number.get())
    print ("COM"+ (number.get()))
    ser.open()
    if ser.is_open:
        action.configure(text='Valid')
        global flg
        flg = True

action = ttk.Button(monty, text="Check Connection",
command=clickMe)
```

- Function2 - limitSizeRate: The function is called everytime a new entry in the sampling rate is entered. Using the trace command in the tkinter library, I can trace each new value is entered in the sampling rate entry to ensure that is is validated according to standards, not less than 1 and not more than 0. It also changes the label to the proper error message to notify the user.

```
def limitSizeRate(*args):

    if int(rateValue.get()) <= 0:
        rateValue.set('1')
        ttk.Label(monty, text="Minimum Rate: 1 - (150
recommended)").grid(column =0,row=2, sticky='W')
    elif int(rateValue.get()) > 1501:
        rateValue.set(1500)
        ttk.Label(monty, text="Maximum Rate: 1500 - (150
recommended)").grid(column =0,row=2, sticky='W')
```

```
rateValue.trace('w', limitSizeRate)
rateEntered = ttk.Entry(monty, width=12,
textvariable=rateValue)
```

- Function 3 - clickMe2: This function is called whenever the user presses the button - start sampling. It basically contains two parts. The first part (before the if condition) is to prepare the message sent across the UART to be in the required shape 'ssXXXX'. Afterwards, it checks the flg (flag indicating if the serial connection has been established) to start reading data from UART and animating it using Matplotlib over time.

```
def clickMe2():

    #cleaning the axises for new plot
    xs = []
    ys = []
    global msg2
    global flg
    #preparing the message to be sent over UART
    if int(rateValue.get()) < 10:
        msg2 = '000'+str(rateValue.get())
    elif int(rateValue.get()) < 100:
        msg2 = '00'+str(rateValue.get())
    elif int(rateValue.get()) < 1000:
        msg2 = '0'+str(rateValue.get())
     elif int(rateValue.get()) > 1000 and int(rateValue.get())
< 1501:
        msg2 = str(rateValue.get())
    elif int(rateValue.get()) > 1501:
        msg2 = '1500'

    msg ='ss' + msg2 + 'ss'
    msg = msg.encode('utf-8')
    #Check if there is a connection
    if flg == True:
        ser.write(msg)
```

```
        def animate(i, xs, ys):
            global count
            global lst
    # Read ADC data
            if count < int(msg2)*60 and flg == True:
                msgg = ser.read(4)
                count = count + 1
                msg3 = msgg.decode("utf-8")
                if msg3[0] == '0':
                        msg3 = '0'
                msg4 = int(msg3)
                lst.append(msg4)
                # Add x and y to lists

xs.append(dt.datetime.now().strftime('%H:%M:%S.%f'))
                ys.append(msg4)

                # Limit x and y lists to 200 items
                xs = xs[-200:]
                ys = ys[-200:]

                # Draw x and y lists
                ax.clear()
                ax.plot(xs, ys)

                # Format plot
                plt.xticks(rotation=45, ha='right')
                plt.subplots_adjust(bottom=0.30)
                plt.title('ECG Signals Over Time')
                plt.ylabel('ECG Signals')

       #Start the timer for 60 seconds
        timer.start()

         ani = animation.FuncAnimation(fig, animate, fargs=(xs,
ys), interval=1000.0/int(msg2))
        plt.show()
```

```
action2    =    ttk.Button(monty,    text="Start    Sampling",
command=clickMe2)
```

- During the testing phase, an error has been found that if the serial connection has been open, it basically makes the GUI to crash; therefore, I use a timer that start counting when the plot starts collecting data and for 60 seconds. Then it excuses a function to close the serial connection and turn off the flag. This means that the user has to establish a new connection every time he/she wants to collect new data.

```
def close_event():
    global flg
    flg = False
    global count
    global msg2
    global lst
    ser.close()
    count = 0
    action.configure(text='Check Connection')
    timer.stop()

timer = fig.canvas.new_timer(interval = int(msg2)*60*1000)
timer.add_callback(close_event)
```

- Calculating the BPM:
  Using the measured ECG signals, I try to roughly see the measure the interval between each two peaks and then calculate the BPM accordingly.

```
if msg4 > MAX_ECG:
     if Pulse_1 == False:
      PulseTime_1 = dt.datetime.now()
      Pulse_1 = True
     else:
      PulseTime_2 = dt.datetime.now()
      PulseInterval = PulseTime_2 - PulseTime_1
      if PulseInterval.total_seconds() >= 1:
       BPM = (1.0/PulseInterval.total_seconds()) * 60.0
       ttk.Label(monty,  text=str(BPM)).grid(column  =1,row=3,
sticky='W')
       Pulse_1 = False
```