EE2028 Assignment 2 Report

R N Nandetha Group Number: 23

Tan Jo-Wayn Lab Day: Lab[02], Tuesday Afternoon

Table of Contents

Introduction and Objectives

In this report, we will be explaining the logic behind our COvid Patients Enhanced MONitoring (COPEMON) system and the methods we used to build this system. Our objective is to create a user-friendly system that allows us to monitor a COVID patient's health by sending readings and warnings to the CHIPACU terminal program using the STM32 chip.

Flowcharts and Explanations

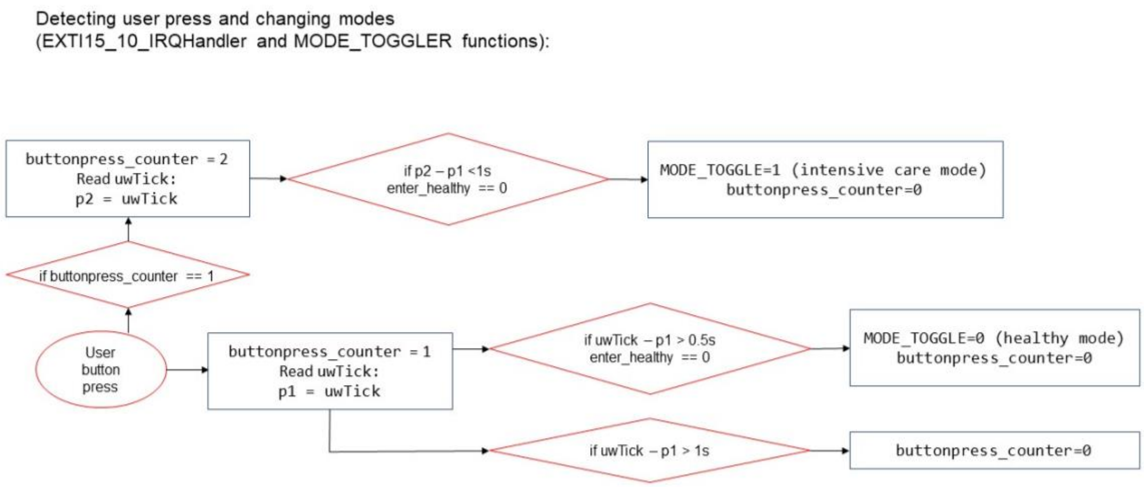

Fig. 1 Flowchart for Detecting User Press and Changing Modes

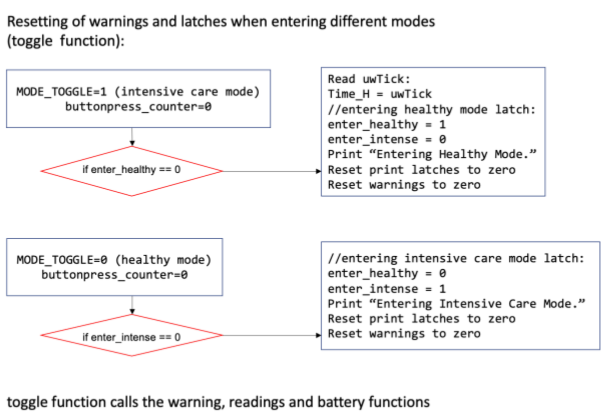

Fig. 2 Flowchart for Resetting of Warnings and Latches When Entering Different Modes

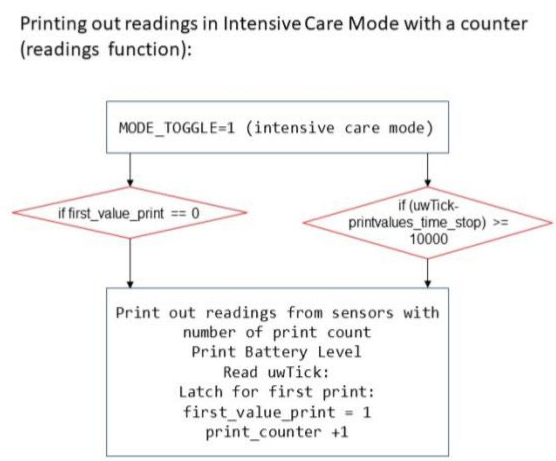

Fig. 3 Flowchart for Printing Out Readings in IC Mode with Counter

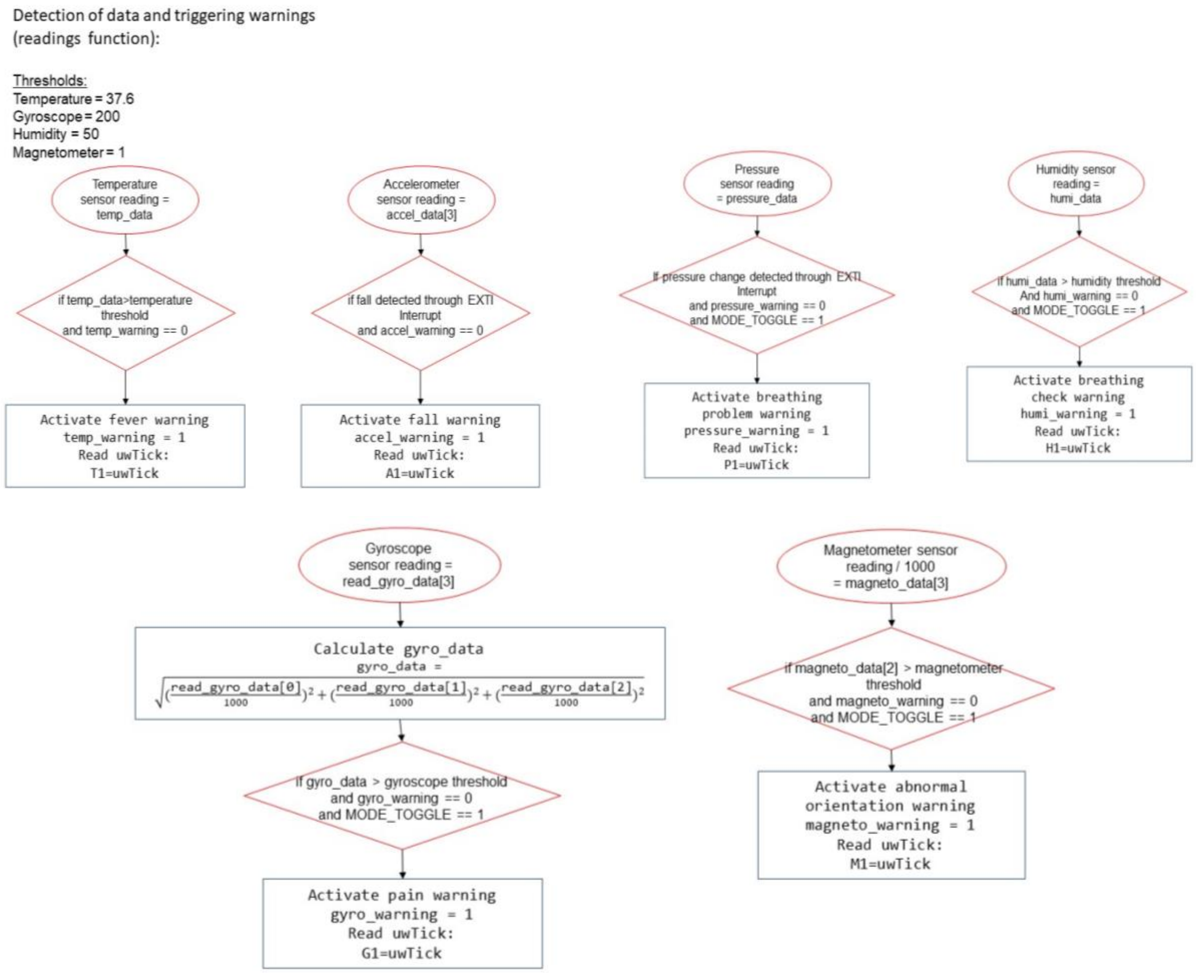

Fig. 4 Flowcharts for Detection of Data and Triggering Warnings

Printing out warnings in Intensive Care Mode
(warnings function):

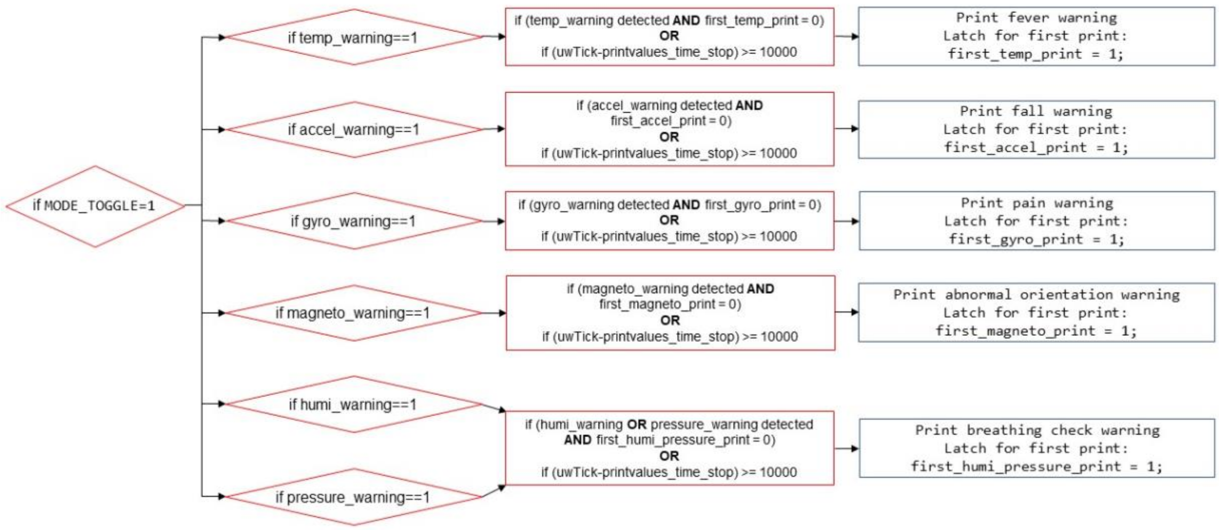

Fig 5. Flowchart for Printing Warnings in Intensive Care Mode

Printing out warnings in Healthy Mode
(warnings function):

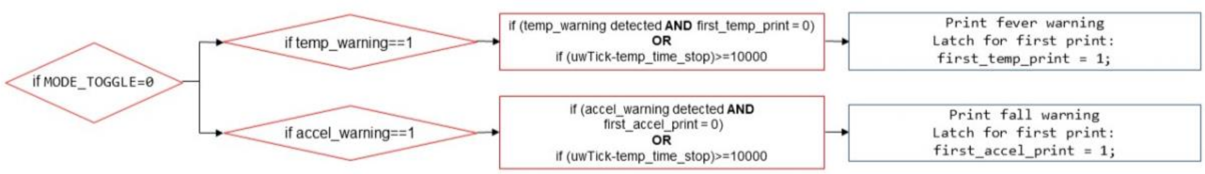

Fig 6. Flowchart for Printing Warnings in Healthy Mode

Toggling the LED when warning detected
(SysTick_Handler function):

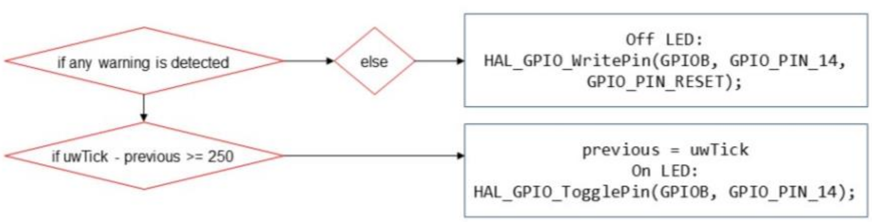

Fig 7. Flowchart for LED to Blink when Warning Triggered

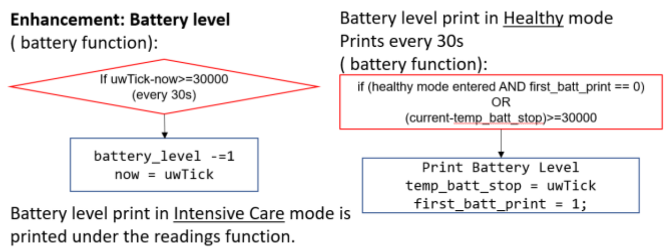

Fig 8. Flowchart for Battery Level Enhancement

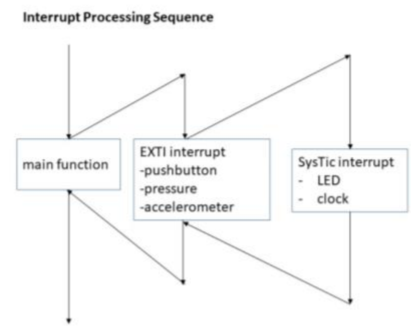

Fig 9. Interrupt Processing Sequence

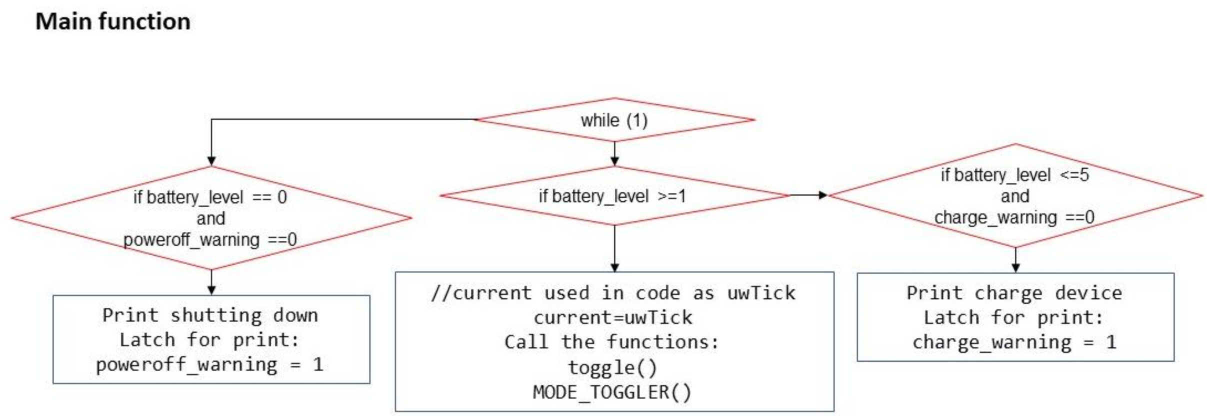

Fig 10. Flowchart for Main Function

Detailed Implementation

System Interrupts

There are two types of interrupts used in our program, namely SysTic and EXTI interrupts. The priorities given to the corresponding interrupts are as follows:

	IRQn	Preempt Priority	Sub Priority
SysTic Interrupts	SysTick_IRQn	0	0
EXTI Interrupts	EXTI15_10_IRQn	1	0

Table 1. Interrupt Priority Configuration

	Pin Name	Feature	Signal
User Button Interrupt	GPIO_PIN_13	GPIO_EXTI13	BUTTON_EXTI13_Pin
Pressure Sensor Interrupt	GPIO_PIN_10	GPIO_EXTI10	LPS22HB_INT_DRDY_EXTI10
Accelerometer Interrupt	GPIO_PIN_11	GPIO_EXTI11	LSM6DSL_INT1_EXTI11

Table 2. Interrupt Pins Configuration

SysTic Interrupt is the highest-priority interrupt in our program, given higher preempt priority than EXTI interrupts. The priority ranking of our programs are in the sequence of SysTic Interrupts > EXTI Interrupts > Main Program.

The **SysTic Interrupt Handler** in our program does the following:

1. Toggles LED, making it blink every 250ms when the following are detected:
    a. In Healthy Mode:
        i. Fever (Temperature)
        ii. Falling (Accelerometer)
    b. In Intensive Care Mode:
        i. Fever (Temperature)
        ii. Falling (Accelerometer)
        iii. Patient in Pain (Gyroscope)
        iv. Breathing Issue (Humidity, Pressure)

2. Calls `HAL_IncTick()` which increases current system Tick (`uwTick`) based on default frequency (`uwTickFreq`) of 1kHz (1 Tick = 1ms)

It is crucial that SysTic interrupts are utilised to manage time-related matters, as we want our program to run as accurately to real-time as possible. Throughout our program, there are many instances where current tick is used to manage time intervals, such as in the case of printing warnings e.g Fever is detected. The steps are as follows:

1. The current Tick is stored into a variable `temp_time_stop` once a fever warning is printed.

2. The difference between the current Tick and the variable `temp_time_stop` is constantly polled. Once the difference between the current Tick and variable `temp_time_stop` reaches 10000ms (10s), another fever warning is printed.

The **EXTI Interrupt Handler** in our program is a function that triggers user-defined reactions whenever an external interrupt request is detected by EXTI and passed to NVIC. The handler function does the following upon interrupt requests:

1. Registers button presses
    a. Through `GPIO_PIN_13`
    b. Adds 1 to `buttonpress_counter`
2. Triggers interrupts when interrupt pin `GPIO_PIN_10` is triggered by pressure sensor
3. Triggers interrupts when interrupt pin `GPIO_PIN_11` is triggered by accelerometer free-fall

<u>Button Press Interrupt</u>

`buttonpress_counter` takes value either 1 or 2 in the program, with 1 corresponding to a single button press and 2 corresponding to a double press. Current Tick is stored into variables p1 and p2 for the first and second button presses respectively. When `buttonpress_counter` is 2, the time difference between p2 and p1 is taken, and if this difference is below an interval of 1000ms (1s), mode is toggled to Intensive Care Mode within a separate function **`MODE_TOGGLER`** (**`MODE_TOGGLER`** function will be elaborated in detail later).

<u>Pressure Interrupt</u>

The LPS22HB digital output pressure sensor has an in-built interrupt pin INT_DRDY that is able to send interrupt signals to EXTI when the pressure increases beyond a certain threshold. BSP Library function SENSOR_IO_Write is used to write values to registers on the sensor device to determine which functions and modes are utilised.

The following lines are written in our code to configure the sensor to desired settings.

```
SENSOR_IO_Write(LPS22HB_I2C_ADDRESS, LPS22HB_CTRL_REG2, 0x10);
SENSOR_IO_Write(LPS22HB_I2C_ADDRESS, LPS22HB_CTRL_REG3, 0x01);
SENSOR_IO_Write(LPS22HB_I2C_ADDRESS, LPS22HB_INTERRUPT_CFG_REG, 0x89);
SENSOR_IO_Write(LPS22HB_I2C_ADDRESS, LPS22HB_THS_P_LOW_REG, 0x08);
```

The following is a control register table extracted from the LPS22HB datasheet which describes the pin register in one of its register controls. This was how we determined how to configure the device to work based on task specifications.

CTRL_REG3 (12h)

Control register 3 - INT_DRDY pin control register

7	6	5	4	3	2	1	0
INT_H_L	PP_OD	F_FSS5	F_FTH	F_OVR	DRDY	INT_S2	INT_S1

INT_H_L	Interrupt active-high/low. Default value: 0 (0: active high; 1: active low)
PP_OD	Push-pull/open drain selection on interrupt pads. Default value: 0 (0: push-pull; 1: open drain)
F_FSS5	FIFO full flag on INT_DRDY pin. Default value: 0 (0: disable; 1: enable)
F_FTH	FIFO watermark status on INT_DRDY pin. Default value: 0 (0: disable; 1: enable)
F_OVR	FIFO overrun interrupt on INT_DRDY pin. Default value: 0 (0: disable; 1: enable)
DRDY	Data-ready signal on INT_DRDY pin. Default value: 0 (0: disable; 1: enable)
INT_S[2:1]	Data signal on INT_DRDY pin control bits. Default value: 00 Refer to *Table 19*.

Table 3: LPS22HB Datasheet, Control Register 3 Pin Control Register Table

0x01 (00000001 in binary) is written to CTRL_REG3, or Control Register 3. CTRL_REG3[1:0] are INT_DRDY pin interrupt configuration control bits. According to the interrupt configuration table, we set INT_S1 to 1 in order to achieve an interrupt configuration which sends an interrupt signal when pressure is increased suddenly. (Pressure High)

INT_S2	INT_S1	INT_DRDY pin configuration
0	0	Data signal (in order of priority: DRDY or F_FTH or F_OVR or F_FSS5
0	1	Pressure high (P_high)
1	0	Pressure low (P_low)
1	1	Pressure low OR high

Table 4: INT_DRDY Pin Configuration extracted from LPS22HB Datasheet

Next, `__HAL_GPIO_EXTI_GET_FLAG(GPIO_PIN_10)` is used to receive interrupt triggers from the pressure sensor. Once an interrupt arising from increased pressure is detected, the warning latch `pressure_warning` is set to 1, and `warning()` function is called to transmit the corresponding pressure warning through UART.

Accelerometer Interrupt

The LSM6DSL Accelerometer/Gyroscope comes with built-in, programmable interrupt pins INT1 and INT2 that are able to send interrupt signals to EXTI upon certain conditions including tilt, free-fall, tap, double-tap etc.

The following lines are written in our code to configure the accelerometer to desired settings.

```
SENSOR_IO_Write(LSM6DSL_ACC_GYRO_I2C_ADDRESS_LOW, LSM6DSL_ACC_GYRO_TAP_CFG1, 0x80);
SENSOR_IO_Write(LSM6DSL_ACC_GYRO_I2C_ADDRESS_LOW, LSM6DSL_ACC_GYRO_FREE_FALL, 0x3F);
SENSOR_IO_Write(LSM6DSL_ACC_GYRO_I2C_ADDRESS_LOW, LSM6DSL_ACC_GYRO_MD1_CFG, 0x10);
```

0x80 (10000000 in binary) is written to `TAP_CFG1` which enables basic interrupts (6D/4D, free-fall, wake-up, tap). 0x3F (00111111 in binary) is written to `LSM6DSL_ACC_GYRO_FREE_FALL` to configure the threshold for free-fall function (in mg) as well as the duration of the free-fall event. 0x10 (00010000 in binary) is written to `LSM6DSL_ACC_GYRO_MD1_CFG`, which enables the routing of free-fall event on INT1 interrupt signal. MD1_CFG controls a multiplexer that is able to send signal of ‘1’ to INT1 pin, hence the naming convention.

Similar to the pressure sensor, `__HAL_GPIO_EXTI_GET_FLAG(GPIO_PIN_11)` is used to retrieve an interrupt signal upon detection of a free-fall event.

Enhancement: Battery Indicator

It is highly likely that in the design of a portable medical monitoring device, a compact power supply such as a rechargeable battery is used, hence our enhancement helps users identify the battery level left in the monitoring device to ensure the sustained operation of the device.

The battery level starts at 100% and is set to decrease by 1% every 30 seconds, simulating the power usage of a portable medical device. The battery level percentage value prints every 1% drop in healthy mode and prints every 10 seconds along with the readings in intensive care mode, as it is important that the medical device does not shut down during the critical period just because the user missed out the battery level display.

When the battery level is very low, at 5%, a warning ,’Battery Low! Please Charge Device!’ prints on the display to alert and remind users to charge the device. When the battery level reaches 0%, the system prints ‘No Battery! System Powering Down......’ before shutting down.

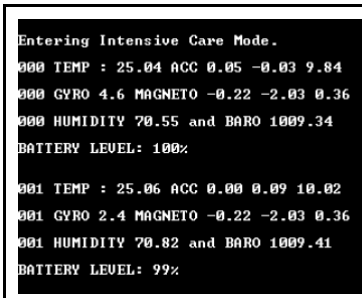

Fig 11. Battery Indicator in IC Mode

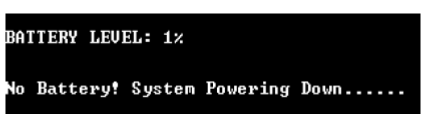

Fig 12. Battery Indicator in IC Mode

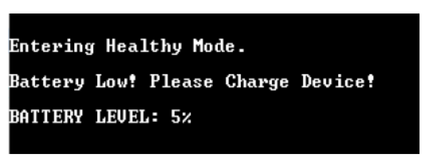

Fig 13. Battery Indicator in IC Mode

<u>Significant problems encountered and solutions proposed</u>

One realisation we made during the project is the memory-intensive nature of UART transmission. Initially, UART transmission was used to transmit the peripheral readings once for each peripheral at the intervals. This was very memory-intensive and caused our program to delay significantly, such that the readings did not transmit all at one go or had delays in the printing intervals.

To maximise the continuity of polling features (polling for temperature, etc.), UART is used sparingly, and the peripheral values are transmitted in one chunk instead of calling several transmission instances in order to minimise the use of time-consuming instructions.

Hence, the indicator ( ) is modified in the handler, and the time consuming instructions such as ( ) based on the indicator's state are running in the (main, OTHER NAME??) program

Another realisation we made while doing this project is to allow for time allowances in the execution of function such as printing. As UART telemetric transmission is very taxing on memory, we used

1. `if (current-temp_time_stop)>=10000` instead of

2. `if (current-temp_time_stop)==10000`

To poll for the next printing of warnings. By using method (1) and using a latch to indicate that the line has been stepped into, this method allows for millisecond-delays in the program. In the case of (2), the difference between the current Tick and `temp_time_stop` has to be *exactly* 10000 at the instant where the program is executing this particular line, for it to be stepped into. The problem with this is that if the hardware is currently executing another line in the program during the instant where the difference between the current Tick and `temp_time_stop` is 10000, the UART transmission will be missed if the line is not stepped into during that particular instant. By using method (1), millisecond delays will still allow the line to be stepped into and execute UART transmission of the warning, for example in the case where `(current-temp_time_stop)` value is 10002. Once the line is stepped into, the new current tick is fetched and stored into `temp_time_stop` and the cycle repeats itself.

<u>Assignment Feedback</u>

We felt that the method of instruction, which involved catering several lab sessions to help students reach the final product was extremely helpful to achieve the project requirements towards the end. The progressive mode of learning helped us understand how to interface peripherals with the board through the smaller-scale lab exercises, then make use of interrupts to drive certain functions and lastly configure telemetric transmission between the board and our PC. The sequence of the functions we learnt made sense as well because we were able to use our newfound knowledge in addition to what we have learnt, for example by the time we learnt about the use of interrupts, we had already learnt how to interface the device peripherals with the board, hence we could practice triggering interrupts using the built in accelerometer, as an example.

<u>Conclusion</u>

Overall, with the use of SysTic, IRQHandler, interrupts and multiple latches, we were able to build a system that monitors a COVID patient's health and sends signals in case of emergency. This assignment has allowed us to further explore the use of microcontrollers and learn how to program them as well as interface them with external peripherals, which will be extremely helpful in our future endeavours in the IoT field. The basic principles of hardware functionality such as interrupts and the protocoling of GPIO and I2C interfaces are transferable to the context of other hardware models, which function under similar principles.