Overview of the code

The basic idea of how the code works will be explained in this section. At the beginning when we first enter main, all the variables are declared and initialised and all the required hardware is also initialised. Then we attempt to establish a connection through wifi to a server. Once a connection has been established we then enter the main loop where all the polling of sensors and data processing and transmission will be done. While inside the while loop, we first check for the mode that is currently being used, i.e. `healthy` or `intensive`.

If the mode is `intensive` then we execute a block of code that polls the temperature, humidity, accelerometer, gyroscope, magnetometer and pressure sensor every 1 second. If it detects if any thresholds have been crossed and if so, it will set the warning flag for the sensor and will immediately send a warning message to telemetry. Every 10 seconds, it will send the data readings from the 6 sensors mentioned above to telemetry. To account for timing delays, we take note of the start tick and end tick before and after executing the block of code for intensive mode. This will allow us to more accurately implement a delay of 250ms at the end of each loop by doing:

```
difference = endcount - startcount;

delay(250-difference);
```

This is so the LED can be toggled every 250ms which is also the same as blinking at 2Hz frequency.

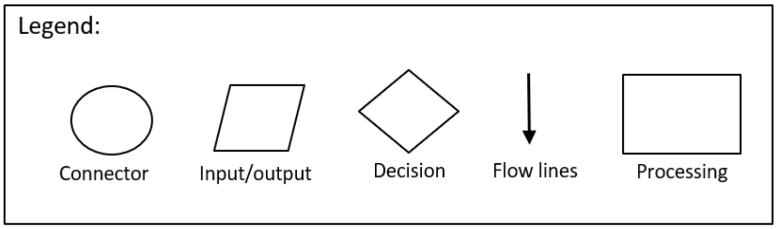
If the mode is `healthy`, code will be executed such that we only poll the temperature sensor and use raise warning flags through interrupts for the accelerometer. Upon detection of temperature readings that exceed the threshold temperature, or if the accelerometer flag is raised, a warning will be sent to telemetry immediately. If any warnings are raised, we programmed the

LED to blink at 2Hz frequency, otherwise it is turned off in `healthy` mode. And every 10 seconds we send any warnings to telemetry, otherwise we send an ‘all good’ message.

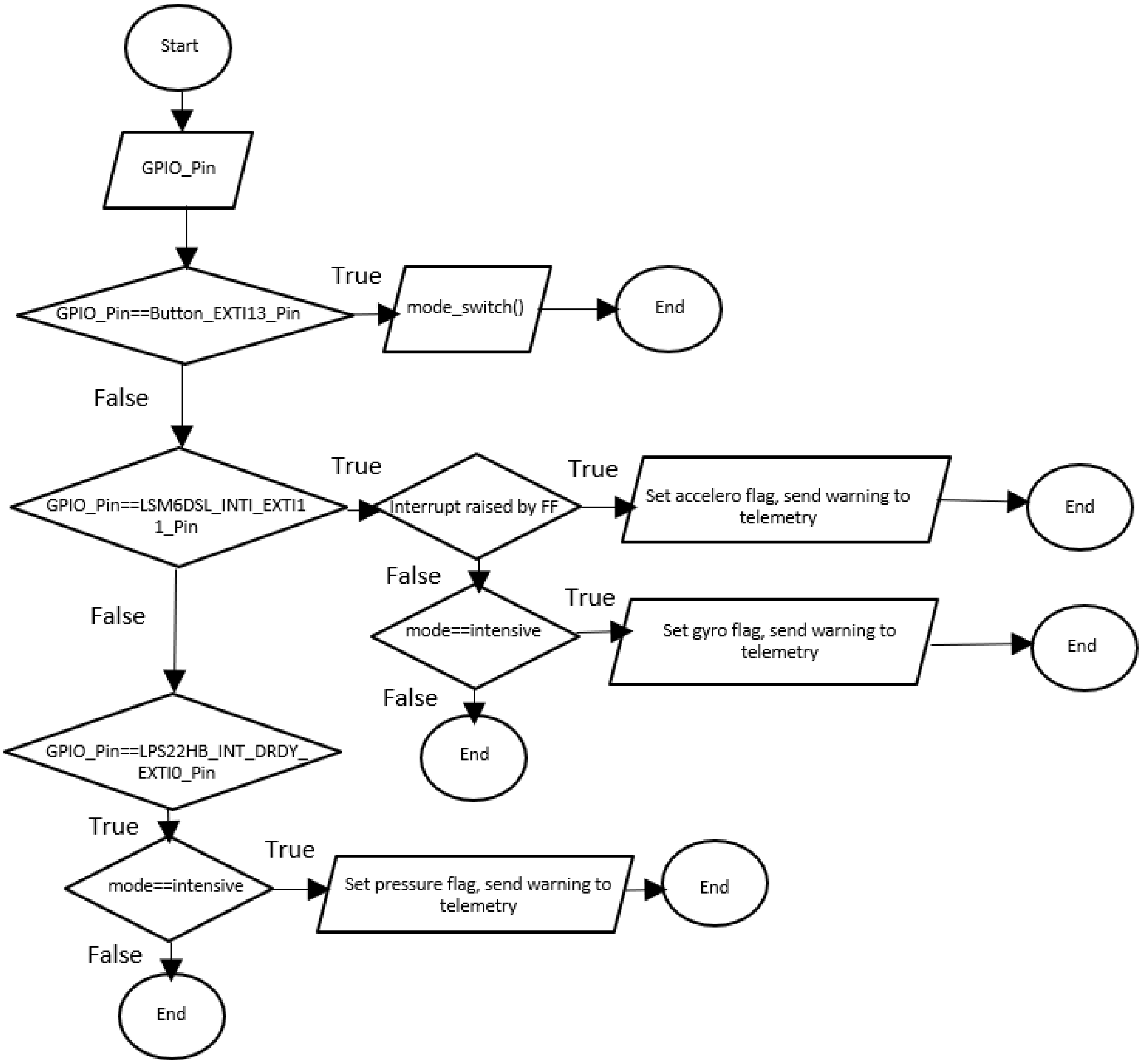
For the accelerometer, gyroscope and pressure sensor, we raised warning flags through interrupts. Thus in our main loop, only some devices will have codes that raise flags and send warnings immediately. For temperature, humidity and magnetometer, detection of exceeding threshold values were done through polling.

To switch modes between healthy and intensive, a function was created called `mode_switch()`. A variable called `button_timer` was created and placed in the Systick handler. Every 1ms `button_timer` is incremented by 1. When we first enter the function, it checks if `button_timer` has exceeded 1000. If so, it resets `button_timer` and the variable (`button_count`) to keep track of the number of button presses. `button_count` is then incremented by 1. After that, the function basically checks `button_count` and the current `mode` to determine if mode should be switched to `intensive` or `healthy`.

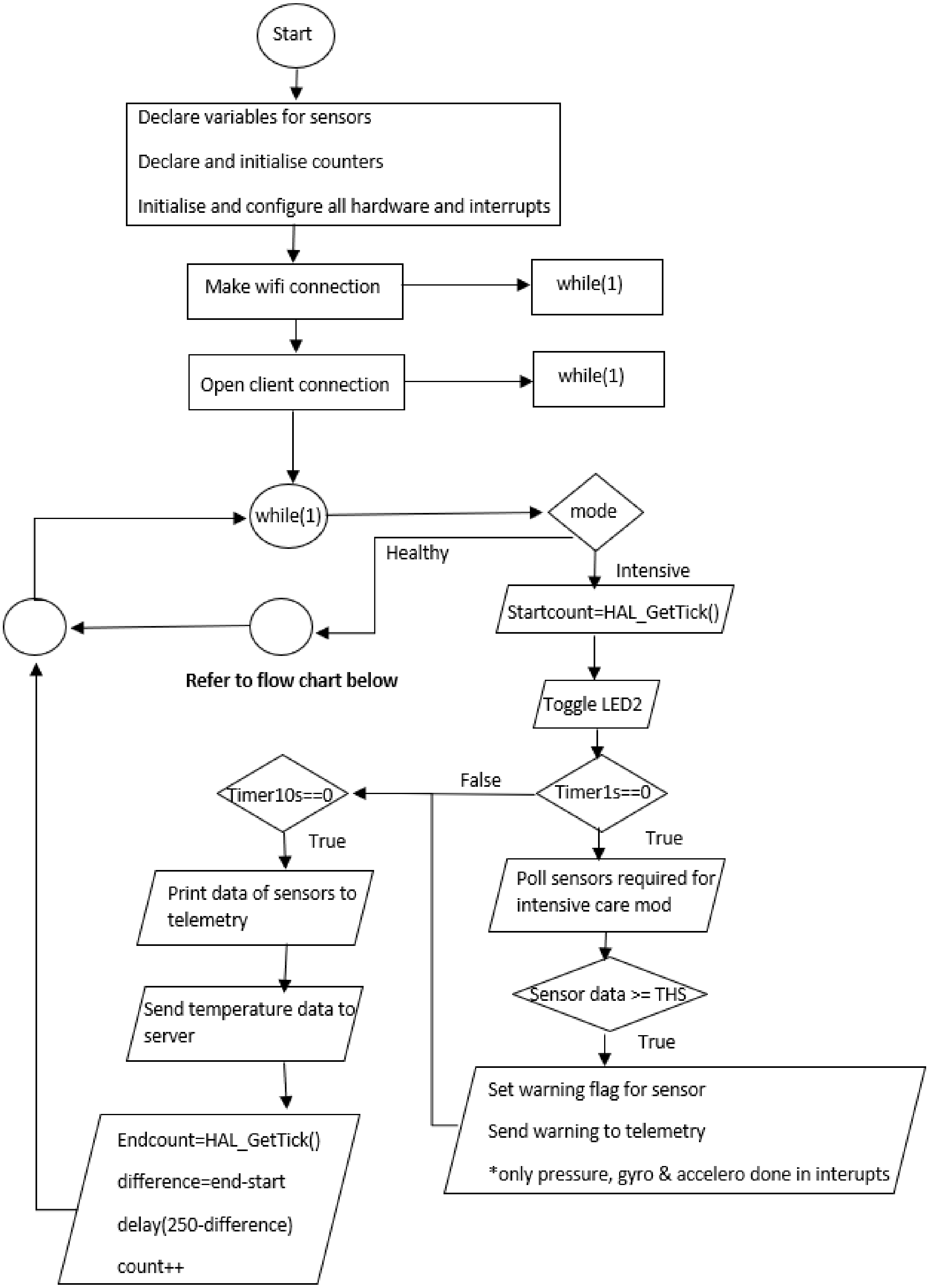
Flowchart



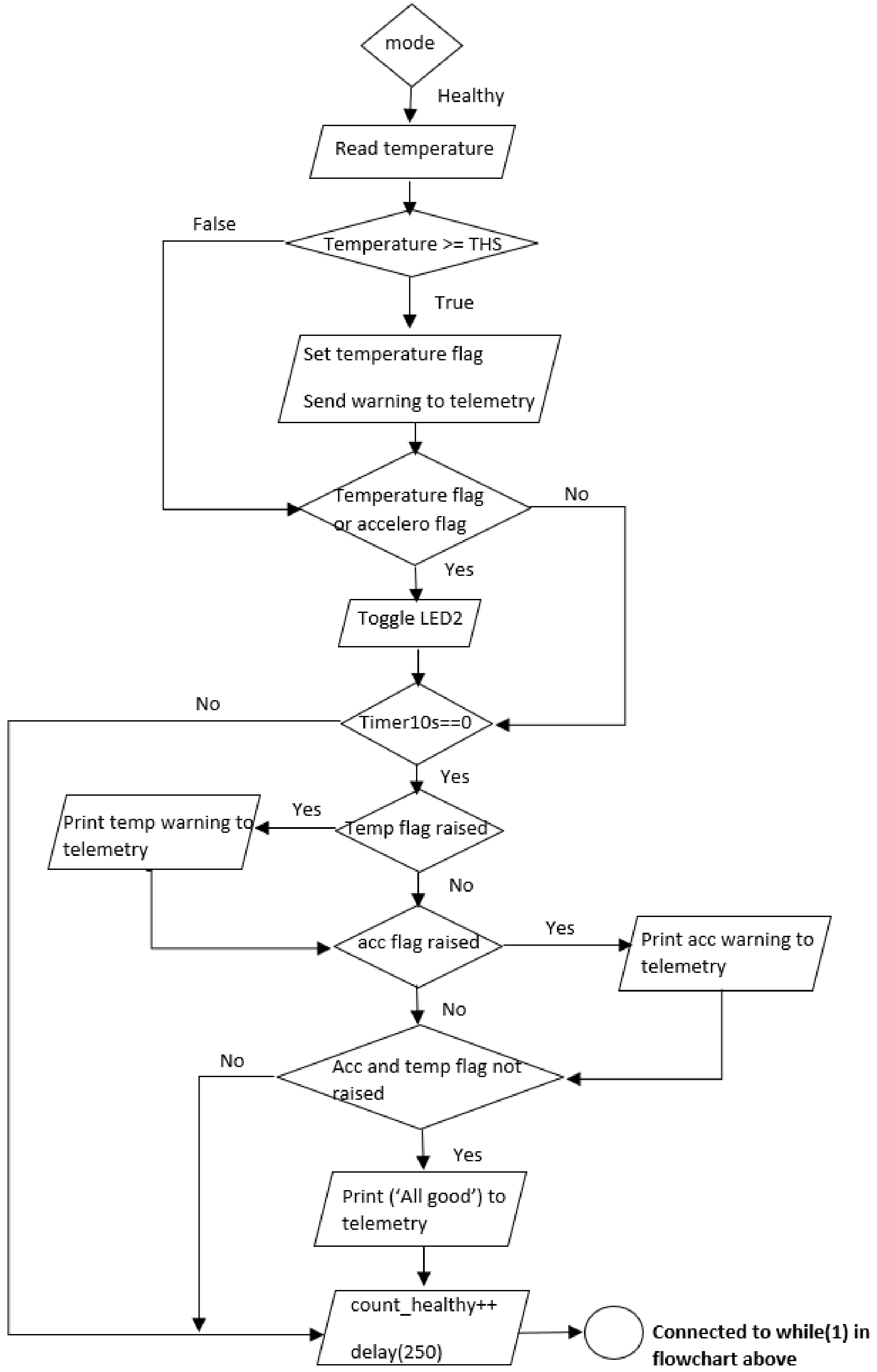
HAL_GPIO_EXTI_Callback (uint16_t GPIO_Pin)


Start
GPIO_Pin
GPIO_Pin==Button_EXTI13_Pin
True
mode_switch()
End
False
GPIO_Pin==LSM6DSL_INTI_EXTI11_Pin
True
Interrupt raised by FF
True
Set accelero flag, send warning to telemetry
End
False
mode==intensive
True
Set gyro flag, send warning to telemetry
End
False
End
False
GPIO_Pin==LPS22HB_INT_DRDY_EXTI0_Pin
True
mode==intensive
True
Set pressure flag, send warning to telemetry
End
False
End

main() (Part 1)

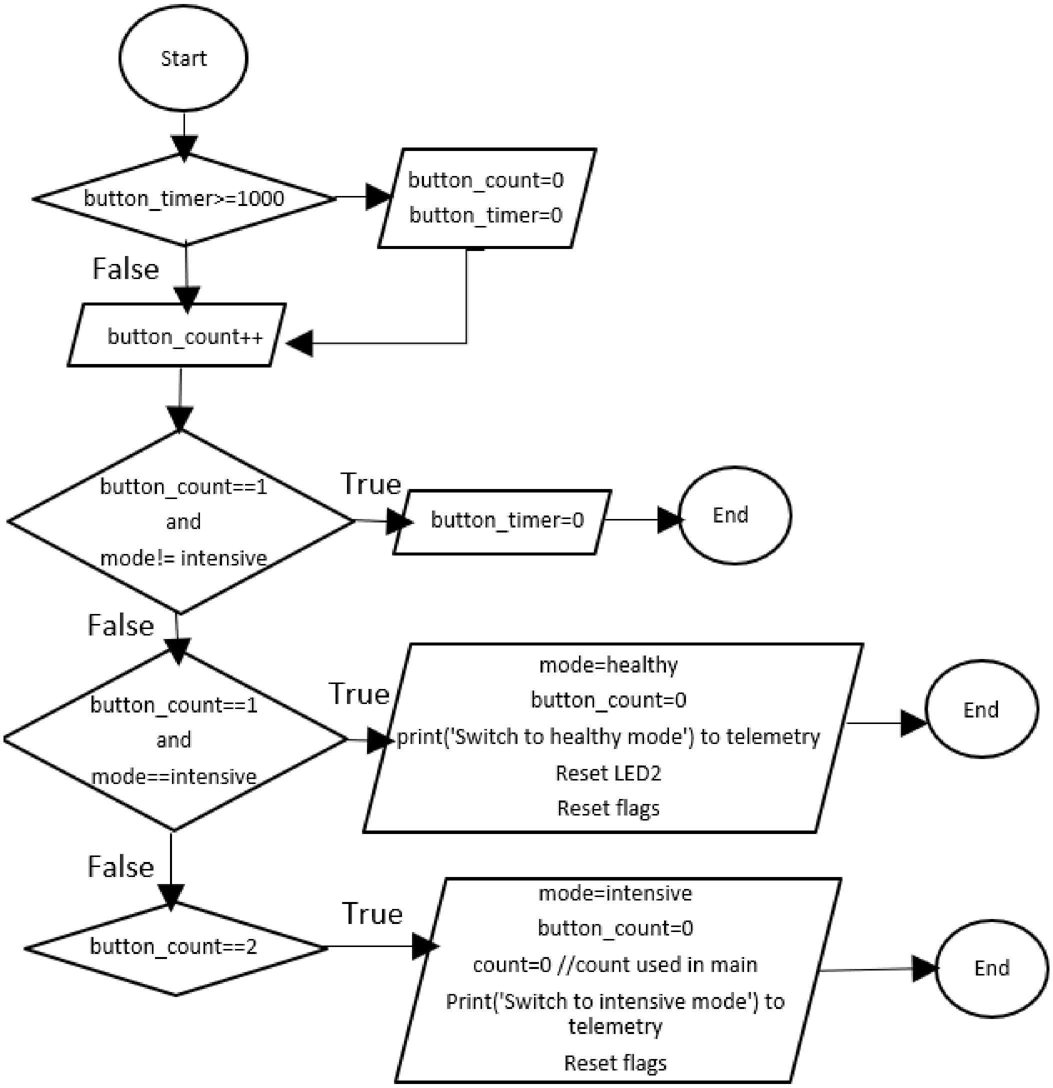

Start
Declare variables for sensors
Declare and initialise counters
Initialise and configure all hardware and interrupts
Make wifi connection
while(1)
Open client connection
while(1)
while(1)
mode
Healthy
Intensive
Startcount=HAL_GetTick()
Refer to flow chart below
Toggle LED2
False
Timer10s==0
Timer1s==0
True
True
Print data of sensors to telemetry
Poll sensors required for intensive care mod
Send temperature data to server
Sensor data >= THS
True
Set warning flag for sensor
Send warning to telemetry
*only pressure, gyro & accelero done in interupts
Endcount=HAL_GetTick()
difference=end-start
delay(250-difference)
count++

main() (Part 2)


mode
Healthy
Read temperature
False
Temperature >= THS
True
Set temperature flag
Send warning to telemetry
Temperature flag or accelero flag
No
Yes
Toggle LED2
No
Timer10s==0
Yes
Print temp warning to telemetry
Yes
Temp flag raised
No
acc flag raised
Yes
Print acc warning to telemetry
No
No
Acc and temp flag not raised
Yes
Print ('All good') to telemetry
count_healthy++
delay(250)
Connected to while(1) in flowchart above

mode_switch(void)

**button_timer++ every 1ms by Systick_handler



Devices

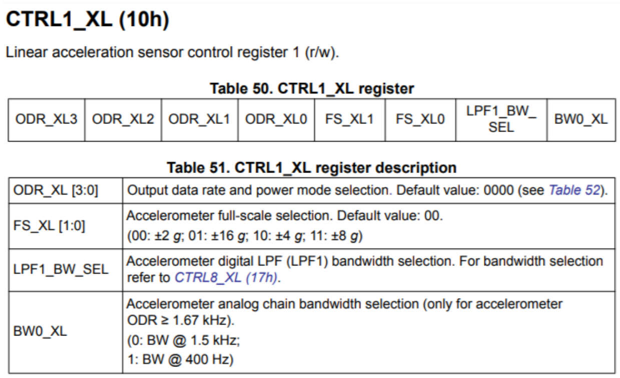
Most devices in the stm32L475 discovery node make use of i2c to interface with the sensors on the board.

LSM6DSL (3D accelerometer and 3D gyroscope)

LSM6DSL features a 3D digital accelerometer and a 3D digital gyroscope.

The LSM6DSL has an i2c address of 110110x where x denotes the Read/Write bit. This notation will be used for the other devices as well. The control registers for accelerometer and gyroscope are 0x10 and 0x11 respectively.

Different settings can be obtained by reading different bits from these registers. We can obtain the full-scale selection by reading the bits within these registers to determine the sensitivity settings being used.



CTRL1_XL (10h)

Linear acceleration sensor control register 1 (r/w).

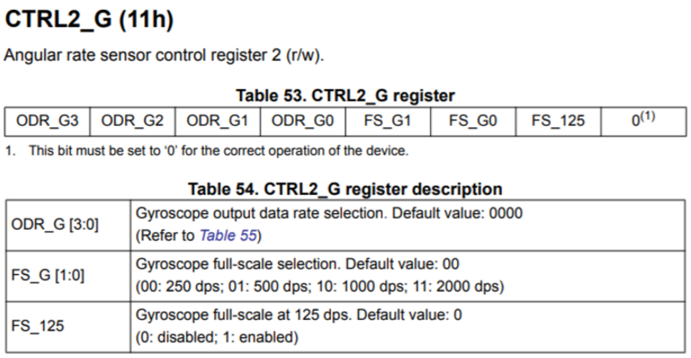
Table 50. CTRL1_XL register

ODR_XL3	ODR_XL2	ODR_XL1	ODR_XL0	FS_XL1	FS_XL0	LPF1_BW_SEL	BW0_XL

Table 51. CTRL1_XL register description

ODR_XL [3:0]	Output data rate and power mode selection. Default value: 0000 (see *Table 52*).
FS_XL [1:0]	Accelerometer full-scale selection. Default value: 00. (00: ±2 *g*; 01: ±16 *g*; 10: ±4 *g*; 11: ±8 *g*)
LPF1_BW_SEL	Accelerometer digital LPF (LPF1) bandwidth selection. For bandwidth selection refer to *CTRL8_XL (17h)*.
BW0_XL	Accelerometer analog chain bandwidth selection (only for accelerometer ODR ≥ 1.67 kHz). (0: BW @ 1.5 kHz; 1: BW @ 400 Hz)

Fig. 1. Accelerometer sensitivity control registers



CTRL2_G (11h)

Angular rate sensor control register 2 (r/w).

Table 53. CTRL2_G register

ODR_G3	ODR_G2	ODR_G1	ODR_G0	FS_G1	FS_G0	FS_125	0[1]

1. This bit must be set to '0' for the correct operation of the device.

Table 54. CTRL2_G register description

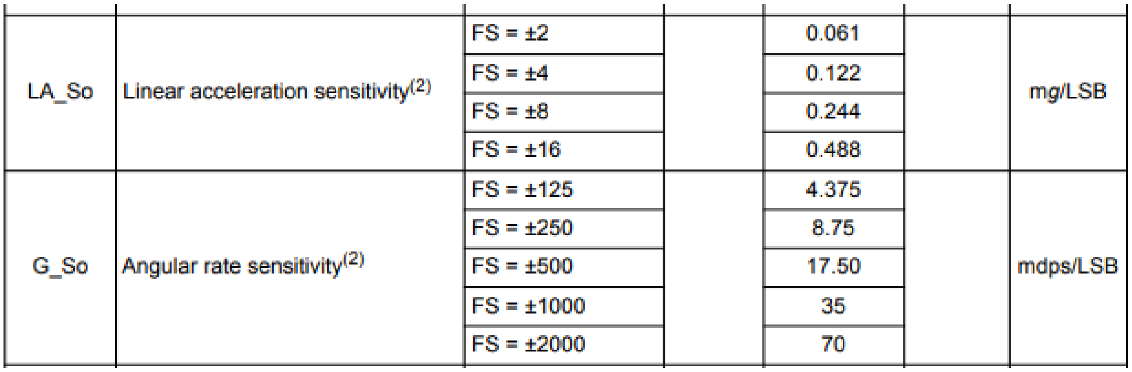
ODR_G [3:0]	Gyroscope output data rate selection. Default value: 0000 (Refer to *Table 55*)
FS_G [1:0]	Gyroscope full-scale selection. Default value: 00 (00: 250 dps; 01: 500 dps; 10: 1000 dps; 11: 2000 dps)
FS_125	Gyroscope full-scale at 125 dps. Default value: 0 (0: disabled; 1: enabled)

Fig. 2. Gyroscope sensitivity control registers

The raw data for these devices can be obtained by reading the values from the output data registers:

- The registers for the gyroscope output data ranges from: OUTX_L_G(22h) - OUTZ_H_G (27h)
- The registers for the accelerometer output data ranges from: OUTX_L_XL(28h) - OUTZ_H_XL (2D h)

- The raw data for gyro x-axis will be OUTX_L_G + OUTX_H_G. The raw data for each axis can be thus retrieved in this way. This applies the raw data for the accelerometer as well.
- By performing $Raw\_data \times Sensitivity$, we can obtain the physical value used in real life

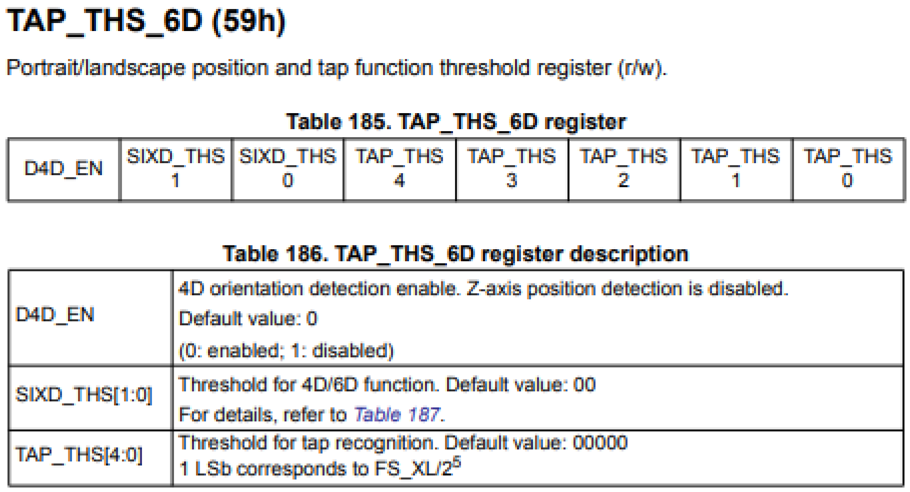


LA_So	Linear acceleration sensitivity(2)	FS = ±2		0.061		mg/LSB
		FS = ±4		0.122		
		FS = ±8		0.244		
		FS = ±16		0.488		
G_So	Angular rate sensitivity(2)	FS = ±125		4.375		mdps/LSB
		FS = ±250		8.75		
		FS = ±500		17.50		
		FS = ±1000		35		
		FS = ±2000		70		

Fig. 3. Linear acceleration and angular rate sensitivities

Interrupts for LSM6DSL (freefall and 6D interrupts)

The LSM6DSL supports interrupts as well through the INT1 line. We can enable basic interrupts such as freefall (FF) and 4D/6D interrupts. This is done by setting bit[7] of the TAP_CFG (58h) register to 1.



TAP_THS_6D (59h)

Portrait/landscape position and tap function threshold register (r/w).

Table 185. TAP_THS_6D register

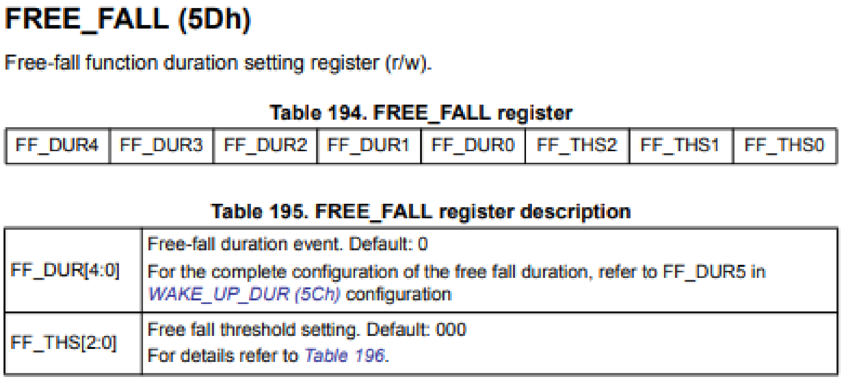
D4D_EN	SIXD_THS 1	SIXD_THS 0	TAP_THS 4	TAP_THS 3	TAP_THS 2	TAP_THS 1	TAP_THS 0

Table 186. TAP_THS_6D register description

D4D_EN	4D orientation detection enable. Z-axis position detection is disabled. Default value: 0 (0: enabled; 1: disabled)
SIXD_THS[1:0]	Threshold for 4D/6D function. Default value: 00 For details, refer to *Table 187*.
TAP_THS[4:0]	Threshold for tap recognition. Default value: 00000 1 LSb corresponds to $FS\_XL/2^5$

Fig. 4. Tap function threshold (and portrait/landscape position) register

The threshold settings for 6D interrupts are set in bits [6:5] of TAP_THS_6D (59h). FF (freefall) threshold and duration can be set through FREE_FALL (5Dh). Bits [7:3] are for FF_DUR (freefall duration) while bits [2:0] are for FF_THS (freefall threshold).



FREE_FALL (5Dh)

Free-fall function duration setting register (r/w).

Table 194. FREE_FALL register

FF_DUR4	FF_DUR3	FF_DUR2	FF_DUR1	FF_DUR0	FF_THS2	FF_THS1	FF_THS0

Table 195. FREE_FALL register description

FF_DUR[4:0]	Free-fall duration event. Default: 0 For the complete configuration of the free fall duration, refer to FF_DUR5 in *WAKE_UP_DUR (5Ch)* configuration
FF_THS[2:0]	Free fall threshold setting. Default: 000 For details refer to *Table 196*.

Fig. 5. Freefall duration and threshold registers

The different threshold values corresponding to different bit settings can be referenced from the datasheet. Additionally, bit[4] and bit[2] must be set in MD1_CFG (5Eh) to enable FF and 6D interrupts to INT1.

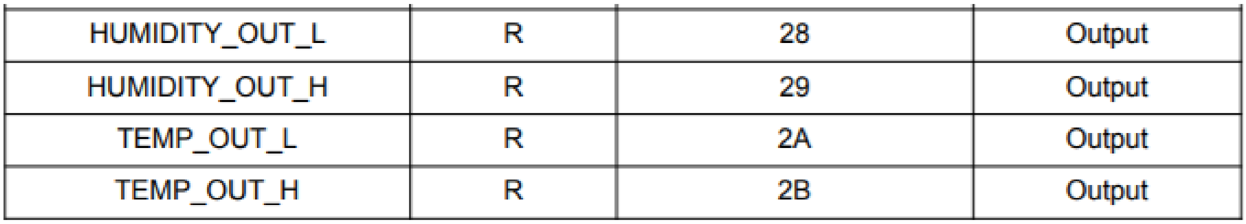
The interrupts are routed to GPIOD Pin11. To determine the source of the interrupt, we can read the WAKE_UP_SRC (1Bh) register and TAP_SRC (1Ch) register.

HTS221 (Humidity and temperature sensor)

The HTS221 features a humidity and temperature sensor.

The HTS221 has an i2c address of 1011111x where x denotes the read/write bit.

The raw data for humidity and temperature can be retrieved by reading the value from registers HUMIDITY_OUT_L (28h), HUMIDITY_OUT_H (29h) and TEMP_OUT_L (2Ah), TEMP_OUT_H (2Bh) respectively.



HUMIDITY_OUT_L	R	28	Output
HUMIDITY_OUT_H	R	29	Output
TEMP_OUT_L	R	2A	Output
TEMP_OUT_H	R	2B	Output

Fig. 6. Register address map of HTS221

To convert the raw data into its physical value, interpolation must be done using calibration registers.

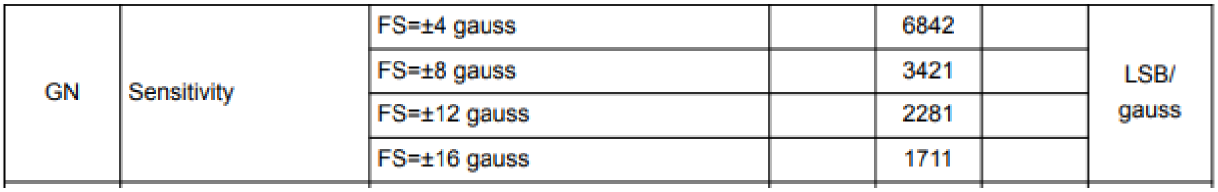
There are no interrupts available for this device

LIS3MDL (magnetometer)

The LIS3MDL features a magnetometer.

The LIS3MDL has an i2c address of 0011110x where x denotes the read/write bit.

By default the LIS3MDL is initialised with FS = ±4 which gives us a sensitivity of 6842LSB / gauss.



GN	Sensitivity	FS=±4 gauss		6842		LSB/ gauss
		FS=±8 gauss		3421		
		FS=±12 gauss		2281		
		FS=±16 gauss		1711		

Fig. 7. Mechanical characteristics of LIS3MDL

The raw data for the magnetometer can be accessed through OUT_X_L (28h)... OUT_Z_H (2Dh) registers. The value is represented in the form of two’s complement.

$Raw\_data \times \frac{1}{Sensitivity}$ will give the physical value in terms of gauss.

Interrupt for this device is not connected to the microprocessor, hence it is not used.

LPS22HB (barometer)

This LPS22HB features a barometer (pressure sensor).

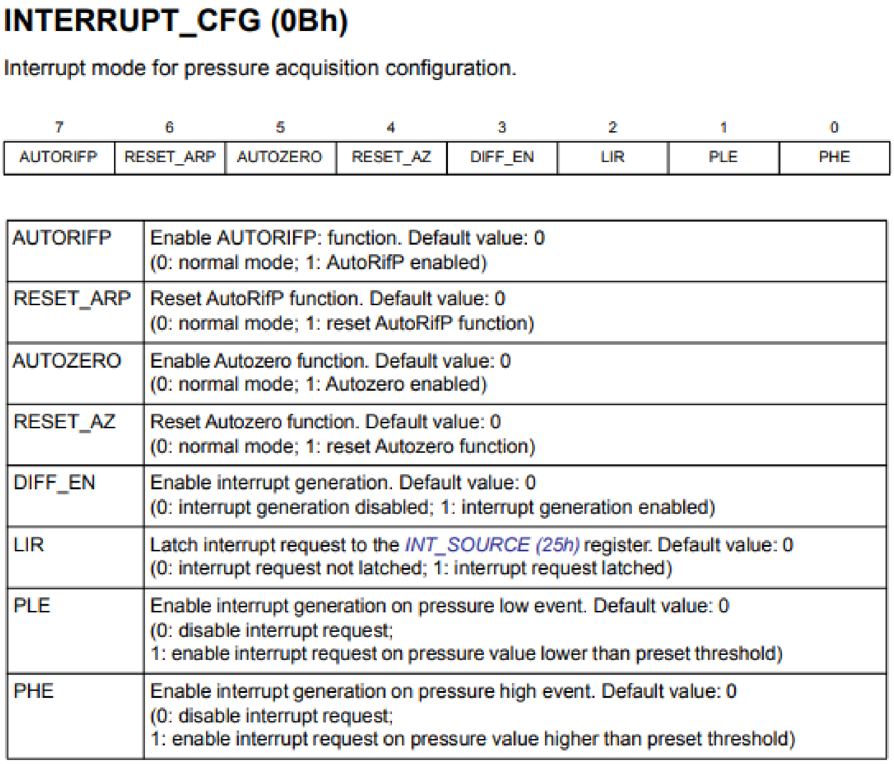
It has an i2c address of 1011101x where x denotes the read/write address.

Interrupts for LPS22HB (high pressure event)

Interrupts can be generated by LPS22HB and are sent through INT_DRDY.

To enable interrupts, DIFF_EN was set in the INTERRUPT_CFG (0Bh) register.

AUTOZERO was set for our case for ease of visualisation. PHE (pressure high event)was also enabled to generate interrupts upon pressure high events.



INTERRUPT_CFG (0Bh)

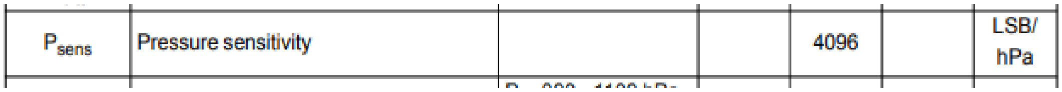
Interrupt mode for pressure acquisition configuration.

7	6	5	4	3	2	1	0
AUTORIFP	RESET_ARP	AUTOZERO	RESET_AZ	DIFF_EN	LIR	PLE	PHE

AUTORIFP	Enable AUTORIFP: function. Default value: 0 (0: normal mode; 1: AutoRifP enabled)
RESET_ARP	Reset AutoRifP function. Default value: 0 (0: normal mode; 1: reset AutoRifP function)
AUTOZERO	Enable Autozero function. Default value: 0 (0: normal mode; 1: Autozero enabled)
RESET_AZ	Reset Autozero function. Default value: 0 (0: normal mode; 1: reset Autozero function)
DIFF_EN	Enable interrupt generation. Default value: 0 (0: interrupt generation disabled; 1: interrupt generation enabled)
LIR	Latch interrupt request to the *INT_SOURCE (25h)* register. Default value: 0 (0: interrupt request not latched; 1: interrupt request latched)
PLE	Enable interrupt generation on pressure low event. Default value: 0 (0: disable interrupt request; 1: enable interrupt request on pressure value lower than preset threshold)
PHE	Enable interrupt generation on pressure high event. Default value: 0 (0: disable interrupt request; 1: enable interrupt request on pressure value higher than preset threshold)

Fig. 8. Interrupt configuration registers for LPS22HB

Threshold settings can be set by writing to THS_P_L (0Ch) and THS_P_H(0Dh) registers. The formula for interrupt threshold is as follows: *Interrupt threshold (hPa) = ±THS_P / 16*



P_{sens}	Pressure sensitivity			4096		LSB/ hPa

Fig. 9. Pressure sensitivity of the device

The raw data can be read from PRESS_OUT_XL (28h), PRESS_OUT_L (29h) and PRESS_OUT_H (2Ah). The value is expressed as two's complement. The physical value can be calculated by taking PRESS_OUT * (1/sensitivity). The interrupt is routed to GPIOD Pin10.

Interrupts through internal peripherals

Once an interrupt is generated by a sensor, it will be routed to the internal peripherals of the microprocessor. These internal peripherals must be enabled and configured to detect the appropriate interrupt at the specified port and pin. These can be done by:

1. Enable NVIC to detect interrupt for the appropriate IRQ
2. Configure SYSCFG_EXTICR register to the appropriate GPIO port and pin where the interrupt is being raised
3. Configure IMR register to ensure line is not masked
4. Configure either RTSR or FTSR to detect rising or falling edge
5. Set Pending Register
6. Clear Pending Register

However, all these steps need not be carried out manually during Assignment 2 as it is handled by the driver functions and we can specify the mode of a certain GPIO port and pin as GPIO.Mode = GPIO_MODE_IT_RISING for example. For GPIO in open drain mode, a pull up/ pull down resistor network must be configured.

Enhancement

Our enhancement is a real time data visualisation feature. As we are dealing with COVID-19 patients, their temperature is of top priority as fevers are one of the main symptoms of COVID-19. To make it easier to understand how a patient's temperature is fluctuating over time, we have implemented a server over a local area network that is able to receive temperature data and display a real time graph.

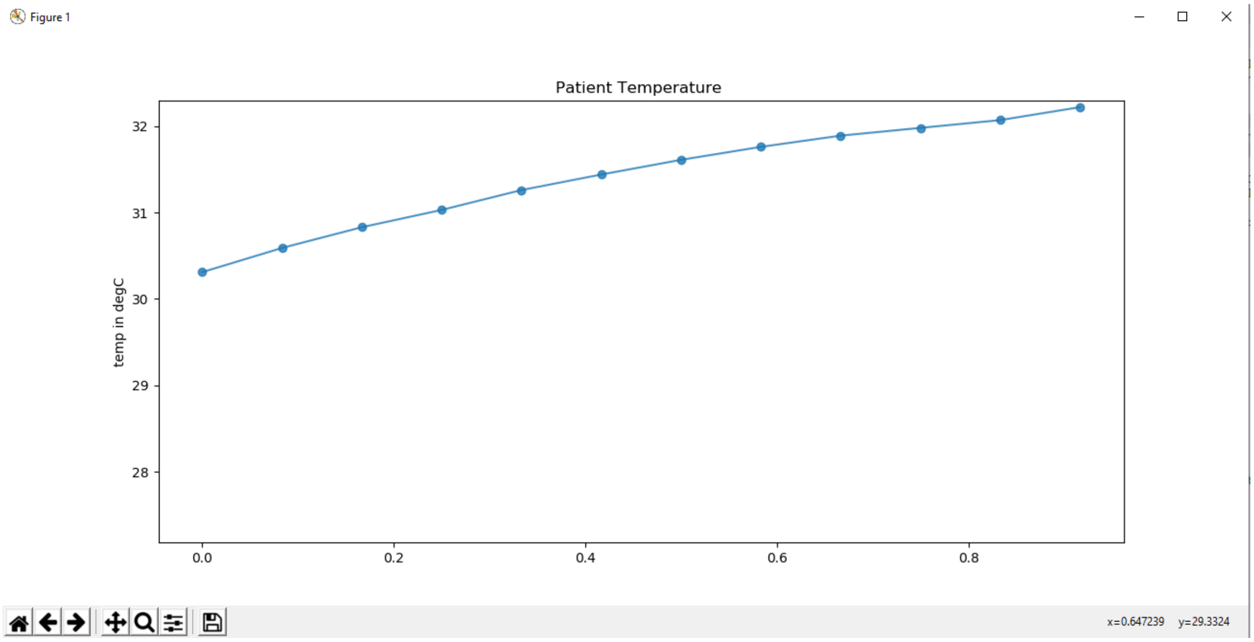


Fig. 10. Snapshot of the temperature-time graph

The enhancement makes use of Python Sockets (Sockets and the socket API are used to send messages across a network. They provide a form of inter-process communication (IPC).) such that Python is able to act as a server. In our case, a TCP connection uses IPv4 internet address family. The board is able to make a connection using WiFi to the server given that both are on the same LAN network. The board will then send data to the server every 10 seconds which the server then plots into a real time graph displaying the patient's temperature.

Future iterations of this enhancement could allow for multiple connections to the server allowing hospital staff to easily monitor the temperature of multiple patients.