|  |
| --- |
| EE2028 Microcontroller Programming and Interfacing |
| Project Report |
| Neo Yu Yao Terence A0164651E  Neo Jia Wen Rachel A0176763R  GA: |



**FitNUS** Fitness Tracking System Prototype Design

Table of Contents

[2 Introduction and objectives 2](#_Toc529984031)

[3 Flowcharts describing the system design and processes 2](#_Toc529984032)

[4 Detailed implementation 5](#_Toc529984033)

[4.1 Temperature sensor – Body temperature monitoring 6](#_Toc529984034)

[4.2 Light sensor – Ambient light detection 6](#_Toc529984035)

[4.3 Accelerometer – Fall detection 7](#_Toc529984036)

[4.4 Wireless UART 7](#_Toc529984037)

[4.5 General Code Structure 7](#_Toc529984038)

[5 Enhancement 8](#_Toc529984039)

[5.1 7 Segment 8](#_Toc529984040)

[5.2 Debugger Mode with UART interrupt 8](#_Toc529984041)

[5.3 Music Player 10](#_Toc529984042)

[5.3.1 Motivation 10](#_Toc529984043)

[5.3.2 Frontend: Graphical User Interface Design 10](#_Toc529984044)

[5.3.3 Backend: Song Library Storage and Retrieval 11](#_Toc529984045)

[5.3.4 Backend: Processing 11](#_Toc529984046)

[6 Significant problems encountered and solutions proposed 12](#_Toc529984047)

[6.1 Reading of Temperature Sensor causes system lag 12](#_Toc529984048)

[6.2 Jumper Modification for Pin Conflicts 13](#_Toc529984049)

[6.2.1 Green RGB conflicts with OLED 13](#_Toc529984050)

[6.2.2 Blue RGB conflicts with Speaker 13](#_Toc529984051)

[6.2.3 Default Jumper settings prevents affects SW4 functionality 13](#_Toc529984052)

[6.3 Calibration of TEMP\_THRESHOLD 14](#_Toc529984053)

[6.4 UART Interrupt not working because of Tera Term Terminal Setup 14](#_Toc529984054)

[7 Issues or suggestions 14](#_Toc529984055)

[8 Conclusion 14](#_Toc529984056)

# Introduction and objectives

In this assignment, our group is tasked to implement a fitness tracking system, **FitNUS**. The main purpose of **FitNUS** is to boost daily workouts and make them easier to achieve. **FitNUS** detects acceleration, light and temperature changes. **FitNUS** sends data periodically to a server known as **FiTrackX**. The XBee RF module acts as a low powered wireless communication device that sends collected data to **FiTrackX**.

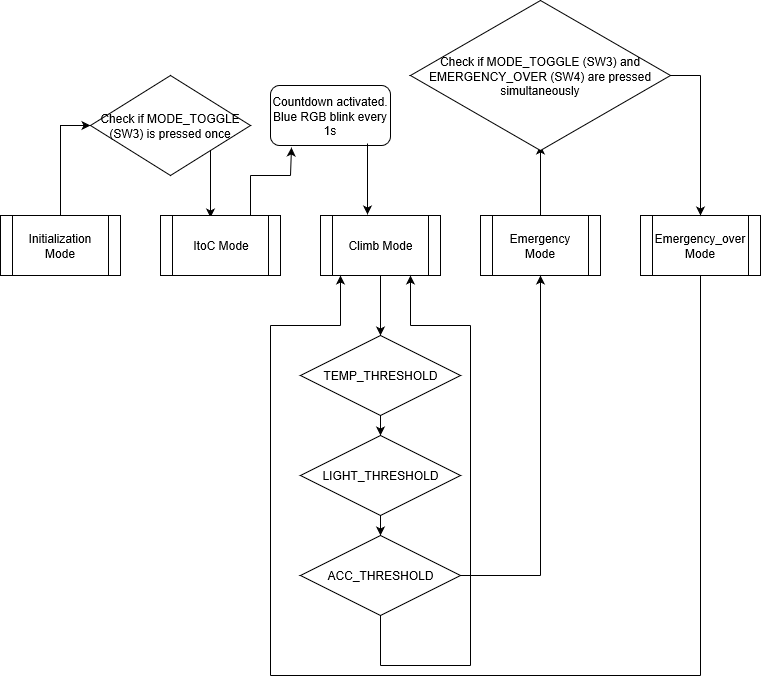
**FitNUS** has 3 modes of operation: Initialization, Climb and Emergency modes, and will transmit to **FiTrackX** is certain conditions are met. We also implemented ItoC and Emergency\_over modes which is later described in our report. Initialization mode would be active when the FitNUS system is first switched on. Climb mode would be active when MODE\_TOGGLE (SW3) is activated. Emergency mode would be active when the user decides to trigger fall detection by slightly shaking the board in Climb mode.

The accelerometer has an extremely wide range of applications. For **FitNUS**, the accelerometer is assumed to be mounted on the system to detect falling event of the climber. Light sensor is used for **FitNUS** to monitor the ambient environment, where the intensity decreases as the ambient light is dim. The temperature module on **FitNUS** is used to monitor the body temperature of the climber.

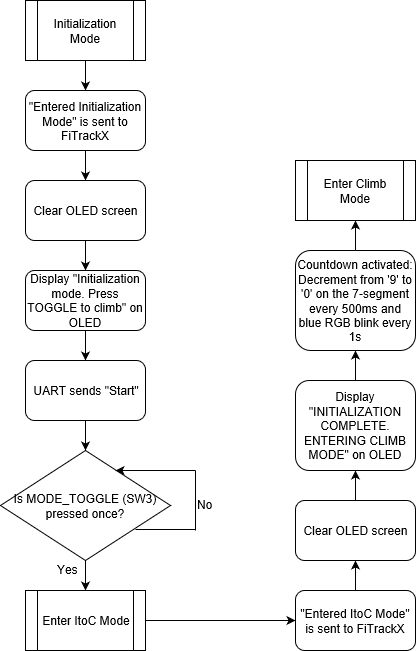
After completing assignment 2, we would have learnt how to apply system design approaches, such as using flowcharts, to design embedded applications, understand the interfaces between microcontrollers and peripherals and have the ability to develop C embedded programming controller-based applications.

# Flowcharts describing the system design and processes

## int main



## Initialization Mode



|  |  |
| --- | --- |
| Climb Mode | Emergency Mode |

# Detailed implementation

|  |  |
| --- | --- |
| **int** **main** (**void**) {  init\_everything();  acc\_read(&x, &y, &z);  xoff = 0-x;  yoff = 0-y;  zoff = 64-z;  moveBar(1, dir);  oled\_clearScreen(*OLED\_COLOR\_BLACK*);  led7seg\_setChar(0xFF, *TRUE*);  **while** (1){  **if** (state == *Initialization*){  do\_Initialization();}  **if** (state == *ItoC*){  do\_toclimb();}  **if** (state == *Climb*){  do\_Climb();}  **if** (state == *Emergency*){  do\_Emergency();}  **if** (state == *Emergency\_over*){  do\_Emergency\_over(); }}} | When the system is first switched ON, all the peripherals will be initialized and the interrupts will be enabled. **FitNUS** will then enter Initialization Mode. The OLED should display “Initialization mode. Press TOGGLE to climb”. Sensors will not be reading any data and no UART transmission should be sent to **FitNUS**.  When SW3 is pressed, **FitNUS** will enter Climb Mode. The OLED should display “CLIMB”. The 7-segment will display the countdown, decrementing from ‘9’ to ‘0’. The sensors will obtain values and store them in variables to be utilised by other functions. The OLED should then display the values obtained.  When fall detection is triggered, **FitNUS** will enter Emergency Mode. The OLED should display “EMERGENCY!”.  In order to make our code less complex, we performed functional abstraction and segmented the codes for the operation modes into several functions outside of the main function and then integrated them back using the conditional while loop. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The function “init\_everything()” initializes all the peripherals required for **FitNUS** to work properly (e.g. i2c, GPIO, uart, OLED, etc). We integrated the segmented codes for the initialization of the peripherals back into the function. Additionally, when **FitNUS** is first switched ON, it would configure SysTick to generate an interrupt every 1ms. The SysTick handler updates msTicks every 1ms to give a real time reference to the system.   |  | | --- | | **void** **SysTick\_Handler**(**void**){  msTicks++;} |   We initialized the interrupts for EINT0 (SW3) and EINT3 (SW3, light sensor, temperature sensor and all functions of the joystick). The function will clear any pending interrupts first before clearing the interrupt flags, configuring them and then enabling the interrupt handler. The function will then enable the GPIO interrupt. For instance, to enable the GPIO interrupt for SW3, a ‘1’ would be shifted 4 bits into the register IO0IntEnF, for the GPIO interrupt P0.4 to be enabled, after which the EINT3 interrupt handler will be enabled. Furthermore, as higher pre-empt priority means lower priority, SysTick will come first, followed by EINT3, EINT0 then UART3. This is because SysTick is a clock that controls everything. If a tick is missed, timing is off, EINT3 is used to measure temperature, so it is the 2nd most important. If one interrupt timestep is missed, temperature goes slightly off. EINT0 is used for SW3. Between SW3 and UART, we chose SW3 to be more important so when SW3 is pressed, it is more responsive. | **static** **void** **init\_everything**(){  init\_i2c();  init\_ssp();  init\_GPIO();  init\_uart();  SysTick\_Config(SystemCoreClock/1000);  temp\_init(&Get\_Time);  pca9532\_init();  joystick\_init();  acc\_init();  oled\_init();  led7seg\_init();  speaker\_init();  rgb\_init();  lightSenIntInit();  LPC\_GPIOINT->IO2IntEnF |= 1<<10;  NVIC\_ClearPendingIRQ(*EINT0\_IRQn*);  NVIC\_EnableIRQ(*EINT0\_IRQn*);  LPC\_GPIOINT ->IO0IntEnF |= 1<<4;  LPC\_GPIOINT ->IO2IntEnF |= 1<<5;  LPC\_GPIOINT ->IO0IntEnR |= 1<<2;  LPC\_GPIOINT ->IO0IntEnF |= 1<<17;  LPC\_GPIOINT ->IO0IntEnF |= 1<<15;  LPC\_GPIOINT ->IO0IntEnF |= 1<<16;  LPC\_GPIOINT ->IO2IntEnF |= 1<<3;  LPC\_GPIOINT ->IO2IntEnF |= 1<<4;  NVIC\_ClearPendingIRQ(*EINT3\_IRQn*);  NVIC\_EnableIRQ(*EINT3\_IRQn*);  UART\_IntConfig(LPC\_UART3, *UART\_INTCFG\_RBR*, *ENABLE*);  NVIC\_ClearPendingIRQ(*UART3\_IRQn*);  NVIC\_EnableIRQ(*UART3\_IRQn*);  NVIC\_SetPriorityGrouping(5);  uint32\_t ans = NVIC\_EncodePriority(5, 0, 0);  NVIC\_SetPriority(*SysTick\_IRQn*,ans);  ans = NVIC\_EncodePriority(5, 1, 0);  NVIC\_SetPriority(*EINT3\_IRQn*,ans);  ans = NVIC\_EncodePriority(5, 2, 0);  NVIC\_SetPriority(*EINT0\_IRQn*,ans);  ans = NVIC\_EncodePriority(5, 3, 0);  NVIC\_SetPriority(*UART3\_IRQn*,ans);} | | |
| **void** **EINT3\_IRQHandler**(**void**){  **if** ((LPC\_GPIOINT ->IO0IntStatR>>2) & 0x1){  LPC\_GPIOINT ->IO0IntClr = 1<<2;  temp\_periods\_cnt++;  **if**(temp\_periods\_cnt == 1) t1 = Get\_Time();  **else** **if**(temp\_periods\_cnt == temp\_periods+1) t2 = Get\_Time();  **else** **if**(temp\_periods\_cnt == temp\_periods\*2) temp\_periods\_cnt = 0; | | | Temperature sensor – Body temperature monitoring The EINT3 interrupt obtains the time taken for 11 rectangular wave periods. The temp\_periods value represents the number of times temperature interrupt has happened. This only resets after 20 rectangular wave periods have past due to the computation time needed for the system to run smoothly. |
| int32\_t **fast\_temp\_read** (**void**){  **if** (~t2 && ~t1){  time\_diff = **abs**(t2-t1);  temp = ((time\_diff\*1000.0) / (temp\_periods\*10\*TEMP\_SCALAR\_DIV10))-273.15;}  **if** (temp < 16) temp = 0;  **return** temp\*10;} | | | This code replaces the slow temp\_read() driver function. The time difference t2-t1 is later used to calculate the temperature. |
| **if**(temp\_on){  tempvalue = (fast\_temp\_read() != 0)? fast\_temp\_read():tempvalue;  **if** (tempvalue > TEMP\_THRESHOLD && temp\_flag == 0){  **if**(Climb\_State != *Rest*) Saved\_State = Climb\_State;  Climb\_State = *Rest*;}  **if** (tempvalue <= TEMP\_THRESHOLD && temp\_flag == 1){  temp\_flag = 0;}} | | | While in Climb mode, the temperature sensor should display the reading (to 1d.p.) on the OLED in the following format: “Temp: xx.x deg” where xx.x is the temperature reading in oC. If temperature exceeds TEMP\_THRESHOLD, the OLED should display “REST NOW” for 3 seconds before returning to CLIMB Mode. This should only be triggered once every time the temperature exceeds TEMP\_THRESHOLD unless the temperature goes below TEMP\_THRESHOLD and exceeds it again. Update Code. In a while loop, using printf() might result in timing issues. Each call to printf() takes significant time to process and execute: about 100,000 instructions or 10 milliseconds, depending on the clock speed. |
|  | | |
| Light sensor – Ambient light detection While in Climb mode, the light sensor should be continuously read and display the reading on the OLED in the following format: “Light: xx lux”, where xx is the reading. If the reading falls below LIGHT\_THRESHOLD, the lights on LED\_ARRAY should light up proportionately to how low the ambient light is (the dimmer the ambient light, the more the number of LEDs should be lit.) Otherwise, LED\_ARRAY should not be lit. In order to calculate | | **if**(light\_on){  light\_enable();  luminI = light\_read();  shift = luminI / 18.75;  ledOn = (shift <= 16)?(1 << (16-shift)) - 1: 0x0; //max 1<<16 - 1 = 0xffff (16 1's), min 1<<0 - 1 = 0  // if (shift <= 16){ ledOn = (1 << (16-shift)) - 1;}  // else ledOn = 0x0;  pca9532\_setLeds(ledOn, 0xffff);} | |

|  |
| --- |
| how far an object is from the threshold to convert to the number of LEDs that should be lit, we left shifted the total number of logic ‘1’ in the 16 bits at different bit positions. This value is then implemented into the pca9532\_setLeds function, which is used for controlling 16 LEDs over and the I2C bus, including the logic to act as an I2C slave device as well as the drive capability for directly driving LEDs. Instead of hardcoding how many LEDs will be lit for different ranges of light intensity, we used the shift expression to shift bits by a certain number of positions specified. The bit positions that have been vacated by the shift operation are zero-filled. The range for shift is from 0 to 16. The range for (16-shift) is the inverse of that: from 16 to 0. By inversing the shift expression, we can then find out how many LEDs will be lit up. |

|  |  |  |
| --- | --- | --- |
| **if**(acc\_on){  acc\_read(&x, &y, &z);  x = x+xoff;  y = y+yoff;  z = z+zoff;  net\_acc = (**sqrt**(x\*x + y\*y + z\*z))/ 64;  **if**(net\_acc > ACC\_THRESHOLD){  **printf**("State changed from Climb to Emergency\n");  state = *Emergency*;}} | Accelerometer – Fall detection While in Climb mode, Emergency mode may be triggered through fall detection (shaking the board slightly), where net acceleration exceeds ACC\_THRESHOLD in Climb mode. No other modes should be able to trigger Emergency mode. The net acceleration (to 2 d.p.) should be displayed on the OLED in the following format: “Acc: x.xx” where x.xx is the net acceleration in ‘g’s (1g = 9.8m/s2). | |
| Wireless UART UART is a standard for long-distance, asynchronous, serial, full-duplex peer-peer communication. It makes transmission easier and more readily available as compared to I2C although having less control over parity. We first initialized the UART to enable it on LPC1769 then configured the pins required to set up UART. We then initialized UART3 to supply power and set up working parts and enabled transmission for UART3. We removed jumper B on J7 allow for the XBee module to be mounted on the LPCXpesso Baseboard. | | **void** **init\_uart**(**void**){  UART\_CFG\_Type uartCfg;  uartCfg.Baud\_rate = 115200;  uartCfg.Databits = *UART\_DATABIT\_8*;  uartCfg.Parity = *UART\_PARITY\_NONE*;  uartCfg.Stopbits = *UART\_STOPBIT\_1*;  PINSEL\_CFG\_Type PinCfg;  PinCfg.Funcnum = 2;  PinCfg.Pinnum = 0;  PinCfg.Portnum = 0;  PINSEL\_ConfigPin(&PinCfg);  PinCfg.Pinnum = 1;  PINSEL\_ConfigPin(&PinCfg);  UART\_Init(LPC\_UART3, &uartCfg);  UART\_TxCmd(LPC\_UART3, *ENABLE*);} |
| An example of how we implemented UART into our code is illustrated in the Emergency\_over mode. This sends a message to FiTrackX that reads “Emergency is cleared! Time consumed for recovery: %lu sec\r\n”, where lu = emer\_dur. Since protocols add extra bits, we have to increase the number of characters in char[] array. | | **char** uart\_msg[64];  **sprintf**(uart\_msg,"Emergency is cleared! Time consumed for recovery: %lu sec\r\n", emer\_dur);  UART\_Send(LPC\_UART3, (uint8\_t \*)uart\_msg, **strlen**(uart\_msg), |

## General Code Structure

The general code structure that governs each state and their submachines (if any) in our programme is as such:

1. Firstly, there is an if conditional where the current state of the programme is checked via polling. This conditional is the entry point at which the programme first enters each state, be it a main state or a sub-state. In the if conditional block, we execute instructions to initialise the state. For example, we cleared the OLED and 7-segments, and saved the accelerometer readings as we entered the Climb state.

**if**(state == *CurrentState*){

//Initialization of CurrentState

**while** (state == *CurrentState*){

//Remains in CurrentState

//Refreshes CurrentState values

//which may include changes to substates

}}

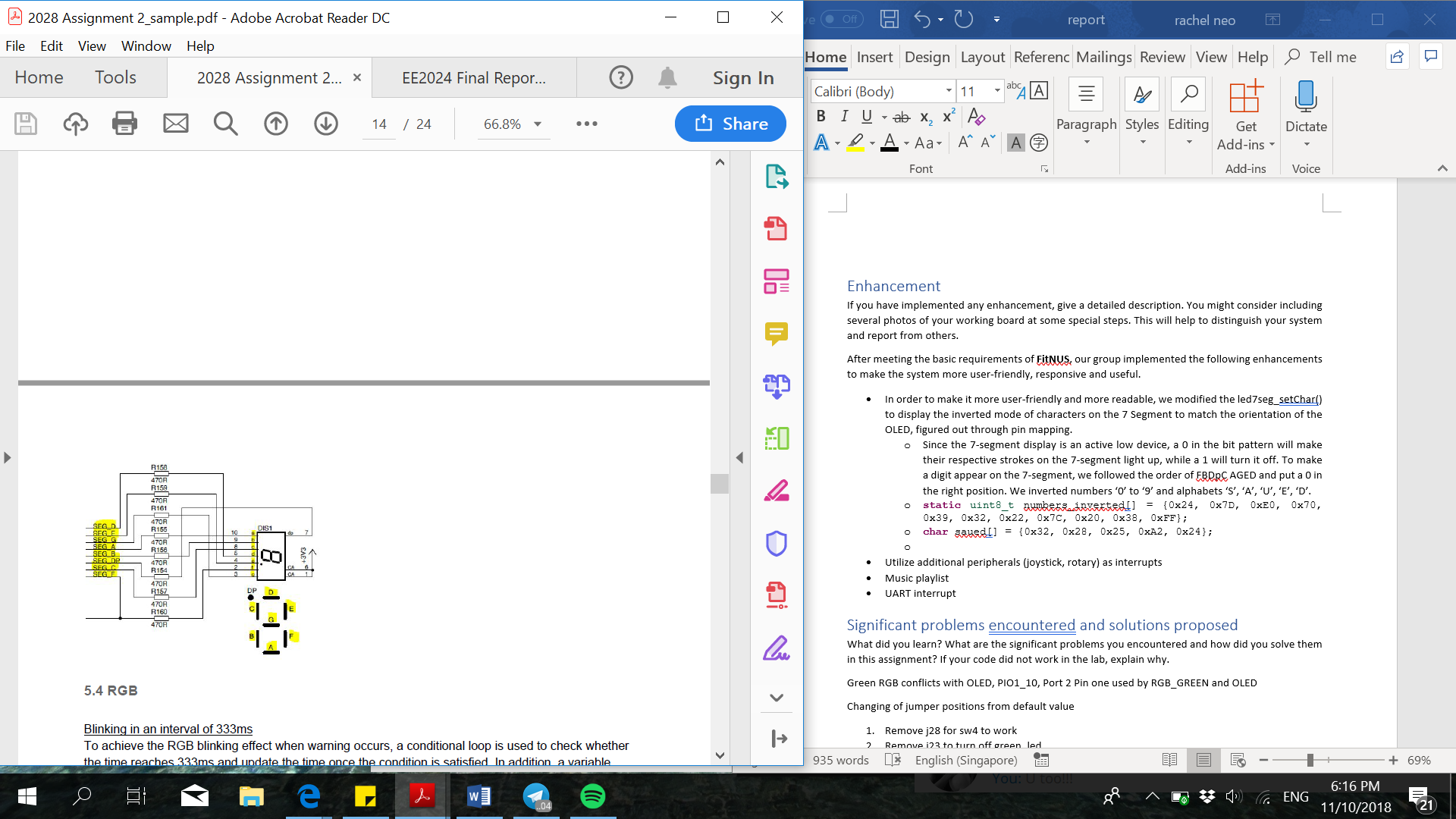
1. Secondly, after the initialization of the state is executed in the if block, there is a while conditional that cycles through instructions to update values and outputs as the programme remains in the state. For example, refreshing sensor readings and refreshing their values on the OLED.

# Enhancement

*If you have implemented any enhancement, give a detailed description. You might consider including several photos of your working board at some special steps. This will help to distinguish your system and report from others.*

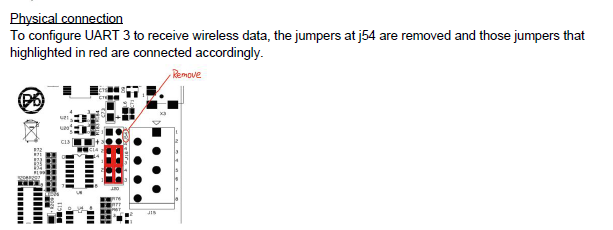
After meeting the basic requirements of **FitNUS**, our group implemented the following enhancements to make the system more user-friendly, responsive and useful.

## 7 Segment

In order to make it more user-friendly and more readable, we modified the led7seg\_setChar() to display the inverted mode of characters on the 7 Segment to match the orientation of the OLED, figured out through pin mapping.

* Since the 7-segment display is an active low device, a 0 in the bit pattern will make their respective strokes on the 7-segment light up, while a 1 will turn it off. To make a digit appear on the 7-segment, we followed the order of FBDpC AGED and put a 0 in the right position. We inverted numbers ‘0’ to ‘9’ for the countdown function in the ItoC mode and also the alphabets ‘S’, ‘A’, ‘U’, ‘E’, ‘D’ for the save function in the Emergency\_over mode.
* **static** uint8\_t numbers\_inverted[] = {0x24, 0x7D, 0xE0, 0x70, 0x39, 0x32, 0x22, 0x7C, 0x20, 0x38, 0xFF};
* **char** saued[] = {0x32, 0x28, 0x25, 0xA2, 0x24};

## Debugger Mode with UART interrupt



|  |  |
| --- | --- |
| **void** **UART3\_IRQHandler**(**void**){  **if**((LPC\_UART3->IIR & 0xE) == 0b0100) //RDA {  **if**(UART\_Receive(LPC\_UART3, (uint8\_t\*)&rxbuf[rx\_count], 1, *BLOCKING*) == 1){  **if**(rxbuf[rx\_count] == '\r'){  **printf**("Command Received\n");  rxbuf[rx\_count+1] = 0;  **if** (**strcmp**(rxbuf, "On Acc\r")==0) acc\_on = 1;  **else** **if** (**strcmp**(rxbuf, "Off Acc\r")==0) acc\_on = 0;  **else** **if** (**strcmp**(rxbuf, "On Temp\r")==0) temp\_on = 1;  **else** **if** (**strcmp**(rxbuf, "Off Temp\r")==0) temp\_on = 0;  **else** **if** (**strcmp**(rxbuf, "On Light\r")==0) light\_on = 1;  **else** **if** (**strcmp**(rxbuf, "Off Light\r")==0) light\_on = 0;  **else** UART\_Send(LPC\_UART3, (uint8\_t\*)UART\_Error\_msg, **strlen**(UART\_Error\_msg), *BLOCKING*);  rx\_count = 0;  **printf**("Entered: %s\n", rxbuf);}  **else** rx\_count = (rx\_count == 64)? 0: rx\_count+1; }}} | We decided to implement UART interrupt because there was a delay in playing music. We figured that it was the temperature sensor that was causing this delay so with the UART interrupt, we turned off the temperature sensor and the system became a lot more responsive. UART3 is used to receive commands. We implemented this function into all the sensors: Accelerometer, temperature sensor and light sensor. For instance, if “Off Acc” is entered in Tera term, the system will receive this command and turn off accelerometer. Likewise, for the rest of the commands. If a command received is not implemented into the UART interrupt, an error message will pop up. By using if else in EINT3 instead of just if, which lowers the worse case execution time (WCET) of ISR, and allows for prioritizing of diff inputs in same interrupt handler. |

## [Image result for ipod nano](https://www.google.com.sg/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjEuveDvszeAhXMuY8KHcxwCZsQjRx6BAgBEAU&url=https://www.amazon.com/Apple-Graphite-Generation-Discontinued-Model/dp/B002L6HE9G&psig=AOvVaw1UF4kVDoqh-tkpb_VkBXEP&ust=1542030667570324)Music Player

### Motivation

Learning about interrupts that are able to pre-empt currently running processes, we wondered if we are able to design a programme in such a way to re-enter the thread mode, from the Interrupt Service Routine (ISR), at an instruction line different from where we left off. One such application of this would be to interrupt a playing song with the press of a button, pause the song and execute another function in thread mode with no intention of returning to play the remainder of the song. This can be done with hyper-threading which is unfortunately out of the scope of this project. Fortunately, this can also be done with logic enhancements to the code.

We shall hence attempt to implement this in the form of a Music Player, inspired by the iPod Nano.

### Frontend: Graphical User Interface Design

**Controls:** user input is done through the joystick using GPIO interrupts via EINT3 interrupt.

* **Up/ Down**: scroll up/down for song selection when no song is playing, next/ previous when a song is playing
* **Center**: play/ pause a song by selecting it
* **Left/ Right**: move to the menu screen on the left or right. For our case, as we only have two user selectable menu screens in Climb mode (the third begin Rest which can only be triggered from temperature), moving left or right from Music mode will result in the sensor reading screen and vice versa.



**Display:** Similar to how a handphone or an MP3 player displays songs, our GUI will take the form of the OLED, where song titles are displayed for the user to select from.

* As shown in the image, the currently selected song will be highlighted white
* **Scrolling**
  + One limitation we faced was that the OLED updates with a noticeable delay. Thus, it will not be viable to refresh the entire OLED each time there needs to be a change in some part of the screen. Our code will hence need to identify the parts of the screen to be updated each time the user scrolls to select a different song, and only refresh the identified pixels. This was done by setting flags and counters each time the up or down joystick buttons are pressed.

//JOYSTICK\_DOWN

**else** **if** ((LPC\_GPIOINT ->IO0IntStatF >> 15) & 0x1){

LPC\_GPIOINT ->IO0IntClr = 1<<15; //clear the interrupt

prev\_song\_index = song\_index;

song\_index = (song\_index < number\_of\_songs-1)? song\_index+1: 0;

scroll\_updated = 0;

song\_changed = 1;

}

//JOYSTICK\_UP

**else** **if** ((LPC\_GPIOINT ->IO2IntStatF >> 3) & 0x1){

LPC\_GPIOINT ->IO2IntClr = 1<<3; //clear the interrupt

prev\_song\_index = song\_index;

song\_index = (song\_index > 0)? song\_index-1: number\_of\_songs-1;

scroll\_updated = 0;

song\_changed = 1;

}

The flag, scroll\_updated, is set to 0 each time the button is pressed to indicate that the OLED is due for a refresh. It is checked via polling in the Music submachine.

song\_index and prev\_song\_index are counters that indicate the pixels to be refreshed.

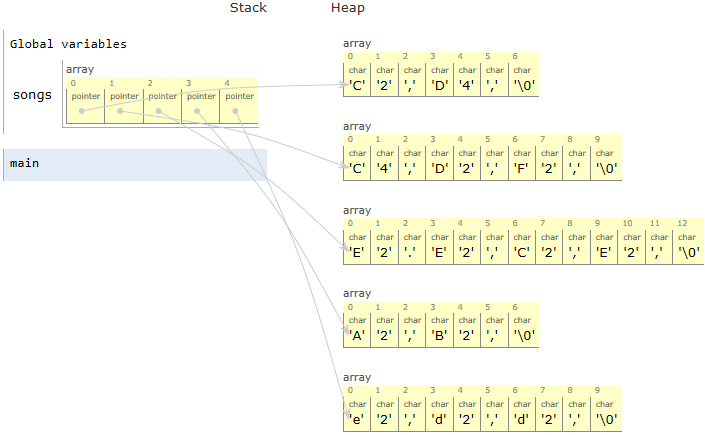
**if**(scroll\_updated == 0){

oled\_putString(0, prev\_song\_index\*10 + 10, (uint8\_t \*) song\_titles[prev\_song\_index],…

oled\_putString(0, song\_index\*10 + 10, (uint8\_t \*) song\_titles[song\_index],…

}

### Backend: Song Library Storage and Retrieval

Our library of songs is stored in a 2-dimensional array, songs, as illustrated in Figure 1. Each element in songs contains the pointer to the first note of each song which is stored in a heap. This way, when we want to access the starting address of the second song in the array, we do not have to recurse through the first song and can instead move to the next element in songs. The length of each song is hence allowed to be arbitrary, increasing the robustness of the design.

To keep track of the progress as songs are played, we implemented a form of memory to save last location with the use of 2 global variables:

1. song\_index: variable that tracks the Y-axis of the 2D array which is the starting address of the song currently selected relative to the starting address of the array.
2. song\_pointer\_count: variable that tracks the X-axis, the address of the last note that was played relative to the address of the first note in the current song.

Figure 1: Song Library

Here, we note that every adjacent note is 3 intervals apart. Hence, the pointer will have to increase by 3 to point to the next note (line 25). We increase this relative pointer, song\_pointer\_count, each time a note finishes playing.

Each time the **playSong()** function is called, it checks if the selected song is different from the song that was previously played (line 3). If song has been changed, the pointer points to the start of the new song (line 4). Else, the pointer is incremented to one note after the last note that was played (lines 7-9).

**static** **void** **playSong**(uint8\_t \*song){

……

**if**(song\_changed){

song\_pointer\_count = 0;

song\_changed = 0;}

**else**{

**while**(i<song\_pointer\_count){

\*song ++;

i++;}}

**if**(\*song != '\0' && play\_flag){

note = getNote(\*song++);

**if** (\*song == '\0'){

song\_pointer\_count = 0;

play\_flag = 0;

**return**;}

dur = getDuration(\*song++);

**if** (\*song == '\0'){

song\_pointer\_count = 0;

play\_flag = 0;

**return**;}

pause = getPause(\*song++);

playNote(note, dur);

Timer0\_Wait(pause);

song\_pointer\_count += 3;}

**if** (\*song == '\0'){

song\_pointer\_count = 0;

play\_flag = 0;}}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

If the end of the song has been reached, (\*song == '\0'), song\_pointer\_count is reset to the start of the song in preparation to play the next song, doesn’t matter which song.

**Evaluation of Approach**

A while loop is undesirable in general as it hogs processor time. However, song\_pointer\_count is not expected to be large in our case, and this small number of loop runs will result in an insignificant delay.

### Backend: Processing

Firstly, the while loop in the original **playSong()** is removed and replaced with non-blocking logic. This allows the full functionality of *Climb* mode such as sensor readings and triggering of *Emergency* mode in the Music submachine.  
Next, we implemented two flags to indicate of input processing:

1. song\_changed: set to 1 when up/down joystick is used to select another song. This flag is checked via polling in **playSong()**and used to reset song\_pointer\_count to the starting address of the song when a new song is selected.
2. play\_flag is toggled each time center joystick is pressed. **playSong()**is only executed while play play\_flag == 1. When play flag is toggled to 0, the last location played is saved and the song is paused.

# Significant problems encountered and solutions proposed

What did you learn? What are the significant problems you encountered and how did you solve them in this assignment? If your code did not work in the lab, explain why.

## Reading of Temperature Sensor causes system lag

The temp\_read() function provided in the temperature sensor library is a blocking function, containing lines such as shown below that waits for GET\_TEMP\_STATE to change before the function proceeds to the next instruction.

**while**(GET\_TEMP\_STATE == state);

Furthermore, there is a for loop that loops for up to 340 times when both pins U7-TSI0 and U7-TSI1 are set to 0. The purpose of this, we presume, is to do a smoothing on the temperature reading across the set time interval to minimise noise. As our programme is not reading the sensor in real time, but rather at a fixed time interval set by sensor\_refresh\_ticks, smoothing in sensor readings are not necessary for us. We will however, take the average temperature reading over 10 periods to get a more reliable reading. We have hence written our own temp read function using an interrupt.

**void** **EINT3\_IRQHandler**(**void**){

// Temperature sensor

**if** ((LPC\_GPIOINT ->IO0IntStatR>>2) & 0x1){

LPC\_GPIOINT ->IO0IntClr = 1<<2; //clear the interrupt

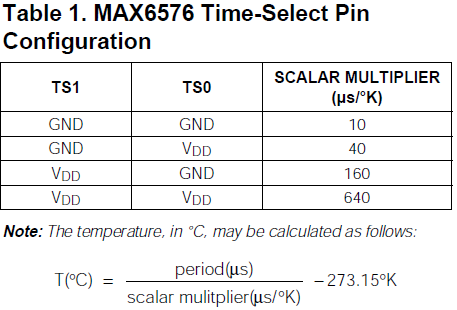
temp\_periods++;

**if** (temp\_periods == 1) t1 = Get\_Time();

**else** **if** (temp\_periods == 10) t2 = Get\_Time();

**else** **if** (temp\_periods == 20) temp\_periods = 0;

}

By this formula given in the datasheet, we can define our function as shown.

// Replacement for slow temp\_read() driver function

int32\_t **fast\_temp\_read** (**void**){

//10T(C) = (period (us) / scalar\_div10) - 2731 K

**return** **abs**(t2-t1)\*1000/(10\*TEMP\_SCALAR\_DIV10)-2731;

}

## Jumper Modification for Pin Conflicts

### Green RGB conflicts with OLED

**void** **rgb\_setLeds** (ledMask){

…

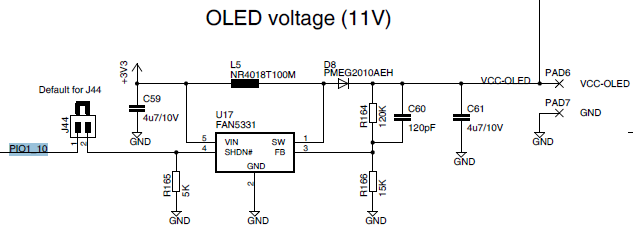
**if** ((ledMask & RGB\_GREEN) != 0)

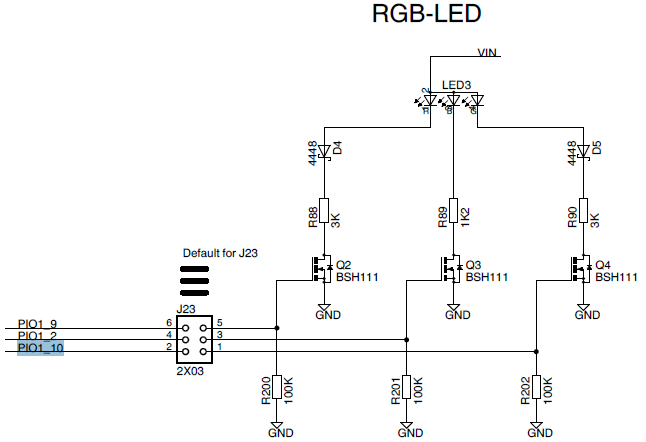
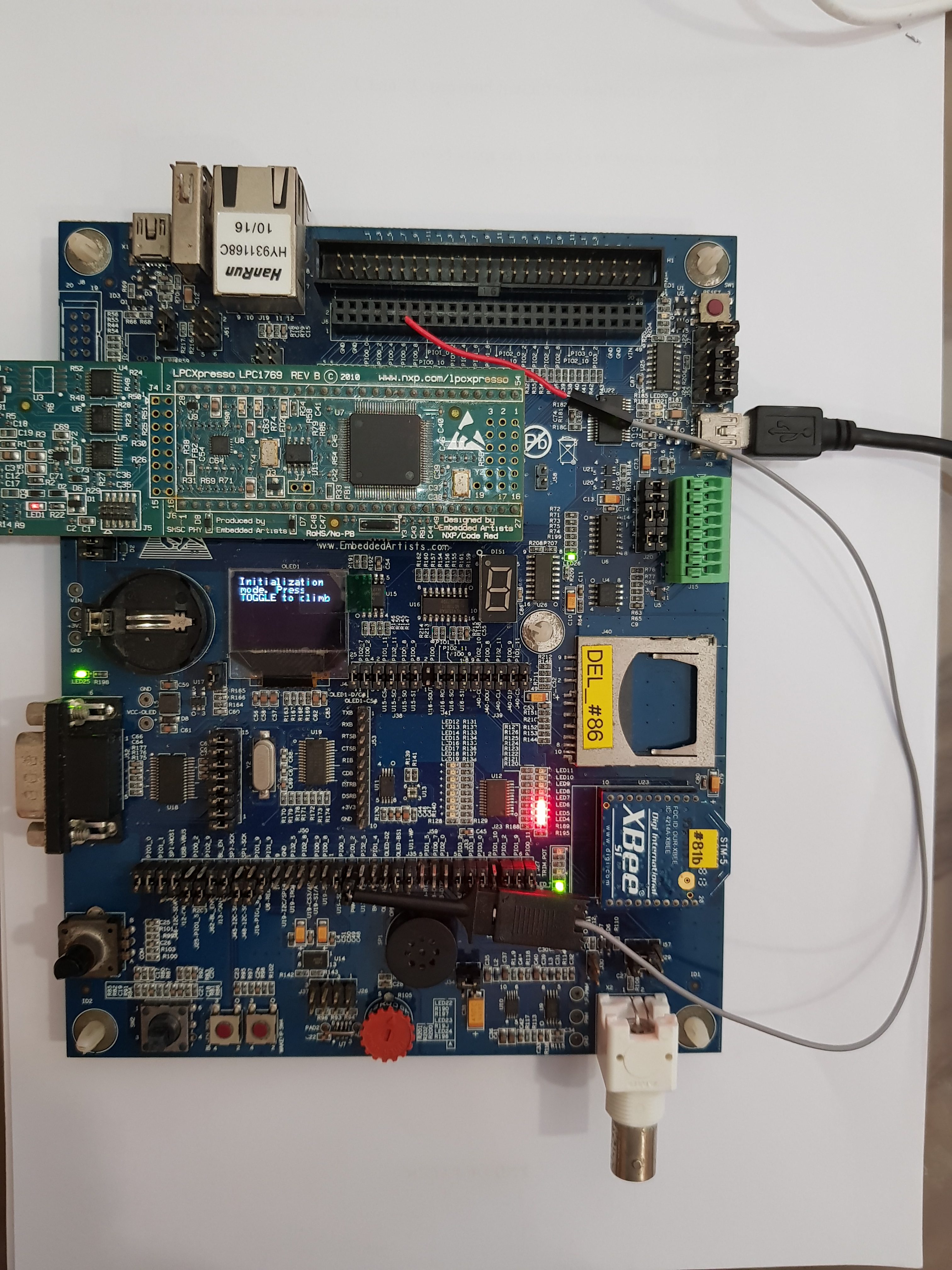
GPIO\_SetValue( 2, (1<<1) );

**else**

GPIO\_ClearValue( 2, (1<<1) );

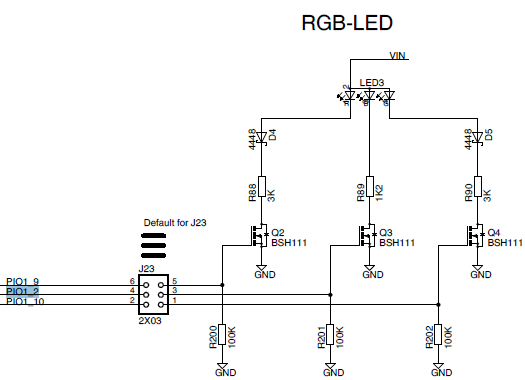
PIO1\_10, which is connected to P2.1 is used by both RGB\_GREEN and OLED voltage. The default driver function **rgb\_setLeds** clears P2.1 each time the function is called and RGB\_GREEN is not turned on, affecting OLED functionality. For the blue and red LEDs to alternate every 500ms, the green LED should also be turned off throughout.

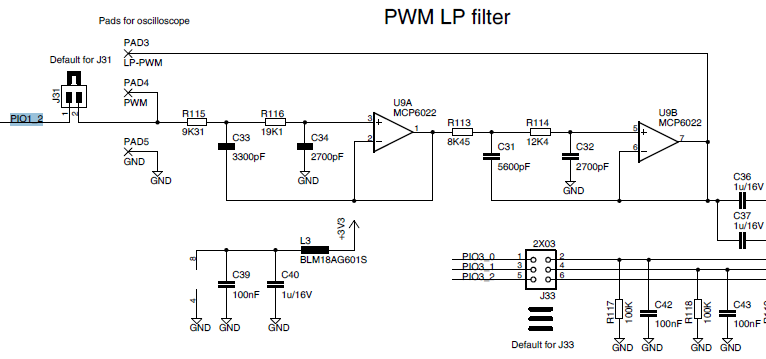
As RGB\_GREEN is not required for this project, we have written our own **setRGBLeds** function that does not clear P2.1 and removed the jumper J23 shown below to physically turn off RGB\_GREEN.



### Blue RGB conflicts with Speaker

PIO1\_2 which is connected to P0.26 is used by both RGB\_BLUE and the PWM LP Filter input to the speaker circuit. This causes the speaker to ‘pop’ when RGB\_BLUE is toggled, and RGB\_BLUE to flash when an output is sent to the speaker. As shown in the image, we have hence physically rerouted the PWM LP Filter’s input to PIO0\_6 which is connected to P0.21. PIO0\_6 is only used by the USB interface which is not utilised in this project. Codes for initialization of speaker is updated accordingly.





PIO0\_6

### Default Jumper settings prevents affects SW4 functionality

A jumper at J28 grounds PIO1\_4, incorrectly indicating active low SW4 as pressed. We remove jumper J28.

## Calibration of TEMP\_THRESHOLD

## UART Interrupt not working because of Tera Term Terminal Setup

|  |  |
| --- | --- |
|  | We were originally tasked to have both new-line characters for receiving and transmitting set as CR+LF. However, it couldn’t work unless we tweaked it so as the new line for transmission is set as CR. New-line characters (CR or CR+LF) received from the host are converted to CR+LF pairs by Tera Term, and then Tera Term sends them to MACRO. It was not able to work because we should only use the pair CR+LF as a new-line character to send to Tera Term while ASCII code 13 is for CR. Additionally, our code for UART interrupt is relatively longer. There should only be 1/2 lines in the interrupt handler. There cannot be too many lines or too long of a function otherwise program execution will not be as effective. |

# Issues or suggestions

These feedbacks, whether positive or negative, will not affect your marks in any way, but will make the report more complete.

As this is the first hardware programming project we did on LPC, there were many times when we got stuck, baffled by lines of codes that seems to work but did not. Fortunately, we were able to readily consult the various teachers, lab staff, and graduate assistances who are not only very knowledgeable, spotting our errors instantly, but also extremely patient when explaining the concepts to us. And we are very grateful for you all!

# Conclusion