Neo Yu Yao Terence A0164651E, Neo Jia Wen Rachel A0176763R GA:

Table of Contents

[Introduction and objectives 1](#_Toc529643541)

[Flowcharts describing the system design and processes 2](#_Toc529643542)

[Detailed implementation 6](#_Toc529643543)

[Enhancement 6](#_Toc529643544)

[Significant problems encountered and solutions proposed 7](#_Toc529643545)

[Reading of Temperature Sensor causes system lag 7](#_Toc529643546)

[Issues or suggestions 8](#_Toc529643547)

[Conclusion 8](#_Toc529643548)

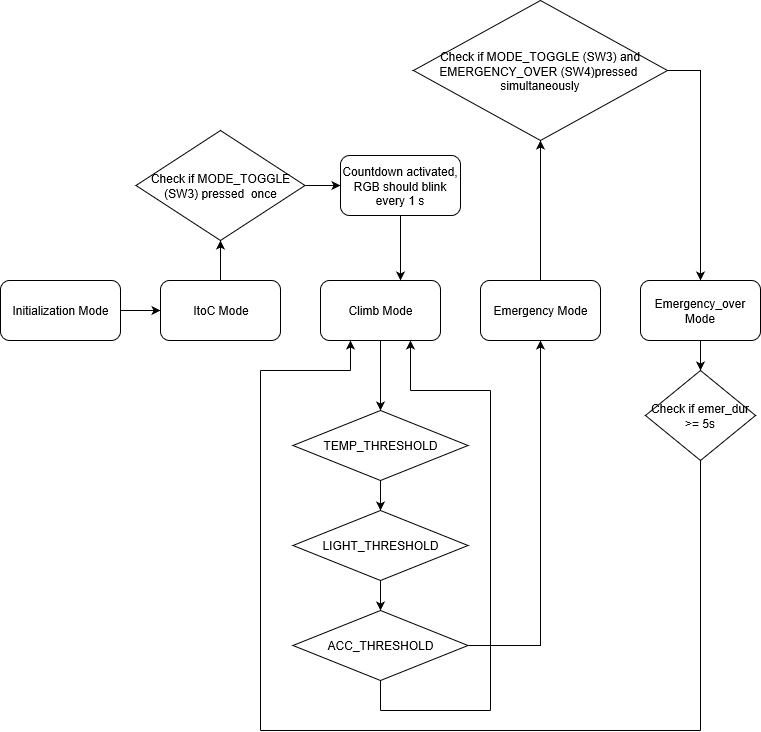
# Introduction and objectives

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**.

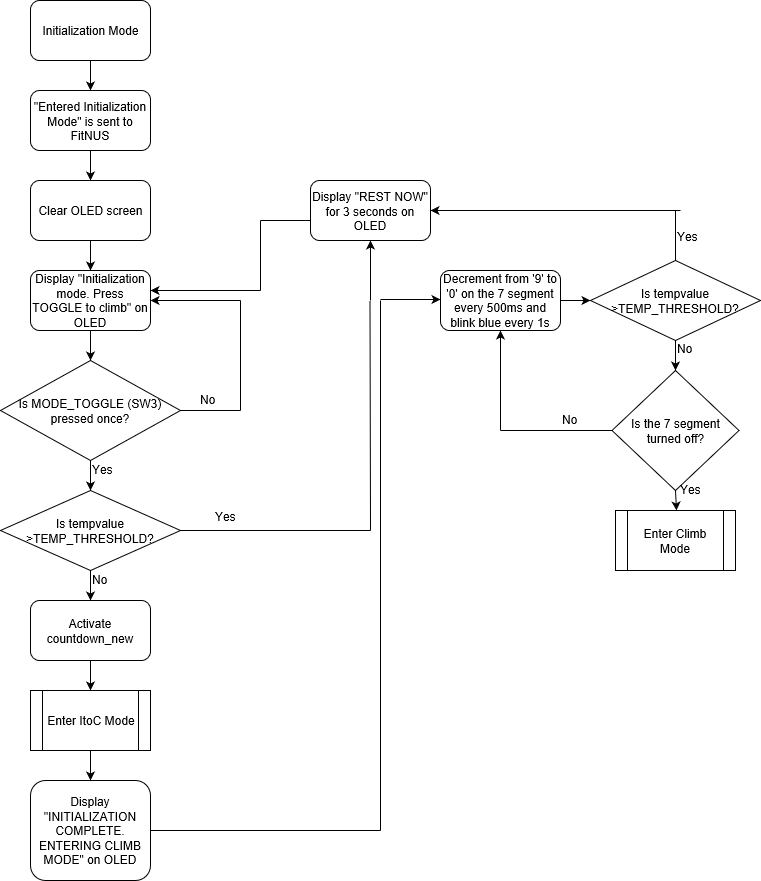
**FitNUS** has 3 modes of operation: Initialization, Climb and Emergency modes, and will transmit to **FiTrackX** is certain conditions are met. We also implemented ItoC and Emergency\_over modes which is later described in our report. Initialization mode would be active when the FitNUS system is first 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.

# Flowcharts describing the system design and processes

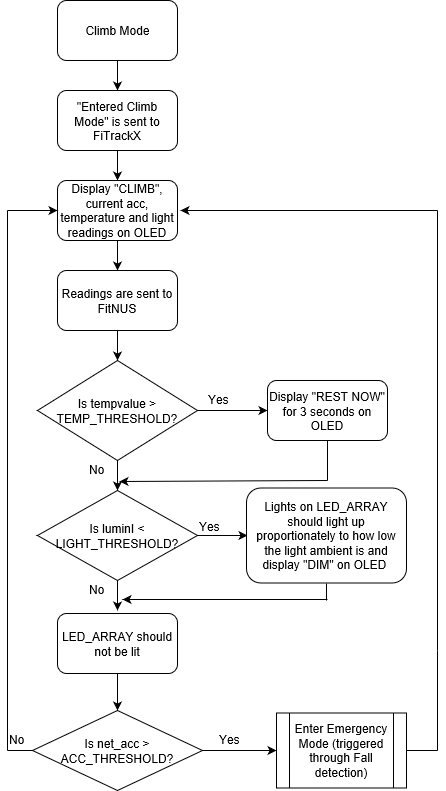
**int main**



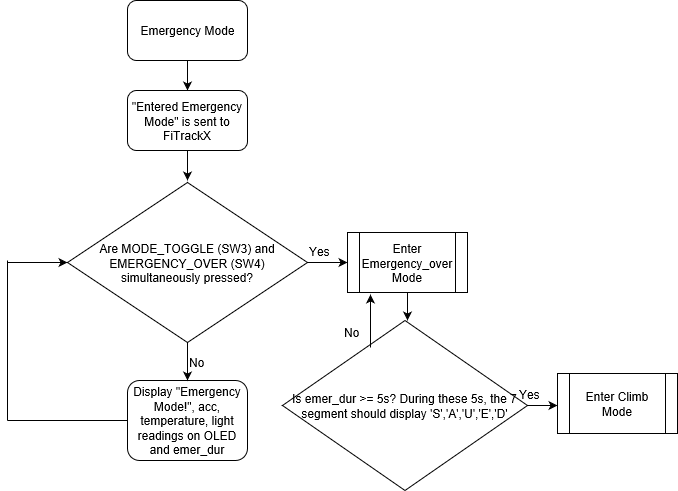
**Initialization Mode**



**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 connected to BL\_EN) 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, EINT0 would be set to a higher priority than EINT3 and SysTick so that the interrupts pertaining to SW3 will come before any other interrupts. | **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);} |

|  |  |  |
| --- | --- | --- |
| In order to deconflict the interrupts used by RGB\_GREEN and OLED caused by the internal connection of the OLED and Green LED, we removed the jumper from PI01\_10 (highlighted in the diagram below) to prevent any possible interferences. For the blue and red LEDs to alternate every 500ms, the green LED should also be turned off throughout.  |  | | --- | | **void** **ALTERNATE\_LED**(){    **if**(Get\_Time() - prev\_alternateled\_ticks >= 500){  **if**(RGB\_FLAG == 0){  setRGBLeds (RGB\_BLUE);  RGB\_FLAG = 1;  prev\_alternateled\_ticks = Get\_Time();  }**else**{  setRGBLeds (RGB\_RED  RGB\_FLAG = 0;  prev\_alternateled\_ticks = Get\_Time();  }}} | | The reason why the time taken is >= the value that we want is because it takes time for the system to obtain and store values. We cannot determine the exact time taken for this to happen because the time taken is negligible. However, if we just assume the time taken is exactly 500ms, the code will not be able to run. |

## General Code Structure

The general code structure that governs each state routine and their subroutines (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

}}

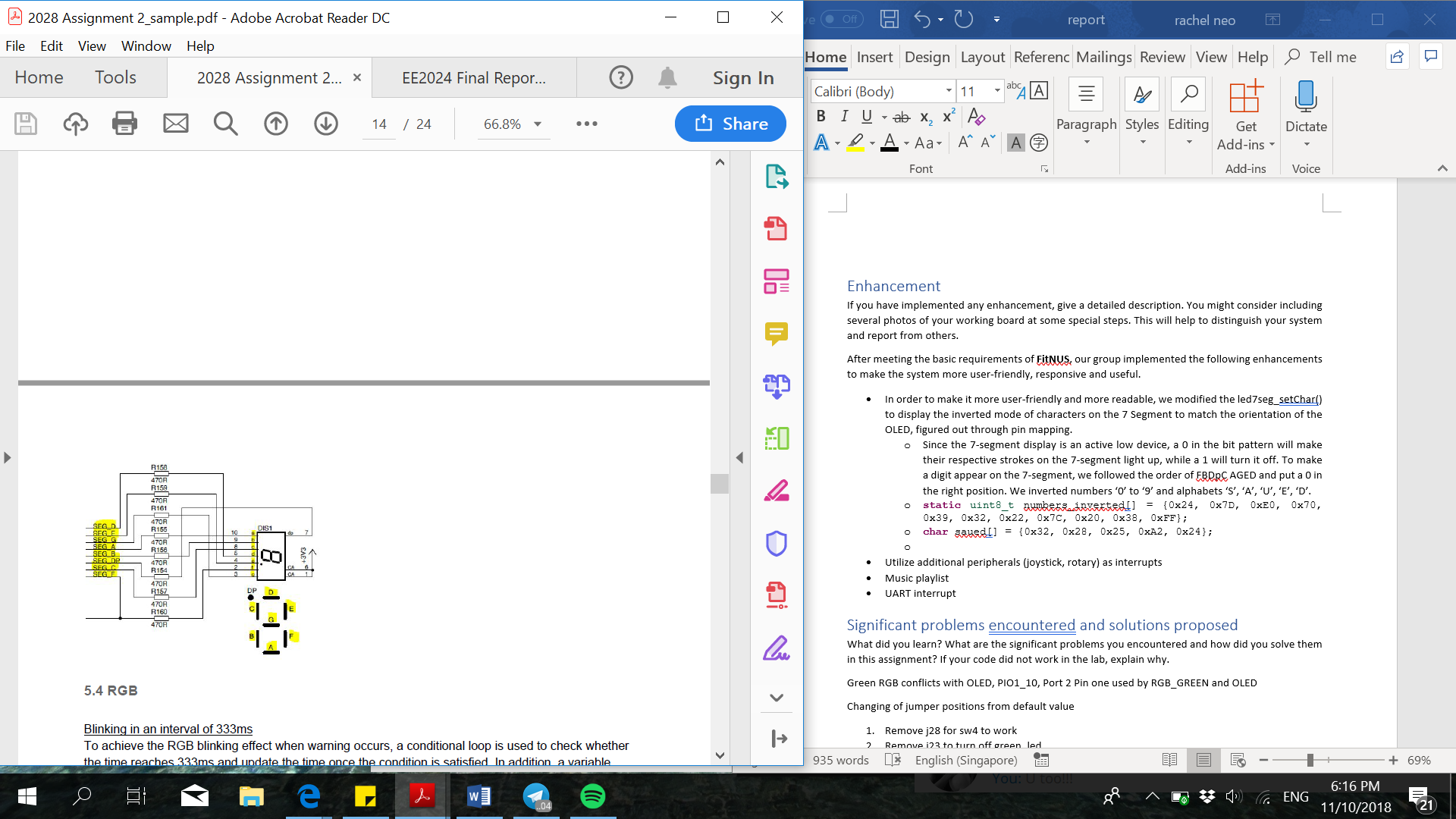
1. Secondly, after the initialization of the state is executed in the if block, there is a while conditional that cycles through instructions to update values and outputs as the programme remains in the state. For example, refreshing sensor readings and refreshing their values on the OLED.

# 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’ and alphabets ‘S’, ‘A’, ‘U’, ‘E’, ‘D’.
* **static** uint8\_t numbers\_inverted[] = {0x24, 0x7D, 0xE0, 0x70, 0x39, 0x32, 0x22, 0x7C, 0x20, 0x38, 0xFF};
* **char** saued[] = {0x32, 0x28, 0x25, 0xA2, 0x24};

## Music playlist

### 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 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.

### Frontend: Graphical User Interface Design

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

* Up/ Down: scroll up down for song selection when no song is playing, next song/ previous song 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 variables.

### Backend: Song Library Storage and Processing

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 second song in the array, we do not have to recurse through the first song to reach the address of the second song and can instead move to the next element in songs.

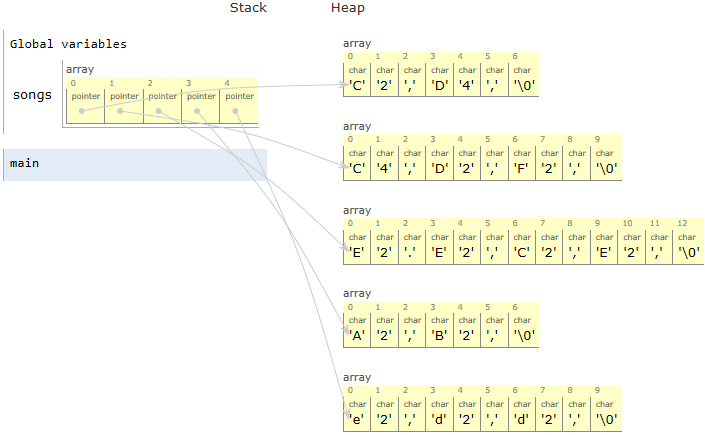


Figure 1: Song Library

To keep track of the progress as songs are played, we used 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 address of the start 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.

## UART interrupt

# 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

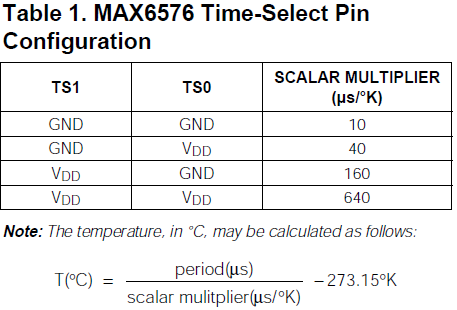
temp\_periods++;

**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;

}

## Green RGB conflicts with OLED

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

…

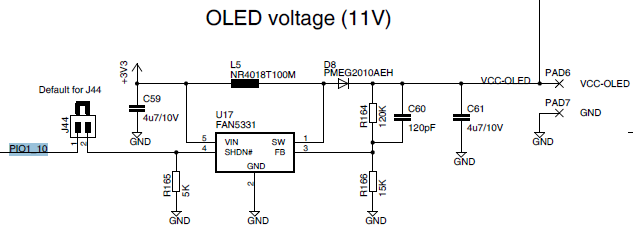
**if** ((ledMask & RGB\_GREEN) != 0)

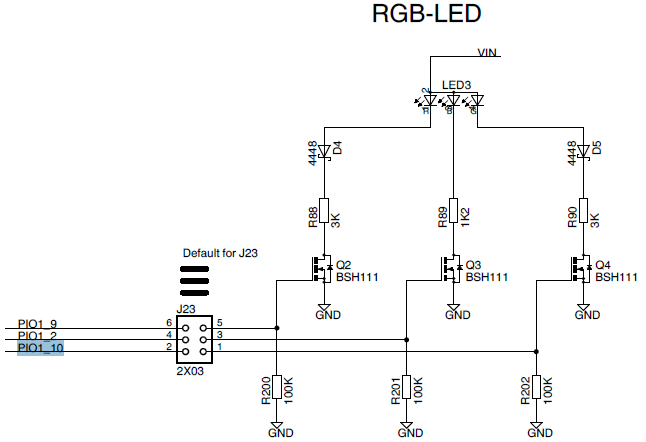
GPIO\_SetValue( 2, (1<<1) );

**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.

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.



## 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.

## Blue RGB conflicts with Speaker

# 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