

EE2028

Microcontroller Programming and Interfacing

Project Report

Neo Yu Yao Terence A0164651E

Neo Jia Wen Rachel A0176763R

GA: Ng Yu Wei, Kenneth



FitNUS Fitness Tracking System
Prototype Design

1 Table of Contents

2	Introduction and objectives.....	2
3	Flowcharts describing the system design and processes	2
3.1	Main Programme Outline	2
3.2	Initialization and ItoC Modes	3
3.3	Climb Mode and Climb Subroutines	4
3.4	Emergency and Emergency_over Modes	5
4	Detailed implementation.....	6
4.1	Temperature sensor – Body temperature monitoring	7
4.2	Light sensor – Ambient light detection	7
4.3	Accelerometer – Fall detection	7
4.4	Wireless UART	8
4.5	EINT0 for sw3	8
4.6	General Code Structure	8
5	Enhancement.....	9
5.1	7 Segment	9
5.2	Debugger Mode with UART interrupt.....	9
5.3	Music Player	10
5.3.1	Motivation.....	10
5.3.2	Frontend: Graphical User Interface Design	10
5.3.3	Backend: Song Library Storage and Retrieval	11
5.3.4	Backend: Processing.....	11
6	Significant problems encountered and solutions proposed.....	12
6.1	Reading of Temperature Sensor causes system lag	12
6.2	Jumper Modification for Pin Conflicts	13
6.2.1	Green RGB conflicts with OLED	13
6.2.2	Blue RGB conflicts with Speaker	13
6.2.3	Default Jumper settings prevents SW4 functionality	13
6.3	Calibration of TEMP_THRESHOLD	13
6.4	UART Interrupt not working because of Tera Term Terminal Setup	14
7	Issues or suggestions.....	14
8	Conclusion	14

2 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**.

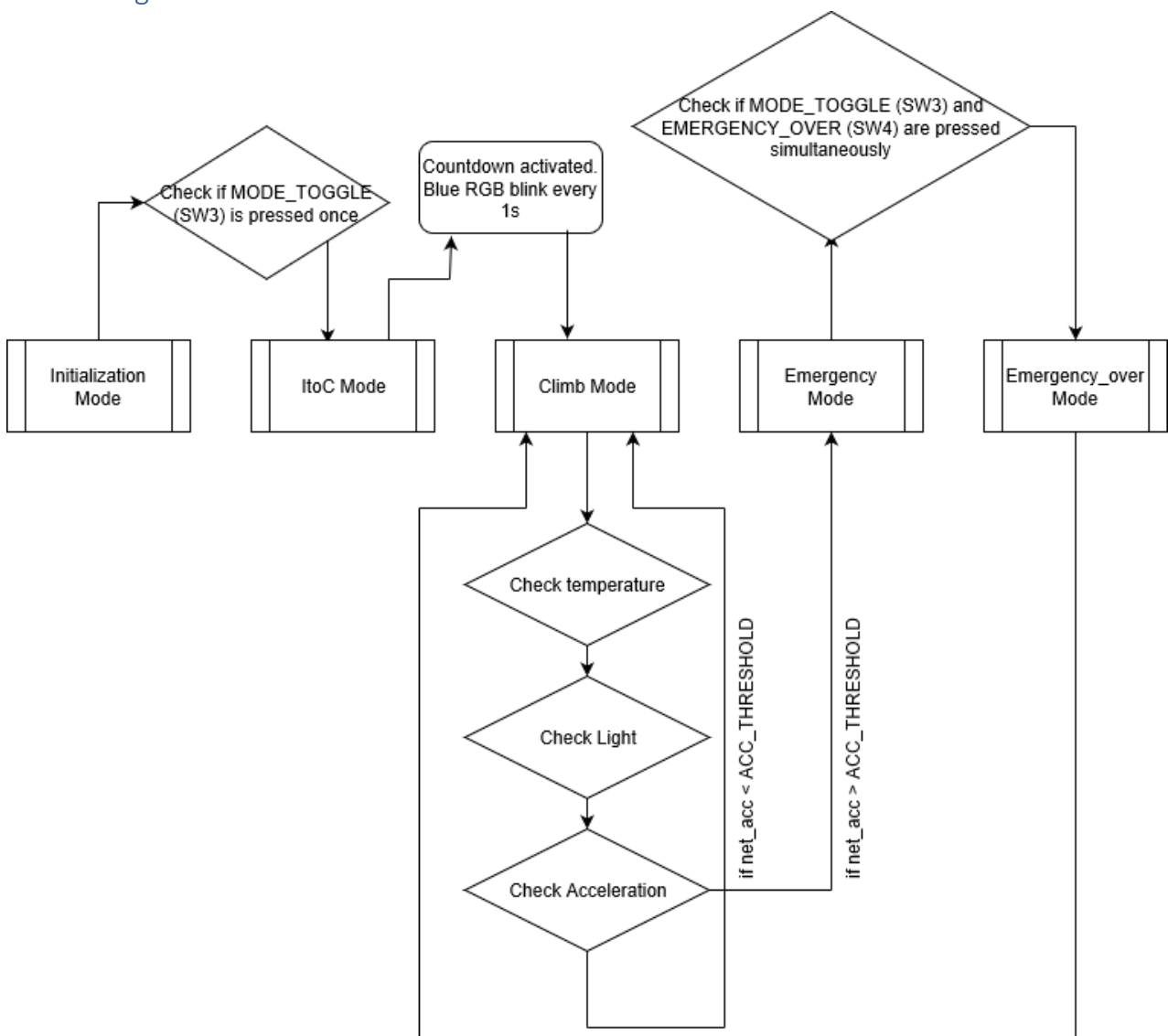
FitNUS has 3 modes of operation: Initialization, Climb and Emergency modes, and will transmit to **FiTrackX** is certain conditions are met. We also implemented ItoC and Emergency_over modes which is later described in our report. Initialization mode would be active when the FitNUS system is first switched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

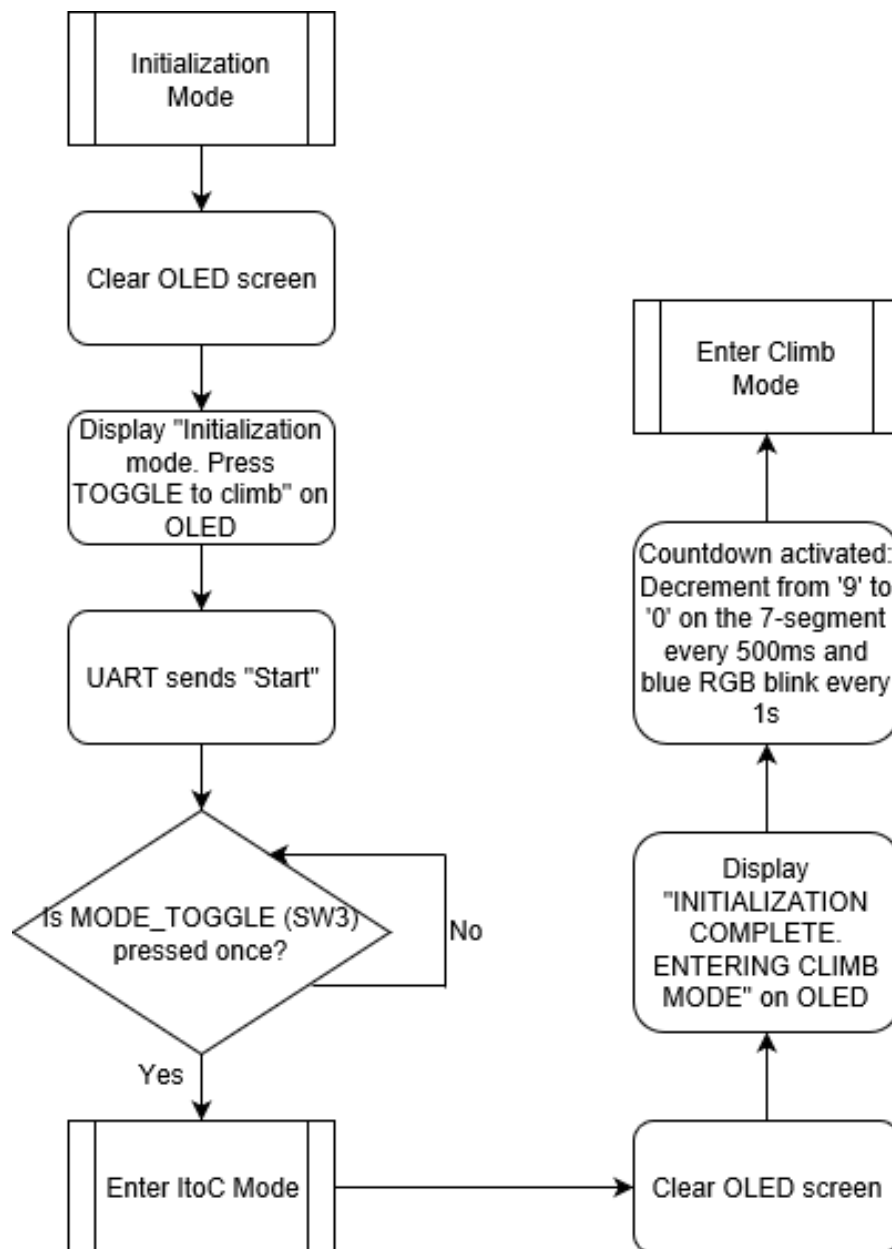
After completing assignment 2, we would have learnt how to apply system design approaches, such as using flowcharts, to design embedded applications, understand the interfaces between microcontrollers and peripherals and have the ability to develop C embedded programming controller-based applications.

3 Flowcharts describing the system design and processes

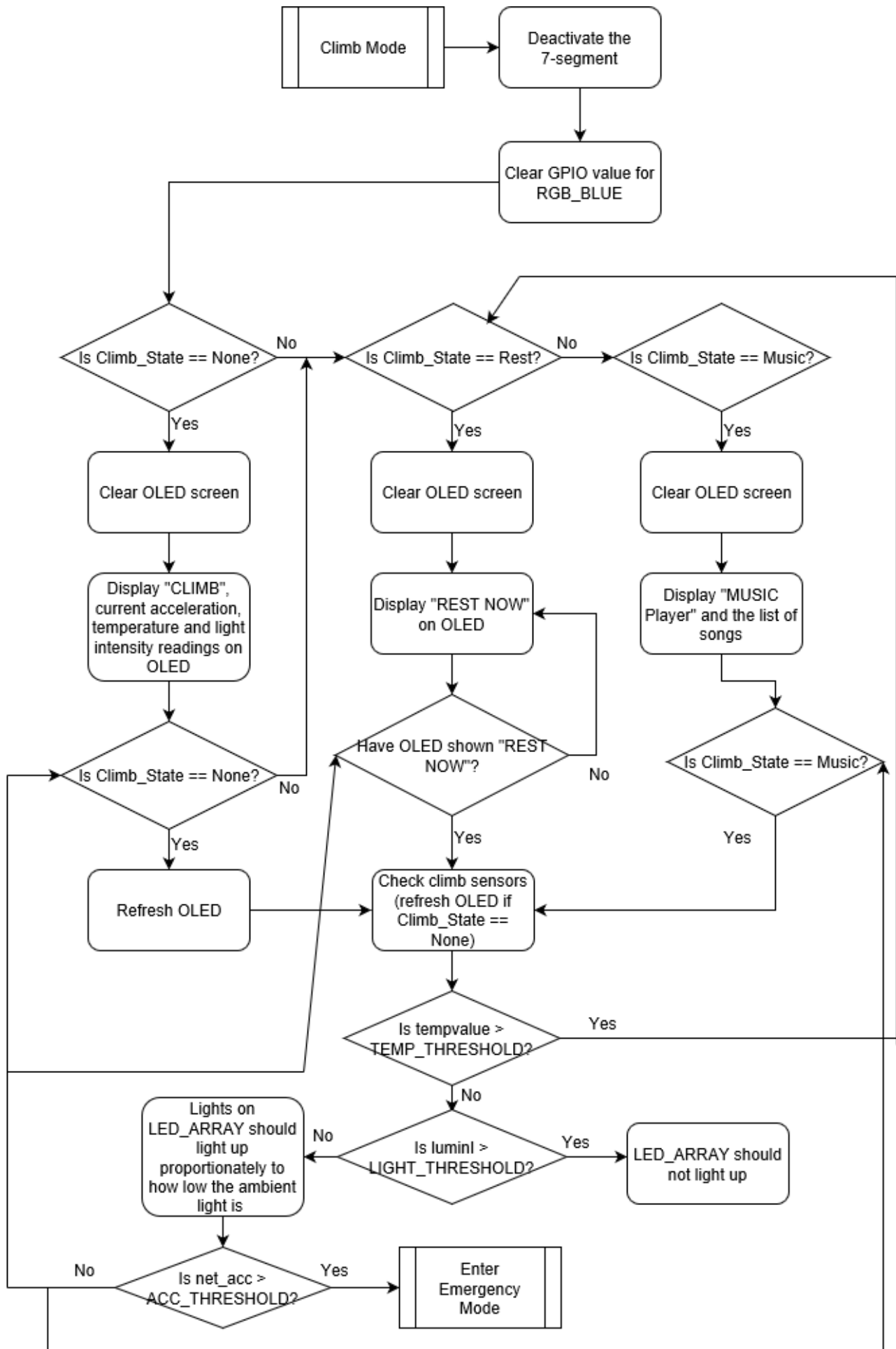
3.1 Main Programme Outline



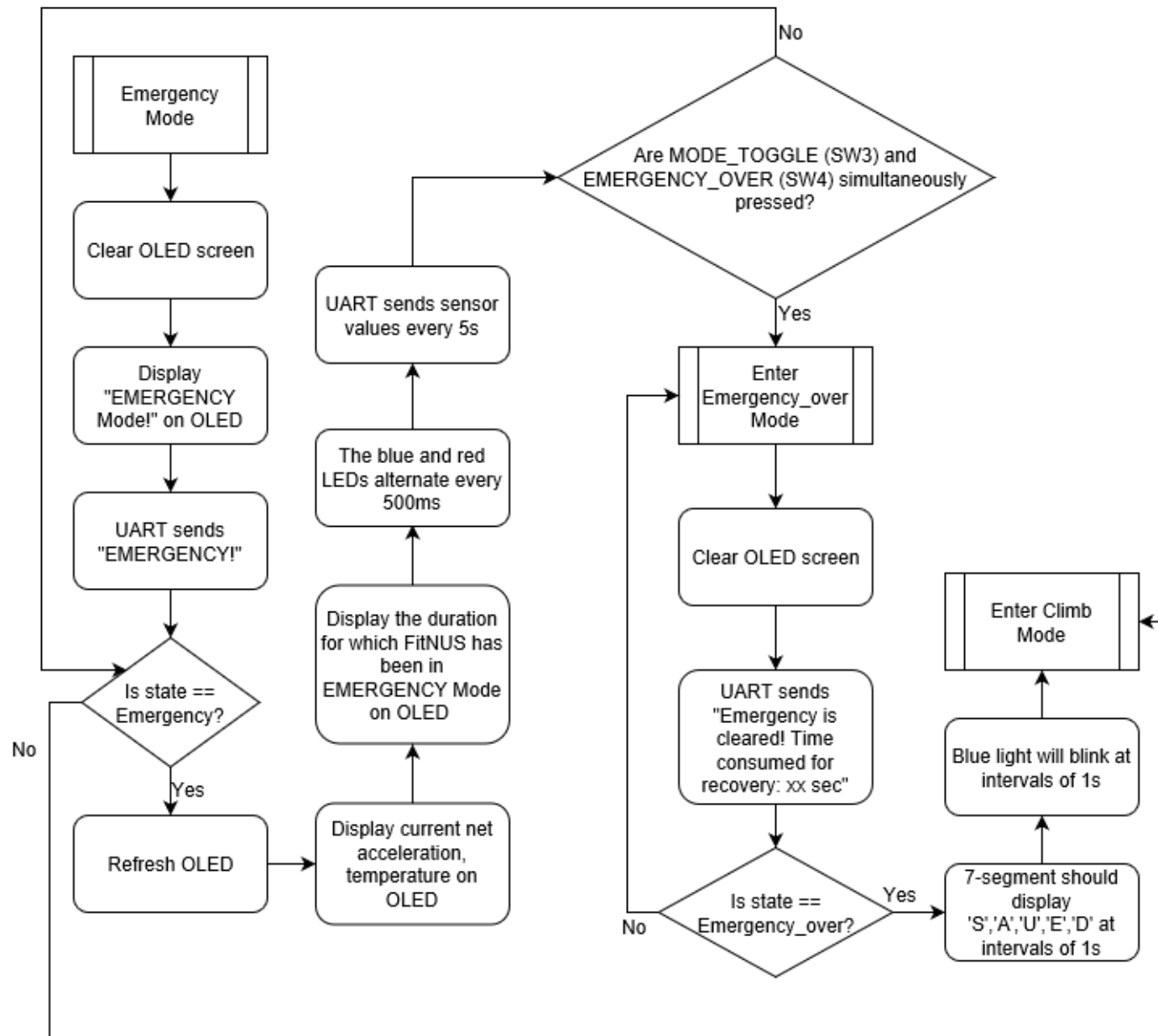
3.2 Initialization and ItoC Modes



3.3 Climb Mode and Climb Subroutines



3.4 Emergency and Emergency_over Modes



4 Detailed implementation

When the system is first switched ON, all the peripherals will be initialized and the interrupts will be enabled. **FitNUS**

```
int main (void) {
    init_everything();
    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();
        }
    }
}
```

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 display the values obtained. When fall detection is triggered, by gently shaking the board, **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 protocols and 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** (temperature sensor and all functions of the joystick) and **UART3** (RDA Rx Interrupt). The function will clear any interrupt pending first before clearing the interrupt flags, configuring them and then enabling the interrupt handler. The function will then enable the respective interrupts. For instance, to trigger the GPIO interrupt at the Rising edge of the temperature sensor reading, a ‘1’ would be shifted 2 bits into IO0IntEnR, for P0.2, after which the **EINT3** interrupt handler will be enabled.

Furthermore, as lower priority level will mean higher pre-empt priority, **SysTick_IRQn** will be set at pre-empt priority level of 0 to give it the highest priority, 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 second most important. If one **EINT3** interrupt time step is missed, temperature goes slightly off. **EINT0** is used for SW3. Between SW3 and UART, we chose SW3 to be more important so the system will be more responsive to sw3 in the event a UART Character is sent when sw3 is pressed.

```
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_SC -> EXTINT |= 1<<0; //Clearing EINT0
LPC_SC -> EXTMODE |= 1<<0; //edge sensitive
LPC_SC->EXTPOLAR&=0xFFFFF0; //falling-edge
NVIC_ClearPendingIRQ(EINT0_IRQn);
NVIC_EnableIRQ(EINT0_IRQn);
```

```
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);
}
```

4.1 Temperature sensor – Body temperature monitoring

While in Climb mode, the temperature sensor should display the temp reading on the OLED in the following format: “Temp: xx.x deg”. If temp exceeds TEMP_THRESHOLD, the OLED displays “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.

```
void check_ClimbSensors() {
    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;
        }
        else if (tempvalue <= TEMP_THRESHOLD && temp_flag == 1){
            temp_flag = 0;}}}

```

4.2 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”. 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.

Instead of hardcoding how many LEDs will be lit for different light intensities using conditionals, we used binary arithmetic to set a mask for the LEDs to be turned on for a given value of the light sensor reading.

Every 18.75 decrement in the light sensor reading, luminI, from 300 to 0, an additional LED will be lit in the LED_ARRAY, up to a maximum of 16 LEDs when the luminI=0. If the light sensor reading falls between 2 decrements of 18.75, it will be rounded off to the nearest 18.75 for calculations. For example, when luminI=265, we get the following:

$$\begin{aligned} 300 - 2 \times 18.75 < \text{luminI} = 265 < 300 - 1 \times 18.75 \\ \text{shift} &= \frac{\text{luminI}}{18.75} \approx 14, \quad \text{rounded off as shift is defined as an int} \\ \text{ledOn} &= [1 \ll (16 - \text{shift})] - 1 \\ &= [1 \ll 2] - 1 \\ &= 0b0000\ 0000\ 0000\ 0011 \end{aligned}$$

Sending this value as an input into the library function `pca9532_setLeds()` will turn on the last 2 LEDs and turn off all other LEDs in the LED_ARRAY. This is used for controlling 16 LEDs over and the I2C bus, including the logic to act as an I2C slave device and the drive capability for directly driving LEDs.

```
void check_ClimbSensors() {
    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){
            state = Emergency;
        }}}

```

```
void check_ClimbSensors() {
    if(light_on){
        light_enable();
        luminI = light_read();
        shift = luminI / 18.75;
        ledOn = (shift <= 16)?(1<<(16-shift))-1:0;
        pca9532_setLeds(ledOn, 0xffff);
    }
    // turns on ledOn and off everything
    else, ledOn takes priority
}

```

4.3 Accelerometer – Fall detection

While in Climb mode, Emergency mode may be triggered through fall detection by gently shaking the board, 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”, in ‘g’s (1g = 9.8m/s²).

4.4 Wireless UART

UART is a standard for long-distance, asynchronous, serial, full-duplex peer-peer communication. We configured the UART so that Baud rate value is of 115200. 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 LPCXpresso Baseboard.

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 refers to `emer_dur`.

```
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);
}

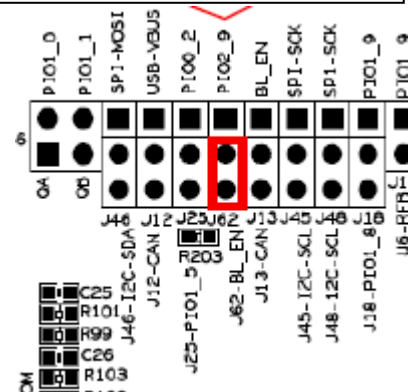
sprintf(uart_msg, "Emergency is cleared!
Time consumed for recovery: %lu
sec\r\n", emer_dur);
UART_Send(LPC_UART3, (uint8_t
*)uart_msg, strlen(uart_msg), BLOCKING);
```

4.5 EINT0 for sw3

We initialized the interrupts for `EINT0` for the use of SW3. By taking sw3 out of `EINT3` and placing it in a separate interrupt, we will not only be able to minimise the conditionals to be checked in `EINT3`, but will also be able to set different priority levels for interrupts triggered by sw3 and other GPIO inputs thus resulting in a faster, more responsive system. While in `Initialization` mode, if SW3 is pressed, FitNUS enters `ItoC` mode. While in `Climb` mode, if SW3 is pressed, `TEMP_THRESHOLD` calibration will take place. Lastly, while in `Emergency` mode, when SW3 and SW4 are pressed together, FitNUS will enters `Emergency_over` mode.

We enabled `EINT0` to be falling edge sensitive to detect when sw3 is depressed. This is done by writing to registers `EXTMODE` and `EXTPOLAR` as shown at the start of section 4. For `EINT0` to work, we removed the jumper indicated in the diagram and connected sw3 to `PIO2_9` instead.

```
void EINT0_IRQHandler(void) {
    if ((LPC_SC -> EXTINT >> 0) & 0x1) {
        LPC_SC -> EXTINT |= (1<<0);
        if (state == Initialization) {
            state = ItoC;
        } else if (state == Climb) {
            //TEMP_THRESHOLD Calibration
            TEMP_THRESHOLD = tempvalue + 5;
        } else if (state == Emergency &&
        (((GPIO_ReadValue(1) >> 31) & 0x1) == 0)) {
            state = Emergency_over;
        }
    }
}
```



5 Enhancement

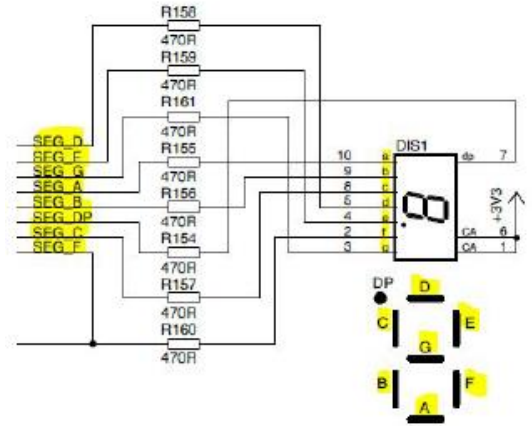
If you have implemented any enhancement, give a detailed description. You might consider including several photos of your working board at some special steps. This will help to distinguish your system and report from others.

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

5.1 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-seg display is an active low device, a 0 in the bit pattern will make their respective strokes on the 7-seg light up, while a 1 will turn it off. To make a digit appear on the 7-seg, 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};`



5.2 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'){
                rxbuf[rx_count+1] = 0; //end of string
                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;
            }
            else rx_count=(rx_count== 63)?0:rx_count+1;
        }
    }
}
```

```
Start
CLIMB mode
Acc: 0.02 g Light: 13 lux Temp: 27.5 deg
Acc: 0.03 g Light: 35 lux Temp: 27.7 deg
Acc: 0.03 g Light: 36 lux Temp: 27.7 deg
EMERGENCY!
Temp: 27.9 deg
Emergency is cleared! Time consumed for recovery: 6 sec
CLIMB mode
Acc: 0.02 g Light: 13 lux Temp: 27.7 deg
Acc: 0.02 g Light: 33 lux Temp: 27.7 deg
Acc: 0.02 g Light: 32 lux Temp: 27.7 deg
Acc: 0.02 g Light: 15 lux Temp: 27.7 deg
Acc: 0.04 g Light: 32 lux Temp: 27.7 deg
Acc: 0.03 g Light: 30 lux Temp: 27.7 deg
Acc: 0.02 g Light: 32 lux Temp: 27.7 deg
Acc: 0.04 g Light: 33 lux Temp: 27.7 deg
Acc: 0.02 g Light: 32 lux Temp: 27.5 deg
Acc: 0.02 g Light: 32 lux Temp: 27.7 deg
Acc: 0.03 g Light: 34 lux Temp: 27.7 deg
Error: Unrecognized Command
Accepted Commands:
On Acc
Off Acc
On Temp
Off Temp
On Light
Off Light
Acc: 0.02 g Light: 35 lux Temp: 27.7 deg
Acc: 0.02 g Light: 16 lux Temp: 27.7 deg
Off Temp
Acc: 0.02 g Light: 23 lux
```

We experienced system lag when playing songs in our music function. Hence, to debug our fitness tracker prototype, we implemented a way to turn on and off the various sensors while the system is running in the form of a UART interrupt. With this, we were able to identify the temperature sensor as the cause of the lag (we then fixed this lag in section 6.1). To execute this Enhancement, type in a command in TeraTerm and press 'Enter'. For instance, typing "Off Temp", all the functions corresponding to temperature, such as reading the sensor, displaying on OLED and TermTerm as well as triggers caused by the sensor, will be disabled. If a command not implemented is entered, an error message will pop up which includes a list of recognised commands. We implemented this function into all the sensors: Accelerometer, Temperature and Light sensors.

5.3 Music Player

5.3.1 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.



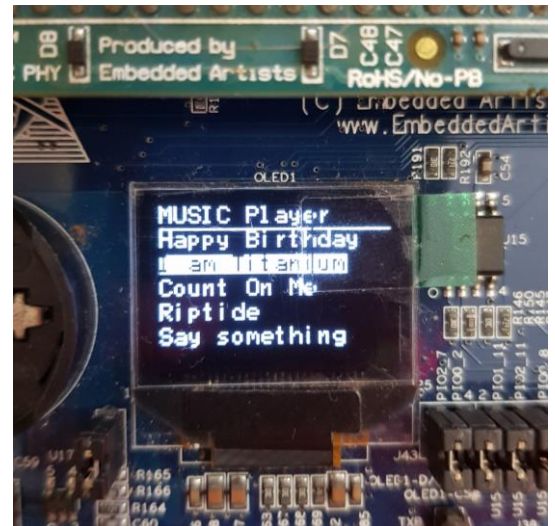
5.3.2 Frontend: Graphical User Interface Design

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 `scroll_updated` flag, 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],...
}
```

5.3.3 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.

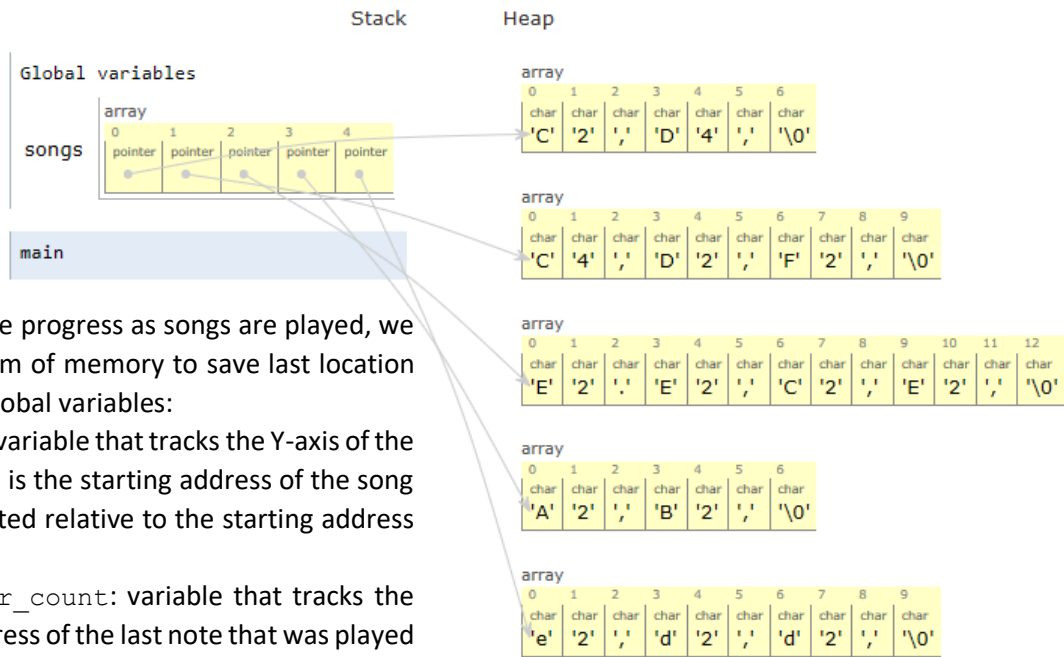


Figure 1: Song Library

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

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 24). We increase this relative pointer, `song_pointer_count`, each time a note finishes playing.

```

1  static void playSong(uint8_t *song) {
2      .....
3      if(song_changed){
4          song_pointer_count = 0;
5          song_changed = 0;
6      } else{
7          while(i < song_pointer_count){
8              *song ++;
9              i++;}
10         if(*song != '\0' && play_flag){
11             note = getNote(*song++);
12             if (*song == '\0'){
13                 song_pointer_count = 0;
14                 play_flag = 0;
15                 return;}
16             dur = getDuration(*song++);
17             if (*song == '\0'){
18                 song_pointer_count = 0;
19                 play_flag = 0;
20                 return;}
21             pause = getPause(*song++);
22             playNote(note, dur);
23             Timer0_Wait(pause);
24             song_pointer_count += 3;}
25         if (*song == '\0'){
26             song_pointer_count = 0;
27             play_flag = 0;}

```

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).

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.

5.3.4 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_flag == 1`. When play flag is toggled to 0, the last location played is saved and the song is paused.

6 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.

6.1 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 and also to increase the precision of the sensor. 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 7 periods to get an acceptable precision for our reading. We have hence written our own `temp read` function using an interrupt with the formula given in the datasheet, we can define our `fast_temp_read()` function as shown.

```
void EINT3_IRQHandler(void) {
    // Temperature sensor
    if ((LPC_GPIOINT ->I00IntStatR >> 2) & 0x1) {
        LPC_GPIOINT ->I00IntClr = 1 << 2; //clear the interrupt
        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;
    }
```

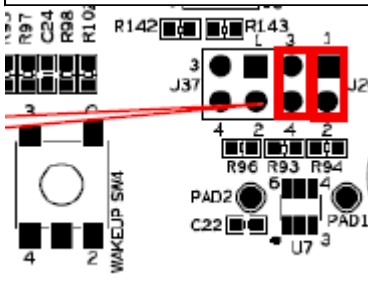
```
// Replacement for slow temp_read() driver function
int32_t fast_temp_read (void){
    if (~t2 && ~t1){
        time_diff = abs(t2-t1);
        temp = ((time_diff*1000.0)/(temp_periods*10*TEMP_SCALAR_DIV10))-273.15;
    }
    if (temp < 16) temp = 0;
    // printf("Temp: %f\n", temp); //offending line
    return temp*10;
}
```

Table 1. MAX6576 Time-Select Pin Configuration

TS1	TS0	SCALAR MULTIPLIER ($\mu\text{s}/^{\circ}\text{K}$)
GND	GND	10
GND	VDD	40
VDD	GND	160
VDD	VDD	640

Note: The temperature, in $^{\circ}\text{C}$, may be calculated as follows:

$$T(^{\circ}\text{C}) = \frac{\text{period}(\mu\text{s})}{\text{scalar multiplier}(\mu\text{s}/^{\circ}\text{K})} - 273.15^{\circ}\text{K}$$



We chose to set TS1 and TS0 both to 1 by removing the jumpers as illustrated. This gives the greatest scalar multiple in the denominator of the formula, resulting in the smallest change in temp per change in unit

temp_diff and hence a higher sensor precision. By choosing to sample the timing every `temp_periods=7` number of temperature sensor periods, we get the precision of our `fast_temp_read()` function to be as follows:

$$\text{precision} = \frac{1\text{ms} \times 1000}{\text{temp periods} \times \text{scalar}} = \frac{1000}{7 \times 640} = 0.22^{\circ}\text{C}$$

Increasing the number of periods required to get a sensor reading will increase the precision of the sensor, but it comes at the expense of system responsiveness with a greater delay before a change in temperature is displayed. We experimented with different `temp_periods` values and found that 7 periods was the most optimal given our purposes.

Problems Encountered

One problem we faced when implementing this solution was that adding a `printf()` anywhere (such as that in the code above) affects interrupt handling and hence timing, which in turn affects the reading of the temperature. Each call to `printf()` takes significant time to process and execute: about 100,000 instructions or 10 milliseconds, depending on the clock speed. This was unintuitive as interrupts are expected to pre-empt all other tasks in thread mode no matter what the task was. A `printf()` was expected to be no different. We removed the offending line, and all was working again.

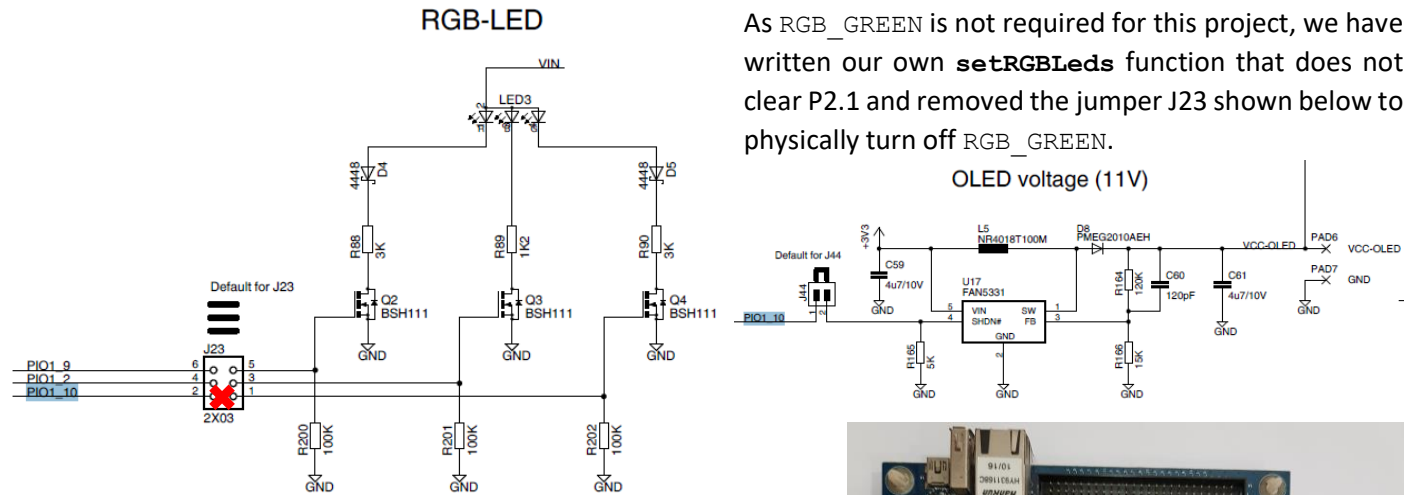
6.2 Jumper Modification for Pin Conflicts

6.2.1 Green RGB conflicts with OLED

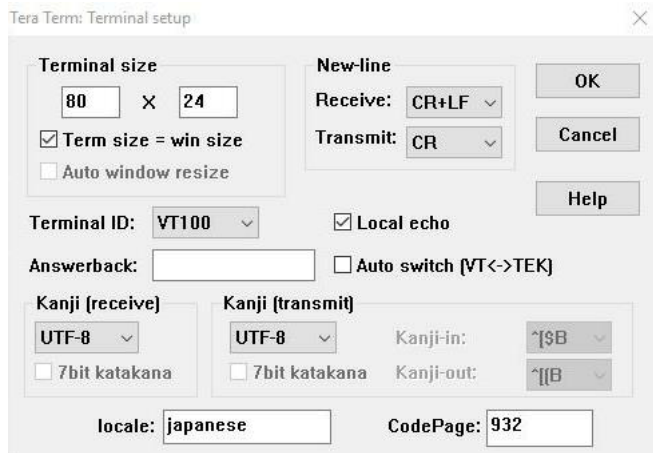
PIO1_10, which is connected to P2.1 is used by both RGB_GREEN and OLED voltage. The default driver function `rgb_setLeds` clears P2.1 each time the function is called and RGB_GREEN is not turned on, affecting OLED functionality. For the blue and red LEDs to alternate every 500ms, the green LED should also be turned off throughout.

```
void rgb_setLeds (ledMask) {  
    ...  
    if ((ledMask & RGB_GREEN) != 0)  
        GPIO_SetValue( 2, (1<<1) );  
    else  
        GPIO_ClearValue( 2, (1<<1) );  
}
```

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.



6.4 UART Interrupt not working because of Tera Term Terminal Setup



We are tasked to have the new-line characters set as CR+LF. However, it didn't work until we tweaked the new line for transmission to be set as CR. When enter is registered as `\r`, we will only need to check for 1 character (`\r`) to know that enter is pressed, as opposed to checking for 2 characters (`\r\n`). For CR+LF, enter is registered as `\r\n`, which is why we use this as new-line for `uart_send`. 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.

7 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!

8 Conclusion

We came to a conclusion to know how important an embedded system is and what it is capable of doing. Embedded systems often work with real-time computing constraints, where the response to external event has hard boundary for execution. Embedded systems are designed with a purpose that is unique to their user/s. They are able to perform specific functions, within a range of harsh environments. Thus, they work really well for applications where response to an external event is critical.

Furthermore, we learnt how and when to use keywords to store data such as `uint_t`, `static` and `volatile`, different types of interrupts, UART, how to work with Tera Term and also how the device communicates with the processor, etc.

We learnt how to design this for real life situations (i.e. fitness tracker) which are very much in used in today's context and how to compile and make the code a lot less complex by calling it outside of the main function. This project has given us thorough and great insights on what we might possibly face in our near future career.