EE2024 Assignment 2 Project Report
Frequency Fluttering System

Muhammad Muneer B Gulam M A0101168R
Chew Yi Ming Nicholas A0101885E

Graduate Assistant: Kang Seungmin

1.0 Introduction

In this assignment, we will be implementing a Frequency Fluttering System (FFS). The main purpose of the FFS is to ensure the safety of the airplane by detecting any potential structural failures at the wing. The system achieves its purpose by monitoring the frequency of the flutter of the wings and to send an alert warning signal to the monitoring system when the plane is compromised.

There are three main modes namely Calibration mode, Standby mode and Active mode:

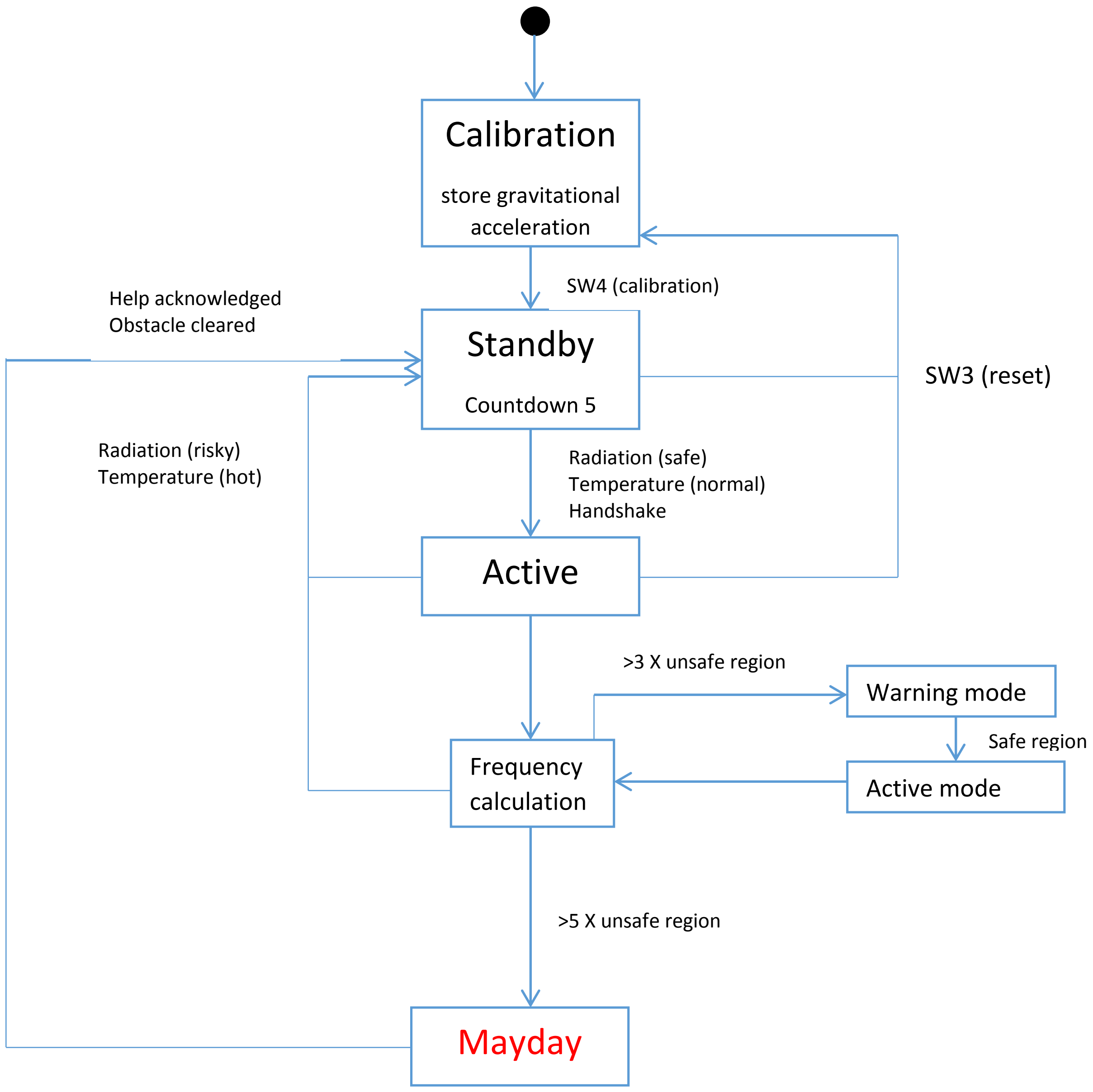
- Calibration is the first mode that will be entered when the FFS is started. This mode primarily calculates and stores the initial gravitational acceleration.
- The second mode is the Standby Mode. This mode ensures a smooth transition into the Active Mode. It only moves to the other modes if the airplane is in a controlled environment. In addition, it is also responsible to “handshake” with the monitoring station.
- The last mode is the Active Mode. In this mode, the system monitors the frequency of the flutter and regularly updates the monitoring station.

Lastly, we have extended the functionality of the system to include a May Day Mode. This mode is activated when the airplane is experiencing the flutter of a prolonged period. Further detail will provided in the upcoming sections.

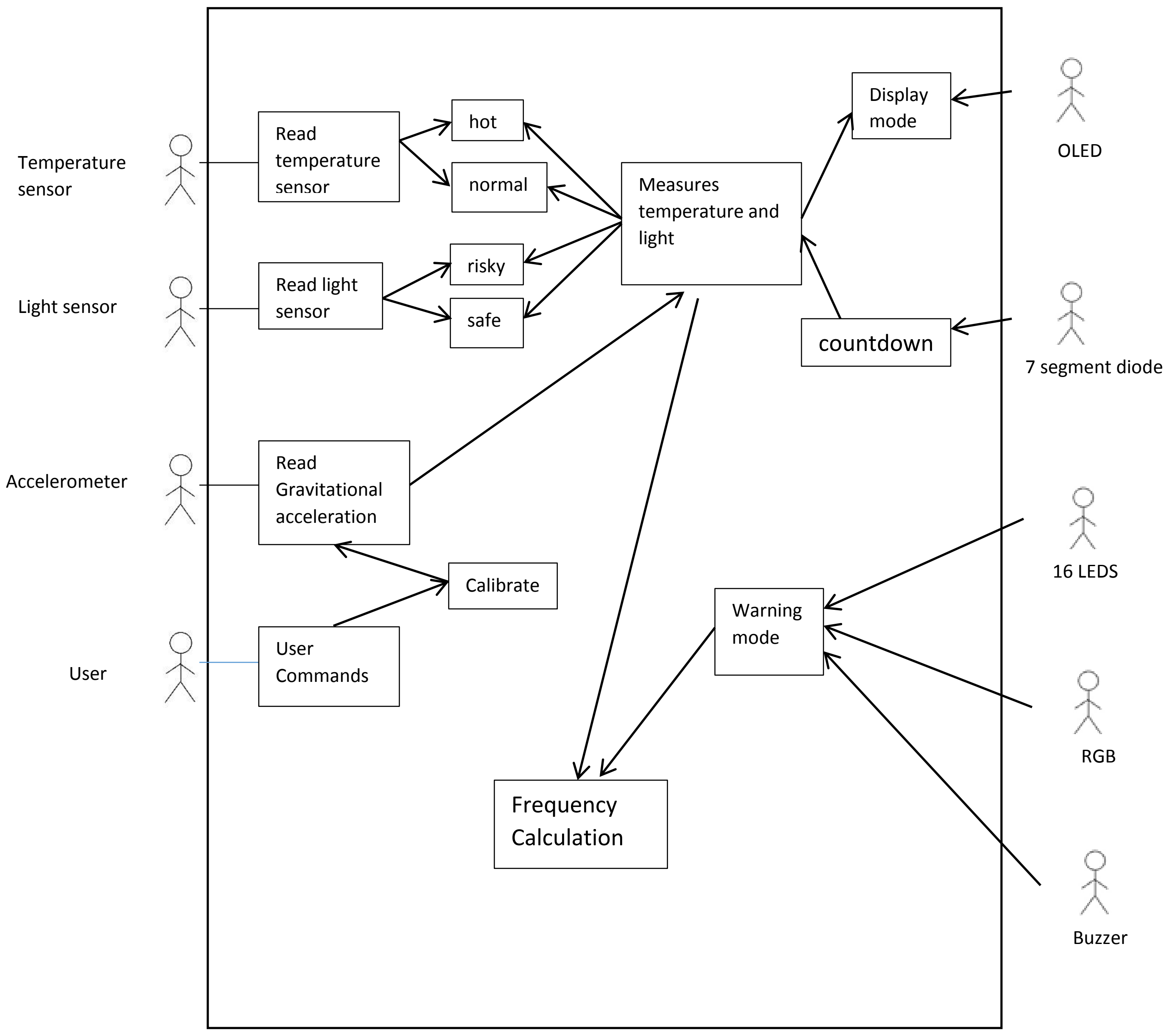
2.0 UML Diagram

The following section contains both UML State Chart Diagram and UML Use Case Diagram of the Frequency Flutter System.

2.1 State Chart



2.2 Use Case Diagram



3.0 Detailed Implementation

The Frequency Flutter System operates in three modes: Calibration, Standby and Active, as seen in the UML diagrams above. Now, let's take a look into the details of every mode.

3.1 The Calibration Mode

The calibration mode is the first mode that the Frequency Fluttering System immediately enters when the system starts. Its primary objective is to allow the user to calibrate the gravitational acceleration.

I. Initialisation

```
void calibrateInit(void) {
      disable7Segment();
      disableRGB();
      disableResetBtn();
      enableCalibrateBtn();
      initAccelerometer();
      disableLEDS();
      initOled();
}
```

Figure 1: Initialisation of Calibration Mode

From Figure 1, it can be seen that we are enabling peripherals that will be used in this mode such as accelerometer, OLED Display and SW4 button. In addition, we are also disabling peripherals that are not used in this mode, although this was not explicitly stated in the project manual.

For example, we are disabling the reset button since we feel that the system had just begun and no calibration has been done, and so there is no reason to allow the user to reset the system

II. Calibration of Gravitational Acceleration

```
uint8_t isCalibrated(uint8_t* accReading){

        int8_t x,y,z;

        acc_read(&x,&y,&z);

        uint8_t input = (GPIO_ReadValue(1)>>31)&0x01;

        *accReading = z;

        return !input;

}
```

Figure 2: Calibration

The function in Figure 2 is constantly called in this mode. When the user presses the calibration button (SW4), the z-value of the accelerometer, the gravitational acceleration, is stored in a global variable, *accReading*. In addition, it also causes the value of variable *input* to change. As a result, the programs exits the Calibration Mode and enters the Active Mode.

It is important to note that since this function is constantly called, awaiting for user's input. Therefore, the variable *input* is updated via the polling method. Polling is inefficient in comparison with other methods such as, interrupt. However, since SW4 is configured using GPIO port 1, it can't be configured as a GPIO interrupt.

3.2 Standby Mode

Standby Mode is the second mode of the FFS. This mode ensures a smooth and safe transition to the next mode. It is responsible in ensuring the system is in a controlled environment: the current temperature and radiation level. In addition, handshaking with monitoring station is established.

I. Measuring Temperature

```
void runTemp(int* tempBool) {

        int32_t tempRead = temp_read();

        *tempBool = (tempRead/10.0 < TEMP_THRESHOLD);

        displayTemp(tempRead,*tempBool);

}
```

Figure 3: Measuring the Temperature

Figure 3 contains the function that measures the temperature, compares it with the constant *TEMP_THRESHOLD* (value of 26) and calls another function *displayTemp* with the necessary parameters to display the appropriate value and status (NORMAL/HOT) on the OLED. It is interesting to note that this function is polled in the Standby Mode. Since temperature readings have to be read regularly, it is not appropriate to configure it as an interrupt.

```
static void initTemp() {
        .
        .
        .
    GPIO_SetDir(0, 1<<2, 0);
    temp_init(&getSystick);
}
```

Figure 4a: Initialising the temperature sensor

```
void SysTick_Handler(void) {
      msTicks++;
}

uint32_t getSystick(void){
      return msTicks;
}
```

Figure 4b: Initialising the temperature sensor

Unlike other peripherals, to initialise the temperature sensor, it requires a function pointer that returns the current time. Hence, a *SysTick_Handler* function that increments the current time by 1 every 1000ms and a *getSystick* function that returns the current time in terms of ticks. The address of the latter function was then passed to *temp_init* as a parameter.

II. Measuring Radiation

```
void initLight() {
      LPC_GPIOINT -> IO2IntClr = 1<<5;
      light_clearIrqStatus();
      light_enable();
      light_setRange(LIGHT_RANGE_4000);
      light_setWidth(LIGHT_WIDTH_16BITS);
      LPC_GPIOINT -> IO2IntEnF |= 1<<5;
      light_setHiThreshold(800);
      light_setLoThreshold(0);
      NVIC_EnableIRQ(EINT3_IRQn);
}
```

Figure 5a: Initialising the Light sensor

Since only the status of the radiation level (Safe/Risky) is to be displayed and it is called at unpredictable time, light sensor should be configured as an interrupt. Fortunately, light sensor is a P2.5 peripheral, so it can be configured as a GPIO interrupt.

From the figure above, it can be seen that we cleared any pending interrupt status by writing to the *I02IntClr* register and using the *light_clearIrqStatus* function. This is a good practice, in order to prevent previous interrupt to be called after the enabling of the interrupt.

Next, we increase both the range and precision of the readings. Since the INT port of the light sensor is active low, we enable falling edge interrupt. It is also important to initialise the high and low threshold of the interrupt so that it interrupts when the radiation gets above the high threshold or below the low interrupt.

Lastly, we have to enable the *EINT3_IRQn* interrupt handler since all these interrupt handler is invoked for all GPIO interrupts.

```
void  EINT3_IRQHandler() {
      if ((LPC_GPIOINT -> IO2IntStatF>>5) & 0x01) {
            if(!isInAccRead){
                  isSafe = !isSafe;
                  if (isSafe) {
                        light_setLoThreshold(LIGHT_LOW);
                        light_setHiThreshold(LIGHT_THRESHOLD );
                  }
                  else {
                        light_setHiThreshold(LIGHT_HIGH);
                        light_setLoThreshold(LIGHT_THRESHOLD );
                  }
            }
            LPC_GPIOINT -> IO2IntClr = 1<<5;
            light_clearIrqStatus();
      }
}
```

Figure 5b: Measuring the Temperature

Figure 5b contains the implementation of the GPIO interrupt handler. Since we want to toggle between these two states when the interrupt handler is invoked, we want to keep a flag *isSafe* to keep track of the states and to toggle, we assign *isSafe = !isSafe.*

Next, we want this interrupt handler to be invoked when the radiation level changes from Safe to Risky and vice-versa. Since we want the interrupt to respond to both changes, we have to change the parameters of the high and low threshold when the interrupt is invoked.

For example, let's say currently the flag *isSafe* is 1, which signifies that the radiation is in the safe region. When the radiation level goes beyond 800, the interrupt will be triggered and the parameter will be modified. Thus, the light interrupt will not be continuously triggered even if the radiation level continues to be in the risky region. When the radiation level falls below 800, another interrupt will be triggered and the parameter will be modified yet again.

III. Handshaking with Monitoring Station

```
void UART_INTERUPT(){
                                        .
                                        .
                                        .
                                        .

if(!strcmp(bufferForUART,"RNACK"))

      UART_Send(LPC_UART3, (uint8_t *)"RDY 036 \r\n" , strlen("RDY 036 \r\n"), BLOCKING);

else if (!strcmp(bufferForUART,"RACK")){

      UART_Send(LPC_UART3, (uint8_t *)"HSHK 036\r\n" ,strlen("HSHK 036\r\n"), BLOCKING);
      isEstablished = 1;

}
                                        .
                                        .
}
```

Figure 6: Handshaking UART Interrupt

Figure 6 contains part of the code from the UART Interrupt Handler. The code seen above is responsible for the establishing handshake with the monitoring station.

The basic idea is that the monitoring system will send “RNACK” (Not Acknowledge) when it is not ready to establish handshake and the FFS will continuously prompt the monitoring system until “RACK” (Acknowledge) is received which will set the global flag, *hasEstablished* to 1.

It is important to note that we have designed the system such that one handshake is required as long as the system stays in either the Standby or Active Mode. If the system is reset, all previous information are lost, including the handshake and so handshake needs to be established again with the monitoring station.

3.3 Active Mode

Active mode is the final mode of the FFS. Its primary role is to measure the frequency of a flutter and determine if it’s within the safety range. A warning will be issued to the monitoring station if the flutter stays in the unsafe zone for the *TIME_WINDOW*.

In addition, this mode will only be entered when handshaking is established, the radiation is in the safe level and the temperature is normal. Any changes in its state will cause the system to go back to the Standby Mode.

I. Calculation of Frequency of flutter

The calculation of the frequency of the flutter is achieved through a 4-staged process. Let’s take a look at one stage at a time.

```
int calculateFreq(){
      int i, j;
      uint32_t runtime;
      int data[41];
      int finalData[37];
      int numOfReadings = 0;
      int numOfSamples = 0;
      int frequency = 0;
      int8_t x,y,z;
      int isMovingUp;
      int isInitialised = 0;
      uint32_t start_time = msTicks;
```

Figure 7a: Declaration of local variables

This stage is rather straight forward, all the local variables that are used in the calculation of the frequency of the flutter is declared here.

```
      while(1){
            runtime = msTicks - start_time;
            if(runtime > 1000) break;
            if(!(runtime%50)){ // get reading every 50ms
                  isInAccRead = 1;
                  acc_read(&x,&y,&z);
                  isInAccRead = 0;
                  data[numOfReadings++] = z;
            }
      }
```

Figure 7b: Getting the readings

In this stage, the accelerometer reading of the z-index is pushed into the *data* array every 50ms for a time period of 1000ms.

```
      for(i=0;i<numOfReadings-4;i++){
            int8_t temp[5];

            for(j=0;j<5;j++)
                  temp[j] = data[i+j];

            quick_sort(temp,0,4);
            finalData[numOfSamples++] = temp[2];
      }
```

Figure 7c: Sampling of the Data

In this stage, median of every 5 reading is pushed into another array *finalData.*

```
for(i=0;i<numOfSamples;i++){
            if(!isInitialised){
                  if(finalData[i] > ACC_TOLERANCE + gAccRead){
                        isInitialised = 1;
                        isMovingUp = 0;
                  }else if(finalData[i] < gAccRead - ACC_TOLERANCE){
                        isInitialised = 1;
                        isMovingUp = 1;
                  }
            }else{
                  if ((finalData[i] > ACC_TOLERANCE + gAccRead && isMovingUp == 1)  ||
                        (finalData[i] < gAccRead - ACC_TOLERANCE  && isMovingUp == 0)){
                        frequency++;
                        isMovingUp = !isMovingUp;
                  }
            }
      }
      return frequency;
}
```

Figure 7d: Calculation of frequency of flutter

In the final stage, the frequency of the flutter is calculated by checking the number of zero-crossing. In other words, whenever the reading in the array *finalData* crosses the gravitational acceleration reading, *gAccRead,* that was previously calibrated is counted as a zero-crossing.

It is also important to minus out any “noise”. We achieve this by making sure the zero-crossing is larger than a threshold, *ACC_TOLERANCE*.

In addition, this function calculates the frequency of the flutter in 1000ms.

II. Warning Mode

```
      if(!safe(freq)){
            isFrequent++;
      }else{
            isFrequent = 0;
      }
      if(isFrequent > MAYDAY_THRESHOLD) isMayDay = 1;

      if(isFrequent == TIME_WINDOW && MODE == ACTIVE_MODE){
            enterWarningMode();
      }else if(isFrequent < TIME_WINDOW && MODE == WARNING_MODE){
            leaveWarningMode();
      }
```

Figure 8: Determining Warning Mode

Figure 8 contains the code to test whether FFS should enter warning or non-warning mode. When the frequency is not within the safe range (2 Hz to 10 Hz), the *isFrequent* flag is incremented by 1. In other words, the FFS is within the unsafe range for 1000ms.

Hence, depending on the *TIME_WINDOW*, the FFS will enter into warning mode after the flag *isFrequent* is equal to the *TIME_WINDOW*. Likewise, when the frequency falls within the safe range, it will return back to the non-warning mode.

III. Updating the monitoring station with the frequency

```
static void sendUARTAct(int freq) {
   if(msTicks - reportingTimeFlag > REPORTING_TIME * 1000){
      reportingTimeFlag = msTicks;
      if (MODE==WARNING_MODE) {
         char messageWarning[16];
         snprintf( messageWarning, sizeof(messageWarning), "013 %02d WARNING\r\n",
freq);
         UART_Send(LPC_UART3, (uint8_t *)messageWarning ,strlen(messageWarning),
BLOCKING);
      }
      else if (MODE==ACTIVE_MODE) {
         char messageActive[8];
         snprintf( messageActive, sizeof(messageActive), "036 %02d\r\n", freq);
         UART_Send(LPC_UART3, (uint8_t *)messageActive ,strlen(messageActive),
BLOCKING);
            }
      }
}
```

Figure 9: FFS updates the monitoring station

Figure 9 contains the code for the UART to communicate with the monitoring system. When in active mode, the monitoring system will send a message of the frequency every *REPORTING_TIME*. Likewise, in warning mode, a warning message with the frequency will be send every *REPORTING_TIME*.

4.0 Extensions

We implemented three extensions namely using of External Interrupt 0 (EINT0), UART interrupt and a new mode: Mayday mode.

I. External Interrupt 0

```
static void initEINT0Interupt(){
	PINSEL_CFG_Type PinCfg;

	PinCfg.Funcnum = 1;
	PinCfg.Pinnum = 10;
	PinCfg.Portnum = 2;
	PinCfg.OpenDrain = 0;
	PinCfg.Pinmode = 0;
	PINSEL_ConfigPin(&PinCfg);
}
```

Figure 10: PIN Select for EINT0

Instead of configuring the reset button (SW3) to be a GPIO Interrupt, we configured the pin P2.10 to be EINT0. By doing so, we have imposed a higher priority on the reset button. This enhances our system as reset button should have the highest priority in the system.

Since it is a different type of interrupt, clearing it is also different. From Figure 11, below, it can be seen that the register that we have to write to clear the pending status of the interrupt is different from the one of GPIO Interrupt

```
void EINT0_IRQHandler () {
	if ((LPC_SC -> EXTINT) & 0x01) {
		resetFlag = 1;
		LPC_SC -> EXTINT = (1<<0);
	}
}
```

Figure 11: EINT0 Interrupt Handler

The mayday mode is a newly designed mode designed by our group. The idea is when the flutter continues for more than 5 cycles, this mode will be triggered. All the LEDS, RGB and buzzer will start up alerting the pilot or cabin crew of the persistent fluttering problems. There will be two options for the pilot to choose from, he may seek help from the monitoring system or try to resolve the issue himself.

II. Configuring UART Interrupt

```
void init_uart(void){
        .....
        // Below is for enabling UART interrupt
        NVIC_EnableIRQ(UART3_IRQn);
        UART_IntConfig(LPC_UART3, UART_INTCFG_RBR, ENABLE);
        UART_SetupCbs(LPC_UART3, 0, UART_INTERUPT);
}

void UART3_IRQHandler(void) {
        UART3_StdIntHandler();
}
```

Figure 12: EINT0 Interrupt Handler

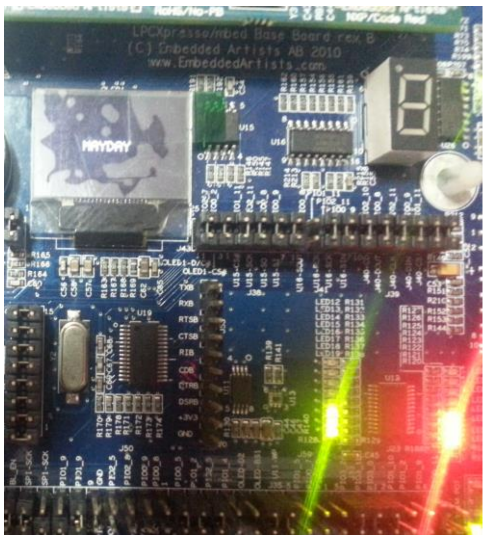
The code in Figure 12 shows the additional code we had to include to enable UART interrupt. Firstly, we have to enable the *UART3_IRQn* interrupt handler. Next , we configured the interrupt to receive from buffer. Lastly, we configured the callback function (*UART_INTERUPT*), the function that will be invoked when interrupt occurs, as a parameter.

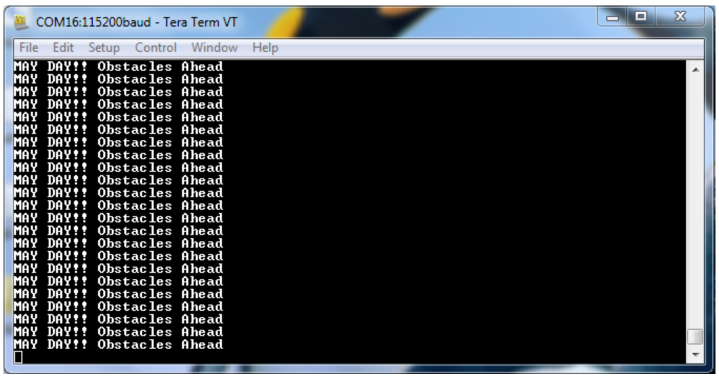
In other words, when there is UART Interrupt, the *UART3_IRQHandler* will be invoked. It will call the function *UART3_StdIntHandler* which will then call our callback function, *UART_INTERUPT.*

III. Mayday Mode

Lastly, we created our own mode: Mayday Mode, to add to the three existing modes in FFS. Let's take a look to how it works.

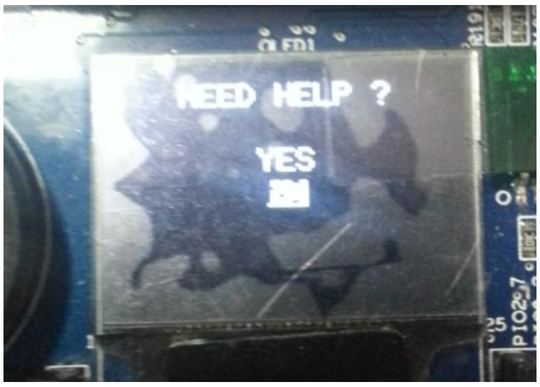
1) In the Active Mode, if the FFS is in "warning" for a threshold period of time, it enters the MAYDAY Mode. This is to simulate the airplane in a serious problem. Therefore, at this stage, the buzzer will continuously send a siren, the LEDs 4 to 19 blinks top to bottom and it sends a distress signal to the monitoring station. This can be seen from the picture below.



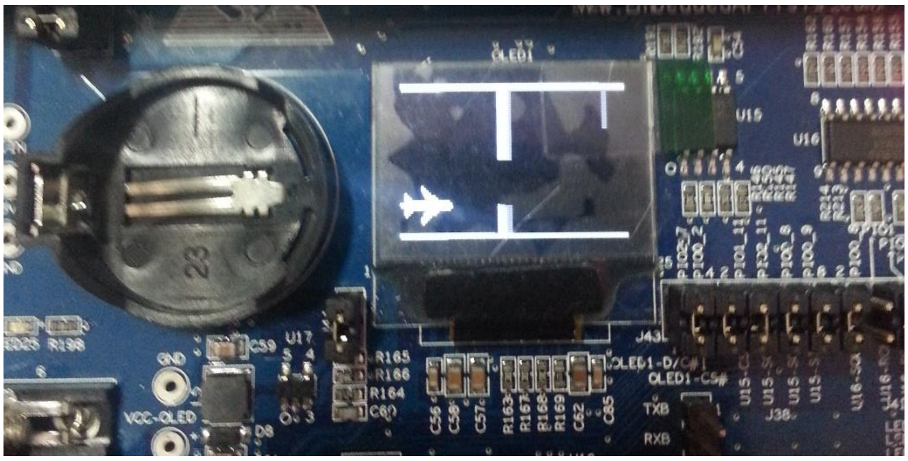

COM16:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
MAY DAY!! Obstacles Ahead
```

2) Then the monitoring station replies the airplane with a "HELP" command asking the pilots of the airplane if they need assistance. The options can be seen from the picture below. Options is to be chosen with the joystick.



3) If the pilots choose to be assisted, the FFS will return to Standby Mode. In other words, assistance was given and airplane is safe now.
4) If the pilot choose to navigate through obstacles without the help from the monitoring station, the FFS will enter a game mode. This game is similar to the Flappy Bird Game. The pilot has to navigate the plane through the obstacles using only the joystick, as seen from the picture below. The 7SegmentDisplay will increment by 1 when the pilot successfully navigates through one obstacle. When the pilot manage to navigate through a total of 9 obstacles, he will be safe and the FFS returns to Standby Mode.



5.0 Problems Faced and its Solutions

I. Light Interrupt

We faced numerous problems when implementing the interrupt for the detection of radiation. Firstly, we initially configured the interrupt such that it will be only called when the radiation level exceeded the threshold level. However, we were clueless to how to invoke the interrupt again when it falls under threshold level. After much discussion, we decided to change the upper and lower threshold setting inside the interrupt handler.

However, after making the necessary changes, the solution, which can be seen from the figure below, did not work as intended.

```
if (isSafe) {
      light_setLoThreshold(0);
      light_setHiThreshold(800);
}
else {
      light_setHiThreshold(973);
      light_setLoThreshold(800);
}
```

Figure 13: Initial Solution

We were baffled by the result. Then, we used the debugger tools from the LPCXpresso IDE to slowly check the error. We then found out that when we set the highest threshold to the largest number, 973, it gets truncated to a small number due to the bits overflow as it was expecting a number 972 and below. So by changing the value to 972, instead of 973, we solved our bug.

II. Light sensor and Accelerometer

Towards, the end of the project, we realised that whenever light interrupt handler was invoked when accelerometer reading was taken in the Active Mode, it gets in to an infinite loop in one of the functions in acc.c file.

After checking our data sheets, we realised that both the accelerometer and the light sensor were using the same I2C protocol to talk to the Master. However, our confusion worsened when we realised that we configured the light sensor to be a GPIO interrupt rather than I2C protocol. After debugging, we realised that even though the light sensor was configured as a GPIO interrupt, it was still using I2C to talk to the Master when the functions *setLoThreshold* and *setHiThreshold* were called.

Since both peripherals were trying to use the same SDA and SCL lines, the program hangs.

```
isInAccRead = 1;
acc_read(&x,&y,&z);
isInAccRead = 0;
```

Figure 14a: In Frequency Calculation function

```
void  EINT3_IRQHandler() {
      if ((LPC_GPIOINT -> IO2IntStatF>>5) & 0x01) {
            if(!isInAccRead){
                  isSafe = !isSafe;
                  if (isSafe) {
                        light_setLoThreshold(LIGHT_LOW);
                        light_setHiThreshold(LIGHT_THRESHOLD );
                  }
                  else {
                        light_setHiThreshold(LIGHT_HIGH);
                        light_setLoThreshold(LIGHT_THRESHOLD );
                  }
            }
            LPC_GPIOINT -> IO2IntClr = 1<<5;
            light_clearIrqStatus();
      }
```

Figure 14b: In Light's interrupt handler

To solve the problem, we set a global variable, *isInAccRead* before and after *acc_read* function is called. By doing so, we prevent the handler from executing the *light_setHiThreshold* and *light_setLoThreshold.* The handler only clears the interrupt.

6.0 Conclusion

For this assignment, we have successfully implemented a safe and reliable Frequency Fluttering system. The FFS is able to measure the fluttering of the aircraft and ensure that the flutter will be within the normal flight envelop. We have also implemented additional safety checks like radiation with interrupts and temperature. The entire system was developed to model reality thus when the plane is in dire situation, a message will be automatically send to the monitoring system which is able to communicate back through UART to resolve and assist the plane. Furthermore, we have also implemented a brand new mode; mayday mode. This mode is triggered only when the plane is in grave situation and immediate attention have to be given. We made use of the different peripherals present on the baseboard like the light sensor, temperature sensor and made use of the systick timer to ensure real time simulation of the situation present. Therefore, with this we were able to complete the prerequisite of the assignment with additional features to enhance the FFS.