ACS6127: Real-Time Embedded Systems Final Assignment

**PART A.I**

My Final Embedded System meets the specifications for Level 1 and Level 2 and operates in two modes. The ‘Tilt switch mode’ displays the tilt direction using the onboard LEDs as required by Level 1 and the ‘Standby mode’ toggles the LEDs in the predefined pattern based on a button input as required by level 2. The system uses the CMSIS-RTOS RTX for real-time thread management. The processor clock is configured to 168 MHz.

**Initialisation**

When initialising, I decided to initialise everything in my\_defines.c in order to keep my main file as concise as possible. Additionally, I initialised all the hardware peripherals, such as SPI and GPIO, through dedicated functions with clear individual parameter settings. This ensures that each peripheral is configured correctly before use, reducing the risk of undefined behaviour. This structured approach also helped the debugging process. This overall increases the robustness of my code.

**Thread Initialising**

When calling the Threads in main, they are called through separate ‘Init\_’ initialising functions. These initialisingfunctions are created before the main operational Threads. These separate initialising functions check whether the main thread is created successfully before continuing. If the thread creation fails, the function returns -1, which prevents further issues caused by an uninitialised thread. This error check ensures the system avoids unexpected behaviour, improving the robustness of the code.

**Level 1**

This level requires the board to power up the corresponding lights with the corresponding tilt direction according to measurements taken from the LIS3DSH whilst having an appropriate deadzone for each LED and having an appropriate level of sensitivity, with the LEDs not flickering too much and having little lag.

This was done using a single thread ‘Tilt\_Thread’ and the two functions ‘Blink\_LED’ and ‘Return\_angle’ that I created in my\_defines. The logic of this Thread is as follows: After all the LED parameters are defined, a while(1) loop is entered which isn’t left unless another thread is ready. Within this while loop, the angles in the X and Y axis are constantly being updated into the variables X\_Angle and Y\_angle. This is done using the Return\_Angle function, which reads the current angle from the LIS3DSH depending on the given address and is used to increase efficiency. These angle values are then constantly being compared with 4 if statements, with each checking to see if the angles exceed the given deadzone for each tilt direction. This is done in each if statement by firstly checking if the angle is positive or negative by looking at the value of the MSB and then seeing if the magnitude of the angle exceeds 14 degrees (to act as a deadzone, 14 degrees was chosen to balance sensitivity and stability). This is done by removing the MSB, therefore getting the magnitude of the angle, which is of a value between 0 and 128 for a 180-degree section. Therefore, a value of 0x10 or just 10 decimal gives an angle reading of 10 \* 180/128 = 14 degrees. Additionally, because of the way the angle works, for negative values, the angle has to be less than 128 **– 10**, as opposed to the positive where it is greater than 10. If a statement is met, the corresponding light is turned on and when the statement is no longer met, the light is turned off. This was done using the Blink LED function which was created to increase the efficiency of the code.

**Level 2**

This level requires the board to be able to enter a second mode I have named ‘Standby mode’. It enters this mode only after the blue user button has been pressed and released. It then flashes all the lights on for one second and then off for a second after this. The red light is then flashed intermittently for half a second on and off indefinitely until the button is pressed and released again, where the board returns to the initial Tilt switch mode. This standby mode is continuously available on further clicks.

This was done using two threads ‘Flash\_Thread’ and ‘Button\_Thread’ and one function ‘Blink\_LED’ that I defined in my defines. The logic of the main Thread ‘Button\_Thread’ is as follows: after the one while loop variable ‘ii’ is assigned, a while(1) loop is entered which isn’t left unless another thread is ready. Within this while loop, there is an if condition waiting for the blue button to be pressed. After this there is a small delay (used to act as a switch debounce to improve robustness) and a while loop that loops until the button is no longer pressed i.e. released. Once this While loop is broken out of, it terminates the Tilt thread and initialises a new thread ‘Flash Thread’. This thread contains the code which turns on and off all the lights for a second and then enters a while loop to continually turn on and off the red light every half second. Returning to the Button thread, a 2 second delay is implemented after the Flash thread is initialised to allow the lights to turn on and off before anything can happen, improving robustness. After this a new while loop is entered that isn’t broken out of until the blue button is pressed and released. Once this happens the flash thread is then terminated, and tilt thread is re-initialised. Because of the way this Thread is written, it is able to change between the states an infinite number of times, as required.

**Main**

The simplicity and shortness of my main.c file makes the code efficient and easy to edit. All complex functionality is handled in prewritten functions in my\_defines.c and threads in thread.c, keeping main.c simple and clear. Each function that is called, such as Initialise\_SPI or Init\_Tilt\_Thread, is focused on a specific task making the main.c easy to read. This overall design helped to prevent errors and make debugging easier, which therefore made the program robust and adaptable to future changes.

**Part A.II**

**Thread.h**

extern int Init\_Tilt\_Thread (void); // Declare the initialise Tilt Thread

void Tilt\_Thread (void const \*argument); // Declare the initialise Tilt Thread

extern int Init\_Flash\_Thread (void); // Declare the initialise Flash Thread

void Flash\_Thread (void const \*argument); // Declare the initialise Flash Thread

extern int Init\_Button\_Thread (void); // Declare the initialise Button Thread

void Button\_Thread (void const \*argument); // Declare the initialise Button Thread

**Thread.c**

#include "cmsis\_os.h" // Includes the CMSIS RTOS header file for real time functionality

#include "stm32f4xx.h" // Includes the stm32f4xx.h header file for compatability and specific definitons for STM32F4 controllers

#include "my\_defines.h" // Includes my\_defines for prewritten functions

// Thread Declarations

void Tilt\_Thread (void const \*argument); // Declares the function Tilt\_Thread that is defined after the declarations

osThreadId tid\_Tilt\_Thread; // Declares an ID associated with the thread, allows easy reference when using os functions.

osThreadDef (Tilt\_Thread, osPriorityNormal, 1, 0); // Declares the main thread object. Has normal priority, 1 thread instance and 0 stack size.

void Flash\_Thread (void const \*argument); // Declares the function Flash\_Thread that is defined after the declarations

osThreadId tid\_Flash\_Thread; // Declares an ID associated with the thread, allows easy reference when using os functions.

osThreadDef (Flash\_Thread, osPriorityNormal, 1, 0); // Declares the main thread object. Has normal priority, 1 thread instance and 0 stack size.

void Button\_Thread (void const \*argument); // Declares the function Button\_Thread that is defined after the declarations

osThreadId tid\_Button\_Thread; // Declares an ID associated with the thread, allows easy reference when using os functions.

osThreadDef (Button\_Thread, osPriorityNormal, 1, 0); // Declares the main thread object. Has normal priority, 1 thread instance and 0 stack size.

/\*-------------------------------------------------------------------------

\* Tilt Thread

\*-----------------------------------------------------------------------\*/

// Code to define the initialise Tilt thread to be used in main, when initialised in main, the thread is started

int Init\_Tilt\_Thread (void){

tid\_Tilt\_Thread = osThreadCreate (osThread(Tilt\_Thread), NULL); // Creates the main thread object and assigns it the thread ID declared for it

if(!tid\_Tilt\_Thread) return(-1); // Checks to ensure the thread has been created.

return(0);

}

void Tilt\_Thread (void const \*argument){

// Code to define the operation of the thread, i.e. turning on the corresponding lights when the board is tilted

uint8\_t LED\_on = 1; // Definition of parameter for LED on

uint8\_t LED\_off = 0; // Definition of parameter for LED off

uint8\_t green\_LED = 12; // Definition of parameter for green LED (GPIOD pin 12)

uint8\_t red\_LED = 14; // Definition of parameter for red LED (GPIOD pin 14)

uint8\_t orange\_LED = 13; // Definition of parameter for orange LED (GPIOD pin 13)

uint8\_t blue\_LED = 15; // Definition of parameter for blue LED (GPIOD pin 15)

uint8\_t X\_Register = 0x29; // Definition of the MSB X-Axis (H) data register address

uint8\_t Y\_Register = 0x2B; // Defintion of the MSB Y-axis (H) data register address

uint8\_t X\_Angle, Y\_Angle; // Assigns data type unsigned 8 bit integer to X Angle and Y Angle

while(1){

X\_Angle = Return\_Angle(X\_Register); // Returns current angle in X axis and sets it to variable X\_Angle

Y\_Angle = Return\_Angle(Y\_Register); // Returns current angle in Y axis and sets it to Variable Y\_Angle

if (((X\_Angle & 0x80) == 0x80) && ((X\_Angle & ~0x80) < 0x76)){ // Checks to see if X axis value is Negative and the magnitude of the value is less than 14 degrees from the level poisiton

Blink\_LED(LED\_on,green\_LED); // Turn on Green LED

} else{

Blink\_LED(LED\_off,green\_LED); // Turn off Green LED

}

if (((X\_Angle & 0x80) == 0x00) && ((X\_Angle & ~0x80) > 0x10)){ // Checks to see if the X axis value is Positive and the magnitude of the value is greater than 14 degrees from the level poisiton

Blink\_LED(LED\_on,red\_LED); // Turn on Red LED

} else{

Blink\_LED(LED\_off,red\_LED); // Turn off Red LED

}

if (((Y\_Angle & 0x80) == 0x80) && ((Y\_Angle & ~0x80) < 0x76)){ // Checks to see if Y axis value is Negative and the magnitude of the value is less than 14 degrees from the level poisiton

Blink\_LED(LED\_on,blue\_LED); // Turn on Blue LED

} else{

Blink\_LED(LED\_off,blue\_LED); // Turn off Blue LED

}

if (((Y\_Angle & 0x80) == 0x00) && ((Y\_Angle & ~0x80) > 0x10)){ // Checks to see if the Y axis value is Positive and the magnitude of the value is greater than 14 degrees from the level poisiton

Blink\_LED(LED\_on,orange\_LED); // Turn on Orange LED

} else{

Blink\_LED(LED\_off,orange\_LED);// Turn on Blue LED

}

osThreadYield(); //Tells RTOS to either suspend this thead and run next thread if there is another one is ready. If not this Thread continues.

}

}

/\*-------------------------------------------------------------------------

\* Flash Thread

\*-----------------------------------------------------------------------\*/

// Code to define the initialise Flash thread to be used in main, when initialised in main, the thread is started

int Init\_Flash\_Thread (void){

tid\_Flash\_Thread = osThreadCreate (osThread(Flash\_Thread), NULL); // Creates the main thread object and assigns it the thread ID declared for it

if(!tid\_Flash\_Thread) return(-1); // Checks to ensure the thread has been created.

return(0);

}

void Flash\_Thread (void const \*argument){

// Code to define the operation of the thread, i.e. flashing the lights in the given pattern

uint8\_t LED\_on = 1; // Definition of parameter for LED on

uint8\_t LED\_off = 0; // Definition of parameter for LED off

uint8\_t green\_LED = 12; // Definition of parameter for green LED (GPIOD pin 12)

uint8\_t red\_LED = 14; // Definition of parameter for red LED (GPIOD pin 14)

uint8\_t orange\_LED = 13; // Definition of parameter for orange LED (GPIOD pin 13)

uint8\_t blue\_LED = 15; // Definition of parameter for blue LED (GPIOD pin 15)

Blink\_LED(LED\_on,orange\_LED); // These 4 lines turn on the orange, blue, green and red LED

Blink\_LED(LED\_on,blue\_LED);

Blink\_LED(LED\_on,green\_LED);

Blink\_LED(LED\_on,red\_LED);

osDelay(1000); // Waits one second

Blink\_LED(LED\_off,orange\_LED); // These 4 lines turn off the orange, blue, green and red LED

Blink\_LED(LED\_off,blue\_LED);

Blink\_LED(LED\_off,green\_LED);

Blink\_LED(LED\_off,red\_LED);

osDelay(1000); //Waits one second

while(1){ //This Repeatedly Blinks the Red LED infinitley until a new thread is called

Blink\_LED(LED\_on,red\_LED);

osDelay(500);

Blink\_LED(LED\_off,red\_LED);

osDelay(500);

osThreadYield(); //Tells RTOS to either suspend this thead and run next thread if there is another one is ready. If not this Thread continues.

}

}

/\*-------------------------------------------------------------------------

\* Button Thread

\*-----------------------------------------------------------------------\*/

// Code to define the initialise Button thread to be used in main, when initialised in main, the thread is started

int Init\_Button\_Thread (void){

tid\_Button\_Thread = osThreadCreate (osThread(Button\_Thread), NULL); // Creates the main thread object and assigns it the thread ID declared for it

if(!tid\_Button\_Thread) return(-1); // Checks to ensure the thread has been created.

return(0);

}

void Button\_Thread(void const \*argument){

// Code to define the operation of the thread, i.e. switching between tilt switch and 'standby mode'.

uint8\_t ii; //Assigns data type unsigned 8 bit integer to ii (while loop variable)

while(1){

if(((GPIOA->IDR)&0x00000001) == 0x00000001){ // if button is pressed

osDelay(100); // wait 0.1 seconds to act as a switch debounce

while(((GPIOA->IDR)&0x00000001) == 0x00000001){} // while button is pressed, wait until it is released

osThreadTerminate(tid\_Tilt\_Thread); // terminates tilt thread

Init\_Flash\_Thread(); // initialises flash thread

osDelay(2000); // wait for 2 seconds

ii = 0; // reset while loop variable

while(ii == 0){ // continue flashing until if statement is met

if(((GPIOA->IDR)&0x00000001) == 0x00000001){ // if button is pressed

osDelay(100); //wait 0.1 seconds to act as a switch debounce

while(((GPIOA->IDR)&0x00000001) == 0x00000001){} // while button is pressed, wait until it is released

ii = 1; // breakout of while loop

}

}

osThreadTerminate(tid\_Flash\_Thread); // terminates flash thread

Init\_Tilt\_Thread(); // initialises tilt thread

}

osThreadYield(); //Tells RTOS to either suspend this thead and run next thread if there is another one is ready. This will always occur as we initialise TIlt thread

}

}

**my\_defines.h**

#include "stm32f4xx.h"

void Initialise\_SPI(void); // Declaration for the function to initialise SPI

void Initialise\_GPIOA(void); // Declaration for the function to initialise GPIOA

void Initialise\_GPIOE(void); // Declaration for the function to initialise GPIOE

void Initialise\_GPIOE\_I(void); // Declaration for the function to initialise GPIOE\_I

void Enable\_SPI(void); // Declaration for the function to Enable SPI

void Initialise\_LED\_and\_Button(void); // Declaration for the function to initialise the LED and Button

void Initialise\_ControlReg(void); // Declaration for the function to initialise the Control Register

void Blink\_LED(uint8\_t,uint8\_t); // Declaration for the function to blink the LED - the two inputs are: one for the state and one for the colour LED

uint8\_t Return\_Angle(uint8\_t address); // Declaration for the function to Read the angle - the input is the address of either Y axis or X axis data register

**my\_defines.c**

#include "stm32f4xx.h" // Includes the stm32f4xx.h header file for compatability and specific definitons for STM32F4 controllers

SPI\_HandleTypeDef SPI\_Params; //SPI parameters structure handle

I2C\_HandleTypeDef I2C\_Params; //I2C parameters structure handle

I2S\_HandleTypeDef I2S\_Params; //I2S parameters structure handle

GPIO\_InitTypeDef GPIOA\_Params; // GPIOA parameters structure handle

GPIO\_InitTypeDef GPIOB\_Params; // GPIOB parameters structure handle

GPIO\_InitTypeDef GPIOD\_Params; // GPIOD parameters structure handle

GPIO\_InitTypeDef GPIOE\_Params; // GPIOE parameters structure handle

GPIO\_InitTypeDef GPIOE\_Params\_I; // GPIOE\_I parameters structure handle

uint8\_t data\_to\_send\_receive[1]; // Declares an array used to store required LIS3DSH register address or data in.

uint16\_t data\_size=1; //Declares a variable specifying that only a single address is accessed in each transaction.

uint32\_t data\_timeout=1000; //Sets a maximum time of 1000 to wait for the SPI transaction to complete in - this ensures program won’t freeze if there is a problem with the SPI/I2C communication channel.

//Definition for the function to initialise the SPI connection

void Initialise\_SPI(void){

RCC->APB2ENR |= RCC\_APB2ENR\_SPI1EN; // Enable clock for SPI1

SPI\_Params.Instance = SPI1; // Selects SPI1 interface

SPI\_Params.Init.Mode = SPI\_MODE\_MASTER; // Sets STM32F407 as Master

SPI\_Params.Init.NSS = SPI\_NSS\_SOFT; // Sets slave to be controlled by the software

SPI\_Params.Init.Direction = SPI\_DIRECTION\_2LINES; // Sets SPI to full-duplex

SPI\_Params.Init.DataSize = SPI\_DATASIZE\_8BIT; // Sets the data packet size to 8-bit

SPI\_Params.Init.CLKPolarity = SPI\_POLARITY\_HIGH; // Sets idle polarity for clock line to high

SPI\_Params.Init.CLKPhase = SPI\_PHASE\_2EDGE; // Sets the data line to be sampled on the second transition of the clock line

SPI\_Params.Init.FirstBit = SPI\_FIRSTBIT\_MSB; //Sets the transmission to MSB first

SPI\_Params.Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_32; // Sets clock prescaler to divede main APB2 clock by 32 to give an SPI clock of 2.625MHZ, to be less than the max value of 10MHz for SPI.

HAL\_SPI\_Init(&SPI\_Params); // Configures the SPI using the specified parameters

}

//Definition for the function to initialise pins 5-7 of GPIOA

void Initialise\_GPIOA(void) {

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIOAEN; //Enable clock for GPIOA

GPIOA\_Params.Pin = GPIO\_PIN\_5 | GPIO\_PIN\_6 | GPIO\_PIN\_7; // Selects pins 5,6,7

GPIOA\_Params.Alternate = GPIO\_AF5\_SPI1; // Selects alternate function 5 which corresponds to SPI1

GPIOA\_Params.Mode = GPIO\_MODE\_AF\_PP; // Selects alternate function to push pull

GPIOA\_Params.Speed = GPIO\_SPEED\_FAST; // Selects fast speed

GPIOA\_Params.Pull = GPIO\_NOPULL; //Selects no pull-up or pull down activation

HAL\_GPIO\_Init(GPIOA, &GPIOA\_Params); // Configures GPIOA into specidified modes in GPIOA\_Params

}

//Definition for the function to initialise pins 3 of GPIOE

void Initialise\_GPIOE(void){

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIOEEN; // Enable clock for GPIOE

GPIOE\_Params.Pin = GPIO\_PIN\_3; // Selects pin 3

GPIOE\_Params.Mode = GPIO\_MODE\_OUTPUT\_PP; // Selects push-pull mode

GPIOE\_Params.Speed = GPIO\_SPEED\_FAST; // Selects fast speed

GPIOE\_Params.Pull = GPIO\_PULLUP; // Selects pull-up activation

HAL\_GPIO\_Init(GPIOE, &GPIOE\_Params); // Configures GPIOE into specified modes in GPIOE\_Params

}

//Definition for the function to initialise pin 0 of GPIOE\_I

void Initialise\_GPIOE\_I(void) {

GPIOE\_Params\_I.Pin = GPIO\_PIN\_0; // Selects pin 0

GPIOE\_Params\_I.Mode = GPIO\_MODE\_IT\_RISING; // Selects rising edge trigger detection mode

GPIOE\_Params\_I.Speed = GPIO\_SPEED\_FAST; // Selects fast speed

HAL\_GPIO\_Init(GPIOE, &GPIOE\_Params\_I); // Configures GPIOE\_I into specified modes in GPIOE\_Params\_I

}

//Definition for the function to Enable SPI

void Enable\_SPI(void) {

GPIOE->BSRR = GPIO\_PIN\_3; // Sets the serial port enable pin to CS high

\_\_HAL\_SPI\_ENABLE(&SPI\_Params); // Enables SPI

}

//Definition for the function to initialise LEDs and Button

void Initialise\_LED\_and\_Button(void) {

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN;

GPIOD->MODER |= GPIO\_MODER\_MODER14\_0; // Red LED port 14 to output

GPIOD->MODER |= GPIO\_MODER\_MODER12\_0; // Green LED port 12 to output

GPIOD->MODER |= GPIO\_MODER\_MODER13\_0; // Orange LED port 13 to output

GPIOD->MODER |= GPIO\_MODER\_MODER15\_0; // Blue LED port 15 to output

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIOAEN; // Enable Port A clock

}

//Definition for the Control Register

void Initialise\_ControlReg(void){

data\_to\_send\_receive[0] = 0x20; // Address for control register 3 on the LIS3DSH

GPIOE->BSRR = GPIO\_PIN\_3<<16; // Set SPI communication enable line low to initiate communication

HAL\_SPI\_Transmit(&SPI\_Params,data\_to\_send\_receive,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

data\_to\_send\_receive[0] = 0x13; // Sample rate of 3.125hz, continous update, x and y enabled

HAL\_SPI\_Transmit(&SPI\_Params,data\_to\_send\_receive,data\_size,data\_timeout); // Send the new register value to the LIS3DSH through the SPI channel

GPIOE->BSRR = GPIO\_PIN\_3; // Set the SPI communication enable line high to signal the end of the communication process

}

// Definition for the function to blink the LED

void Blink\_LED(uint8\_t LED\_state, uint8\_t LED\_colour){

if(LED\_state == 1){ // Checks to see if the request is to turn the LED on or off

GPIOD->BSRR = 1<<LED\_colour; // Turn on the LED

}

else{

GPIOD->BSRR = 1<<(LED\_colour+16); // Turn off the LED

}

}

uint8\_t Return\_Angle(uint8\_t address){ // Returns current angle from LIS3DSH depending on address

data\_to\_send\_receive[0] = 0x80 | address; // Address is passed for either the MSB Y-axis (H) data register or the MSB X-axis (H) data register on the LIS3DSH

GPIOE->BSRR = GPIO\_PIN\_3 << 16; // Set the SPI communication enable line low to initiate communication

HAL\_SPI\_Transmit(&SPI\_Params, data\_to\_send\_receive, data\_size, data\_timeout); // Send the address of the register to be read on the LIS3DSH

data\_to\_send\_receive[0] = 0x00; // Set a blank address because we are waiting to receive data

HAL\_SPI\_Receive(&SPI\_Params, data\_to\_send\_receive, data\_size, data\_timeout); // Get the data from the LIS3DSH through the SPI channel

GPIOE->BSRR = GPIO\_PIN\_3; // Set the SPI communication enable line high to signal the end of the communication process

return(data\_to\_send\_receive[0]); // Return the data read from the specified LIS3DSH register

}

**Main.c**

#define osObjectsPublic // Define objects in main module

#include "osObjects.h" // Includes RTOS object definitions

#include "stm32f4xx.h" // Includes the stm32f4xx.h header file for compatability and specific definitons for STM32F4 controllers

#include "my\_defines.h" // Includes my\_defines for prewritten functions

#include "Thread.h" // Includes Thread for prewritten threads

int main(void){

SystemCoreClockUpdate(); // Updates system core clock update

osKernelInitialize(); // Initialise CMSIS-RTOS

Initialise\_SPI(); // Initialises SPI using Function from my\_defines.c

Initialise\_GPIOA(); // Initialises GPIOA using Function from my\_defines.c

Initialise\_GPIOE(); // Initialises GPIOE using Function from my\_defines.c

Initialise\_GPIOE\_I(); // Initialises GPIOE\_I using Function from my\_defines.c

Enable\_SPI(); // Enables SPI using Function from my\_defines.c

Initialise\_LED\_and\_Button(); // Initialises LED and Button using Function from my\_defines.c

Initialise\_ControlReg(); // Initiallses Control Registers using Function from my\_defines.c

Init\_Tilt\_Thread(); //Initialises Tilt thread from Thread.c

Init\_Button\_Thread(); //Initialises Button thread from Thread.c

osKernelStart(); // start thread execution

while(1){}; // while loop to prevent program terminating

}