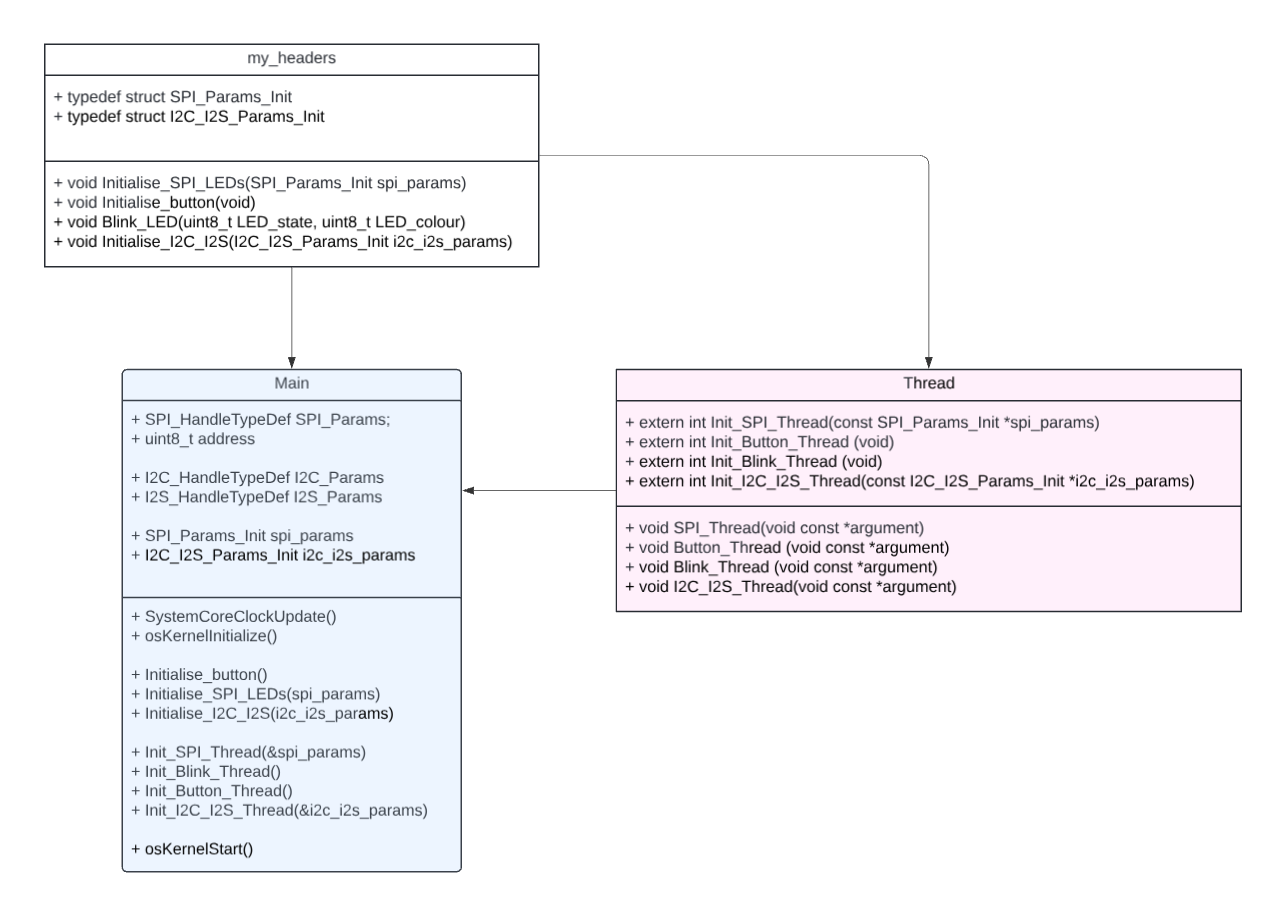
**Overview**

For this assignment, four threads have been created; the SPI thread, LED flasher thread, Button thread, and finally, the I2C and I2S thread. The Button thread will run concurrently with all other threads in the program, providing control over which thread runs at any given time. The execution of SPI thread will run when the program is initially uploaded. And the other threads will start working when the button is clicked. Here is a UML overview of the program implementation:

****

****

**Level 1 - Creating a Dynamic Two-Axis Tilt Switch Interface**

1. In the ‘my\_headers.h’ file, a struct named ‘SPI\_Params\_Init’ will be created to hold SPI initialization parameters. This struct includes four variables, with two of them being pointers to track changes to these variables.
2. A thread with the main function named 'SPI\_Thread' will be created in the ‘Thread.c’ file to accomplish the functionality of this task using an RTOS. Furthermore, the struct created will be passed as a parameter in the initialization function 'Init\_SPI\_Thread.'
3. Access the X and Y axes in the LIS3DSH, with X and Y as signed 16-bit integers holding XL, XH, YL, and YH values read from the corresponding addresses in the LIS3DSH datasheet. Set sensitivity values for X and Y at 500 and -500 for negative X and Y, this is because the acceleration typically ranges from -32768 to +32767. Specific LEDs will turn on and off based on the direction of tilt when the device exceeds these values.
4. Now, before the while loop in the main thread function for SPI, which is 'SPI\_Thread,' the statement ‘osSignalSet(tid\_SPI\_Thread, 0x01)’ is included to enable the execution of SPI when the program is initially uploaded. Inside the while loop, the first line of code will be ‘osSignalWait(0x01, osWaitForever)’. This will cause the program to pause if the push button is clicked, as the flag associated with this thread is cleared.

**Level 2 - Control with Push Button**

1. Two threads will be created in this section, each with main thread functions named 'Button\_Thread' and 'Blink\_Thread' in the file 'Thread.c'.
2. For the blinking thread, a red flasher will be created for 0.5 seconds. The first line of code will include 'osSignalWait(0x01, osWaitForever)' to control its operation using the button.
3. We need the push button to work as a latch switch, maintaining its current state until intentionally toggled. I've implemented a global variable 'flag' with an initial state of zero. Clicking the button with flag at 0 triggers one function and sets the flag to one; clicking with flag at one triggers another function and resets the flag to zero, this will make the button work as a latch switch.
4. When the button is clicked with a state of zero, all LEDs will turn on for one second without the need for a debouncing delay. We will then stop the SPI thread by clearing its flag and set flags for the blinking, I2C, and I2S threads.
5. If the button is clicked with a state of one, a 200-millisecond delay will be added to remove the debouncing effect of the button. The flags for the blinking, I2C, and I2S threads will be cleared, and the SPI thread flag will be set.

**Level 3 - Generating a Beep Sound using CS43L22**

1. Similar to level one, a struct will be created to hold the initialization parameters for I2C and I2S. This is because the STM32F4 controls the audio DAC through the I2C interface and processes digital signals through the I2S connection or analog input signal.
2. Based on the datasheet of the CS43L22, specifically in section 4.11 titled 'Required Initialization Settings', the implementation of the CS43L22 initialization settings using I2C involves writing these values to the crossbounding addresses of CS43L22 through the 'HAL\_I2C\_Master\_Transmit' function. Similarly, in section 7.17, Beep & Tone Configuration (Address 1Eh), 0xC0 will be written to register 0x1E to configure a continuous beep occurrence. The same applies to setting the 'Power Ctl 1' register (0x02) to 0x9E.
3. Finally, a thread with the main function named 'I2C\_I2S\_Thread' will be created in the 'Thread.c' file to transmit the data of the square wave through I2S to generate sound using the function 'HAL\_I2S\_Transmit\_IT'.

**my\_headers.h:**

#include "stm32f4xx.h"

// Structure to hold SPI initialization parameters

typedef struct {

SPI\_HandleTypeDef\* SPI\_Params;

uint8\_t\* address;

uint16\_t data\_size;

uint32\_t data\_timeout;

} SPI\_Params\_Init;

// Structure to hold I2C and I2S initialization parameters

typedef struct {

I2C\_HandleTypeDef\* I2C\_Params;

I2S\_HandleTypeDef\* I2S\_Params;

} I2C\_I2S\_Params\_Init;

// Function declaration to initialize SPI and LEDs

void Initialise\_SPI\_LEDs(SPI\_Params\_Init spi\_params);

// Function declaration to initialize I2C and I2S

void Initialise\_I2C\_I2S(I2C\_I2S\_Params\_Init i2c\_i2s\_params);

// Function declarations for controlling LED and button initialization

void Initialise\_button(void);

void Blink\_LED(uint8\_t state, uint8\_t color);

**my\_headers.c:**

#include "stm32f4xx.h"

#include "my\_headers.h"

// Function to initialise SPI communication protocol and the 4 LEDs

void Initialise\_SPI\_LEDs(SPI\_Params\_Init spi\_params) {

// Declare a handle structure for Port A pins

GPIO\_InitTypeDef GPIOA\_Params;

// Declare a handle structure for Port B pins

GPIO\_InitTypeDef GPIOE\_Params;

// Code to initialise the SPI

RCC->APB2ENR |= RCC\_APB2ENR\_SPI1EN; // Enable the clock for SPI1 which is found in the register APB2.

// (\*spi\_params.SPI\_Params) accesses the member named ‘SPI\_Params’ within the structure pointed to by ‘spi\_params’, and the ‘\*’ sign is used to dereference and access the value of it

(\*spi\_params.SPI\_Params).Instance = SPI1; // Set SPI instance to SPI1

(\*spi\_params.SPI\_Params).Init.Mode = SPI\_MODE\_MASTER; // Set SPI mode to master

(\*spi\_params.SPI\_Params).Init.NSS = SPI\_NSS\_SOFT; // Set SPI NSS (Slave Select) mode to software control

(\*spi\_params.SPI\_Params).Init.Direction = SPI\_DIRECTION\_2LINES; // Set SPI data transfer direction to full-duplex

(\*spi\_params.SPI\_Params).Init.DataSize = SPI\_DATASIZE\_8BIT; // Set SPI data size to 8 bits

(\*spi\_params.SPI\_Params).Init.CLKPolarity = SPI\_POLARITY\_HIGH; // Set SPI clock polarity to high

(\*spi\_params.SPI\_Params).Init.CLKPhase = SPI\_PHASE\_2EDGE; // Data is captured on the second edge of the clock signal

(\*spi\_params.SPI\_Params).Init.FirstBit = SPI\_FIRSTBIT\_MSB; // Set SPI data frame format to MSB first

(\*spi\_params.SPI\_Params).Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_32; // Set SPI Baud Rate Prescaler to 32

HAL\_SPI\_Init(spi\_params.SPI\_Params); // Initialize SPI using the specified parameters

// Code to initialise pins 5-7 of GPIOA

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIOAEN; // Enable clock for GPIOA in the AHB1 peripheral clock register

GPIOA\_Params.Pin = GPIO\_PIN\_5 | GPIO\_PIN\_6 | GPIO\_PIN\_7; // Configure GPIOA pins 5, 6, and 7 for SPI communication

GPIOA\_Params.Alternate = GPIO\_AF5\_SPI1; // Set the alternate function for GPIOA pins to SPI1

GPIOA\_Params.Mode = GPIO\_MODE\_AF\_PP; // Set GPIOA pins mode to alternate function push-pull

GPIOA\_Params.Speed = GPIO\_SPEED\_FAST; // Set GPIOA pins speed to fast

GPIOA\_Params.Pull = GPIO\_NOPULL; // Set GPIOA pins pull configuration to no pull-up/pull-down

HAL\_GPIO\_Init(GPIOA, &GPIOA\_Params); // Initialize GPIOA pins with the specified parameters

// Code to initialise pin 3 of GPIOE

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIOEEN; // Enable clock for GPIOE in the AHB1 peripheral clock register

// Configure GPIOE pin 3 for general-purpose output in push-pull mode

GPIOE\_Params.Pin = GPIO\_PIN\_3;

GPIOE\_Params.Mode = GPIO\_MODE\_OUTPUT\_PP;

GPIOE\_Params.Speed = GPIO\_SPEED\_FAST;

GPIOE\_Params.Pull = GPIO\_PULLUP; // Set GPIOE pin 3 pull configuration to pull-up

HAL\_GPIO\_Init(GPIOE, &GPIOE\_Params); // Initialize GPIOE pin 3 with the specified parameters

GPIOE->BSRR = GPIO\_PIN\_3; // Set GPIOE pin 3 to high (BSRR: Bit Set Reset Register)

\_\_HAL\_SPI\_ENABLE(spi\_params.SPI\_Params); // Enable SPI communication on the specified SPI instance

// Initialize GPIO Port for LEDs

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN; // Enable clock for GPIOD in the AHB1 peripheral clock register

// Configure GPIOD pin 12, 13, 14, and 15 for general-purpose output

GPIOD->MODER |= GPIO\_MODER\_MODER14\_0;

GPIOD->MODER |= GPIO\_MODER\_MODER12\_0;

GPIOD->MODER |= GPIO\_MODER\_MODER15\_0;

GPIOD->MODER |= GPIO\_MODER\_MODER13\_0;

\*spi\_params.address = 0x20; // Address for control register 4 on LIS3DSH

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

// Send the address of the register to be read on the LIS3DSH

HAL\_SPI\_Transmit(spi\_params.SPI\_Params,

spi\_params.address,

spi\_params.data\_size,

spi\_params.data\_size);

\*spi\_params.address = 0x13; // Write a new value to control register 4 of the LIS3DSH by setting register value to give a sample rate of 3.125Hz, continuous update and enable x-axis and y-axis only

// Send the new register value to the LIS3DSH through the SPI channel

HAL\_SPI\_Transmit(spi\_params.SPI\_Params,

spi\_params.address,

spi\_params.data\_size,

spi\_params.data\_size);

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

}

// Function to initialize GPIO for push-button

void Initialise\_button(void){

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIOAEN; // Enable clock for GPIOA in the AHB1 peripheral clock register

}

// Definition for the function to blink the LED

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

// Checks to see if the request is to turn the LED on or off

if(LED\_state == 1){

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

}

else{

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

}

}

// Function to initialize I2C and I2S peripherals

void Initialise\_I2C\_I2S(I2C\_I2S\_Params\_Init i2c\_i2s\_params) {

// Initialize GPIO parameters for B, C and D ports

GPIO\_InitTypeDef GPIOD\_Params;

GPIO\_InitTypeDef GPIOB\_Params;

GPIO\_InitTypeDef GPIOC\_Params;

uint8\_t C[2]; // Initialize an array for I2C data

// Configure GPIOD for reset pin on CS43L22

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN; // Enable the clock for GPIOD

GPIOD\_Params.Pin = GPIO\_PIN\_4; // Select pin 4 in port D

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

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

GPIOD\_Params.Pull = GPIO\_NOPULL; // Selects pull-down activation

HAL\_GPIO\_Init(GPIOD, &GPIOD\_Params); // Sets GPIOD into the modes specified in GPIOE\_Params.

// Configure GPIOB for SCL (PB6) and SDA (PB9) on I2C1

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIOBEN; // Enable the clock for GPIOB

GPIOB\_Params.Pin = GPIO\_PIN\_6 | GPIO\_PIN\_9; // Selects pins 6 and 9

GPIOB\_Params.Alternate = GPIO\_AF4\_I2C1; // Selects alternate function I2C1

GPIOB\_Params.Mode = GPIO\_MODE\_AF\_OD; // Selects alternate function open drain mode

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

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

HAL\_GPIO\_Init(GPIOB, &GPIOB\_Params); // Sets GPIOB into the modes specified in GPIOA\_Params

// Configure I2C peripheral

RCC->APB1ENR |= RCC\_APB1ENR\_I2C1EN ; // Enables the clock for I2C1

// (\*i2c\_i2s\_params.I2C\_Params) accesses the member named I2C\_Params within the structure pointed to by i2c\_i2s\_params, and the ‘\*’ sign is used to dereference and access the value of it

(\*i2c\_i2s\_params.I2C\_Params).Instance = I2C1; // Set I2C instance to I2C1

(\*i2c\_i2s\_params.I2C\_Params).Init.ClockSpeed = 400000; // Set clock speed to 400 kHz

(\*i2c\_i2s\_params.I2C\_Params).Init.DutyCycle = I2C\_DUTYCYCLE\_2; // Set the I2C duty cycle to 2, where '2' indicates a 50% duty cycle. In this context, '2' is used to represent a 1:1 ratio or 50% duty cycle

(\*i2c\_i2s\_params.I2C\_Params).Init.OwnAddress1 = 0x33; // Set the own address for I2C communication to 0x33

(\*i2c\_i2s\_params.I2C\_Params).Init.AddressingMode = I2C\_ADDRESSINGMODE\_7BIT; // Set the addressing mode to 7-bit

(\*i2c\_i2s\_params.I2C\_Params).Init.DualAddressMode = I2C\_DUALADDRESS\_DISABLED; // Disable dual address mode

(\*i2c\_i2s\_params.I2C\_Params).Init.OwnAddress2 = 0; // Set the second own address to 0 (not used in this case)

(\*i2c\_i2s\_params.I2C\_Params).Init.GeneralCallMode = I2C\_GENERALCALL\_DISABLED; // Disable general call mode

(\*i2c\_i2s\_params.I2C\_Params).Init.NoStretchMode = I2C\_NOSTRETCH\_DISABLED; // Disable clock stretching during data transfer

HAL\_I2C\_Init(i2c\_i2s\_params.I2C\_Params); // Initialize the I2C peripheral with the specified parameters

// Configure GPIOC for 7(MCLK), 10(SCLK) and 12(SDIN) on I2S:

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIOCEN; // Enable the clock for GPIOC.

GPIOC\_Params.Pin = GPIO\_PIN\_7 | GPIO\_PIN\_10 | GPIO\_PIN\_12; // Selects pins 7(MCLK), 10(SCLK) and 12(SDIN).

GPIOC\_Params.Alternate = GPIO\_AF6\_SPI3; // Selects alternate function (I2S).

GPIOC\_Params.Mode = GPIO\_MODE\_AF\_PP; // Selects alternate function open drain mode.

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

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

HAL\_GPIO\_Init(GPIOC, &GPIOC\_Params); // Sets GPIOB into the modes specified in GPIOA\_Params.

// Configure I2S peripheral

RCC->APB1ENR |= RCC\_APB1ENR\_SPI3EN ; // Enables the clock for SPI3 (I2S)

(\*i2c\_i2s\_params.I2S\_Params).Instance = SPI3; // Set the I2S instance to SPI3

(\*i2c\_i2s\_params.I2S\_Params).Init.Mode = I2S\_MODE\_MASTER\_TX; // Configure I2S mode as Master Transmit

(\*i2c\_i2s\_params.I2S\_Params).Init.Standard = I2S\_STANDARD\_PHILIPS; // Set the I2S standard to Philips format

(\*i2c\_i2s\_params.I2S\_Params).Init.DataFormat = I2S\_DATAFORMAT\_16B; // Configure data format as 16-bit

(\*i2c\_i2s\_params.I2S\_Params).Init.MCLKOutput = I2S\_MCLKOUTPUT\_ENABLE; // Enable the Master Clock (MCLK) output

(\*i2c\_i2s\_params.I2S\_Params).Init.AudioFreq = I2S\_AUDIOFREQ\_44K; // Set the audio frequency to 44 kHz

(\*i2c\_i2s\_params.I2S\_Params).Init.CPOL = I2S\_CPOL\_LOW; // Set the clock polarity to low

(\*i2c\_i2s\_params.I2S\_Params).Init.ClockSource = I2S\_CLOCK\_PLL; // Use PLL as the clock source

(\*i2c\_i2s\_params.I2S\_Params).Init.FullDuplexMode = I2S\_FULLDUPLEXMODE\_DISABLE; // Disable full duplex mode

HAL\_I2S\_Init(i2c\_i2s\_params.I2S\_Params); // Initialize I2S with the configured parameters

GPIOD->BSRR = GPIO\_PIN\_4; // Sets the reset pin of CS43L22 high

// Required Initialization Settings from CS43L22 datasheet

C[0] = 0x00;

C[1] = 0x99; // Write 0x99 to register 0x00

HAL\_I2C\_Master\_Transmit(i2c\_i2s\_params.I2C\_Params, 0x94, C, 2, 50);

C[0] = 0x47;

C[1] = 0x80; // Write 0x80 to register 0x47

HAL\_I2C\_Master\_Transmit(i2c\_i2s\_params.I2C\_Params, 0x94, C, 2, 50);

C[0] = 0x32;

C[1] = 0x80; // Write ‘1’b to bit 7 in register 0x32

HAL\_I2C\_Master\_Transmit(i2c\_i2s\_params.I2C\_Params, 0x94, C, 2, 50);

C[0] = 0x47;

C[1] = 0x00; // Write ‘0’b to bit 7 in register 0x47

HAL\_I2C\_Master\_Transmit(i2c\_i2s\_params.I2C\_Params, 0x94, C, 2, 50);

C[0] = 0x00;

C[1] = 0x00; // Write 0x00 to register 0x00

HAL\_I2C\_Master\_Transmit(i2c\_i2s\_params.I2C\_Params, 0x94, C, 2, 50);

C[0] = 0x1E; // Beep & tone configuration register

C[1] = 0xC0; // Write 0xC0 (continuous Beep Occurrence) to register 0x1E

HAL\_I2C\_Master\_Transmit(i2c\_i2s\_params.I2C\_Params, 0x94, C, 2, 50);

C[0] = 0x02;

C[1] = 0x9E; // Set the “Power Ctl 1” register (0x02) to 0x9F

HAL\_I2C\_Master\_Transmit(i2c\_i2s\_params.I2C\_Params, 0x94, C, 2, 50);

}

**Thread.h:**

#include "cmsis\_os.h"

#include "stm32f4xx.h"

extern int Init\_SPI\_Thread(const SPI\_Params\_Init \*spi\_params); // Initializing SPI thread

void SPI\_Thread(void const \*argument); // The main thread function for the SPI

extern int Init\_Button\_Thread (void); // Initializing button thread

void Button\_Thread (void const \*argument); // The main thread function for the button

extern int Init\_Blink\_Thread (void); // Initializing blink thread

void Blink\_Thread (void const \*argument); // The main thread function for the blink

extern int Init\_I2C\_I2S\_Thread(const I2C\_I2S\_Params\_Init \*i2c\_i2s\_params); // Initializing I2C and I2S thread

void I2C\_I2S\_Thread(void const \*argument); // The main thread function for I2C and I2S

**Thread.c:**

#include "cmsis\_os.h"

#include "stm32f4xx.h"

#include "my\_headers.h"

#include "stdio.h"

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

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

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

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

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

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

int flag = 0; // Define a flag to make the push-button work as a latch switch

/\* Thread(1): SPI thread declaration and initialisation \*/

void SPI\_Thread(void const \*argument); // Declaration for the main thread function for SPI

osThreadId tid\_SPI\_Thread; // Declares an ID for SPI thread

osThreadDef (SPI\_Thread, osPriorityNormal, 1, 0); // Define SPI\_Thread with normal priority and a stack size of 1 word

// Initialisation function for the SPI thread

int Init\_SPI\_Thread(const SPI\_Params\_Init \*spi\_params) {

// Create an SPI thread that takes a pointer of 'SPI\_Params\_Init' type as parameter that we created in ‘my\_defines.h’, which contain the suitable variables and parameters for initialising SPI communication

tid\_SPI\_Thread = osThreadCreate(osThread(SPI\_Thread), (void \*)spi\_params); // Creates the main thread object that we have declared and assigns it the thread ID that we have declared

if (!tid\_SPI\_Thread) return (-1); // Checks to make sure the thread has been created

return (0);

}

/\* Thread(2): Blink thread declaration and initialisation \*/

void Blink\_Thread (void const \*argument); // Declaration for the main thread function for blinking LED

osThreadId tid\_Blink\_Thread; // Declares an ID for Blink thread

osThreadDef (Blink\_Thread, osPriorityNormal, 1, 0); // Define Blink\_Thread with normal priority and a stack size of 1 word

// Initialisation function for the Blink thread

int Init\_Blink\_Thread (void) {

tid\_Blink\_Thread = osThreadCreate (osThread(Blink\_Thread), NULL); // Creates the main thread object that we have declared and assigns it the thread ID that we have declared

if(!tid\_Blink\_Thread) return(-1); // Checks to make sure the thread has been created

return(0);

}

/\* Thread(3): Button thread declaration and initialisation \*/

void Button\_Thread (void const \*argument); // Declaration for the main thread function for the button

osThreadId tid\_Button\_Thread; // Declares an ID for Button thread

osThreadDef (Button\_Thread, osPriorityNormal, 1, 0); // Define Button\_Thread with normal priority and a stack size of 1 word

int Init\_Button\_Thread (void) {

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

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

return(0);

}

/\* Thread(4): I2C and I2S thread declaration and initialisation \*/

void I2C\_I2S\_Thread (void const \*argument); // Declaration for the main thread function for the I2C and I2S

osThreadId tid\_I2C\_I2S\_Thread; // Declares an ID for I2C and I2S thread

osThreadDef (I2C\_I2S\_Thread, osPriorityNormal, 1, 0); // Define I2C\_I2S\_Thread with normal priority and a stack size of 1 word

int Init\_I2C\_I2S\_Thread (const I2C\_I2S\_Params\_Init \*i2c\_i2s\_params) {

// Create an I2C and I2S thread that takes a pointer of ‘I2C\_I2S\_Params\_Init’ type as parameter that we created in ‘my\_defines.h’, which contain the suitable variables and parameters for initialising I2C and I2S communication

tid\_I2C\_I2S\_Thread = osThreadCreate (osThread(I2C\_I2S\_Thread), (void \*)i2c\_i2s\_params); // Creates the main thread object that we have declared and assigns it the thread ID that we have declared

if(!tid\_I2C\_I2S\_Thread) return(-1); // Checks to make sure the thread has been created

return(0);

}

/\* Thread(1): SPI main thread function \*/

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

SPI\_Params\_Init \*spi\_params = (SPI\_Params\_Init \*)argument; // To access the parameters from the main file using the struct that we have created

uint8\_t XL, XH, YL, YH; // Variables to store the register values for XL, XH, YL and YH of LIS3DSH

int16\_t X, Y; // Variables to combine XL and XH into single variable, and the same done for Y

osSignalSet(tid\_SPI\_Thread, 0x01); // Set the 0x01 flag of the SPI thread to enable its execution when we upload the program

while(1) {

osSignalWait(0x01, osWaitForever); // Waits until flag 0x01 of this thread is set (like if we click on the push button)

osSignalSet(tid\_SPI\_Thread, 0x01); // Set flag 0x01 of the SPI thread so that it resumes next time wait is called

\*spi\_params->address = 0x80 | 0x29; // Address for the 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->SPI\_Params, spi\_params->address, spi\_params->data\_size, spi\_params->data\_timeout); // Send the address of the register to be read on the LIS3DSH

\*spi\_params->address = 0x00; // Set a blank address because we are waiting to receive data

HAL\_SPI\_Receive(spi\_params->SPI\_Params, spi\_params->address, spi\_params->data\_size, spi\_params->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

XH = \*spi\_params->address; // Read the data from the SPI data array into our internal variable

\*spi\_params->address = 0x80 | 0x28; // Address for the x-axis (L) 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->SPI\_Params, spi\_params->address, spi\_params->data\_size, spi\_params->data\_timeout); // Send the address of the register to be read on the LIS3DSH

\*spi\_params->address = 0x00; // Set a blank address because we are waiting to receive data

HAL\_SPI\_Receive(spi\_params->SPI\_Params, spi\_params->address, spi\_params->data\_size, spi\_params->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

XL = \*spi\_params->address; // Read the data from the SPI data array into our internal variable

X = ((XH<<8) | XL); // Combine XL and XH into a single variable by left-shifting XH by 8 units and adding XL to the first 8 bits

\*spi\_params->address = 0x80 | 0x2B; // Address for the MSB y-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->SPI\_Params, spi\_params->address, spi\_params->data\_size, spi\_params->data\_timeout); // Send the address of the register to be read on the LIS3DSH

\*spi\_params->address = 0x00; // Set a blank address because we are waiting to receive data

HAL\_SPI\_Receive(spi\_params->SPI\_Params, spi\_params->address, spi\_params->data\_size, spi\_params->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

YH = \*spi\_params->address; // Read the data from the SPI data array into our internal variable

\*spi\_params->address = 0x80 | 0x2A; // Address for the MSB y-axis (L) 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->SPI\_Params, spi\_params->address, spi\_params->data\_size, spi\_params->data\_timeout); // Send the address of the register to be read on the LIS3DSH

\*spi\_params->address = 0x00; // Set a blank address because we are waiting to receive data

HAL\_SPI\_Receive(spi\_params->SPI\_Params, spi\_params->address, spi\_params->data\_size, spi\_params->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

YL = \*spi\_params->address; // Read the data from the SPI data array into our internal variable

Y = ((YH<<8) | YL); // Combine YL and YH into a single variable by left-shifting YH by 8 units and adding YL to the first 8 bits

if (Y > 500) {

// Y is now a 16-bit signed variable, resulting in an acceleration range typically from -32768 to +32767

// When the device is in static condition the X and Y values should be close to 0 (zero)

// When the device is being tilted in a direction corresponding to the positive Y-axis the value of Y will be increasing

// Here the sensitivity value for positive Y is 500 when the device is being tilted more than this value the orange led will turn on

Blink\_LED(LED\_on, orange\_LED); // If the receive value is more than 500 turn on the orange LED

} else {

Blink\_LED(LED\_off, orange\_LED); // Otherwise turn off the LED

}

if (Y < -500) {

// When the device is being tilted in a direction corresponding to the negative Y-axis the value of Y will be decreasing

// Here the sensitivity value for negative Y is -500 when the device is being tilted less than this value the blue led will turn on

Blink\_LED(LED\_on, blue\_LED); // If the receive value is less than -500 turn on the blue LED

} else {

Blink\_LED(LED\_off, blue\_LED); // Otherwise turn off the LED

}

if (X > 500) {

// When the device is being tilted in a direction corresponding to the positive X-axis the value of X will be increasing

// Here the sensitivity value for positive X is 500 when the device is being tilted more than this value the red led will turn on

Blink\_LED(LED\_on, red\_LED); // If the receive value is more than 500 turn on the red LED

} else {

Blink\_LED(LED\_off, red\_LED); // Otherwise turn off the LED

}

if (X < -500) {

// When the device is being tilted in a direction corresponding to the negative X-axis the value of X will be decreasing

// Here the sensitivity value for negative X is -500 when the device is being tilted less than this value the blue led will turn on

Blink\_LED(LED\_on, green\_LED); // If the received value is positive turn on the green LED

} else {

Blink\_LED(LED\_off, green\_LED); // Otherwise turn off the LED

}

osThreadYield(); // Yield the processor to other ready-to-run threads

}

}

/\* Thread(2): Blink main thread function \*/

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

osSignalClear(tid\_Blink\_Thread, 0x01); // Clear the 0x01 flag of the blink LED thread to prevent it from running when we upload the program

while (1) {

osSignalWait(0x01, osWaitForever); // Waits until flag 0x01 of this thread is set (it will be set the first time when we click on the push button)

osSignalSet(tid\_Blink\_Thread, 0x01); // Set flag 0x01 of the blink LED thread so that it resumes next time the wait is called

// Repeatedly blinking on for 0.5 seconds and off for 0.5 seconds

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

osDelay(500);

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

osDelay(500);

osSignalWait(0x01, osWaitForever); // If we click the push button again, it will pause the blinking. We may click on it during the blinking that’s why we called it again here

osSignalSet(tid\_Blink\_Thread, 0x01); // Set flag 0x01 of the blink LED thread so that it resumes next time the wait is called

osThreadYield(); // Yield the processor to other ready-to-run threads

}

}

/\* Thread(3): Button main thread function \*/

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

while (1) {

if(((GPIOA->IDR & 0x00000001) == 0x00000001) && flag == 0){

// If the push button has been clicked and the value for the flag variable is zero, then turn on the blinking, I2C and I2S threads and turn off the SPI thread

osSignalClear(tid\_SPI\_Thread, 0x01); // Clear the 0x01 flag of the SPI thread when we click on the button

flag = 1; // Set the flag value to 1. This ensures that if we click the button again, it will trigger a different functionality

// All four user LEDs blinking on for 1 second

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

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

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

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

osDelay(1000);

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

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

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

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

osSignalSet(tid\_Blink\_Thread, 0x01); // Set flag 0x01 of the blink LED thread so that it can run now

osSignalSet(tid\_I2C\_I2S\_Thread, 0x01); // Set flag 0x01 of the I2C and I2S thread so that it can run now

}

if (((GPIOA->IDR & 0x00000001) == 0x00000001) && flag == 1) {

// If the push button has been clicked and the value for the flag variable is one, then turn off the blinking, I2C and I2S threads and turn on the SPI thread

osSignalClear(tid\_Blink\_Thread, 0x01); // Clear the 0x01 flag of the blinking thread so it can stop working

osSignalClear(tid\_I2C\_I2S\_Thread, 0x01); // Clear the 0x01 flag of the I2C and I2S thread so it can stop working

osDelay(200); // A delay of 200 milliseconds so that we could remove the debounce effect of the switch

flag = 0; // Set the flag value to 0. This ensures that if we click the button again, it will trigger a different functionality

osSignalSet(tid\_SPI\_Thread, 0x01); // Set flag 0x01 of the SPI thread so that it can run now

}

osThreadYield(); // Yield the processor to other ready-to-run threads

}

}

/\* Thread(4): I2C and I2S main thread function \*/

void I2C\_I2S\_Thread(void const \*argument) {

I2C\_I2S\_Params\_Init \*i2c\_i2s\_params = (I2C\_I2S\_Params\_Init \*)argument; // To access the parameters from the main file using the struct that we have created

uint16\_t S[8] = {0x7FFF, 0x0000, 0x7FFF, 0x0000, 0x7FFF, 0x0000, 0x7FFF, 0x0000}; // A square wave array

while (1) {

osSignalWait(0x01, osWaitForever); // Waits until flag 0x01 of this thread is set (it will be set the first time when we click on the push button)

osSignalSet(tid\_I2C\_I2S\_Thread, 0x01); // Set flag 0x01 of the I2C and I2S thread so that it resumes next time the wait is called

HAL\_I2S\_Transmit\_IT(i2c\_i2s\_params->I2S\_Params, S, 8); // Transmit the data of S through I2S to generate the sound

osDelay(1000); // Delay of one second

osSignalWait(0x01, osWaitForever); // If we click the push button again, it will pause the I2C and I2C thread. We may click on it during the transmission that’s why we called it again here

osSignalSet(tid\_I2C\_I2S\_Thread, 0x01); // Set flag 0x01 of the blink LED thread so that it resumes next time the wait is called

osThreadYield(); // Yield the processor to other ready-to-run threads

}

}

**main.c:**

#define osObjectsPublic

#include "osObjects.h"

#include "stm32f4xx.h"

#include "my\_headers.h"

#include "Thread.h"

SPI\_HandleTypeDef SPI\_Params; // Declares the structure handle for SPI

uint8\_t address; // Declare a variable to hold the address for SPI

I2C\_HandleTypeDef I2C\_Params; // Declares the structure handle for I2C

I2S\_HandleTypeDef I2S\_Params; // Declares the structure handle for I2S

SPI\_Params\_Init spi\_params; // Declare a variable of type ‘SPI\_Params\_Init’ that we have created in ‘my\_defines.h’ for SPI

I2C\_I2S\_Params\_Init i2c\_i2s\_params; // Declare a variable of type ‘I2C\_I2S\_Params\_Init’ that we have created in ‘my\_defines.h’ for I2C and I2S

int main(void) {

SystemCoreClockUpdate();

spi\_params.SPI\_Params = &SPI\_Params; // Assign the value of ‘SPI\_Params’ in the struct ‘spi\_params’ with the variable the we declared outside main ‘SPI\_Params’

spi\_params.address = &address; // Assign the value of address in the struct ‘spi\_params’ with the variable the we declared outside main ‘address’

spi\_params.data\_size = 1; // Assign the value of ‘data\_size’ in the struct ‘spi\_params’ with 1

spi\_params.data\_timeout = 1000; // Assign the value of ‘data\_timeout’ in the struct ‘spi\_params’ with 1000

i2c\_i2s\_params.I2C\_Params = &I2C\_Params; // Assign the value of ‘I2C\_Params’ in the struct ‘i2c\_i2s\_params’ with the variable the we declared outside main ‘I2C\_Params’

i2c\_i2s\_params.I2S\_Params = &I2S\_Params; // Assign the value of ‘I2S\_Params’ in the struct ‘i2c\_i2s\_params’ with the variable the we declared outside main ‘I2S\_Params’

osKernelInitialize(); // Initialize CMSIS-RTOS

/\* Initialise any peripherals or system components \*/

Initialise\_button();

Initialise\_SPI\_LEDs(spi\_params);

Initialise\_I2C\_I2S(i2c\_i2s\_params);

/\* Initialise the main threads \*/

Init\_SPI\_Thread(&spi\_params);

Init\_Blink\_Thread();

Init\_Button\_Thread();

Init\_I2C\_I2S\_Thread(&i2c\_i2s\_params);

osKernelStart(); // start thread execution

while (1) {}; // While loop so the program doesn’t terminate

}