**Serial Communication, Polling and Interrupts**

**Complete code and Task Solutions**

Contents

[Part A 2](#_Toc471747061)

[Task A.1 4](#_Toc471747062)

[Task A.2 4](#_Toc471747063)

[Task A.3 4](#_Toc471747064)

[Task A.4 6](#_Toc471747065)

[Part B.1 10](#_Toc471747066)

[Part B.2 11](#_Toc471747067)

[Task B.1 12](#_Toc471747068)

[Part C 14](#_Toc471747069)

[Task C.1 17](#_Toc471747070)

# Part A

**Complete contents of “main.c” required to setup the SPI interface and read the “who am I” register.**

#include "stm32f4xx.h"

int main(){

// Declarations

SPI\_HandleTypeDef SPI\_Params; // Declares the structure handle for the parameters of SPI1

GPIO\_InitTypeDef GPIOA\_Params; // Declares the structure handle for the parameters of GPIOA

GPIO\_InitTypeDef GPIOE\_Params; // Declares the structure handle for the parameters of GPIOE

uint8\_t data\_to\_send[1]; //Declares an array to store the required LIS3DSH register address in. It has a single element since we will only be accessing a single address in each SPI transaction.

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

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

uint8\_t Who\_am\_I; //Declares the variable to store the who\_am\_I register value in

// Code to initialise the SPI

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

SPI\_Params.Instance = SPI1; // Selects which SPI interface to use

SPI\_Params.Init.Mode = SPI\_MODE\_MASTER; // Sets the STM32F407 to act as the master

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

SPI\_Params.Init.Direction = SPI\_DIRECTION\_2LINES; // Sets the 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 the idle polarity for the clock line to high

SPI\_Params.Init.CLKPhase = SPI\_PHASE\_2EDGE; // Sets the data line to change 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 the clock prescaler to divide the main APB2 clock (previously set to 84MHz) by 32 to give a SPI clock of 2.625MHz, which is less the maximum value of 10MHz for the SPI.

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

// Code to initialise pins 5-7 of GPIOA

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

GPIOA\_Params.Pin = GPIO\_PIN\_5 | GPIO\_PIN\_6 | GPIO\_PIN\_7; // Selects pins 5,6 and 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 push-pull mode

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); // Sets GPIOA into the modes specified in GPIOA\_Params

// Code to initialise pin 3 of GPIOE

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

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

GPIOE\_Params.Mode = GPIO\_MODE\_OUTPUT\_PP; //Selects normal 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); // Sets GPIOE into the modes specified in GPIOE\_Params

GPIOE->BSRR |= GPIO\_PIN\_3; //Sets the serial port enable pin CS high (idle)

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

// Initialize GPIO Port for LEDs

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN; // Enable Port D clock

GPIOD->MODER |= GPIO\_MODER\_MODER14\_0; // Port D.14 output - red LED

GPIOD->MODER |= GPIO\_MODER\_MODER12\_0; // Port D.12 output - green LED

// Read the value from the Who\_am\_I register of the LIS3DSH

data\_to\_send[0] = 0x80|0x0f; // Address for Who\_am\_I register on 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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

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

HAL\_SPI\_Receive(&SPI\_Params,data\_to\_send,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

Who\_am\_I = \*SPI\_Params.pRxBuffPtr; // Read the data from the SPI buffer sub-structure into our Who\_am\_I variable.

if(Who\_am\_I == 0x3f){ // Check to see if the received value is the same as the expected value

GPIOD->BSRR |= (1<<12); // If the receive value is the same turn on the green LED

}

else{

GPIOD->BSRR |= (1<<14); // If the received value is different turn on the red LED

}

}

# Task A.1

The line

SPI\_Params.Init.DataSize = SPI\_DATASIZE\_8BIT;

would become

SPI\_Params.Init.DataSize = SPI\_DATASIZE\_16BIT;

The pre-defined value SPI\_DATASIZE\_16BIT is found on line 195 of stm32f4xx\_hal\_spi.h and SPI control register 1 is defined on page 905 (bit 2) of the “STM32F - Processor Reference Manual”

and the line

SPI\_Params.Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_32;

would become

SPI\_Params.Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_64;

The new baud rate prescaler needs to set a SPI clock speed to a maximum of 2Mhz. The smallest prescaler (and thus largest SPI clock frequency) that meets this is 84/64 = 1.3125MHz (see the end of step 3 in the lab sheet for the reason behind this calculation). The pre-defined value SPI\_BAUDRATEPRESCALER\_64 is found on line 245 of stm32f4xx\_hal\_spi.h and SPI control register 1 is defined on page 904 (bits 5:3) of the “STM32F - Processor Reference Manual”.

# Task A.2

The address of the status register is 0x21 (or 21h) and the value it should contain is 000-0--12 (- indicates it doesn’t matter or isn’t defined)

# Task A.3

#include "stm32f4xx.h"

int main(){

// Declarations

SPI\_HandleTypeDef SPI\_Params; // Declares the structure handle for the parameters of SPI1

GPIO\_InitTypeDef GPIOA\_Params; // Declares the structure handle for the parameters of GPIOA

GPIO\_InitTypeDef GPIOE\_Params; // Declares the structure handle for the parameters of GPIOE

uint8\_t data\_to\_send[1]; //Declares an array to store the required LIS3DSH register address in. It has a single element since we will only be accessing a single address in each SPI transaction.

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

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

uint8\_t Reg\_value; //Declares the variable to store the register value in

// Code to initialise the SPI

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

SPI\_Params.Instance = SPI1; // Selects which SPI interface to use

SPI\_Params.Init.Mode = SPI\_MODE\_MASTER; // Sets the STM32F407 to act as the master

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

SPI\_Params.Init.Direction = SPI\_DIRECTION\_2LINES; // Sets the 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 the idle polarity for the clock line to high

SPI\_Params.Init.CLKPhase = SPI\_PHASE\_2EDGE; // Sets the data line to change 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 the clock prescaler to divide the main APB2 clock (previously set to 84MHz) by 32 to give a SPI clock of 2.625MHz, which is less the maximum value of 10MHz for the SPI.

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

// Code to initialise pins 5-7 of GPIOA

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

GPIOA\_Params.Pin = GPIO\_PIN\_5 | GPIO\_PIN\_6 | GPIO\_PIN\_7; // Selects pins 5,6 and 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 push-pull mode

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); // Sets GPIOA into the modes specified in GPIOA\_Params

// Code to initialise pin 3 of GPIOE

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

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

GPIOE\_Params.Mode = GPIO\_MODE\_OUTPUT\_PP; //Selects normal 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); // Sets GPIOE into the modes specified in GPIOE\_Params

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

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

// Initialize GPIO Port for LEDs

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN; // Enable Port D clock

GPIOD->MODER |= GPIO\_MODER\_MODER14\_0; // Port D.14 output - red LED

GPIOD->MODER |= GPIO\_MODER\_MODER12\_0; // Port D.12 output - green LED

// Write a new value to control register 4 of the LIS3DSH

data\_to\_send[0] = 0x20; // Address for control register 4 on 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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

data\_to\_send[0] = 0x14; // Set register value to give a sample rate of 3.125Hz, continuous update and z-axis enabled only

HAL\_SPI\_Transmit(&SPI\_Params,data\_to\_send,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

// Read the value from control register 4 of the LIS3DSH

data\_to\_send[0] = 0x80|0x20; // Address for control register 4 on 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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

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

HAL\_SPI\_Receive(&SPI\_Params,data\_to\_send,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

Reg\_value = \*SPI\_Params.pRxBuffPtr; // Read the data from the SPI buffer sub-structure into our internal variable.

if(Reg\_value == 0x14){ // Check to see if the received value is the same as the expected value

GPIOD->BSRR |= (1<<12); // If the receive value is the same turn on the green LED

}

else{

GPIOD->BSRR |= (1<<14); // If the received value is different turn on the red LED

}

}

# Task A.4

**Solutions**

A. 0x80|0x2D;

B. HAL\_SPI\_Transmit(&SPI\_Params,data\_to\_send,data\_size,data\_timeout);

C. HAL\_SPI\_Receive(&SPI\_Params,data\_to\_send,data\_size,data\_timeout);

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

**Full code**

#include "stm32f4xx.h"

int main(){

// Declarations

SPI\_HandleTypeDef SPI\_Params; // Declares the structure handle for the parameters of SPI1

GPIO\_InitTypeDef GPIOA\_Params; // Declares the structure handle for the parameters of GPIOA

GPIO\_InitTypeDef GPIOE\_Params; // Declares the structure handle for the parameters of GPIOE

uint8\_t data\_to\_send[1]; //Declares an array to store the required LIS3DSH register address in. It has a single element since we will only be accessing a single address in each SPI transaction.

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

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

uint8\_t Z\_Reg\_H; //Declares the variable to store the z-axis MS 8-bits in

// Code to initialise the SPI

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

SPI\_Params.Instance = SPI1; // Selects which SPI interface to use

SPI\_Params.Init.Mode = SPI\_MODE\_MASTER; // Sets the STM32F407 to act as the master

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

SPI\_Params.Init.Direction = SPI\_DIRECTION\_2LINES; // Sets the 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 the idle polarity for the clock line to high

SPI\_Params.Init.CLKPhase = SPI\_PHASE\_2EDGE; // Sets the data line to change 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 the clock prescaler to divide the main APB2 clock (previously set to 84MHz) by 32 to give a SPI clock of 2.625MHz, which is less the maximum value of 10MHz for the SPI.

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

// Code to initialise pins 5-7 of GPIOA

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

GPIOA\_Params.Pin = GPIO\_PIN\_5 | GPIO\_PIN\_6 | GPIO\_PIN\_7; // Selects pins 5,6 and 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 push-pull mode

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); // Sets GPIOA into the modes specified in GPIOA\_Params

// Code to initialise pin 3 of GPIOE

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

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

GPIOE\_Params.Mode = GPIO\_MODE\_OUTPUT\_PP; //Selects normal 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); // Sets GPIOE into the modes specified in GPIOE\_Params

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

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

// Initialize GPIO Port for LEDs

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN; // Enable Port D clock

GPIOD->MODER |= GPIO\_MODER\_MODER14\_0; // Port D.14 output - red LED

GPIOD->MODER |= GPIO\_MODER\_MODER12\_0; // Port D.12 output - green LED

// Write a new value to control register 4 of the LIS3DSH

data\_to\_send[0] = 0x20; // Address for control register 4 on 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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

data\_to\_send[0] = 0x14; // Set register value to give a sample rate of 3.125Hz, continuous update and z-axis enabled only

HAL\_SPI\_Transmit(&SPI\_Params,data\_to\_send,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

for(;;){

// Read the value from the MSB (h) z-axis data register of the LIS3DSH – Only the higher 8-bits are needed as this contains the sign bit in 2’s complement and only the sign of the z-axis acceleration is needed to determine the orientation of the board in this case.

data\_to\_send[0] = 0x80|0x2D; // Address for the MSB z-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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

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

HAL\_SPI\_Receive(&SPI\_Params,data\_to\_send,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

Z\_Reg\_H = \*SPI\_Params.pRxBuffPtr; // Read the data from the SPI buffer sub-structure into our internal variable.

if((Z\_Reg\_H&0x80) == 0x80){ // Check to see if the received value is positive or negative - the acceleration is a signed 16-bit number so the MSB is the sign bit - 1 is negative, 0 is positive. Refer to the C Programming guide document if you are unclear about this.

GPIOD->BSRR |= (1<<12); // If the receive value is negative turn on the green LED

GPIOD->BSRR |= (1<<(14+16)); // If the receive value is negative turn off the red LED

}

else{

GPIOD->BSRR |= (1<<14); // If the received value is positive turn on the red LED

GPIOD->BSRR |= (1<<(12+16)); // If the received value is positive turn off the green LED

}

}

}

# Part B.1

**Basic functions**

**my\_headers.h**

void Initialise\_LED(void); // Declaration for the function to initialise the LED

void Blink\_LED(void); // Declaration for the function to blink the LED

**my\_headers.c**

#include "stm32f4xx.h"

// Definition for the function to initialise the LED

void Initialise\_LED(void){

// Initialize GPIO Port for LEDs

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN; // Enable Port D clock

GPIOD->MODER |= GPIO\_MODER\_MODER12\_0; // Port D.12 output - green LED

}

// Definition for the function to blink the LED

void Blink\_LED(void){

uint32\_t ii;

GPIOD->BSRR = 1<<12; // Turn on the green LED

for(ii=0;ii<26000000;ii++){} // FOR loop to implement a 1 second delay with a clock speed of 168MHz

GPIOD->BSRR = 1<<(12+16); // Turn off the green LED

for(ii=0;ii<26000000;ii++){} // FOR loop to implement a 1 second delay with a clock speed of 168MHz

}

**main.c**

#include "my\_headers.h"

int main(void){

Initialise\_LED(); // Initialises the LED

for(;;){

Blink\_LED(); // Blinks the green LED on and off once

}

}

# Part B.2

**Functions with Input/Output Parameters**

**my\_headers.h**

#include "stm32f4xx.h"

void Initialise\_LED(void); // Declaration for the function to initialise the LED

void Blink\_LED(uint8\_t); // Declaration for the function to blink the LED

**my\_headers.c**

#include "stm32f4xx.h"

// Definition for the function to initialise the LED

void Initialise\_LED(void){

// Initialize GPIO Port for LEDs

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN; // Enable Port D clock

GPIOD->MODER |= GPIO\_MODER\_MODER12\_0; // Port D.12 output - green LED

}

// Definition for the function to blink the LED

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

uint32\_t ii;

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

GPIOD->BSRR = 1<<12; // Turn on the green LED

}

else{

GPIOD->BSRR = 1<<(12+16); // Turn off the green LED

}

for(ii=0;ii<26000000;ii++){} // FOR loop to implement a 1 second delay with a clock speed of 168MHz

}

**main.c**

#include "stm32f4xx.h"

#include "my\_headers.h"

int main(void){

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

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

Initialise\_LED(); // Initialises the LED

for(;;){

Blink\_LED(LED\_on); // Blinks the green LED on once

Blink\_LED(LED\_off); // Blinks the green LED off once

}

}

# Task B.1

**my\_headers.h**

#include "stm32f4xx.h"

void Initialise\_LED(void); // Declaration for the function to initialise the LED

void Blink\_LED(uint8\_t); // Declaration for the function to blink the LED

void Delay\_in\_seconds(uint8\_t); // Declaration for the delay function specified in seconds

**my\_headers.c**

#include "stm32f4xx.h"

// Definition for the function to initialise the LED

void Initialise\_LED(void){

// Initialize GPIO Port for LEDs

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN; // Enable Port D clock

GPIOD->MODER |= GPIO\_MODER\_MODER12\_0; // Port D.12 output

}

// Definition for the function to blink the LED

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

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

GPIOD->BSRR = 1<<12; // Turn on the green LED

}

else{

GPIOD->BSRR = 1<<(12+16); // Turn off the green LED

}

}

void Delay\_in\_seconds(uint8\_t time\_in\_seconds){

uint32\_t ii;

for(ii=0;ii<(time\_in\_seconds\*26000000);ii++){} // FOR loop to implement a 1 second delay with a clock speed of 168MHz

}

**main.c**

#include "stm32f4xx.h"

#include "my\_headers.h"

int main(void){

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

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

Initialise\_LED(); // Initialises the LED

for(;;){

Blink\_LED(LED\_on); // Blinks the green LED on once

Delay\_in\_seconds(4);

Blink\_LED(LED\_off); // Blinks the green LED off once

Delay\_in\_seconds(1);

}

}

# Part C

#include "stm32f4xx.h"

int main(){

// Declarations

SPI\_HandleTypeDef SPI\_Params; // Declares the structure handle for the parameters of SPI1

GPIO\_InitTypeDef GPIOA\_Params; // Declares the structure handle for the parameters of GPIOA

GPIO\_InitTypeDef GPIOE\_Params; // Declares the structure handle for the parameters of GPIOE

GPIO\_InitTypeDef GPIOE\_Params\_I; // Declares the structure handle for the parameters of the interrupt pin on GPIOE

uint8\_t data\_to\_send[1]; //Declares an array to store the required LIS3DSH register address in. It has a single element since we will only be accessing a single address in each SPI transaction.

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

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

uint8\_t Z\_Reg\_H; //Declares the variable to store the z-axis MS 8-bits in

uint8\_t Z\_Reg\_L; //Declares the variable to store the z-axis LS 8-bits in

int8\_t Z\_Accn; //Declares the variable to store the z-axis acceleration in - note that this is a signed 16-bit number

// Code to initialise the SPI

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

SPI\_Params.Instance = SPI1; // Selects which SPI interface to use

SPI\_Params.Init.Mode = SPI\_MODE\_MASTER; // Sets the STM32F407 to act as the master

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

SPI\_Params.Init.Direction = SPI\_DIRECTION\_2LINES; // Sets the 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 the idle polarity for the clock line to high

SPI\_Params.Init.CLKPhase = SPI\_PHASE\_2EDGE; // Sets the data line to change 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 the clock prescaler to divide the main APB2 clock (previously set to 84MHz) by 32 to give a SPI clock of 2.625MHz, which is less the maximum value of 10MHz for the SPI.

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

// Code to initialise pins 5-7 of GPIOA

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

GPIOA\_Params.Pin = GPIO\_PIN\_5 | GPIO\_PIN\_6 | GPIO\_PIN\_7; // Selects pins 5,6 and 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 push-pull mode

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); // Sets GPIOA into the modes specified in GPIOA\_Params

// Code to initialise pin 3 of GPIOE

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

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

GPIOE\_Params.Mode = GPIO\_MODE\_OUTPUT\_PP; //Selects normal 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); // Sets GPIOE into the modes specified in GPIOE\_Params

//Code to initialise GPIOE pin 0 for the interrupt

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

GPIOE\_Params\_I.Mode = GPIO\_MODE\_IT\_RISING; // Selects the interrupt mode and configures the interrupt to be signalled on a rising edge (low to high transition)

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

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

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

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

// Initialize GPIO Port for LEDs

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN; // Enable Port D clock

GPIOD->MODER |= GPIO\_MODER\_MODER14\_0; // Port D.14 output - red LED

GPIOD->MODER |= GPIO\_MODER\_MODER12\_0; // Port D.14 output - green LED

// Write a new value to control register 4 of the LIS3DSH

data\_to\_send[0] = 0x20; // Address for control register 4 on 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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

data\_to\_send[0] = 0x14; // Set register value to give a sample rate of 3.125Hz, continuous update and z-axis enabled only

HAL\_SPI\_Transmit(&SPI\_Params,data\_to\_send,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

// Write a new value to control register 3 of the LIS3DSH to configure the interrupts

data\_to\_send[0] = 0x23; // Address for control register 3 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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

data\_to\_send[0] = 0xC8; // Enable DRDY connected to Int1, sets Int1 active to high, enables int1

HAL\_SPI\_Transmit(&SPI\_Params,data\_to\_send,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

// TASK C.1 - ADD SOME CODE HERE TO TURN ON THE ORANGE LED AT THE START OF THE CODE (Point B)

for(;;){

if (\_\_HAL\_GPIO\_EXTI\_GET\_IT(GPIO\_PIN\_0)==SET){ // Checks to see if the interrupt line has been set

\_\_HAL\_GPIO\_EXTI\_CLEAR\_IT(GPIO\_PIN\_0); // Clears the interrupt before proceeding to service the interrupt

// TASK C.1 - ADD SOME CODE TO IMPLEMENT THE ORANGE/BLUE LED SWITCHING (Point C)

// Read the value from the MSB z-axis data register of the LIS3DSH

data\_to\_send[0] = 0x80|0x2D; // Address for control register 4 on 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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

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

HAL\_SPI\_Receive(&SPI\_Params,data\_to\_send,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

Z\_Reg\_H = \*SPI\_Params.pRxBuffPtr; // Read the data from the SPI buffer sub-structure into our internal variable.

if((Z\_Accn&0x80) == 0x80){ // Check to see if the received value is positive or negative - the acceleration is a signed 8-bit number so the MSB is the sign bit - 1 is negative, 0 is positive. Refer to the C Programming guide document if you are unclear about this.

GPIOD->BSRR |= (1<<12); // If the receive value is negative turn on the green LED

GPIOD->BSRR |= (1<<(14+16)); // If the receive value is negative turn of the red LED

}

else{

GPIOD->BSRR |= (1<<14); // If the received value is positive turn on the red LED

GPIOD->BSRR |= (1<<(12+16)); // If the received value is positive turn of the green LED

}

}

else{

// Here we could add code to perform another task, or we could modify the code so that the STM32F407 powers down until the interrupt is signalled.

}

}

}

# Task C.1

**Solutions**

A.

GPIOD->MODER |= GPIO\_MODER\_MODER15\_0; // Port D.15 output - blue LED

GPIOD->MODER |= GPIO\_MODER\_MODER13\_0; // Port D.13 output - orange

B.

GPIOD->BSRR |= (1<<13); // Turns on the orange LED

C.

if((GPIOD->ODR & (1<<13))==(1<<13)){ // Checks to see if the orange LED is on

GPIOD->BSRR |= (1<<(13+16)); // Turns off the orange LED

GPIOD->BSRR |= (1<<15); // Turns on the blue LED

}

else{

GPIOD->BSRR |= (1<<(15+16)); // Turns off the blue LED

GPIOD->BSRR |= (1<<13); // Turns on the orange LED

}

**FULL CODE**

#include "stm32f4xx.h"

int main(){

// Declarations

SPI\_HandleTypeDef SPI\_Params; // Declares the structure handle for the parameters of SPI1

GPIO\_InitTypeDef GPIOA\_Params; // Declares the structure handle for the parameters of GPIOA

GPIO\_InitTypeDef GPIOE\_Params; // Declares the structure handle for the parameters of GPIOE

GPIO\_InitTypeDef GPIOE\_Params\_I; // Declares the structure handle for the parameters of the interrupt pin on GPIOE

uint8\_t data\_to\_send[1]; //Declares an array to store the required LIS3DSH register address in. It has a single element since we will only be accessing a single address in each SPI transaction.

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

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

uint8\_t Z\_Reg\_H; //Declares the variable to store the z-axis MS 8-bits in

uint8\_t Z\_Reg\_L; //Declares the variable to store the z-axis LS 8-bits in

int8\_t Z\_Accn; //Declares the variable to store the z-axis acceleration in - note that this is a signed 16-bit number

// Code to initialise the SPI

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

SPI\_Params.Instance = SPI1; // Selects which SPI interface to use

SPI\_Params.Init.Mode = SPI\_MODE\_MASTER; // Sets the STM32F407 to act as the master

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

SPI\_Params.Init.Direction = SPI\_DIRECTION\_2LINES; // Sets the 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 the idle polarity for the clock line to high

SPI\_Params.Init.CLKPhase = SPI\_PHASE\_2EDGE; // Sets the data line to change 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 the clock prescaler to divide the main APB2 clock (previously set to 84MHz) by 32 to give a SPI clock of 2.625MHz, which is less the maximum value of 10MHz for the SPI.

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

// Code to initialise pins 5-7 of GPIOA

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

GPIOA\_Params.Pin = GPIO\_PIN\_5 | GPIO\_PIN\_6 | GPIO\_PIN\_7; // Selects pins 5,6 and 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 push-pull mode

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); // Sets GPIOA into the modes specified in GPIOA\_Params

// Code to initialise pin 3 of GPIOE

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

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

GPIOE\_Params.Mode = GPIO\_MODE\_OUTPUT\_PP; //Selects normal 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); // Sets GPIOE into the modes specified in GPIOE\_Params

//Code to initialise GPIOE pin 0 for the interrupt

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

GPIOE\_Params\_I.Mode = GPIO\_MODE\_IT\_RISING; // Selects the interrupt mode and configures the interrupt to be signalled on a rising edge (low to high transition)

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

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

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

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

// Initialize GPIO Port for LEDs

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIODEN; // Enable Port D clock

GPIOD->MODER |= GPIO\_MODER\_MODER15\_0; // Port D.15 output - blue LED

GPIOD->MODER |= GPIO\_MODER\_MODER14\_0; // Port D.14 output - red LED

GPIOD->MODER |= GPIO\_MODER\_MODER13\_0; // Port D.13 output - orange LED

GPIOD->MODER |= GPIO\_MODER\_MODER12\_0; // Port D.14 output - green LED

// Write a new value to control register 4 of the LIS3DSH

data\_to\_send[0] = 0x20; // Address for control register 4 on 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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

data\_to\_send[0] = 0x14; // Set register value to give a sample rate of 3.125Hz, continuous update and z-axis enabled only

HAL\_SPI\_Transmit(&SPI\_Params,data\_to\_send,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

// Write a new value to control register 3 of the LIS3DSH to configure the interrupts

data\_to\_send[0] = 0x23; // Address for control register 3 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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

data\_to\_send[0] = 0xC8; // Enable DRDY connected to Int1, sets Int1 active to high, enables int1

HAL\_SPI\_Transmit(&SPI\_Params,data\_to\_send,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

GPIOD->BSRR |= (1<<13); // Turns on the orange LED

for(;;){

if (\_\_HAL\_GPIO\_EXTI\_GET\_IT(GPIO\_PIN\_0)==SET){ // Checks to see if the interrupt line has been set

\_\_HAL\_GPIO\_EXTI\_CLEAR\_IT(GPIO\_PIN\_0); // Clears the interrupt before proceeding to service the interrupt

if((GPIOD->ODR & (1<<13))==(1<<13)){ // Checks to see if the orange LED is on

GPIOD->BSRR |= (1<<(13+16)); // Turns off the orange LED

GPIOD->BSRR |= (1<<15); // Turns on the blue LED

}

else{

GPIOD->BSRR |= (1<<(15+16)); // Turns off the blue LED

GPIOD->BSRR |= (1<<13); // Turns on the orange LED

}

// Read the value from the MSB z-axis data register of the LIS3DSH

data\_to\_send[0] = 0x80|0x2D; // Address for control register 4 on 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,data\_size,data\_timeout); // Send the address of the register to be read on the LIS3DSH

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

HAL\_SPI\_Receive(&SPI\_Params,data\_to\_send,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

Z\_Reg\_H = \*SPI\_Params.pRxBuffPtr; // Read the data from the SPI buffer sub-structure into our internal variable.

if((Z\_Accn&0x80) == 0x80){ // Check to see if the received value is positive or negative - the acceleration is a signed 8-bit number so the MSB is the sign bit - 1 is negative, 0 is positive. Refer to the C Programming guide document if you are unclear about this.

GPIOD->BSRR |= (1<<12); // If the receive value is negative turn on the green LED

GPIOD->BSRR |= (1<<(14+16)); // If the receive value is negative turn of the red LED

}

else{

GPIOD->BSRR |= (1<<14); // If the received value is positive turn on the red LED

GPIOD->BSRR |= (1<<(12+16)); // If the received value is positive turn of the green LED

}

}

else{

// Here we could add code to perform another task, or we could modify the code so that the STM32F407 powers down until the interrupt is signalled.

}

}

}