Serial Communication, Functions, Polling and Interrupts

Aim

There are three aims for this lab.

1. Extend your knowledge/skills in digital I/O to serial communication and standard communication protocols, allowing you to design and implement more complex embedded systems.
2. Understand and be able to create programs using functions.
3. To develop your knowledge of techniques to control the timing of tasks and communication in embedded systems to include polling and interrupts and use these to solve appropriate problems in practice.

Structure of the Lab sheet

The lab sheet is split into three parts. Part A looks at serial IO, part B looks at functions and part C looks at task/communication control mechanisms. These parts develop the required skills to be able to use functions and serial IO. Additional tasks allow your understanding and ability to use functions and the serial IO to develop further. At the end of each task save the complete contents of you "main.c" file into a separate text file (copy and paste) so that you can access it easily at a later stage.

Date lab sheet is available from: Week 1 Monday 0900

Date of lab support session: Group 1 – Week 6 Friday 1100-1250; Group 2 – Week 6 Friday 1300-1450

Learning process

This lab sheet is generally self-contained. However, at times you will need to refer to other documents. You will be prompted when this is necessary and please take the effort to do so.

The purpose of the take home kit is to allow you to spend as much time as possible using the kit. It also allows you to learn independently and to explore the capabilities of the kit on your own. Please spend as much time as possible with the kit and work through the lab sheet in your own time. You might complete this lab without any problems and find you do not need much support in the structured lab session. Otherwise, you might need some support from the demonstrators in the structured lab session. Either way, if you attend the structured lab session you must be prepared with specific questions. The demonstrators will not sit down and take you through the laboratory step by step. **It is your responsibility to make a reasonable attempt at the laboratory before you seek assistance.**

A video guide implementing the steps in part A, B and C is also available on MOLE. However, this video does not cover most of the details behind the steps, it is only a guide to how to implement them. **To be able to successfully complete this module you should consult both this document and the video guide for the introductory lab.**

This laboratory is not assessed. However, use of the techniques taught in this lab sheet will be required in the in-class test and the final assignment.

Support

If you have any problems with the equipment please contact Dr S A Pope immediately using the contact details provided below. **These inquiries should only relate to problems with the equipment, such as suspected faults.** Any inquiry should include a clear description of the problem. For example "It isn't working" would not be an acceptable inquiry on its own. **Any learning based inquiries must be brought to the timetabled support sessions (lectures, laboratories and demonstrations).**

Contact details

Dr S A Pope, Email: s.a.pope@sheffield.ac.uk, Internal telephone extension: 25186, Room: C07d AJB

Contents

Initialisation - Setup a project in Keil µVision

For each of the three parts of this lab you will first need to create a new project in Keil µVision using the same process as you used in the introductory lab. Connect the STM32F4Discovery to the PC and follow steps 1 - 7 in the Introductory lab sheet. You will initially need the same components of the STM32F4Discovery that you used in the Introductory lab. So you should select the same components in the "Manage Run-Time Environment" window in step 3.

Part A - Setting up a serial communication routine

Up until this point you have only dealt with the most basic form of digital I/O - i.e. simple on/off devices that can be controlled by reading/writing to individual pins on a port. However, in most cases, such as sensor measurements, receiving/sending data is not so straightforward. For example, the STM32F4Discovery board includes a digital accelerometer, the LIS3DSH, which is capable of measuring three orthogonal axes (x, y and z) across a range of sample rates and measurement ranges. The output from the accelerometer is a signed 16-bit number. There are two possible ways that this 16-bit number can be sent along a data bus - parallel and serial. Please refer to the slides for Lecture/Demo 3 for more information on parallel and serial I/O.

The LIS3DSH uses a serial communication protocol in which each individual bit in a 16-bit measurement is sent in series. Serial communication is by far the most common type of I/O, due to factors such as its flexibility and performance. In particular, only a single pin on a microcontroller is needed for the connection of each serial data line (additional pins might be needed, for example for clock lines). However, unlike with the LED's and push-button switches which required a single on or off state to be written/read, a string of data bits (16 in the case of the LIS3DSH) needs to be sent/received through a serial communication pin. Additionally, the data communication usually follows a particular protocol, i.e. a pre-defined process. For the LIS3DSH this is the SPI and I2C protocol. SPI and I2C are two of the widely used serial communication protocols. With SPI the data is formatted in a particular way and sent at a particular rate. When selecting a sensor to use with a particular microcontroller you need to ensure that both support the same communication protocol (see additional reading document "ADC and DAC - Sensors and Actuators"). With the STM32F407 microcontroller and LIS3DSH accelerometer, both support the SPI protocol and it is this protocol that we will use to communicate between the two devices in this lab.

The next steps guide you through setting up the serial communication routine between the STM32F407 and LIS3DSH. You will need to complete these steps before you can attempt the assessed part of the lab. The complete code resulting from these steps is provided in the "Functions and serial IO - Complete code and task solutions" document for reference. However, it is important that you try and understand this code as you will need to modify it for assessed lab 2. You will need three documents to complete the following steps. These are the: STM32F4Discovery - User Manual, STM32F407 processor reference manual and the LIS3DSH data sheet.

STEP 1 - Determine how the LIS3DSH is connected to the STM32F407

For this step you will need the STM32F4Discovery User Manual.

Section 4.8 of the STM32F4Discovery User Manual provides you with some information on the LIS3DSH (note the boards that you are using have the latest LIS3DSH accelerometer and not the older LIS302DL). This section gives you some basic information on the LIS3DSH and tells you that the SPI interface is used. Next, go to section 4.12 (page 21). This section gives you information on how each peripheral on the Discovery board (listed along the top of the table) is connected to the STM32F407 microcontroller. You will see that the LIS3DSH is listed in the 6th column of the table (4th column of the peripheral list). You now need to follow this column down until you find some entries in the column (you will find the entries on pages 22 and 30). On page 22 you will see three entries:

- SCL/SPC connected to PA5 of the STM32F407 processor
- SDO connected to PA6 of the STM32F407 processor
- SDA/SDI/SDO connected to PA7 of the STM32F407 processor

These are the clock, input data line and output data line of the SPI interface respectively. From the second column you will see that these correspond to the SPI1 channel of the STM32F407 processor (the STM32F407 has more than one SPI channel, so it is important to note SPI1). There is some other important information that can be extracted from this table. The first is that the pins used are PA5-7, which are pins 5-7 of GPIO A. In other words, the physical connection is through GPIO A of the STM32F407 and the SPI protocol is then implement on-board (i.e. there are not separate pins for the SPI interface). Secondly, PA5-7 also correspond to other alternate functions, such as AD/DA conversion. Therefore, only one of these available alternate functions can be used at a time. However, since most functions have multiple channels mapped to different ports of the microcontroller, an arrangement can usually be found that accommodates most system designs. This multiple mapping approach is common on most microcontrollers and allows a processor with a finite amount of pins to offer a wide range of possible design choices. On page 30 you will also see that three pins of GPIOE of the processor are connected to the LIS3DSH.

- PE0 and PE1 are connected to the interrupt lines (we will use these later)
- PE3 is connected to the CS line and is used to initiate data communication

STEP 2 - Determining the parameters needed to setup the SPI

In this step you will need to refer to the STM32F4 processor reference manual and the LIS3DSH data sheet.

The first thing we need to do is understand a bit more about the SPI interface on the STM32F407. Information on the SPI interface is found in section 28 (page 859) of the processor reference manual. The introductory paragraph in section 28.1 says that the SPI interface supports both the SPI and I2S interfaces. We will only use the SPI interface. In this step you will be taken through the sequence required to setup the SPI interface, but you might find it useful to refer to the following sections at various stages so that you fully understand the process.

- Section 28.2 describes the main features of the SPI interface.
- Section 28.3.1 describes the operation of the pins
- Section 28.3.3 provides the information required to setup the SPI interface in master mode
- Section 28.5 describes the registers associated with the SPI interface.

We need a bit more information on the settings that the LIS3DSH requires for the SPI interface. Section 6.2 of the LIS3DSH data sheet provides the information that we need. Four wires are needed for the SPI connection:

- SCL connected to PA5 on the STM32F407 is the clock line
- SDO connected to PA6 on the STM32F407 is the output data line
- SDI connected to PA7 on the STM32F407 is the input data line
- CS connected to PE3 on the STM32F407 is the serial port enable line

We can see from this that the combination of two data lines SDI and SDO, allows the LIS3DSH to both receive and transmit data simultaneously in what is referred to as full-duplex communication. The information that we need from section 6.2 is the following:

- The LIS3DSH acts as a slave device (i.e. it can't initiate communication), thus we need to set the STM32F407 as the master and use it to initiate communication and allow the slave to be controlled through software.
- There are 2 data lines allowing full-duplex communication.
- The data is sent in 8-bit packets (plus 8-bits for the device address) - see figure 8.
- We also see from figure 8 and the text below that the clock remains high during idle and the data line is sampled on the rising edge of the clock line, which is the second clock transition (the first is 1 to 0 and then the second is 0 to 1) after idle.
- We also see from figure 8 and the text below that the address/data is sent most significant bit (MSB) first.
- Finally in table 5 in section 3.3.1 we find the timings for the SPI interfaces. This tells us that the clock needs to operate at a maximum frequency of 10MHz.

STEP 3 - Setting up the SPI interface on the STM32F407

In this step you will need to refer to the STM32F4 processor reference manual

Now that we have these parameters for the LIS3DSH SPI interface let's take a look at the control registers on the STM32F407. Section 28.5.1 (page 903) of the processor reference manual describes CR1 for the SPI interfaces. We see that this register contains most of the settings required to configure the parameters given in the bullet points in the previous step. For example, full-duplex communication is configured by setting bit 15 (BIDIMODE) to 0 - i.e. 2-lines with one sending and one receiving information. As with GPIO ports previously, to configure the SPI interface we can set the bits in the control register as desired. However, we can simplify this by using the data structures and functions provided by the drivers for the STM32F407 SPI interface (we did something similar for the PLL in the introductory lab in Appendix A task A2).

First we need to add the SPI interface to the list of components that we will use in our Keil project. Add the SPI option now using the "Manage Run-Time Environment" window (refer to the Introductory lab sheet if needed). The SPI option can be found in the same list as the timers (TIM).

Next we need to add the line of code given below, right at the start of "main.c" on line 1. This line of code tells the compiler that we are going to use some of the standard pre-defined data types. The rest of our code will go in the main() function in the "main.c" file.

```
#include "stm32f4xx.h"
```

Information on how to use the SPI drivers is contained at the start of the driver file "stm32f4xx_hal_spi.c". The first point in the how to use this driver section says that you should declare a handle structure of type SPI_HandleTypeDef. This is the first line of code that goes in our main() function and is shown below. To configure the SPI we store all of the parameters that we need in the structure `SPI_Params` and then pass this to the initialisation function which uses this information to configure the appropriate options in the control registers.

```
SPI_HandleTypeDef SPI_Params; // Declares the structure handle for the
parameters of SPI1
```

The second line of code to add is to enable the clock that SPI1 uses. By reference to the processor user manual we find that this is the APB2 clock that we configured in the introductory lab. The line of code given below is needed.

```
// Code to initialise the SPI
RCC->APB2ENR |= RCC_APB2ENR_SPI1EN; // Enables the clock for SPI1
```

The next lines of code store the required parameters into the structure `SPI_Params` that we have just defined. The lines of code to do this are given below. The first line defines which SPI interface we will use (i.e. SPI1) and is stored in the sub-structure "Instance". The remaining lines store the specific parameters identified in the bullet points in step 2 into the sub-structure "Init". The sub-structures for the available initialisation parameters are defined on or around lines 65-104 in the driver header file "stm32f4xx_hal_spi.h". The parameter values are then defined from line 181 onwards and these are cross-referenced to values defined in the main processor header file "stm32f407xx.h" on line 5781 onwards.

```
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
```

We need to set one final parameter in `SPI_params`. This is the baude rate prescaler that determines the SPI clock from the main APB2 clock (see the processor reference manual definition for SPI_CR1 for the available prescalar values). In the introductory lab we have set the APB2 clock to 84MHz. To achieve the desired maximum SPI clock of 10 MHz we need to set the prescalar appropriately. We will select 32 to give a clock rate of 84/32 = 2.625MHz.

```
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.
```

Now we write these to the control registers using the driver function `HAL_SPI_Init()` which is defined in the driver file “stm32f4xx_hal_spi.c”. This function saves us the task of manually setting the individual bits in the associated SPI control registers. The required code is provided below. The `&` symbols means that `SPI_Params` is a pointer to the `SPI_Params` structure.

```
HAL_SPI_Init(&SPI_Params); // Configures the SPI using the specified
parameters
```

STEP 4 - Configure the GPIOA pins associated with the SPI

In this step you will need to refer to the STM32F4 processor reference manual

In addition to configuring the SPI we need to configure the GPIOA pins that the SPI uses. In step one we identified these as GPIOA pins 5-7. The second column of table 5 in the STM32F4Discovery user manual tells us that the SPI is mapped to the alternate function mode of GPIOA. So we need to place GPIOA pins 5-7 in alternate function mode. As with the SPI settings and unlike both the Introductory and first assessed lab, we will use the built in parameter defines and functions provided by the GPIO diver to set GPIOA pins 5-7 into the required mode. You can find these definitions in the GPIO ".c" and ".h" driver files in a similar manner to the equivalent SPI functions. First we need to declare a handle structure of the correct type for the GPIO. In this case the correct data type is `GPIO_InitTypeDef` and we place the declaration near the start of the main() function and immediately below the declaration for `SPI_Params`. Remember that all declarations must come at the start of the main() function and before any executable code. The required declaration is given below.

```
GPIO_InitTypeDef GPIOA_Params; // Declares the structure handle for the parameters
of GPIOA
```

The following code can go after the code used to initialise the SPI. First we turn on the clock associated with GPIOA. Next we specify which pins we want to configure (i.e. pins 5, 6 and 7). Next we specify the alternate function that we want to configure. We see from figure 26 in section 8.3.2 of the processor reference manual that SPI1 corresponds to alternate function 5 (i.e. AF5). We then have three lines of code that set some specific options for the alternate function mode and relate to how the voltage is signalled at the pins. The final line of code uses the driver function `HAL_GPIO_Init` to apply the parameters specified in `GPIOA_Params` to `GPIOA` (again note the use of the pointer `&` for the structure handle).

```
// 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
```

STEP 5 - Configure the GPIOE pins associated with the SPI

In this step you will need to refer to the STM32F4 processor reference manual and the LIS3DSH data sheet.

Here we repeat the process from step 4 and configure GPIOE pin 3 to be the serial port enable pin. PE3 needs to be set to normal push-pull output mode. The code required to do this is given below.

`GPIOE_Params` should be defined near the start of main() and the remaining code can go after the code to configure GPIOA.

```
GPIO_InitTypeDef GPIOE_Params; // Declares the structure handle for the parameters
of GPIOE

// 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
```

We now just need to set pin 3 of GPIOE (the CS line of the LIS3DSH) so that it is in the idle state (i.e. high - refer to figure 6 and the text below it in the LIS3DSH data sheet) and then enable the SPI. Following this the SPI is ready to be used. Pin 3 of GPIOE is set in the same way as we have previously done for the LED's (first line below). The SPI is enabled by setting on bit 6 in the SPI_CR1 register. We can use the macro `__HAL_SPI_ENABLE` defined in the SPI driver and the code to do this is given in the second line below.

```
GPIOE->BSRR |= GPIO_PIN_3; //Sets the serial port enable pin CS high (idle)
__HAL_SPI_ENABLE(&SPI_Params); //Enable the SPI
```

STEP 6 - Test the SPI communication

In this step you will need to refer to the STM32F4 processor reference manual and the LIS3DSH data sheet.

Our final step is to test the SPI communication port. We see in the LIS3DSH data sheet in section 8 that the accelerometer has a number of internal registers. Some of these are used to set it into particular modes, some to monitor its operation and some are data registers containing the results of the acceleration measurements. To test the SPI we are interested in the register in section 8.3. This is a "who am I" register and can act as an identifier. We can test the SPI by reading the contents of the register. Its address is given in hexadecimal as 0Fh in the data sheet, which is identical to the notation 0x0F that we are using in our code. If the correct value is returned, i.e. 00111111_2 , then the SPI is setup correctly. In our program we will indicate if the correct value is returned by turning on the green LED, otherwise we turn on the red LED.

Before we write the code to read from the "who am I" register, initialise the green and red LED's using the code below. This can go after the previous initialisation code.

```
// 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
```

Section 28.3.3 of the processor reference manual describes how to send and receive data using the SPI. However, as with the initialisation code we will use the pre-defined functions provided by the drivers. The drivers provide functions to transmit and to receive data. When we read data from the LIS3DSH we first need to transmit the register address that we want to read and then then receive the information. Therefore we will need both transmit and receive functions. The driver also provides a combined function to transmit/receive, but we will focus on the separate functions. The two functions are `HAL_SPI_Transmit` and `HAL_SPI_Receive`, which are defined on lines 324-463 and 466-638 respectively of the driver file "stm32f4xx_hal_spi.c". Both functions need four input arguments:

- Information on the port (this information is in the SPI structure handle that we have already defined),
- The address of the register that we want to access (this is stored in an array that could contain multiple address for multiple register accesses),
- The amount of data to be sent
- The maximum time to wait for the communication to complete.

The first of these arguments we have already defined. However we now need to declare three additional variables for the remaining three. In addition, we want to declare a variable that will store the contents of the "who am I" register that the LIS3DSH returns by the SPI. The code to declare these variables is given below and should be placed with your other declarations at the start of main().

```
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
```

The next lines of code will go after all of the port and LED initialisation code. It could go in an infinite FOR or WHILE loop so that it is repeatedly executed as in the previous labs, but this isn't strictly necessary here as we just want to read the value in the "Who am I" register and then signal the result. To use the SPI we need to specify the address on the LISDSH that we want to access. We assign this information to the "data_to_send" array (note that the first element of the array is accessed using the index 0 - see the first line of code below). We have previously identified this address as 0x0f. However it is important to note from section 6.2 of the LIS3DSH data sheet that the address spans the lower 6 bits of this value. If we want to read an address, we must set the MSB to 1 and to write to an address we set it to 0. In this case we want to read the address so the MSB (bit 7) is set to 1 using the code `0x80|0x0f` (if we want to write to an address then we set the MSB to 0). The next step is to set the SPI enable communication line CS (GPIOE pin 3) low to initiate communication. This is done in the second line below. We then transit the address using the `HAL_SPI_Transmit` function. The `HAL_SPI_Transmit` function then requires us to specify a blank address when we receive data. This is done in the next two lines of code and the received data is stored in the receive buffer sub-structure in our SPI handle structure `SPI_Params`. We then end the communication by setting the SPI enable communication line CS high and finally read the received value into our `Who_am_I` variable from the SPI handle structure. Note the use of star here when reading the data in `SPI_Params`. This indicates that `SPI_Params` is a pointer as opposed to a standard variable.

```
// 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.
```

The final few lines of code given below use an IF statement to compare the received value of the "who am I" register to its known value 0x3F (00111111_2 in binary). After you compile and download your program to the STM32F4Discovery, if the received value is correct the green LED turns on and you know that your SPI communication with the LIS3DSH is working correctly. If the red LED turn on then you have not received the correct value from the LIS3DSH and have made a mistake. If the red

LED turns on you should go back through each step and try to identify any errors before seeking help from the teaching team.

```
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
}
```

The complete code is provided in the “Serial Communication, Polling and Interrupts - Complete code and Task Solutions” document.

Task A.1 - Changing the serial port configuration parameters

The LIS3DSH requires specific settings for the SPI. Other external devices are likely to require different SPI settings. With this in mind, how would you change the configuration code so that the STM32F407 now uses 16-bit data packets and requires a maximum SPI clock of 2MHz (all other settings stay the same).

A solution to this task is provided in the "Serial Communication, Polling and Interrupts - Complete code and Task Solutions" document.

Note – If you try to implement these parameters for the LIS3DSH the SPI won't work. The tasks purpose is just to get you thinking about configuration parameters and consulting the data sheet.

Task A.2 - Other addresses on the LIS3DSH

What is the address of control register 1 of the LIS3DSH and what value should it contain if everything is at default values apart from the bit the enables state machine 1, which is set to enable state machine 1?

A solution to this task is provided in the "Serial Communication, Polling and Interrupts - Complete code and Task Solutions" document.

Task A.3 – Writing to the control registers

Now that you have code that successfully communicates with the LIS3DSH, you can use it to do something more interesting than just read the "who am I" register. In this first exercise you will modify the code so that it sets new values to control register 4 of the LIS3DSH (described in section 8.5 of the LIS3DSH data sheet). This register controls the output data rate, method for data update and enables/disables each of the three measurements axis. Inspection of the default value for the output data rate shows that it is set to power down mode (bits 7-4 set to 0000_2). Therefore, we need to set the output data rate in this register to be able to use the accelerometer. We want to set the accelerometer so that it uses an output data rate of 3.125Hz, continuous update and only the z-axis is enabled. This means that we need a register value of 00010100_2 (the output data rate in bits 7-4 is set to 0001_2, the data update in bit 3 is set to 0_2 and the three measurement axes in bits 2-0 are set to 100_2).

In the "who am I" example we wrote code to receive a value from the accelerometer, which required the register address to be sent using the `HAL_SPI_Transmit` function and then the register value received using the `HAL_SPI_Received` function. **Here we want to send a new value to the register, so you will need to modify the code so that first the address for control register 4 is sent using the `HAL_SPI_Transmit` function and then second, the new register value is sent also using the `HAL_SPI_Transmit` function (i.e. we don't use the `HAL_SPI_Receive` function when writing new values to the LIS3DSH).** The control register is read/write, so when you have done this you will read the value of control register 4 back from the LIS3DSH in the same way as was previously done for the "who am I" register. You should then use the IF statement to indicate if the register value is correct (green LED) or incorrect (red value).

In summary, you will need to do the following:

1. *Identify the address for control register 4 from the LIS3DSH data sheet*
2. *Insert a communication routine using `HAL_SPI_Transmit` to write the new value (00010100_2) to control register 4. This code should be inserted after the code to setup the ports and LED's and before the code to read from the "who am I" register". Note the MSB in the address is the read/write bit and needs to be set as discussed in the "who am I" code description in Part A - step 6.*
3. *Modify the code to read from the "who am I" register and the IF statement so that it is now the current value of control register 4 that is read and signalled as correct/incorrect using the LED's.*

When complete compile and download your program to test/observe its operation.

A solution to this task is provided in the "Serial Communication, Polling and Interrupts - Complete code and Task Solutions" document.

Task A.4 – Getting and decoding data from a sensor using serial communication

In this task we will read some acceleration data from the LIS3DSH and create a simple tilt switch. This will require you to make several modifications described below.

1. Take your program from Task A.3 and insert the variable declaration given below in the usual place near the start of main() – all declarations must appear before any executable code.

```
uint8_t Z_Reg_H; //Declares the variable to store the z-axis MS 8-bits in
```

2. Next, **delete the code that reads the current value of control register 4 and signals the result on the LEDs, i.e. your main() function finishes after setting control register 4 into the modes required in Task A.3, i.e. points 1 and 2 in the requirements for exercise 1 are implemented, but not point 3.**

   Then insert the code given below in place of the code that you have just deleted. From reference to both the LIS3DSH data sheet, you will see that two 8-bit data registers (the high and low measurement registers) a required to be loaded in and combined if you need the full 16-bit signed acceleration measurement. The code below will continuously read the current value in the z-axis high data register (this register contains the upper 8-bits of the 16-bit value). The z-axis is orientated vertically up out of the plane of the LIS3DSH (figure 2 in the LIS3DSH data sheet). The acceleration is a signed 16-bit number, thus the most significant bit is the sign bit. In the code the sign of the acceleration measurement is checked and the green LED is turned on (red LED off) if it is negative and the red LED is turned on (green LED is off) if it is positive. Please refer to the C Programming guide or Bits – Bytes - Logic Functions (Negative numbers) document if you are unclear about signed/negative numbers.

   Your task is to complete the code by adding the missing parts indicated in orange.

```
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] = A - ADD THE CORRECT READ/WRITE BIT AND REGISTER ADDRESS;
// 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
B - ADD THE CORRECT CODE// 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
C - ADD THE CORRECT CODE// Get the data from the LIS3DSH through the SPI channel
D - ADD THE CORRECT CODE; // 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 acceleration value is positive or negative - the acceleration is a signed 16-bit number so the MSB of the upper 8-bits (register H) is the sign bit: 1 is negative, 0 is positive. Refer to the C Programming guide or Bits – Bytes - Logic Functions (Negative numbers) document if you are unclear about this.
GPIOD->BSRR = (1<<12); // If the received value is negative turn on the green LED
GPIOD->BSRR = (1<<(14+16)); // If the received 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
}

}
```

When complete compile and download your program to test/observe its operation. If your code is correct the red LED should be on when the board is placed normally on a horizontal surface. If you rotate the board upside down you will see that the green LED comes on when the board has been rotated through at least 90 degrees.

A solution to this task is provided in the "Serial Communication, Polling and Interrupts - Complete code and Task Solutions" document.

Part B - Defining and using functions in programs

In previous projects all of the code that you have written has been in the “main.c” file. This can make the “main.c” file long, particularly if you need to re-use the same piece of code, such as transmitting/receiving data using the SPI, at numerous points in your program. However, when you have setup projects you have created “my_header.c” and “my_header.h” files in addition to “main.c”, but haven’t as yet used these other two files. Now you will uses these files to hold sections of code, in the form of functions, that can be called in the “main.c” file when you need them. Using functions instead of putting all of our code “inline” can potentially slow down a program when it runs, but it can improve readability of the code. Some compilers provide options that allow functions to be placed inline when code is compiled and thus allows this reduction in speed to be overcome.

Part B takes you through step-by-step instructions for creating and using functions (make sure that you create a fresh project for part B). However, you might find is useful to refer to the “Embedded Software in C for ARM Cortex M” programming guide. The contents of all of the files required in part B are included in the “Serial Communication, Polling and Interrupts - Complete code and Task Solutions” document.

Part B.1 - Basic Functions

You will now modify the simple code from the Introductory lab that blinks the green LED on and off. If we review the code that we wrote to do this (re-produced below) it should become clear that it can be split into two distinct parts. The first is the part that initialises the GPIO for the green LED (the code in red) and the second part is the code that blinks the LED on and off (the code marked in blue).

```
#include "stm32f4xx.h"

int main(void){

	uint32_t ii

	// Initialize GPIO Port for LEDs
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable GPIOD clock
	GPIOD->MODER |= GPIO_MODER_MODER12_0; // GPIOD pin 12 output

	for(;;){

		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

	}

}
```

We will modify this code to declare and define a function `Initialise_LED` that will contain the code in red and a function `Blink_LED` that will contain the code in blue. We could do this in the "main.c" file, but if we have a large number of functions, this doesn't necessarily aid readability. Instead we will use the "my_headers" files that we have previously created.

Function Declarations

First we need to declare the required function types in the "my_headers.h" file. The function declarations specify the name and input/output parameters of the functions that we are going to create. The declarations that need to be added to the "my_headers.h" file are provided below. In both declarations there is no data that is provided or returned from the function so the input/output parameters are defined as "void", i.e. empty.

```
void Initialise_LED(void); // Declaration for the function to initialise
the LED
void Blink_LED(void); // Declaration for the function to blink the LED
```

Function Definitions

The next step is to define the functions in terms of their operation. This code goes in the "my_headers.c" file. To do this we add the same code as in the previous declarations, but in addition add the executable code in { } brackets after the function name. The required definitions are provided below. Firstly, note that we need to include the header for the system parameter

definitions in this .c file and not "main.c", as it is now this file where we use them (other programmes may also need to include the header file in "main.c"). Secondly, note that in the definition for `Blink_LED` the variable `ii` is defined within the function. This is because it is internal and only used by the function `Blink_LED`.

```
#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

}
```

Function Calls

Finally, we modify our "main.c" file to replace the executable code with the defined functions and to include the "my_headers.h" file at the start so that the compiler knows where to find the functions that we have defined. The "main.c" file now looks as below.

```
#include "my_headers.h"

int main(void){

Initialise_LED(); // Initialises the LED

	for(;;){

		Blink_LED(); // Blinks the green LED on and off once

	}

}
```

When compiled and downloaded to the Discovery board this program should operate in an identical way as the main code developed in the Introductory lab.

Part B.2 - Functions with Input/Output Parameters

With these two example functions there are no input/output parameters. However, many functions are likely to have input and/or output parameters that are passed to the function and used internally within the function, or returned from the function to be used by other parts of the code. We will now modify this program so that the `Blink_LED` function only blinks the LED on or off. The desired on or off is specified on the input argument of the function.

In the following, the declaration, definition and use of `Initialise_LED` remains the same.

Declaration

For "my_headers.h" we include the system header "stm32f4xx.h" and change the declaration for `Blink_LED` to include an input argument as follows.

```
#include "stm32f4xx.h"
void Blink_LED(uint8_t); // Declaration for the function to blink the LED
```

Definition

For "my_headers.c" we change the definition for `Blink_LED` to include an input argument as follows. `uint8_t` is the data type and `LED_state` is the internal variable name.

```
// 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

}
```

Call

Finally we first change the “main.c” file to include parameter definitions for LED on and off. Then second we change it so that `Blink_LED` is called twice, once to turn on the LED and once to turn off the LED. The new version of “main.c” is provided below.

```
#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 - Function declaration, definition and call

In the example on function declaration, definition and call that we have just completed we could use a further function for a delay instead of using the same FOR loop twice. Your task is to modify this code so that a single function to implement a delay is declared, defined and called in “main.c” to implement the required delay. The delay function should have an input that specifies the required delay in seconds. You should then use this delay function so that the LED turns on for four seconds and turns off for one second. As in the example above you can use the FOR loop for the delay in this task.

When complete, compile and download your program to test/observe its operation.

A solution to this task is provided in the “Serial Communication, Polling and Interrupts - Complete code and Task Solutions” document.

Part C – Polling & Interrupts

In Task A.4 the program repeatedly communicates with the LIS3DSH each time the FOR loop is executed. This is an example of Polling and is a very simple way to initiate and control the timing of a communication routine. However, it does have some draw backs which can be clearly illustrated with this example.

Each time the FOR loop is executed the microcontroller communicates directly with the LIS3DSH by sending the register address values of the z-axis data registers and receiving the contents of the data registers. For this embedded system, each iteration of this FOR loop takes about 252µs. In contrast, we have set the LIS3DSH to have a sample rate of 3.125Hz, which means that a new measurement appears in the data register every 0.32 seconds. So in this case the code reads the same data measurement over 1200 times. In addition, during each iteration of the FOR loop most of the clock cycles (about 99%) are spent completing the communication with the LIS3DSH and thus a lot of clock cycles are wasted completing unnecessary communication routines.

This is an example of an inefficient implementation of polling as a timing control mechanism. It could be improved by adding a delay so that the FOR loop doesn't execute as often. However, if the delay is too long this can lead to problems with synchronisation and could risk samples being missed. Despite these problems, polling a device can be beneficial in some situations if it is implemented well, such as if an embedded system need to perform in a predictable manner.

There is an alternative mechanism that can be used to control the timing of communication. This mechanism is Interrupts. With interrupts a device can signal to another device if it wants to initiate communication. Both the STM32F407 and LIS3DSH support interrupts. In this case we can configure the embedded system so that the LISD3DSH sends a signal to the STM32F407 when a new acceleration measurement is made and the contents of the data registers are updated. This allows us to design an efficient embedded system that only initiates communication when new data is available, which means that in-between these times it can do other tasks. In addition, it also provides a mechanism through which communication that occurs at irregular times can be effectively accommodated. A restriction of interrupts, is that it requires the devices involved to have the required hardware built in. However, it is widely used and many devices support interrupts. Please refer to lecture 4 for further background on polling and interrupts.

You will now modify the code that you wrote in Task B.4 to use interrupts. With this new program communication will only be initiated when new data appears in the LIS3DSH data register. In part A when we identified which pins on the STM32F407 that the LISD3DSH was connected to, in addition to the four main serial communication pins, GPIOE pins 0 and 1 of the STM32F407 were connected to the interrupt request lines of the LIS3DSH. When setup correctly, it is on these lines that the LIS3DSH will signal to the SMT32F407 that new data has been placed in the measurement data registers. We therefore need to:

STEP 1 - Determine which of these two interrupt request lines to use

In Table 2 on page 11 of the LIS3DSH data sheet we see that pin 11 of the LIS3DSH is the INT1/DRDY pin. This is the same INT 1 that is connected to GPIOE pin 0 of the STM32F407. DRDY stands for "Data Ready" and thus it is this interrupt line that is associated with a new data measurement being placed in the data registers.

STEP 2 – Configure the related pin of GPIOE

Now that we have identified INT 1 as the required interrupt line, we can configure the required GPIO pin on the STM32F407, i.e. GPIOE pin 0.

First add the declaration for the structure handle to store the configuration data for GPIOE pin 0 to the usual part of the code at the start of main ().

```
GPIO_InitTypeDef GPIOE_Params_I; // Declares the structure handle for the parameters of the interrupt pin on GPIOE
```

Next add the following lines of code that specify the parameters and sets up GPIOE pin 0. This code should go immediately after the code to initialise pin 3 of GPIOE and before the code that enables the SPI interface. Note, that you do not need to enable the clock for GPIOE as you have previously done this when configuring GPIOE pin 3.

```
//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
```

STEP 3 – Configure the LIS3DSH and enable the interrupt line

The next thing to do is configure the interrupt line on the LIS3DSH. The register that does this is control register 3 defined in section 8. 4 (page 32) of the LIS3DSH data sheet. From Table 20 and 21 we see that:

- Bit 7 (DR_EN) needs to be set to 1 to connect the data ready signal to INT1
- Bit 6 (IEA) needs to be set to 1 to configure the interrupt line to be high when an interrupt is signalling
- Bit 3 (INT1_EN) needs to be set to 1 to enable interrupt 1 (note the corrected typo in the data sheet here).

The code to do this should come after the code to set configuration register 4, but before the infinite FOR loop in your solution from exercise 3. The code required is provided below.

```
// 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
```

STEP 4 – Modify the code that reads the data registers so that it uses an interrupt routine

Now that we have configured both the STM32F407 and the LIS3DSH to use interrupts, we need to modify our infinite FOR loop to only communicate with the LIS3DSH when it signals to the STM32F407 on the interrupt request line. A simple way to do this is to introduce an IF statement at the start of the FOR loop. This IF statement checks the status of the interrupt request line (GPIOE pin 0) and only proceeds if the interrupt request line is set. All of the code that implements the SPI communication then goes inside this IF statement. In addition, after the interrupt line is detected as "set on" we also need to reset the interrupt request line so that it can be used again. This is done in the first line inside the IF statement.

To check and reset the interrupt we can use the macros provide by the drivers. The two macros we need are defined on lines 204-218 of the header file "stm32f4xx_hal_gpio.h". With these macros we just need to specify the GPIO pin that the interrupt is connected to. We don't need to specify which port, since the STM32F407 has a limited number of external interrupt request lines. The infinite FOR loop containing the code to implement an interrupt request routine to create the basic tilt switch from Task A.4 is given below. You should replace your existing FOR loop from Task A.4 with this new FOR loop.

```
// 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] = ADD THE CODE THAT YOU WROTE IN TASK A.4; // Address for control
        register 4 on LIS3DSH
        GPIOE->BSRR = GPIO_PIN_3<<16; // Set the SPI communication enable line low to initiate
        communication
        ADD THE CODE THAT YOU WROTE IN TASK A.4 // 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
        ADD THE CODE THAT YOU WROTE IN TASK A.4 // Get the data from the LIS3DSH through the
        SPI channel
        ADD THE CODE THAT YOU WROTE IN TASK A.4 // 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 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.
```

```
}
}
```

If you compile and download this code, visually it should operate the same as the code from Task A.4. The full code is provided in the "Serial Communication, Polling and Interrupts - Complete code and Task Solutions" document.

Task C.1 – Visually showing the signalling of interrupts

To show the operation of the interrupts, your task is to modify this code so that in addition to signalling the tilt angle using the green and red LED's, you use the orange and blue LED's to alternately signal when the interrupt signal is detected, i.e. the orange LED is set on when the code initialises the when the interrupt is detected for the first time the orange LED turns off and the blue LED turns on, the next time the interrupt is detected the blue LED turns off and the orange LED turns on. This pattern should repeat and the switching frequency of the LED's should be 3.125Hz due to the sample rate of the accelerometer (each LED should stay on for 0.32 seconds). To do this, you will need to:

A. *Add some code to initialise the orange and blue LED's.*
B. *Add some code to turn on the orange LED after the system initialises. This code should be placed just before the main FOR loop and after the code to setup the LIS3DSH (its location is indicated in orange just above the FOR loop in the code in step 4).*
C. *Add some code at the start of the IF statement to implement the alternate orange/blue LED switching to indicate that the interrupt has been detected (this location is also indicated in orange in the code in step 4).*

When complete compile and download your program to test/observe its operation. If it is correct the orange and blue LED's should come on alternately at a rate of 3.125Hz. This shows that the STM32F407 only communicates with the LIS3DSH when the interrupt is signalled. To test this you can modify your code to change the LIS3DSH sample rate in control register 4 to 6.25Hz (refer to the LIS3DSH data sheet to determine the required register value). The effect will be that the frequency of this LED switching doubles since the interrupt is signalled twice as often.

A solution to this task is provided in the "Serial Communication, Polling and Interrupts - Complete code and Task Solutions" document.

IN THE FINAL LAB

So far we have written programs that consist of a single piece of executable code that we place in the main() function and which repeatedly executes from start to finish (a super loop). In more complex embedded systems this is not a particularly attractive approach and becomes inefficient when dealing with inputs from a large number of external devices, each with different communication timings. A much more flexible approach and one which can allow us to design a system to meet a range of real-time constraints, is to us a real-time operating system that manages the execution of tasks (discrete sections of code designed to do specific operations). In the next lab you will learn how to use a real-time operating system and use it to re-design some of the programs that you have previously developed.