Assessed Laboratory 2 - Serial Communication, Polling and Interrupts

This lab contributes 18% towards ACS335 and 15% towards ACS6335

Aim

There are two aims for this second assessed lab.

1. To further extend your knowledge/skills in serial communication and standard communication protocols.
2. 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 exercises. Some exercises will require you to write complete code, while others will require you to modify/complete some code and then use it.

At the end of each exercise 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: 1pm Friday 11th November

Date of assessed lab sessions: (The members of each group are listed on MOLE)

Group 1: Friday 18th November 1100-1300 (you will have 1h40m (1110-1250) to complete the lab)

Group 2: Friday 18th November 1300-1500 (you will have 1h40m (1310-1450) to complete the lab)

Marks awarded: This lab is marked out of 15 and the mark for each exercise is clearly indicated.

Learning, Assessment and Feedback process

You will have 1 week from the release time/date of this lab sheet until the lab session associated with this lab sheet. Please use this time to prepare and attempt the exercises in this lab sheet. 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.

Assessment: There are two marked exercises:

Exercise 1 should be completed before the assessed lab session. You should submit your code to the Turn-it-In link on MOLE before the start of the assessed lab session (DEADLINE Friday 18th November 1100).

Exercise 4 will be provided to you during the assessed lab session. During the two hours of the assessed lab you will be required to develop code to solve the problem posed in the exercise. You should arrive at the start of your allocated lab sessions and spend the first 10 minutes preparing (i.e. opening Keil and starting a new project). You will have 1h40m to complete the exercise. The final 10 minutes should be used to upload the contents of any code that you have developed to the Turn-it-In link on MOLE (details will be provided in the exercise).

The assessed lab will be conducted under "exam conditions" - i.e. No verbal or other communication (such as phones or email) will be allowed. Use of web browsers is not permitted.

Written feedback will be provided within 2 weeks and a general feedback video posted on MOLE the Monday following the lab sessions.

Unfair Means

The lab should be completed individually. Any suspicions of the use of unfair means will be investigated and may lead to penalties. See http://www.shef.ac.uk/ssid/exams/plagiarism for more information.

Special and Extenuating Circumstances

If you have medical or personal circumstances which cause you to be unable to complete this lab on time or that may have affected your performance, please complete and submit a special circumstances form along with documentary evidence of the circumstances. See http://www.shef.ac.uk/ssid/forms/special, particularly noting point 6 (Medical Circumstances affecting Examinations/Assessment).

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.

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

Initialisation - Setup a project in Keil µVision

To start this lab you will first need to create a project in Keil µVision and setup the same serial IO port that you used in part A of the "serial IO and functions" lab. Connect the STM32F4Discovery to the PC and follow steps 1 - 6 in part A in the "serial IO and functions" lab sheet. You will need the same components of the STM32F4Discovery that you used in this and the introductory lab. So you should select the same components in the "Manage Run-Time Environment" window in step 3 of the introductory lab.

Please make sure that you use the same clock and PLL settings as in the main Introductory lab (i.e. you should use a main clock speed of 168MHz, APB1 clock of 42MHz and APB2 clock of 84MHz).

The exercises start on the next page. There are four exercises and each exercise starts on a new page.

Exercise 1 - Serial communication **3 marks**

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.

Please upload the contents of your fully and clearly commented main.c file (and any personal header files that you have used, such as my_header.c or my_header.h) in a .doc file (copy and paste as editable text - DO NOT PASTE AS AN IMAGE) to MOLE using the correct link and by the deadline listed on the cover sheet.

Exercise 2 - Getting and decoding data from a sensor using serial communication **Not Assessed**

This exercise is not assessed, but the use of data from the accelerometer is required in Exercise 4. **No additional support will be provided for this exercise (I will not respond to emails/queries relating to it.** The answers to the questions are provided at the end of the exercise.

In this exercise 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 exercise 1 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 exercise 1, 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 blue (0.5 mark is awarded for each correct missing part).

```
for(;;){

// Read the value from the MSB (H) z-axis data register of the LIS3DSH
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 (this will be demonstrated in Lecture 4).

After your solution to exercise 2 is marked the assessor will correct any mistakes that you have made so that your solutions to exercises 3-4 are not penalised due to any mistakes made in this exercise.

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
```

Exercise 3 - Polling & Interrupts **Not Assessed**

This exercise is not assessed, but the use of interrupts is required to achieve a good mark in Exercise 4. It is broken down into a step by step guide and the LIS3DSH interrupts are covered in Lecture 4 - **No additional support will be provided for this exercise (I will not respond to emails/queries relating to it.** The answers to the questions are provided at the end of the exercise.

In exercise 2 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 3 for further background on polling and interrupts.

You will now modify the code that you wrote in exercise 2 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:

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.

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
```

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
```

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 exercise 2 is given below. You should replace your existing FOR loop from exercise 2 with this new FOR loop.

```
// 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

        // ADD SOME CODE TO IMPLEMENT THE ORANGE/BLUE LED SWITCHING DESCRIBED BELOW (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 EX2; // 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 EX2 // 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 EX2  // Get the data from the LIS3DSH through the SPI
        channel
        ADD THE CODE THAT YOU WROTE IN EX2 // 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.

}
}
```

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 blue just above the FOR loop in the code above).*
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 blue in the code above).*

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.

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
}
```

Exercise 4 – This exercise will be provided in the assessed lab session **12 marks**

In this exercise your task will be to create a new project and use the techniques that you have learnt in exercises 1-3 and in the earlier labs to implement a solution for a given system design.

In preparation you should create a new project for exercise 4 (following the guidelines given in the initialisation section on page 2 of this lab). This will allow you to make maximum use of the 1h40m assessment period.

You will be allowed to view the documentation, previous labs and your previous work when conducting this lab. However, you will not be allowed to use a web-browser. Please make sure that you have downloaded all relevant information that you will need before the lab.

You can either bring your own laptop, or use the managed desktop.

Please remember to bring your STM32F4Discovery kit to the lab session.

IN THE NEXT 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.