ESS: Lab 3

Inputs

Task 1:

Accelerometer Sensing

One of the main requirements of the project is the ability to measure the shocks and impacts that the pallet has undergone on its journey. **R1** requires that we be able to measure vibration, **R2** that we measure impact and **R3** that we measure tilt. The STM32 discovery board has an accelerometer chip, which physically is the small chip in the middle of the four LEDs in the centre of the board. This is an LIS3DH device, which is a 3 axis accelerometer. The ess_helper contains basic code to communicate with the accelerometer over SPI. The API of the SPI exposes the following functions:

```
// Initialize the SPI
void SPIAcc_Init(void);

// Send a byte on SPI
// @param address address of register to write to
// @param databuffer to send
// @param send size
void SPIAcc_Send(uint8_t address, uint8_t* databuffer, uint8_t size);

// Receive a byte on SPI
// @param address address of register to read from
// @param databuffer to send
// @param read size
void SPIAcc_Get(uint8_t address, uint8_t* databuffer, uint8_t size);
```

The second parameter of the `SPIAcc_Send` and `SPIAcc_Get` function is the address of a `uint8_t` array. And the third parameter of the two functions is the number of bytes to write/read. For example, we can use the following code piece to read 2 bytes from address A. The value read will be saved in the first two bytes of the buffer (`buffer[0]` and `buffer[1]`).

```
uint8_t buffer[4];
SPIAcc_Get(A, buffer, 2);
```

Once we have initialized the peripheral, we can read and write bytes in particular registers (addresses) in the accelerometer chip itself. Each register has an address and is one byte wide. At the end of this document are two relevant pages from the datasheet. We will use the register map to program the sensor.

(a) Initialize the SPI peripheral. The first thing to do to check that we can actually communicate with the device is to read a byte from the `WHO_AM_I` register, which should return the default value `0x3F`. With the aid of the register map, work out the hex address of the register and use the supplied functions to read this byte. Use the debugger or red/green LEDs to demonstrate that this first test passes.

Solution:

```
uint8_t who_am_i;
uint8_t buffer[4];
SPIAcc_Init();
SPIAcc_Get(0x0F, buffer, 1);
who_am_i = buffer[0];
```

(b) Once the previous test passes, we now need to turn the device on and make it sample data. For simplicity, we will just use the fastest sampling rate. We need to write `0x87` to the control register 1 `CTRL_REG1`.

Solution:

```
uint8_t who_am_i;
uint8_t buffer[4];
SPIAcc_Init();
SPIAcc_Get(0x0F, buffer, 1);
who_am_i = buffer[0];
buffer[0] = 0x87;
SPIAcc_Send(0x20, buffer, 1);
```

(c) We can now sample data from the registers. Start by measuring the x axis acceleration. The high byte is found in register `OUT_X_H` and the low byte in register `OUT_X_L`. We read two `uint8_t` which we combine to make a single 16 bit signed integer `int16_t`. We can do this with the following:

```
int16_t x_axis = (data_h<<8)+data_l;
```

With the aid of the debugger check that you can read data from the x axis of the accelerometer and it changes as you tilt the device.

Solution:

```
uint8_t who_am_i;
uint8_t buffer[4];
SPIAcc_Init();
SPIAcc_Get(0x0F, buffer, 1);
who_am_i = buffer[0];
buffer[0] = 0x87;
SPIAcc_Send(0x20, buffer, 1);
while(1){
    SPIAcc_Get(0x28, buffer, 2);
    datax_l = buffer[0];
    datax_h = buffer[1];
    datax = (datax_h<<8)+datax_l;
```

```
        delay_usec(100);
    }
```

(d) Use the LEDs (red and green) to show whether the tilt is positive or negative - show this working in real-time.

Solution:

```
    uint8_t who_am_i;
    uint8_t buffer[4];
    SPIAcc_Init();
    SPIAcc_Get(0x0F, buffer, 1);
    who_am_i = buffer[0];
    buffer[0] = 0x87;
    SPIAcc_Send(0x20, buffer, 1);
    while(1){
        SPIAcc_Get(0x28, buffer, 2);
        datax_l = buffer[0];
        datax_h = buffer[1];
        datax = (datax_h<<8)+datax_l;
        if (datax < 0)
        {
            led_on(&led_green);
            led_off(&led_red);
        }
        else
        {
            led_on(&led_red);
            led_off(&led_green);
        }
        delay_usec(100);
    }
```

(e) Further modify to use the PWM driver to fade the LEDs smoothly as you tilt the device. When the device is horizontal, no LEDs should be on. As the device is tilted, the respective LED corresponding to the angle of tilt should illuminate more and more brightly. When the angle to the horizontal is more than 45○, it should be fully illuminated.

(f) Extend your solution to read the y axis reading as well and use this to control the blue and orange LEDs in the same way. You should now have a device which shows in real-time how you are tilting the device.

(g) Add code to read the z axis as well and populate a struct with the following fields:

```
    struct acc3
    {
```

```
  int16_t x;
  int16_t y;
  int16_t z;
};
typedef struct acc3 acc3_t
```

(h) Refactor your code to have an accelerometer driver with the following API:

```
// Initialize accelerometer
void AccInit(void);
// Obtain a reading
void AccRead(acc3_t * reading);
```

Do you think that returning nothing from the `AccInit()` function makes sense?

Solution:
We can read the `WHO_AM_I` register and check whether or not the sensor is attached and working. If it fails this basic test, we should signify to the calling function that there was a failure.

(i) Rather than using a delay, use hardware timer 3 to generate an interrupt (look at Lab 2 to modify code for TIM4) that samples the accelerometer at a rate of 32 Hz. Store the readings in a buffer that can contain half a second of data.

(j) Write a function that averages the tilt over the half-second window, displaying it on the LEDs. You will need to use a flag to tell the main loop that the buffer is ready. This will satisfy one of the main requirements of the project, and will demonstrate progress to the customer. What do you need to be careful of?

Solution:
We need to be careful that the producer (the ISR) does not change or modify values halfway through calculating the average. There are a number of ways of achieving this. One of the simplest is to use a buffer that is twice as big as necessary. Whilst the ISR is writing to the bottom buffer, the average is computed on the top buffer. They then swap, so they are never reading and writing from the same buffer at the same time. In this implementation, we tell the main loop (the consumer) that the buffer is ready and also whether it is the lower or upper buffer that should be used. Another approach is to disable interrupts or use a mutex. It must be noted though that in this example, it is unlikely that there will be concurrent access, as the buffer is only updated every 30 msec and our computation is simple.

Task 2:

Signal Processing

Now we have the ability to sense accelerometer data, we can do some processing on it to meet **R1** and **R2**.

(a) Write a function that will take in a buffer of accelerometer data and check whether there have been any impacts.

> **Solution:**
> Instead of averaging (which is a low-pass filter), we want to find the spikes in the data (which is a high-pass filter). There are lots of ways of doing this, one of the simplest is to compute the sample-by-sample magnitude ($x^2 + y^2 + z^2$) of the accelerometer data and check whether it exceeds a threshold.

(b) Write a function that will take in a buffer of accelerometer data and calculate the average vibration levels.

> **Solution:**
> The mean vibration can also be computed in many ways. A useful start is to compute the variance or standard deviation of the buffer, which is the sum of the deviations from the mean. Approximations to this can also be made by taking the sum of the sample-by-sample differences, which is a first order derivative.

(c) Using these three functions (tilt, vibration and impact), come up with a struct that represents the state of the system at a particular point in time. Write a fictious logger API that could take in these samples and store them (this will become useful once the hardware team produces the first prototype, which has a microSD card for logging).

Task 3:

⋆ **Temperature Sensing**

The STM32F4 has an internal temperature sensor attached to ADC channel 16. To satisfy **R4**, we will measure this and display whether it exceeds a particular value. We have found some sample code, but it needs to be refactored. To set up the reading:

```
#include "stm32f4xx_hal_adc.h"
  /**********************************************************
    * This enables the A/D converter for sampling the on board
    * temperature sensor.
    * ******************************************************/
ADC_HandleTypeDef hadc1;
ADC_ChannelConfTypeDef sConfig;
hadc1.Instance = ADC1;
hadc1.Init.Resolution = ADC_RESOLUTION_12B;
hadc1.Init.ScanConvMode = DISABLE;
hadc1.Init.ContinuousConvMode = ENABLE;
hadc1.Init.DiscontinuousConvMode = DISABLE;
hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
```

```c
hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
hadc1.Init.NbrOfConversion = 1;
hadc1.Init.DMAContinuousRequests = DISABLE;
hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
HAL_ADC_Init(&hadc1);
// Enable clocks to ADC1
__HAL_RCC_ADC1_CLK_ENABLE();
// ADC1 Configuration, ADC_Channel_TempSensor is actual channel 16
sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
sConfig.Rank = 1;
sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
HAL_ADC_ConfigChannel(&hadc1, &sConfig);
```

To acquire a reading:

```c
uint32_t temp_value;
HAL_ADC_Start(&hadc1);//Start the conversion
HAL_ADC_PollForConversion(&hadc1, 100);//Processing the conversion
temp_value = HAL_ADC_GetValue(&hadc1);//Return the converted data
HAL_ADC_Stop(&hadc1);//Stop the conversion
```

(a) Test that this code actually works and that you get (scaled) readings from the temperature sensor. These are typically around 1000 at room temperature. To do this, look at the `temp_value` in the watch window.

(b) Refactor this code so that it is modular and clean.

(c) The temperature sensor returns a 12 bit reading, where 4095 corresponds to 3.3V. According to the datasheet, the temperature in °C can be calculated as:
$Temp = \frac{(V_{sense} - V_{25})}{Slope} + 25$
where:
V_{sense} is the actual converted voltage
V_{25} = 0.76V
$Slope$ = 2.5 mV/°C
Write a function (using floats/doubles) that returns an approximately calibrated version of the temperature in °C [hint, you might need to put your board in the freezer].

(d) What are the advantages and disadvantages of using the on-chip sensor?

Solution:
Advantages:

- No extra cost.
- No extra board space.

Disadvantages:

- Accuracy/precision may not be good enough.

- Sensors might need to be individually calibrated.
- Microcontroller might not be physically in the best place to measure the ambient temperature.

(e) ⋆ Write a fixed point function that achieves a similar result. What are the advantages and disadvantages of the fixed point version?

Solution:
Advantages:

- The fixed point version is portable to devices without hardware floating point units
- The fixed point version is faster on devices without hardware floating point units

Disadvantages:

- The fixed point version needs careful range scaling
- The fixed point version should only really be used if needed

(f) Use an interrupt timer (e.g. Timer 3) to sample the temperature sensor at a rate of 10 Hz. Average these results over a 1 second window to reduce the noise. How are you going to prevent concurrency problems?

LIS3DH

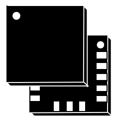
MEMS digital output motion sensor ultra low-power high performance 3-axes “nano” accelerometer

Features

- Wide supply voltage, 1.71 V to 3.6 V
- Independent IOs supply (1.8 V) and supply voltage compatible
- Ultra low-power mode consumption down to 2 μA
- ±2*g*/±4*g*/±8*g*/±16*g* dynamically selectable full-scale
- I^2C/SPI digital output interface
- 16 bit data output
- 2 independent programmable interrupt generators for free-fall and motion detection
- 6D/4D orientation detection
- Free-fall detection
- Motion detection
- Embedded temperature sensor
- Embedded self-test
- Embedded 96 levels of 16 bit data output FIFO
- 10000 *g* high shock survivability
- ECOPACK® RoHS and “Green” compliant

Applications

- Motion activated functions
- Free-fall detection
- Click/double click recognition
- Intelligent power saving for handheld devices
- Pedometer
- Display orientation
- Gaming and virtual reality input devices
- Impact recognition and logging
- Vibration monitoring and compensation



LGA-16 (3x3x1 mm)

Description

The LIS3DH is an ultra low-power high performance three axes linear accelerometer belonging to the “nano” family, with digital I^2C/SPI serial interface standard output. The device features ultra low-power operational modes that allow advanced power saving and smart embedded functions.

The LIS3DH has dynamically user selectable full scales of ±2*g*/±4*g*/±8*g*/±16*g* and it is capable of measuring accelerations with output data rates from 1 Hz to 5 kHz. The self-test capability allows the user to check the functioning of the sensor in the final application. The device may be configured to generate interrupt signals by two independent inertial wake-up/free-fall events as well as by the position of the device itself. Thresholds and timing of interrupt generators are programmable by the end user on the fly. The LIS3DH has an integrated 32-level first in, first out (FIFO) buffer allowing the user to store data for host processor intervention reduction. The LIS3DH is available in small thin plastic land grid array package (LGA) and it is guaranteed to operate over an extended temperature range from -40 °C to +85 °C.

Table 1. Device summary

Order codes	Temp. range [°C]	Package	Packaging
LIS3DH	-40 to +85	LGA-16	Tray
LIS3DHTR	-40 to +85	LGA-16	Tape and reel

7 Register mapping

The table given below provides a listing of the 8 bit registers embedded in the device and the related addresses:

Table 17. Register address map

Name	Type	Register address		Default	Comment
		Hex	Binary		
Reserved (do not modify)		00 - 06			Reserved
STATUS_REG_AUX	r	07	000 0111		
OUT_ADC1_L	r	08	000 1000	output	
OUT_ADC1_H	r	09	000 1001	output	
OUT_ADC2_L	r	0A	000 1010	output	
OUT_ADC2_H	r	0B	000 1011	output	
OUT_ADC3_L	r	0C	000 1100	output	
OUT_ADC3_H	r	0D	000 1101	output	
INT_COUNTER_REG	r	0E	000 1110		
WHO_AM_I	r	0F	000 1111	00111111	Dummy register
Reserved (do not modify)		10 - 1E			Reserved
TEMP_CFG_REG	rw	1F	001 1111		
CTRL_REG1	rw	20	010 0000	00000111	
CTRL_REG2	rw	21	010 0001	00000000	
CTRL_REG3	rw	22	010 0010	00000000	
CTRL_REG4	rw	23	010 0011	00000000	
CTRL_REG5	rw	24	010 0100	00000000	
CTRL_REG6	rw	25	010 0101	00000000	
REFERENCE	rw	26	010 0110	00000000	
STATUS_REG2	r	27	010 0111	00000000	
OUT_X_L	r	28	010 1000	output	
OUT_X_H	r	29	010 1001	output	
OUT_Y_L	r	2A	010 1010	output	
OUT_Y_H	r	2B	010 1011	output	
OUT_Z_L	r	2C	010 1100	output	
OUT_Z_H	r	2D	010 1101	output	
FIFO_CTRL_REG	rw	2E	010 1110	00000000	
FIFO_SRC_REG	r	2F	010 1111		
INT1_CFG	rw	30	011 0000	00000000	