18-847: Wireless Software Systems Architecture

Lab 2:
Serial Communication between devices

Introduction

MEMS or Micro Electro-Mechanical Systems is a term used to describe devices such as sensors actuators that consist of components usually in the range of 1 to 100 micrometers. Along with the small size, MEMS devices offer the advantages of low cost, low power consumption and mass production. As a result, they have found profound usage in a variety of application domains.

For IoT and wireless systems, the advantages offered by MEMS devices are particularly useful. In this lab, we will use a MEMS based magnetic sensor and make it interact with the PowerDue. For embedded systems, serial buses are the preferred mode of communication between devices. Two of the most widely used protocols are I2C and SPI. The lab focuses on understanding how they work, and the energy implications of both protocols.

The objectives of this lab are:

- Writing drivers for interfacing sensors with embedded processors
- Managing I/O on an embedded device using FreeRTOS
- Observing the I2C and SPI protocol transactions
- Learning how to read sensor datasheets
- Learning how to build and use device libraries
- Observing the energy per useful bit transmitted for both protocols

Part 1: Interfacing sensors using I2C

In this part, we will interface the PowerDue to the LSM303C magnetometer using the I2C protocol, which is part of the SparkFun LSM303C IMU (Inertial Measurement Unit) breakout board.

Resources:

For this part of the lab, you will need:

1. The PowerDue connected to the SparkFun LSM303C breakout board on the right bench.
2. LSM303C Arduino library
3. LSM303C datasheet

Setting up the environment:

We will write all our code on the Arduino IDE. The LSM303C hookup guide shows how to install Arduino libraries for the sensor. We recommended you to go through the entire page to understand the hardware connections that have been set up for the lab.

Modifying the Arduino library to work with I2C:

Your task is to download the LSM303C library and modify it to work with I2C on the PowerDue instead of an Arduino. The PowerDue uses the Arduino Wire library for I2C communication. As the PowerDue is based on the Arduino Due, it has two I2C interfaces, which are controlled with the Wire and Wire1 objects. For our lab, we are using the Wire1 interface instead of the Wire interface.

Running an example:

Once you have the library set up, write a program in FreeRTOS which consists of two threads: one to read the data from the magnetometer using the modified library, and the other to print the values to the serial port.

Configuring the IMU:

For our experiments, we will use the same setting as the *ConfigureExample.ino* example that is provided in the library itself. Once you start getting the readings from the sensor, answer the following questions:

- What is the default I2C clock speed that the library uses?
- What is the maximum clock speed that can be used with the LSM303C?
- At what frequency should you read data from the magnetometer?
- How many bytes do you need to send for every read? How many of these bytes are actual data?
- Illustrate the I2C transaction sequence in your lab report

Observe the transaction on an oscilloscope:

Use the oscilloscope in the lab to observe the I2C transaction.

- Does the I2C sequence on the oscilloscope match the sequence that you predicted?
- Take screenshots of the output of the oscilloscope and identify the different parts of the sequence in your lab report.

Implement a WHO_AM_I function:

Most devices have a device ID that is stored in a register on the device. This register can be read to get the device ID, and serves as a great way to verify if the device interfaces are correctly configured. Unfortunately, this function is missing from the Sparkfun library.

Add a WHO_AM_I function to the Sparkfun library which reads the value from the WHO_AM_I register of the magnetometer. The function should compare the read value to the default WHO_AM_I value of the magnetometer and return a IMU_SUCCESS or IMU_HW_ERROR based on whether the results match or not. You should run the WHO_AM_I check in your setup function. The address and default value of the WHO_AM_I register can be found in the datasheet.

NOTE: You do not need to print or read the accelerometer data as is done in the *ConfigureExample.ino* example. Only the magnetometer data is required.

Part 2: Interfacing sensors using SPI

In this part, we will interface the PowerDue to the FXOS8700CQ magnetometer using the SPI protocol, which is part of the BRKT-STBC-AGM01 breakout board. Unlike Part 1 of the lab, there was no existing Arduino library for this magnetometer, so we will start from scratch to develop one.

Resources:

For this part of the lab, you will need:

1. The PowerDue connected to the BRKT-STBC-AGM01 breakout board on the left bench.
2. SPITemplate.zip: Containing function templates from Canvas
3. FXOS8700CQ datasheet
4. The PowerDue SPI library: It uses the same functions as the Arduino SPI library.

Setting up the environment:

We will write our code in the Arduino IDE. Download and extract SPITemplate.zip from Canvas, and put the folder in your Arduino sketch folder. Load the code in the Arduino IDE.

The code consists of the following files:

1. MagSPITemplate.ino: The Arduino sketch file that will run the code
2. FXOS8700CQ.h: Header file for FXOS8700CQ library. Contains the sensor's register addresses, parameter information and function prototypes
3. FXOS8700CQ.cpp: Contains function definitions for the magnetometer library
4. spi_trans.h: Header file for SPI driver
5. spi_trans.cpp: Function definitions for the SPI driver

Requirements:

For implementing this driver, we will need to configure the sensor according to the requirements listed below:

● Sensor mode:	Magnetometer Only Mode
● Magnetometer output data rate (ODR) :	100 Hz
● Magnetometer oversampling rate (OSR) :	5

Writing the sensor library and SPI driver:

Your task is to complete the function templates that have been provided to you in the files FXOS8700CQ.cpp and spi_trans.cpp. For this task, you will need to use the mini-datasheet. Read the function descriptions given in the code for helpful hints for the task of every function.

SPI consists of a 4-wire bus, which have the following lines:

- SPI Clock
- MOSI (Master-Out Slave-In)

- MISO (Master-In Slave-Out)
- Chip Select

Among these four, the Powerdue SPI library can configure the clock, MISO and MOSI pins. We will configure the Chip Select pin using the GPIO pin 51 on the PowerDue. The wiring for the pin has been done for you, so you can directly control it through software.

In addition, the master must also configure the clock polarity (CPOL) and phase(CPHA) with respect to the data. On the basis of these values, SPI can work in 4 modes:

Mode	**CPOL**	**CPHA**
0	0	0
1	0	1
2	1	0
3	1	1

In addition, SPI can transfer serial data in the order of most significant bit (MSB) first or least significant bit (LSB) first.

- What is the SPI mode in which the sensor operates?
- What is the order of data transfer for the sensor?

You will need to initialize the PowerDue SPI library with parameters that will work with our sensor. Once you have the SPI bus working, answer the following questions in your lab report

- How many bytes do you need to send for every read? How many of these bytes are actual data?
- Illustrate the SPI transaction sequence in your lab report

NOTE: By default, the SPI library sets up the bus in Mode 0 and order MSB first.

NOTE: Important registers for the lab:

- WHO_AM_I
- CTRL_REG1
- M_CTRL_REG1
- M_OUT_X_MSB, M_OUT_X_LSB
- M_OUT_Y_MSB, M_OUT_Y_LSB
- M_OUT_Z_MSB, M_OUT_Z_LSB

Observe the transaction on an oscilloscope

Use the oscilloscope in the lab to observe the SPI transaction.

- Can you find out the SPI clock speed from the oscilloscope?
- Does the SPI sequence on the oscilloscope match the sequence that you predicted?
- Take screenshots of the output of the oscilloscope and identify the different parts of the sequence in your lab report.

Part 3:

While designing wireless systems, it is essential that we transfer the data from the sensor to the processor and vice versa in the least amount of time possible. Our wireless systems will be running on a small battery, so it is important to minimize the cost of communication between the peripheral and the processor. In the first two parts, we learned how to do serial communication between devices using SPI and I2C. However, which one of the two serial protocols would you use when designing a wireless system? To answer that question, we will do an energy consumption comparison by the processor for both the protocols. To do this, we will make use of the instrumentation port of the PowerDue.

Resources:

For this part of the lab, you will need:

- Your code from Parts 1 and 2
- The PowerDue Scope
- The PowerDue Scope writeup
- The PowerDue setups in the lab

Setting up the environment:

For calculating the energy consumption, we will need to observe the power consumption by the processor. For this task, we can use the instrumentation port of the PowerDue.

A small writeup on the principles used by the PowerDue instrumentation port is detailed here. The PowerDue Scope is a tool that can be installed on your systems that we will use collect the samples from the instrumentation processor of the PowerDue. Details for setting up and using the Scope as listed in the writeup.

IMP NOTE: It is essential to know which Serial COM ports are used by the instrumentation port and the target port. Be cautious and ensure that you flash your code to the correct port.

Measuring the power:

We now need to measure the power consumed by a I2C and SPI transaction. The PowerDue instrumentation port can only read a sample after every 40 us. Given that we are using clock speeds of 400 KHz and 4 MHz for I2C and SPI respectively, the board will not be able to capture a single transaction. As a result, we will need to do multiple consecutive reads to actually be able to see the transactions happening on the Instrumentation port.

1. Modify the programs you wrote in the first two parts on the PowerDue to periodically read any register 10,000 times. It is essential here to remove any print statements, as well as any

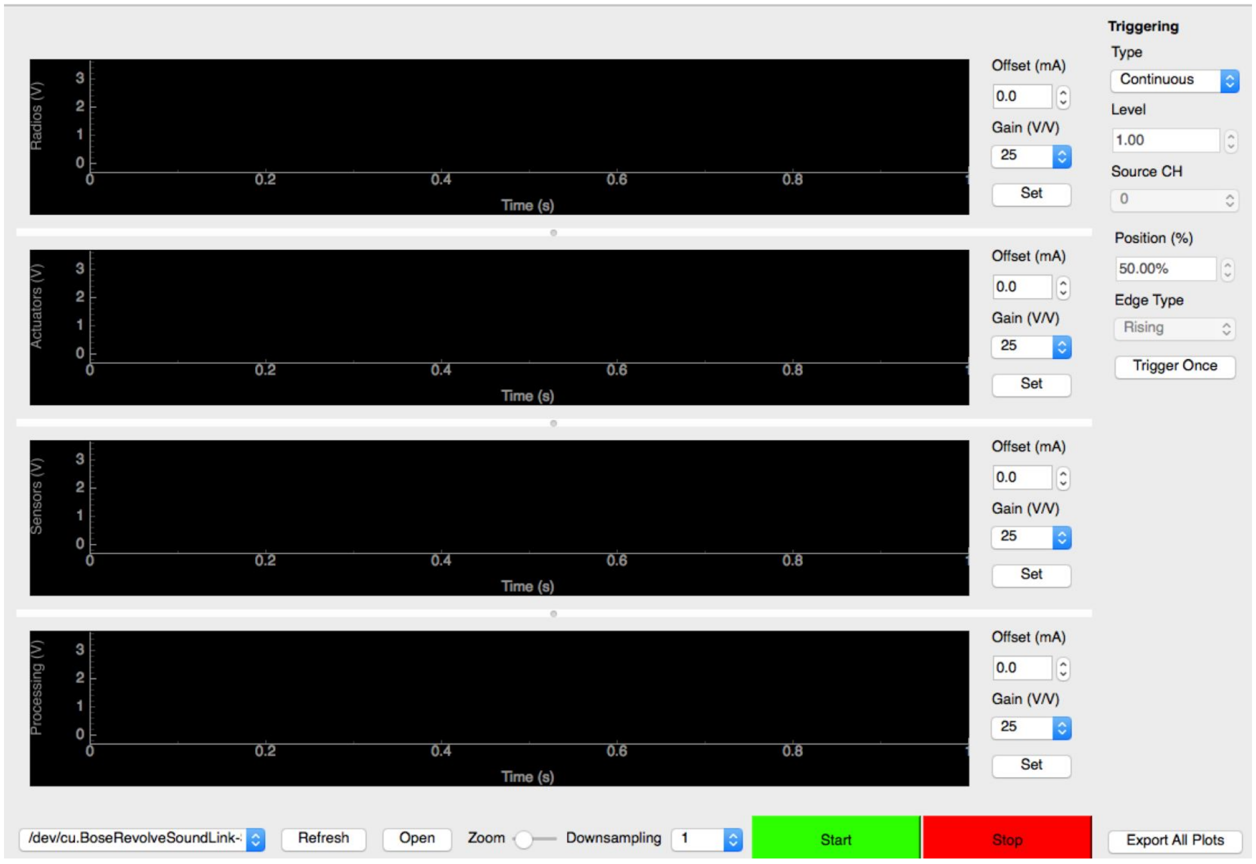
code which is not required because it will add error to our calculations by spending processor cycles on unnecessary tasks.

2. Flash the programs to the respective PowerDues for I2C and SPI

Do the next parts one by one for each protocol

3. Hook up your system to the PowerDue's instrumentation port and start the PowerDue Scope application on your system
4. We will be interested in the processing channel to find out how much time and energy the processor has to use for the transfer of data from the sensor to the processor.

Can you make out when the transaction is happening by observing the processing channel? The processor is always awake in our code and will show constant activity on the processing channel. As a result, it will be hard to know when the transaction starts and ends. For fixing this, we can light an LED before starting a transaction, and turn it off when the transaction is over. This activity can be seen on the actuator channel of the PowerDue. So we can find the starting time *t_start* and ending time *t_end* for the 10000 consecutive reads.



Radios: Offset (mA) 0.0; Gain (V/V) 25
Actuators: Offset (mA) 0.0; Gain (V/V) 25
Sensors: Offset (mA) 0.0; Gain (V/V) 25
Processing: Offset (mA) 0.0; Gain (V/V) 25

Type: Continuous
Level: 1.00
Source CH: 0
Position (%): 50.00%
Edge Type: Rising

/dev/cu.BoseRevolveSoundLink-
Downsampling: 1

5. Based on the starting and ending times, estimate the average power consumed by the processor for one loop of 10000 reads.

6. Estimate the total energy consumed for the entire loop, and estimate the energy per transaction.
7. How many bits of useful information are you getting per transaction?
8. Find the energy per useful bit for a single transaction.

Show your calculations for both I2C and SPI in the lab report. Can you answer the following questions?

- Which one of SPI and I2C is more efficient for energy transfer? Why?
- Can you estimate the amount of error that is there in our calculations? (Hint: Check the oscilloscope)
- Is there a way with I2C and SPI to further minimize the energy consumption when reading data?