18-877 WSSA Lab 3
Serial Communications on Embedded Devices

Reese Grimsley Carnegie Mellon University November 2, 2019

***Abstract*—The purpose of this lab is to explore the usage and characteristics of serial communication protocols such as I^{2}C and SPI, which are two of the most commonly used protocols in embedded systems. Two additional components of this lab are 1) Learning to read datasheets and 2) Writing device drivers, both of which are essential to embedded systems development. In the final part of this lab, we also look to the power consumption of embedded devices as we compare the energy profiles of I^{2}C and SPI.**

I. Introduction

This lab is designed to explore serial communications. Truly, this lab is a substantially deeper dive into embedded systems than the previous lab, in which we familiarized ourselves with Arduino and FreeRTOS. Software for embedded systems is unlike application or back-end programming, in that the developer must be intimately aware of the hardware's capabilities to get anything useful accomplished. This is not to say that hardware knowledge is unnecessary to other programming environments; however, it is unavoidable when working directly with microcontrollers (although environments like Arduino and mbed shield the programmer from many low-level details).

Since embedded systems are usually build to interact with the physical environment, it would follow that some tasks are external to the processor. One must communicate with sensors and actuators to obtain data about and act upon the environment. Serial data protocols are the means of communication whenever high bandwidth is not absolutely necessary, in which case parallel data protocols would be more appropriate. In this exercise, the two protocols of interest are Inter-Integrated Circuit (I^{2}C), a.k.a. the Two Wire interface (TWI), and Serial Peripheral Interface (SPI). In short, TWI is used to refer to I^{2}C whenever the publisher does not wish to deal with licensing issues since Philips patented I^{2}C decades ago.

I^{2}C was created as a standard within Philips for transmitting data between integrated circuits on the same board. This protocol uses just two wires between the devices: a clock (SCL) and a data (SDA) line. These lines are shared between the processor or "master" and multiple other devices or "slaves". Each device listens on the bus whenever the master starts transmitting, which will start by sending a 7 bit address corresponding to the device of interest. These addresses are hard-coded. Unfortunately, $2^7 = 128$ options makes collisions a reason for concern; solved in a similar way to the birthday problem, it takes 13 devices to make a collision near 50% likely! This makes it especially difficult to use multiple of the same device on a board if there is no physical pin that can be used to select the address. For this reason, Philips updated the specification to allow 10-bit addresses and faster speeds than

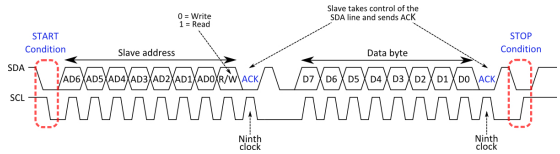

Fig. 1: Illustration of I^{2}C communication

the typical 400kHz; however there are physical limitations that necessitate lower rates (which are beyond the scope of this class and lab report). Generally, the master controls the clock, and both the master and slave share the data line for half-duplex communication. View Figure 1 for a diagram of this transaction[1]. In short, different sequences of raising clock and data lines begin or stop the transaction, and data is written for a longer period than a clock's active pulse, during which data is read.

The SPI communication protocol is a less standardized protocol that sacrifices simplicity by adding wires, but adds higher data rates and well-behaved physical conditions. This protocol has three essential lines: 1) the clock signal (SCK) provided by the master, 2) Master-out-slave-in (MOSI), and 3) Master-in-slave-out (MISO). In addition, a "chip select" (CS) line is provided for each device on the bus. This last bit does not scale well, unfortunately, but it allows much simpler physical modeling of the system because each end device simply disconnected itself from the data bus unless its CS line is held low. For this reason, the bus can be run much faster, with typical speeds of 10 MHz, but sometimes going as high as 25 MHz. An example transaction[2] is shown in Figure 2. Once the chip select goes low, the master begins sending clock pulses along SCK and data along MOSI. The exact data and behavior is device dependent, as SPI is more conceptual than provisional. For the device used in this lab, the first byte includes a write/read bit and the low 7 bits of a device addresses, the second byte contains the highest bit of the address, and then the third byte contains the register to read or write from. The following byte is data, sent over MOSI or MISO depending on the read write bit. Just like the data structure, the logical operations tied to the clock are not standardized. There are two binary options for the clock: polarity and phase. Polarity effectively refers to the inactive state of the clock, where zero polarity means it is low in the inactive state whereas polarity of one means it is logic high in the inactive state. Clock phase determines the logical operation performed on the rising and falling edge. These logical operations are simply whether data bits are set

[1]Image courtesy of https://www.artekit.eu/i2c-communication-protocol/
[2]Image courtesy of https://www.corelis.com/education/tutorials/spi-tutorial/

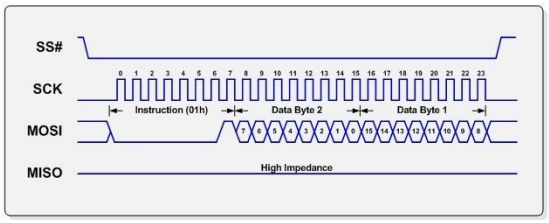

Fig. 2: Illustration of SPI communication

or read on the corresponding edge. The combination of these two clock characteristics define the SPI "mode" that needs to be accounted for for each device on the bus.

The remainder of this lab report is structured to follow the lab manual [1], with each of the three portions partitioned in the II section. These three portions are for the I^{2}C driver for the LSMOC303, the SPI driver for FXOS8700CQ, and power measurements for transactions in the two previous parts. At the request of the TA, the answers to questions from the manual are marked with a reference to the document and appropriate context. The supporting github repository for this lab is at the following link: https://github.com/reese-grimsley/wssa-lab3.

II. Approach

This lab is divided into three sections. The first focuses on communicating with a mostly-written driver for the SparkFun LSM303C IMU breakout board using I^{2}C . This is placed inside of FreeRTOS tasks for accessing the device and displaying the magnetometer values to the serial port. The second section is about writing a driver to facilitate SPI communication with the magnetometer on the FXOS8700CQ IMU, a part of the BRKT-STBC-AGM01 breakout board; this is similar to the previous section, except most of the lower level functions must be written. The final part of this lab focuses on power measurement on the PowerDué . The goal here is to compare power consumption for I2C vs. SPI.

A. I^{2}C Driver

For this portion of the lab, we were given a set of files for the LSM303C driver that need to be modified to work with the PowerDué . I found two primary things to change. The first was changing all *Wire* calls to *Wire1*; nothing a simple find and replace cannot fix. The second was to remove code that prevented compilation. There were a few defined values that references other defined values that are nonexistent in the files for PowerDué. I commented all of these out to resolve the compilation errors; as it turns out, this mostly has the effect of rendering SPI communication with the LSM303C impossible. There is no effect on I^{2}C.

We were also asked to add a few functions for referencing "Who-am-I" values, which are read-only values in the IC that can be used to ensure communication is goign through properly. With the existing functions in the driver, this was as simple as reading from the appropriate register.

The following list answers the questions within the lab manual [1], in the same order they appear:

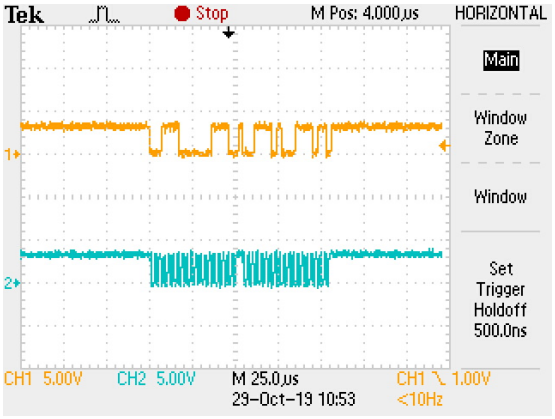

Fig. 3: I^{2}C Transaction on Oscilloscope

- The default I^{2}C speed appears to be 400kHz, though it also supports 100kHz. No true default is specified in the datasheet.
- The maximum clock speed is 400kHz.
- The speed to read the magnetometer at is dependent on the setting used for this. By default, this is 40Hz (so 25ms period). At max, it can be run at 80 Hz.
- For every transaction, there are two overhead bytes: the device address byte and the register address byte. Generally, there is only one byte of data, but multi-byte options exist as well.
- Please refer to Figure 1 for an illustration of a transaction.

Figure 3 displays the output of the oscilloscope for the Who-am-I function call. From my understanding, the master sends two bytes, which I can tell to be 0x1E and 0x0F. These are the device address and register address. The response is not what I would have expected, mainly because there is an additional byte present; regardless, I can see that the first byte reflects the expected value of 0x3E. The additional byte was not requested, but perhaps the other device sent an additional byte over the line. There is also a short delay between the first two bytes and the third, but this be explained as the source code for the read function shows that the transmission is ended and then restarted just before doing the actual read. In general, the transaction appears quite noisy. I had little expectation to see clean values here, especially for the logic level transitions, but I see more noise than expected at the logic levels themselves.

B. SPI Driver

For this portion of the lab, we were given the basic components for a SPI driver for the FXOS8700CQ . The files provided include function definitions for the basic requirements of a driver, such as read and write from registers on the device using SPI, configuring the device, and reading data values. These portion of the lab required more information from the datasheet because SPI has a few additional parameters to set that are device specific.

The first task was to write the SPI driver such that it would read and write correctly. This requires the clock speed and mode to be set. Based on the datasheet, this device can run at 10MHz, uses most-significant-bit first data ordering, and

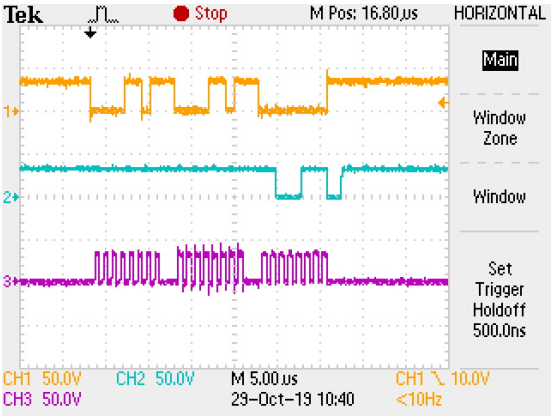

Fig. 4: SPI Transaction on Oscilloscope

requires SPI mode 0, which defines its clock polarity and phase. While not all SPI devices have this, the FXOS8700CQ has a who-am-I register just like the LSM303C does, making this the obvious method to test the connection.

Once the low-level reads and writes are working as expected, I implement the remaining functions for configuring things like sensor sampling rate, putting the device into standby or active mode, and reading magnetometer data from the device itself. It is important to set the device into standby mode before changing registers related to the operation of the sensor, such as sampling rate, because this could cause some instability (if the writes even go through). The device should be put back into active mode on the registers are set. I suppose that setting the device to standby could also have some power savings, but I did not test this hypothesis.

Similar to the previous subsection, the following list items answer the questions asked by the lab manual [1] in the order they appear in that document

- This sensor operates in SPI mode 0.
- The sensor expects data as MSB first.
- Two bytes of overhead are required for each send for this particular device. The device needs 9 bits of information (r/w bit and 8-bit address), which means 2 bytes must be sent before useful data is read/written.
- Please see Figure 2 for a representation of a SPI transaction.

Figure 4 shows the actual tranaction on the oscilloscope. After looking closely at this and seeing that the clock rate is just 1 MHz, I looked into my source code and realized that I had accidentally set this device to run at that speed of 1 MHz. The output matches what I expected to see more closely than the I^2C transaction did, and I would expect this to be due to the physical characteristics of the I^2C bus. In the diagram, there are clearly 3 set of clock pulses that correspond to the three bytes in this transaction. The first two contain the 9 bits of the r/w bit and 8 bit address, and then the last contains the returned value from the FXOS8700CQ of 0xC7. The data lines reflect what I would expect for the SPI bus, but I was not aware that there was (or could be) a gap in clock pulses between consecutive bytes.

TABLE I: Energy/Password Byte

	I^2C	SPI
Clock Speed (Hz)	400 k	1 M
Time to complete (s)	1.0799	0.3503
Total Energy/transaction (μJ)	14.961	4.4016
Energy/Useful bit ($\mu J/bit$)	4.4329	1.4672
Efficiency (%)	0.2963	0.3333

C.

D. Power Measurements for I^2C and SPI transactions

In this final part of the lab, we look to the power and energy consumption for these two serial communication methods. Since PowerDué has power measurement hardware built into the board, we can obtain these measurements directly from it via the instrumentation port. The caveat is that the sampling rate is 50kHz, which is well below the clock rates used by either communication bus. For this reason, we look at the overall power consumption by doing many reads over a known period of time to find the average power consumption and amount of time taken to perform these transactions. To perform these measurements, the programs from the previous two parts of the lab are modified to remove everything that is not strictly necessary for communication to work. Then, I read the Who-Am-I register 10,000 times in a loop; to show where these transactions begin, an LED is turned on such a synchronization signal exists in the 'actuation' channel of power measurements.

It must be noted that the measurements shown in the plots and saved CSV files are *not* the actual voltage readings on the device; they are the amplified voltage across a sensing resistor. This voltage can be converted to current based on Equation 1. This voltage is the value given in the plot; A is the amplifier gain of 25 and R_{sense} is the value of the sensing resistor, which is 1.33 Ohms for the processor channel [2].

$$I_{meas} = \frac{v_{meas}}{A * R_{sense}} \tag{1}$$

Table I shows information derived from PowerScope capture of 10,000 transactions for I^2C and SPI, and in turns answers many questions from the lab manual [1]. Time to complete refers to the time difference between the start and end of this entire block of transactions. I made a mistake with the SPI clock speed; it could have been run ten times faster, but it is still useful for looking at differences between SPI and I^2C . It may be the case that running 10 times faster would make the entire block of 10k transactions similarly 10 times faster. Let us assume this to be the case for approximation, but the true case is likely slightly less than a 10x speed-up due to frequency invariant overhead from processor instructions.

The measurements are calculated in Excel by calculating current for the processor based on the sensed voltage, shown in Equation 2. To get the energy, I average the power measurements and multiply by the sampling period of 20 μs. Efficiency is calculated as $\frac{\#useful\ bits}{\#total\ bits}$. Note that this does not account for time efficiency and thus energy efficiency w.r.t. bits; doing

this would require accounting for any timing delays between bytes, which we see in the oscilloscope waveforms. I believe this can be ignored, as it is highly likely that this is an artifact of the Arduino I^2C and SPI implementations. SPI has a higher efficiency because it does not use ACKs/NACKs to confirm the receipt of a byte. Even taking into account the efficiency of protocols, SPI is 2.74x more energy efficient that I^2C . It possible that this is due to the open-drain construction of I^2C that requires pull-up resistors, which will dissipate power as heat. Even discounting the 2.5x speed-up (which would easily be 25x), SPI is still measured to be 23% more efficient than I^2C , mainly due to the absence of ACKs and pull-up resistors.

$$P = I_{meas} * (3.3 - \frac{v_{meas}}{A}) \tag{2}$$

Without discounting speed differences, SPI can easily be an order of magnitude or two more efficient than I^2C . The concession here is that SPI requires more wires; a minimum of 4. Each individual device will require an additional line for a chip select. This can be avoided by using a daisy chained configuration, but this comes at the cost of additional overhead [3], thereby reducing efficiency. To reduce energy consumption during data reads, the best method would be to batch reads by reading from multiple data registers at once. This requires the registers of interest to be consecutive in the device's register map, but will allow the efficiency to tend towards 100% as the number of bytes read tends to infinity.

III. Conclusion

In this lab, I learned more about the I^2C and SPI serial communication protocols. I had existing background in this area, but I had not done oscilloscope and power analysis to compare these protocols; this was a particularly valuable outcome of this lab. As a result of this analysis, it is clear that SPI is substantially more efficient that I^2C . A great deal of this efficiency comes from the fact that all data wires are half-duplex rather than full-duplex, and also the fact that chip-select lines help prevent bus contention.

References

[1] S. V. Kalluru Srinivas, "Lab 3 manual." [Online]. Available: https://canvas.cmu.edu/courses/11109/files/folder/Labhandouts?preview=4033469

[2] B. Iannucci, "The CMU PowerDué." [Online]. Available: http://ccsg.ece.cmu.edu/wiki/doku.php?id=public:powerdue:home

[3] M. Integrated, "Daisy-Chaining SPI Devices." [Online]. Available: https://www.maximintegrated.com/en/design/technical-documents/app-notes/3/3947.html