EEC 172 Lab 2: Serial Interfacing Using SPI and I²C I Section: A03

Manprit Heer
912839204
mkheer@ucdavis.edu

Cathy Hsieh
912753571
cchsieh@ucdavis.edu

OBJECTIVE

In this lab, we utilize Serial Peripheral Interface, or SPI, to interface the CC3200 Launchpad to an OLED display. The application will further detect XY tilt functioning based on the BMA222 accelerometer's readings, and use the data to control he object's behavior on the OLED.

INTRODUCTION

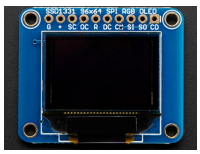
In the first part, we implement the SPI Demo to obtain a better understanding of how the SPI connection works, with a master-to-slave, and slave-to-master option. By downloading these two projects we are able to secure a connection and see the slave echo thee master commands from the PuTTy terminals. We then implement the OLED interface by using the SPI. Our test program must include a myriad of various functions that display various shapes and texts onto the screen, some of which include "Hello World!", a horizontal and vertical pattern of colors, as well as various shapes of different sizes. We verify the SPI waveforms by using a Saleae logic analyzer for exposure of the debugging tool and its interface.

In the second part, we construct an I2C connection and use a demo to secure a connection with the Bosch BMA222 accelerometer on the Launchpad. We then use our tools from part one to control the OLED display with the acceleration data coming from the accelerometer. We proceed to capture these waveforms to confirm thee proper readings from the x and y-axis. Although the z-axis information is available, it is omitted from functionality as we only need 2-D access to the movement of the board.

MATERIALS AND METHODS

The equipment needed for this lab are listed as follows:

1. CC3200 LaunchPad (CC3200-LAUNCHXL)
2. USB Micro-B plug to USB-A plug Cable
3. Adafruit OLED Breakout Board[1]
4. Saleae USB Logic Analyzer



PROCEDURE

Part I. Interfacing to the OLED using the Serial Peripheral Interface (SPI)

SPI_demo on two CC3200 Launchpads

The purpose of this demo is to familiarize us with the SPI connection between devices. We create two separate workspaces for the launchpads, one acting in MASTER_MODE while the other as a slave. MASTER_MODE set to 1 signifies master, and 0 signifies slave.

```
#define MASTER_MODE      1

#define SPI_IF_BIT_RATE  100000
#define TR_BUFF_SIZE     100

#define MASTER_MSG       "This is CC3200 SPI Master Application\n\r"
#define SLAVE_MSG        "This is CC3200 SPI Slave Application\n\r"
```

LAB II, PART I | SPI MASTER

```
#define MASTER_MODE      0

#define SPI_IF_BIT_RATE  100000
#define TR_BUFF_SIZE     100

#define MASTER_MSG       "This is CC3200 SPI Master Application\n\r"
#define SLAVE_MSG        "This is CC3200 SPI Slave Application\n\r"
```

LAB II, PART I | SPI SLAVE

OLED signal	LaunchPad interface
MOSI (SI)	SPI_DOUT (MOSI)
SCK (CL)	SPI_CLK (SCLK)
DC	GPIO
RESET (R)	GPIO
OLEDCS (OC)	GPIO
SDCS (SC)	n.c. (no connection)
MISO (SO)	n.c. (no connection)
CD	n.c. (no connection)
3V	n.c. (no connection)
Vin (+)	3.3V
GND (G)	GND

LP 1 Pin	LP 2 Pin	Signal Name
P1.7	P1.7	SCLK
P2.3	P2.3	CS
P2.6	P2.6	MOSI
P2.7	P2.7	MISO
P2.1	P2.1	GND

LAB II, PART I | SOFTWARE + HARDWARE CONFIGURATIONS AMONGST LAUNCHPADS

[1] https://www.adafruit.com/product/684?gclid=Cj0KCQiAsbrxBRDpARIsAAnnz_PATckp_fWf0mom4VPLEF7osaKOk78D4VWOKufqBx_EhoA3Zs9L6vQaAkC5EALw_wcB

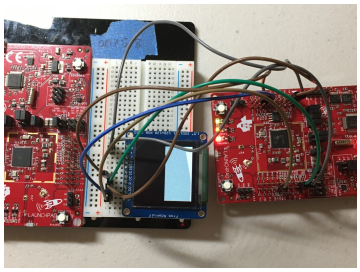
After setting up this connection, we continue to wire the two Launchpad boards according to the table given above.

To compile the two projects, we run the SPI master project first then the SPI slave to get the correct connection and output between the two launchpads. Once the two projects finished compiling, the SPI master terminal displays an instruction which then allows us to send messages to SPI slave through the terminal.

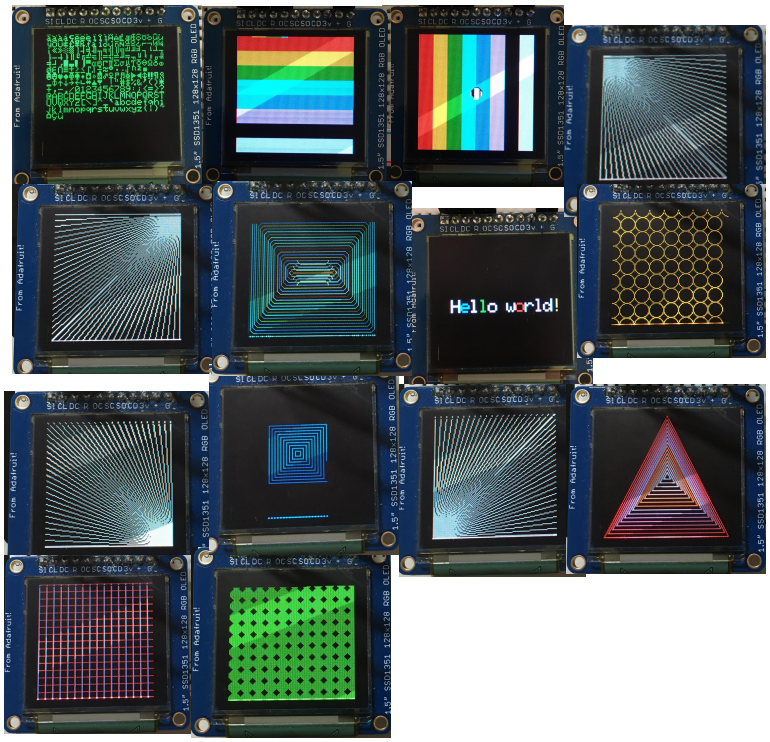
OLED interface using SPI

We will use this SPI connection, specifically in MODE 1 (master) to interface to the OLED. After choosing an unused GPIO signal as a Data/Command output, we proceed to connect up the signals below

After modifying WriteData() and WriteCommand() to work for the CC3200 LaunchPad, we run a test program using Adafruit's open source graphics library to ensure that the display was working properly.



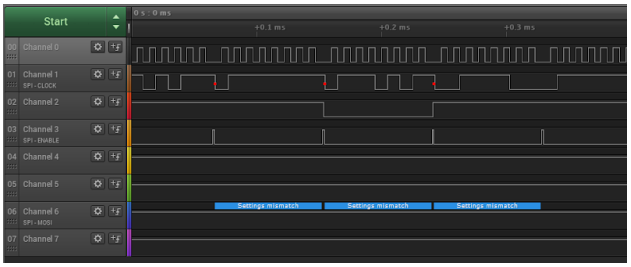
OLED signal	LaunchPad interface
MOSI (SI)	SPI_DOUT (MOSI)
SCK (CL)	SPI_CLK (SCLK)
DC	GPIO
RESET (R)	GPIO
OLEDCS (OC)	GPIO
SDCS (SC)	n.c. (no connection)
MISO (SO)	n.c. (no connection)
CD	n.c. (no connection)
3V	n.c. (no connection)
Vin (+)	3.3V
GND (G)	GND



LAB II, PART I | DEMO ROUTINE

Verifying SPI waveforms using Saleae Logic

After measuring various outputs and pins, we resulted in the chart below. Starting from the top channel, we recorded the clock, DC Enable, OC, and, MOSI respectively. The signals that we receive appear accurate as our clock, which is not supposed to change, looks symmetric and constant on the software. Also the DC channel is consistent from block-to-block which help us differentiate between the Data and Command Write function calls. The enable pin, then is on and off for long durations of time, which is exactly what happens with our OC, or chip select pin, and so it describes how the chip select is enabled for.



LAB II, PART I | SPI SIGNAL

A bug we encountered at this stage was that the Saleae Analyzer on our lab computer refused to recognize the device, and therefore gave garbage values and random spikes for each of the channels. We knew they were garbage values because we decided to take a sample after disconnecting the LaunchPad and we received the exact same stream of noise for the channels. To combat this, we started testing out different variables that could potentially hinder the connection. We tried using new wires, as well as different LaunchPads. The success happened when we finally connected our program with the analyzer on the computer next to ours. The second the "Start Simulation" button on the top left turned to "Start", we knew immediately that the Logic program picked up our device and we were ready to go.

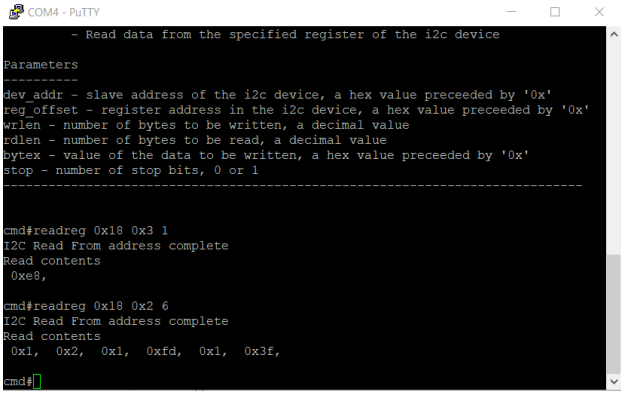
Part II. Implementing I2C to Communicate with BMA222 Accelerometer

The purpose of this part is to utilize the I2C connection and become familiar with its syntax and format while connecting it to the CC3200 LaunchPad format. We will also familiarize ourselves with the acceleration data and a way to connect it to the ball on the OLED.

Implement i2c_demo project on CC3200 LaunchPad

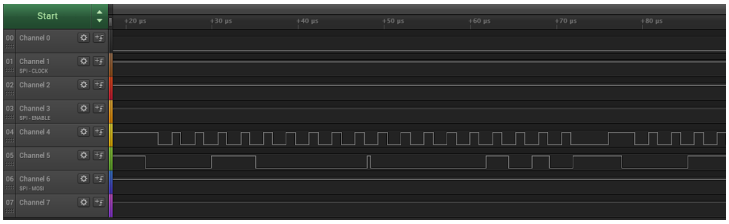
We first implement the demo project, which simply takes in a command as an input and returns the contents of the respective registers requested. This gave us an idea of how to utilize the lower-level functions in the next section.

In this section, we were having trouble reading from the registers and we would get a large unavailable command exception thrown to the terminal console window. We knew it was not picking up the values for some reason, although our device was properly connected. We then lifted the LaunchPad and held it at an angle, and that did the trick. We received constant values. To ensure that they were right, we would check the registers at certain angles



twice or thrice to make sure we were receiving back similar values.

LAB II, PART II | ACCELERATION DATA FOR X,Y, AND Z-AXIS



Verifying i2c waveforms using Saleae Logic

LAB II, PART I | SALEAE ANALYSIS

We set up our logic probe to capture i2c waveforms by enabling the analyzer and connecting ground, I2C_SCL and I2C_SDA. This waveform enraptures both x/y performance as we captured this waveform while the ball was rolling around the OLED, which constantly calls the WRITE and READ functions to acquire the register readings. Channel 4 represents the clock, and Channel 5 represents the WRITE and READ commands.

Application Program

We use the data that we grabbed from the accelerometer and make a small ball move around the screen via the position of the BMA222. The output is able to control the motion of the ball and make it seem like it seamlessly rolls around the OLED. The ball does not move any further than the screen edges.

```
//wall
x = x + x_dir;
if(x < ball_radius)
    x = ball_radius;

if(x > SSD1351WIDTH - ball_radius)
    x =  SSD1351WIDTH - ball_radius;

y = y + y_dir;
if(y < ball_radius)
    y = ball_radius;
if(y > SSD1351WIDTH - ball_radius)
    y =  SSD1351WIDTH - ball_radius;
```

Some problems we encountered on this part included the feature of acceleration. We did not know how to speed up the ball, and it would roll around very slowly. We realized we needed to make the change in position much more quick by dividing by a smaller delay value, which then allowed for the ball to change positions faster.

As a test to make sure that the ball was not glitching at any point, we changed its color every couple seconds while it continued to roll around the board. This confirmed that the process was smooth and the code was functioning correctly. We were able to utilize I2C connections and functions to retrieve measurements that we were able to then manipulate and produce a ball that reflects these fluctuations in the x-y angles.

```
// read X
I2C_IF_Write(ucDevAddr,&ucRegOffsetX,1,0);
I2C_IF_Read(ucDevAddr, &aucRdDataBufX[0], 1);

// read Y
I2C_IF_Write(ucDevAddr,&ucRegOffsetY,1,0);
I2C_IF_Read(ucDevAddr, &aucRdDataBufY[0], 1);

float bufx = (float)((signed char)aucRdDataBufX[0]) / delay;
float bufy = (float)((signed char)aucRdDataBufY[0]) / delay;


int x_dir = bufx*-1;
int y_dir = bufy;
```

LAB II, PART II | DIVIDING BY A DELAY VALUE TO INCREASE SPEED

SUMMARY

This lab has introduced quite a couple of new tools to our belt as engineering students. We learned about the SPI and I2C connections and applied them in different ways. For example, the SPI connection was used to connect to the OLED device and send data over, as simple as the instruction drawPixel(), which draws a single pixel a color of programmer's choice. We also utilized I2C connection, which is a two-wired interface that is able to send data back and forth to multiple devices. In this case however, we only implemented this connection over the BMA222 accelerometer to make our ball move from the OLED. This interaction of multiple devices across the LaunchPad helped us better understand the pathways of these connections starting from the implementation of code in CCS.

EXTRA CREDIT EXTENSION - PAC-BALL

PAC-BALL | INSPIRATION[2]

To test our newfound skills, we decided to create a Pac-Man game where the accelerometer controlled the position of Pac-Man. The aim of the game is to gather all the coins without touching any of the edges. If an edge is touched, then a red circle appears at the bottom of the screen, and more appear as you hit more edges to keep track of each collision, hence your final score (the aim being to avoid as many collisions as possible).

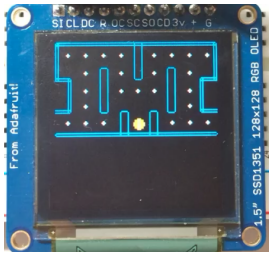
[2] http://www.indieretronews.com/2016/03/pac-man-20-neave-games-remakes-arcade.html

```
void map(){
    drawRoundRect(51,54,7,20,2,BLUE);
    drawRoundRect(74,54,7,20,2,BLUE);

    //center left
    drawRoundRect(35,20,7,36,2,BLUE);
//    drawRoundRect(35,54,16,3,2,BLUE);

    //center right
    drawRoundRect(88,20,7,36,2,BLUE);
//    drawRoundRect(79,54,16,3,2,BLUE);

    //center top
    drawRoundRect(62,0,7,30,2,BLUE);

    //top left
    drawRoundRect(0,0,3,31,2,BLUE);
    drawRoundRect(0,0,128,3,2,BLUE);

    //top right
    drawRoundRect(125,0,3,31,2,BLUE);

    //bottom left
    drawRoundRect(0,71,128,3,2,BLUE);

    //entrance left
    drawRoundRect(-2,28,15,3,2,BLUE);
    drawRoundRect(10,28,3,28,2,BLUE);
    drawRoundRect(-2,53,15,3,2,BLUE);

    //entrance right
    drawRoundRect(114,28,15,3,2,BLUE);
    drawRoundRect(114,28,3,28,2,BLUE);
    drawRoundRect(114,53,15,3,2,BLUE);
}
```

```
void coins(){
    fillCircle(10,10,1,WHITE);
    fillCircle(24,10,1,WHITE);
    fillCircle(38,10,1,WHITE);
    fillCircle(52,10,1,WHITE);
    fillCircle(52,24,1,WHITE);
    fillCircle(52,38,1,WHITE);

    fillCircle(65,38,1,WHITE);

    fillCircle(120,10,1,WHITE);
    fillCircle(106,10,1,WHITE);
    fillCircle(92,10,1,WHITE);
    fillCircle(78,10,1,WHITE);
    fillCircle(78,24,1,WHITE);
    fillCircle(78,38,1,WHITE);

    fillCircle(10,63,1,WHITE);
    fillCircle(24,63,1,WHITE);
    fillCircle(38,63,1,WHITE);

    fillCircle(120,63,1,WHITE);
    fillCircle(106,63,1,WHITE);
    fillCircle(92,63,1,WHITE);

    fillCircle(24,24,1,WHITE);
    fillCircle(24,38,1,WHITE);
    fillCircle(24,50,1,WHITE);

    fillCircle(106,24,1,WHITE);
    fillCircle(106,38,1,WHITE);
    fillCircle(106,50,1,WHITE);
}
```

‘PAC-BALL | FORMING THE MAP

We first create a function to create the map and coins on to the display before the start of the game. We map it out to be as synchronous and symmetric as possible to maintain the look of the classic game.



To accommodate for the edges, we test if the ball hits each and every edge, and the second the ball hits an edge, it prints out a red circle on the bottom half of the screen symbolizing a loss in life. (We could have started the other way around and made each circle disappear but we decided it would be more stimulating to make the circles pop up after each edge-collision) . Further more, the code ensures that the ball cannot go over the rectangles, and is constrained within the walls of the system.

```
        //center right
        if(x < 86 && x > 69 && y > 48 && y < 70){
            if(x < 86 && x > 78 && y > 48 && y < 70){
                x = 86;
                fillCircle(70,lifey,2,RED);
            }
            if(x > 69 && x < 78 && y > 48 && y < 70){
                x = 69;
                fillCircle(80,lifey,2,RED);
            }
        }
```

PAC-BALL | COLLISION CONTROL + LIFELINES

Demo: https://youtu.be/HLsnqbrbg0g