Lab 9

Serial Peripheral Interface (SPI) & LCD Pixel Display

In this lab, we will learn using the Serial Peripheral Interface (SPI) to communicate data and use it to interface the pixel display that's on the BoosterPack board. We will also learn how the graphics software stack works and use it to draw on the pixel display.

9.1 Serial Peripheral Interface (SPI)

SPI is a synchronous protocol that transmits data in a full-duplex manner over two data wires. One device in SPI is designated as the master and is responsible for initiating communication and for driving the clock signal. In its basic form, SPI has four wires: MOSI (Master Out Station In), MISO (Master In Station Out), SCLK (Serial Clock) and CS (Chip Select). MOSI pins at the master and device are connected together, and so are MISO pins. The master's SCLK and CS pins are also connected to the same pins at the device.

Each of the master and device has an internal shift register (e.g. 8-bit) and these shift registers are connected into a loop. The output of the master's shift register is tied to the input of the device's shift

register and vice versa. The communication starts when the master asserts the CS pin to activate the device (CS is usually active low). The master then drives eight clock cycles on the SCLK pin which exchanges the contents of the master's and the device's shift registers. At the end, the master deasserts the CS pin. In some configurations, the device's CS pin is permanently asserted (e.g. tied to ground since it's active low) and this is referred to as 3-pin SPI.

SPI was introduced by Motorola but is not officially considered to be a standard, therefore, the terminology differs across manufacturers. MISO and MOSI can be referred to as SOMI or SIMO, respectively, or they may simply be called SDO (Serial Data Out) and SDI (Serial Data In) or any other similar names. The CS pin may be referred to as STE (Station Enable) or other similar names.

In order to make the SPI operation reliable, the output of each shift register is first latched into a D flip-flop that's inside the device. At one clock edge (e.g. falling edge), the bit going out of the shift register is latched in the flip-flop and, at the opposite clock edge (e.g. rising edge), the shift registers are shifted. Based on this approach, the shift registers read stable data from the flip-flops. The communication between two 8-bit shift registers therefore consists of eight latch/shift operations, which are also referred to as latch/communicate operations.

Four modes of operation are defined for SPI based on whether the clock signal is high or low at idle (clock polarity) and based on what clock edges (rising or falling) trigger the latch and communicate actions (clock phase). The possible setups are shown below. The four cases of (polarity/phase) equal to (0/0), (0/1), (1/0), (1/1) are known as SPI modes 0, 1, 2, 3, respectively. The same mode should be used at the master and the device. In practice, Mode 0 is the most popular mode.

```
Polarity 0:    Clock idle at low
Polarity 1:    Clock idle at high
Phase 0:       Latch at trailing edge, communicate at leading edge
Phase 1:       Latch at leading edge, communicate at trailing edge
```

BoosterPack's Pixel Display

The display module that's on the BoosterPack is the CrystalFontz CFAF128128B-0145T. The display is square, has a diagonal size of 1.45", has 128x128 pixel resolution, and supports 16-bit color for a total of 262,000 colors. The display module has a built-in controller which is the Sitronix ST7735S. The controller incorporates an SPI interface that is used to communicate with the MCU.

The interface between the display and the MCU/BoosterPack is shown in Figure 9.1. The first three pins on the display are the SPI pins. There's no Serial Data Out (MISO) pin since the display doesn't transmit back any information. The pin SPI3W/SPI4W is connected to Vcc on the BoosterPack, which means the SPI interface has four wires (it uses the Chip Select pin). The Data/Command (DC) pin is used to label each byte that's received via SPI. The software driver sets the DC pin to 0/1 to indicate that the byte being transmitted over SPI is either a command byte or a data byte, respectively. The Reset pin is

used to start up and shut down the display's controller. After reset, the display's controller is not active until the reset pulse, which has specific duration requirements, is applied on the reset pin by the software driver. Finally, the Backlight pin is used to control the display's brightness. Writing low or high to this pin corresponds to the minimum and maximum brightness levels, respectively. Intermediate brightness levels can be implemented by driving a PWM on this pin.

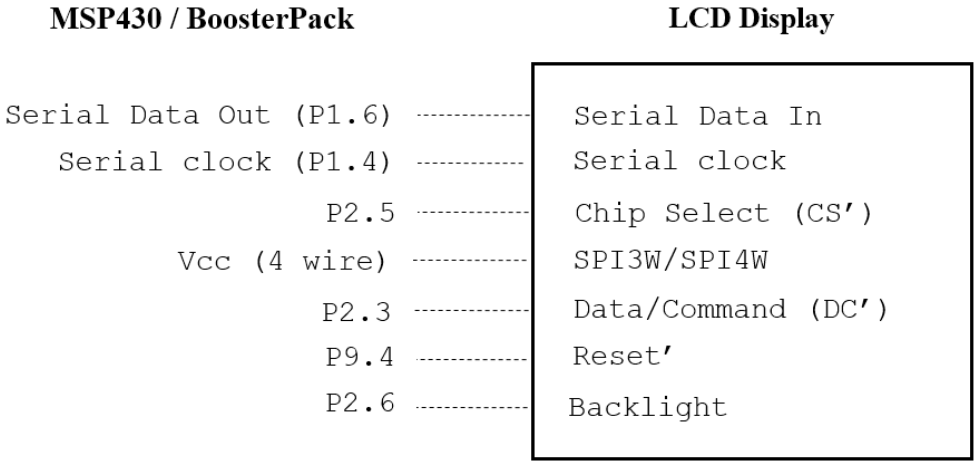

Figure 9.1: BoosterPack's Pixel Display Interface

The left side of the figure shows the MCU/BoosterPack side of the interface. These mappings can be found by looking in the BoosterPack User's Guide (slau599a) Figure 3 and Table 7 and matching the pins to the LaunchPad User's Guide (slau267a) Figure 10 and Figure 3.

On the MCU/BoosterPack side, the SPI data and clock lines are connected to eUSCI_B0, which is eUSCI module #0 Channel B. The display's Chip Select (CS) pin is wired to a GPIO on the MCU rather than eUSCI's Chip Select (CS) pin. This means that the eUSCI module cannot control the CS pin. To keep things simple, our code will set this pin to low permanently so that the display's SPI interface is active all the time. The remaining three pins (DC, Reset and Backlight) are connected to GPIO pins at the MCU.

SPI Configuration

We will configure the eUSCI module to setup an SPI link to the display. Download the Code Composer Studio (CCS) project and the documentation that are found at the link below. The CCS project contains the pixel display driver (in the folder LcdDriver) and the graphics library (in the folder GrLib). The main function prints a welcome message to the display. However, for this code to work, you need to write two functions in the lower driver. These functions configure the pins and setup the SPI interface.

http://www.ece.ucf.edu/~zakhia17/EEL4742C

Open the file LcdDriver/lower_driver.c and write the body of the two functions that are listed

below. The first function[1] configures the GPIO pins and the SPI pins by redirecting their functionality. By default, pins in MSP430 are configured are GPIOs. To change the functionality of the pins, look in the MCU's data sheet (slas789c) starting on p. 95. Find the corresponding values of PxSEL, like we did in earlier labs. Note that for SPI, only two pins need to be configured: the pin that transmits data out of the master and the clock pin. The pin that brings data into the master is not used and the Chip Select (CS) pin is not used since the display's chip select is connected to a GPIO pin and we'll set it to low permanently.

```
// Configure the pins
void HAL_LCD_PortInit(void);

// Configure eUSCI for SPI operation
void HAL_LCD_SpiInit(void);
```

The second function configures eUSCI Module #0 Channel B for SPI operation. The configuration registers of eUSCI in SPI mode are found in the FR6xx Family User's Guide (slau367o) at the end of Chapter 31. Note that SPI is implemented in both Channels A and B of eUSCI since it's a simple protocol. We're using Channel B, therefore, we'll use the configuration registers UCBx at the end of the chapter (rather than UCAx).

The main file of the project has a function that configures the clock system. The function keeps DCO at the default frequency of 8 MHz and sets up MCLK at fDCO/2 = 4 MHz and SMCLK at fDCO/1 = 8 MHz. In the SPI configuration function, your goal is to select SMCLK as the clock signal and divide it by 1 so that SPI runs at a clock frequency of 8 MHz. This results in a reasonably fast refresh rate.

The comments in the two functions will guide you to doing the configuration. Write the two functions and compile the code. The code should print a welcome message to the display.

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Which SPI mode does the configuration correspond to?
- Submit the code in your report. Submit only the two functions that you wrote.

9.2 Using the Graphics Library

In this part, we will use TI's Graphics Library (grlib) to draw shapes and text on the display. The graphics software stack consists of multiple layers. The lowest layer is the driver and is implemented in two source files in our code: lower_driver.c and lcd_driver.c. The first file represents the lower level of the driver and has the task of establishing the physical link to the display, which is the SPI link in our case. The first file implements SPI write functions. The second file implements the general driver tasks such as driving the reset pulse to activate the display and drawing a pixel on any x-y coordinates on

[1]The term HAL refers to Hardware Abstraction Layer which is a low-level software that is similar to a driver.

the display. The function that draws a pixel on the display accomplishes its task by calling the SPI write functions of the lower driver. In general, each layer in the software stack uses the services of lower layers and provides services to upper layers.

The next layer up in the software stack is the graphics library. The library uses the draw pixel function of the driver and provides intuitive functions that can be used in the application code that we're writing. Such functions draw circles, rectangles, lines, text and images and enable changing the background and foreground colors. The grlib library that we're using is open source and is included in the project as source files. Browse the library's API documentation file to see what functions are available. You can also browse the file `grlib.h` in the project to see the functions' headers. This file also contains color definitions as 24-bit constants.

To draw a picture on the display, the picture file needs to be converted to a C file using a tool called Image Reformer Tool. This tool is available for download from TI. The resulting C file contains an object of type `Graphics_Image`, or its alias `tImage`. The picture is drawn on the display by passing the image object to the grlib function `Graphics_drawImage`. The project provided to you has a file called `logo.c` that contains an image object.

Write a demo that demonstrates the graphics library's capabilities. The demo at this video is an example.

https://youtu.be/ZAwfe9gDpdE

Your demo should meet the following requirements.

1. Set a new background color
2. Set a new foreground color
3. Draw at least one of each: an outline circle, a filled circle, an outline rectangle, a filled rectangle and a horizontal line.
4. Use at least three different colors on your screen (open the file `grlib.h` in the project to see the available colors)
5. Draw an image on the first screen (you can use the image object in `logo.c` or make a new one)
6. Pushing the button transitions back and forth between the two screens
7. Use two different fonts on the screen (the project contains fonts in the folder `GrLib/fonts`)

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Include pictures of the two screens of your demo in your report.
- Submit the code in your report. Submit only the main.c file.

9.3 Application: Optical Counter

As an embedded engineer, you are in charge of designing an optical counter based on the BoosterPack's light sensor. The optical counter detects objects passing in close proximity to it, e.g. one inch or closer, and counts all such occurrences. Such a counter has many applications in industrial settings (e.g. counting items passing on a conveyor belt), automated environments and sport equipment. The counter should have a minimum accuracy of 90% and should be able to count objects passing at a rate of once per second or slower. The counter should also detect objects that are passing very slowly.

In the application, print the sensor's real-time value on the display and draw a progress bar that reflects the sensor's value. The progress bar should have a full-range of 1000 lux. For example, if the reading is 400 lux, the progress bar should be at 40% full. When the lux value goes beyond 1000 lux, the progress bar should max out at 100%.

The CCS project provided to you in this lab configures SMCLK to 8 MHz, therefore, you need to update your old I2C initialization function by modifying the I2C clock divider. Your old function was based on the default SMCLK frequency of 1 MHz. Modify the clock divider so that it results in an I2C clock frequency of 320 KHz.

A demo of the application can be found at the link below:

https://youtu.be/7BKjwMoeFvo

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Submit the code in your report. Submit the main.c file only.

Student Q&A

1. Is SPI implemented as simplex or full-duplex in this experiment?
2. What SPI clock frequency did we set up in this lab?
3. What I2C clock frequency did we set up in this lab?
4. What is the maximum SPI clock frequency that is supported by the eUSCI module? Look in the microcontroller's data sheet in Table 5-18.
5. Show how you computed the I2C clock divider in the last part.