Experiment 8: SPI and DMA

1.0 Introduction

The Serial Peripheral Interface (SPI) is a widely-used method for communicating with digital devices with an economy of wires and connections. You have used such devices in the past by "bit-banging" the interface using GPIO, but your microcontroller has support for SPI devices that simplifies the use of such interfaces. This support also allows for the use of Direct Memory Access (DMA) to automatically transfer a region of memory to the device. In this lab, you will gain experience using SPI and DMA with a display device.

STEP	DESCRIPTION	POINTS
4.0	Prelab Questions	20
6.1	Wiring and Testing the LCD Display	10
6.2	Initializing and using the LCD Display with SPI	10
6.3	Writing to SPI with DMA	10
6.4	Using Circular DMA	10
6.5	Observing and Testing SPI Clock Rates	10
6.6	Updating the display with an interrupt	10
7.0	Postlab questions and code submission	*
	TOTAL	80

** All points for the lab experiment are contingent upon submission of code completed in lab.*

2.0 Objectives

1. To understand the Serial Peripheral Interface format
2. To use and observe an SPI device
3. To use DMA to automatically transfer data to an SPI device

3.0 Equipment and Software

1. Standard ECE362 Software Toolchain (Installation instructions available in Experiment 0: Getting Started)
2. STM32F0-DISCOVERY development board (For procurement instructions, see Experiment 0: Getting Started)

4.0 Prelab

Read section 5 of this document for background information. Pre-wire your LCD display to your microcontroller and test that it works. Answer the prelab questions on the ECE 362 web page.

5.0 Background

5.1 Serial Peripheral Interface

When communication speed is not high priority, it is helpful to minimize wiring by using a *serial* communication protocol. It is named so because bits are sent one at a time, one after another. The Serial Peripheral Interface (SPI) is a common way of doing so. SPI turns words into a stream of bits and vice-versa. To send an entire word through the serial output, a programmer need only write the word into the SPI data register, and the hardware takes care of converting into a bit stream.

SPI is a synchronous protocol, so it requires a clock signal to indicate when each bit of data sent should be *latched-in* to the receiver. SPI defines devices that act in two distinct rôles: A master is responsible for directing operations by asserting a slave select line, driving the clock, sending data on a MOSI (master out, slave in) pin, and optionally listening to input on a MISO (master in, slave out) pin. A slave responds to operations when its slave select pin ($\overline{\text{SS}}$ or NSS) is asserted, reads data on the MOSI pin, and sends data on the MISO pin on each clock pulse. Because SPI is synchronous, there is no need for devices to agree, in advance, on a particular baud rate to communicate with each other. As long as the master device does not drive the clock at a frequency that is higher than a slave device can tolerate, data will be received correctly.

The SPI driver in the STM32 can be configured for several different modes of operation. For instance, the clock output can be configured to latch data on the rising edge or falling edge. Also, the NSS output can be set to automatically pulse low for each word written, but only when the clock is in a specific configuration. NSS pulse generation is generally not useful in situations where multiple slave devices share the same MOSI, MISO, and SCK pins. For that, you would want to control multiple individual $\overline{\text{SS}}$ pins. Since we are using a single device, and since that device demands that NSS go high after every word written to it, we will use the NSSP feature.

The baud rate (another name for the rate at which bits are sent) for an STM32 SPI channel can be set to a fraction of the system clock. The SPIx_CR1 register has a BR field that defines a prescale divisor for the clock. The size of the word to be sent and received by an STM32 SPI channel is set with the SPIx_CR2 DS field. This 4-bit field is unique among other I/O registers in that ‘0000’ is not a legal value. An attempt to clear this field before setting it to something new will result in it being reset to ‘0111’ which defines an 8-bit word size. For this lab experiment, we will connect a CFAL1602 OLED display which communicates in bytes with two extra bits to indicate a register selection and read/write selection–10 bits total. To set the DS field to a 10-bit word, it is necessary to write the bit pattern directly without clearing the DS field first. This should be the first thing done to the CR2 register. Thereafter, other bits can be ‘OR’ed to CR2.

5.2 Direct Memory Access

The STM32F0 microcontroller has five Direct Memory Access (DMA) channels that can autonomously move one or more words of various sizes between memory and peripherals. Once values are written to a DMA channel’s configuration registers, the CPU can be used for other purposes while the transfers

operations continue. Each incremental DMA operation can be triggered by a peripheral's readiness to be written to or read from. Some microcontrollers allow DMA channels to be arbitrarily associated with peripheral devices, but the STM32F0 requires that each peripheral be used with only the specific DMA channel that it is wired to. Table 30 of the Family Reference Manual shows the peripheral-to-DMA channel mapping.

Under normal circumstances, when moving a number of words of memory (indicated by the CNDTR register) to or from a single peripheral address (in the CPAR register), the register that points to the memory address (CMAR) will be incremented by the size of the word. This increment is optional and must be set with the MINC bit of the channel's CCR register. A DMA request is activated by setting the EN bit of the channel's CCR register. Once the requested number of words has been moved, the EN bit for that channel is cleared. Each DMA channel can be configured for *circular* operation by setting the CIRC bit in the CCR register. In circular mode, the DMA channel moves the specified number of words (CNDTR) between the addresses in CMAR and CPAR, incrementing these registers as requested, but when finished, instead of disabling the EN bit, the CMAR, CPAR, and CNDTR registers are reset to their original values, and the channel request is restarted. A circular request is one that continually moves words between a region of memory and a peripheral register.

Each DMA channel can give updates on its progress by invoking interrupts. Three interrupts are possible. First, the Transfer Complete interrupt indicates that all words of the DMA request have been moved. Second, the Half Transfer interrupt indicates that half of the words of the DMA request have been moved. (This is useful when using a circular buffer to indicate that the first half of the buffer is ready.) Finally, a Transfer Error interrupt indicates that an access was attempted to a memory location that was not permitted for the type of operation requested.

5.3 Hardware Configuration

You will use a CFAL1602 OLED LCD display for this experiment. Despite what the label on the back might say, the full model name of your OLED display is "CFAL1602C-PB." It only works with SPI protocol. Connect the CFAL1602's V_{DD} connection to 3V and V_{SS} to zero volts. Even though the display's specifications clearly state that it won't work at less than 4.8V, it seems to work perfectly at 3V. Doing so avoids logic level problems that occur when powered at 5V. Connect the SPI:SS (pin 16), SPI:MOSI (pin 14), and SPI:SCK (pin 12) pins to the corresponding pins for SPI channel 2 of the STM32. (You may leave the SPI:MISO pin unconnected.) The pin numbers for the display are conveniently located on the *bottom* of the circuit board where you cannot see them. Examine Figure 1 for a slightly more helpful reference.

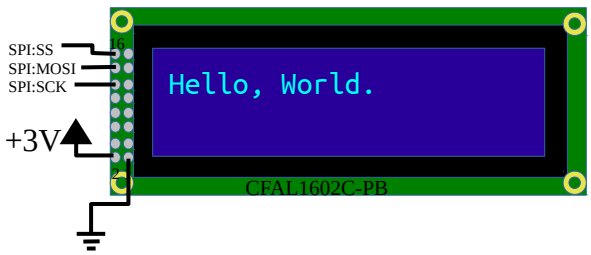

Figure 1. A slightly more helpful diagram of the LCD pins

5.4 SPI Protocol for LCD Display

A controlling computer uses three pins to send data to the CFAL1602 LCD. First, for any communication to take place, the SCK:SS pin must be low. This corresponds to the NSS (negative slave select) line of the STM32. Since we are using only one SPI slave device, the STM32 will be the master for all SPI protocol operations, and we can use automatic NSS protocol to control the display. Data is sent to the display on its SPI:MOSI pin, and it is "clocked in" on the rising edge of the signal on the SPI:SCK pin.

Most SPI devices use a 4-, 8-, or 16-bit word size. The CFAL1602 uses a 2+8-bit word size to send two bits of configuration information plus an 8-bit character on each transfer. The first two bits are, respectively, the *register selection* and read/write. Since we will always be writing data to the display and never reading it, we will always make the second bit a '0'. The register selection determines whether a write is intended as a *command* or a character to write to the display. Commands are needed to, for instance, initialize the display, configure the communication format, clear the screen, move the cursor to a new position on the screen, etc. The CFAL1602 implements an old and well-known LCD protocol. It is documented on the ECE 362 References page in the Winstar Display Datasheet. There are many commands that can be used to implement complex operations on the CFAL1602 that we will not use for this lab experiment. For the sake of this lab, we will be concerned only with the initialization sequence, moving the cursor, and writing characters to display.

When the register select bit is zero, the transmission issues an 8-bit command to the display. When the register select bit is one, the transmission represents a character to write to the display.

Page 24 of the Winstar Display Datasheet describes the command format. Page 18 shows the sequence of operations needed to initialize the display. Until initialization is done, no character data will be displayed on the screen. The operations to be done are as follows:

1) Wait 100ms for the display to power up and reset itself.
2) Change the interface from 4 bits to 8 bits, and configure the display for two-line output, and select the 5x8 character font. This is done with the command 001DNF00 where D configures the data length, N configures the number of lines, and F configures the font size. For our purposes, we want to send the command byte 0011 1000 (0x38). To send this as a 10-bit SPI command transmission, it would be 00 0011 1000.
3) Turn on the display, disable the blinking cursor. This is done with the command 00001DCB where D turns the display on, C turns the cursor on, and B sets it to blink. We will use the command 0000 1100 (0x0c) to turn on the display without a cursor.
4) Clear the entire display. This is one with the 0000 0001 (0x01) command.
5) Wait 62ms for the display to clear.
6) Set the cursor to position zero (the top left corner) of the display. This is done with the 0000 0010 (0x02) command.
7) Set the *entry mode* to move the cursor right after each new character is displayed without shifting the display. This is done with the 0000 01DS command, where D represents the direction to advance to, and S configures shifting the display. We will use the 0000 0110 (0x06) command to set up the display.

After these initialization steps are complete, the LCD is ready to display characters starting in the upper left corner. Data can be sent with a 10-bit SPI transfer where the first bit is a 1. For instance, the 10-bit word 10 0100 0001 (0x241) would tell the LCD to display the character ‘A’ at the current cursor position. In the C programming language, a character is treated as an 8-bit integer. In general, any character can be sent to the display by adding the character to 0x200 to produce a 16-bit result that can be sent to the SPI transmitter. For instance, to write an ‘A’ to the display after initialization, the following statement could be used:

```
while((SPI2->SR & SPI_SR_TXE) == 0)
      ; // wait for the transmit buffer to be empty
SPI2->DR = 0x200 + ‘A’;
```

5.5 Software Configuration

Use the Standard Peripheral firmware for this lab experiment. If you are unable to download the Standard Peripheral firmware, you may use the template directory provided to you as lab8_template.zip on the lab experiments web page. We provide a template file, **`main.c`**, in which you will write all of your code. We also provide a **`support.c`** file that contains several subroutines that you can examine. These subroutines will be used to get started on understanding SPI protocol and to test your wiring and subroutines that you write.

6.0 Experiment

Description:

For this experiment, you will write the subroutines to initialize and write to the CFAL1602 OLED display through the SPI interface and using DMA. You will write and test subroutines similar to the **`init_lcd()`**, **`display1()`**, and **`display2()`** subroutines that you became familiar with in a past lab. This time, these three symbols are declared as function pointers. They are initially NULL and you must *assign* functions to them to configure them to be used. You will gradually change their values as you progress from using bitbanged I/O to SPI to DMA.

6.1 Wiring and testing the LCD Display

In the support.c file, we provide five functions: **`bitbang_cmd()`**, **`bitbang_data()`**, **`bitbang_init_lcd()`**, **`nondma_display1()`**, and **`nondma_display2()`**. The first two functions, **`bitbang_data()`** and **`bitbang_cmd()`**, set and clear the appropriate SPI bits programmatically without using the SPI hardware. It is instructive to examine these functions to understand what they do. The initialization function sets up the GPIO pins, and issues the delays and commands to configure the LCD. The **`bitbang_cmd()`** function sends a 10-bit transfer where the first two bits are always 0, followed by the 8-bits specified by the character passed as an argument. The **`bitbang_data()`** function works similarly, except that the first two bits are '10' to represent a character to be displayed rather than a command to issue. The functions **`bitbang_display1()`**, and **`bitbang_display2()`** send commands to position the cursor at offset zero and offset 64 (0x40), respectively. (Offset 64 is the leftmost character of the second line.)

Uncomment only the **`step1()`** in **`main()`** and check that your display works as expected. Then connect three leads of your lab station's oscilloscope to the SPI:SS, SPI:MOSI, and SPI:SCK pins of the LCD.

CAUTION: You should never connect any measurement device directly to the pins on your development board. If you accidentally short 3V to GND or 5V to GND, you will destroy one of the diodes on your development board, rendering it useless. Always connect wires to the development board that lead away to a safe part of your breadboard. Attach probes to the wire leads.

Configure the scope to trigger on the falling edge of the signal on the SPI:SS pin. Press the "Single" button to capture one full 10-bit transfer on the scope. (Leave these scope leads in place for later sections of the lab.)

Demonstrate your work to your TA.

6.2 Initializing and using the LCD Display with SPI

Implement the **spi_cmd()** and **spi_data()** subroutines so that they wait for the TXE bit to be clear indicating that the SPI channel 2 transmitter bit is empty. Then they should deposit the data in the argument into the SPI channel 2 data register. The **spi_data()** subroutine should add in an extra 0x200 to the data written so that it sets the register select bit as described in Section 5.4.

Complete the **init_lcd_spi()** subroutine so that it configures the appropriate I/O pins so that they are routed to SPI channel 2. It should also initialize SPI channel 2 so that it is configured for bidirectional mode, with bidirectional output enabled, currently set as master, and the slowest possible baud rate. Configure the clock to be 0 when idle, and use the first clock transition as the data capture edge. Also configure it to use a 10-bit word size, slave select output enable, and automatic NSS pulse generation. Enable the SPI2 channel by setting the SPE bit. Finally, this subroutine should invoke the initialization procedures for the LCD by calling **generic_lcd_startup()**.

Uncomment only the **step2()** call in **main()**. This will set the function pointers to the right values for this step of the lab experiment and should result in a new display output. Capture a single frame on the oscilloscope that shows a 10-bit SPI transfer.

Show the working result to your TA.

6.3 Writing to SPI with DMA

The **nondma_display1()** subroutine used in step 6.2 called the **cmd()** and **data()** subroutines which continually checked the SPI status register until the transmitter was empty. Wasting time like this is excusable for a startup initialization subroutine, but to do so during regular operation might prevent some other activity from having enough time to run. For this step, you will modify the **dma_display1()** subroutine so that it initializes the **dispmem** array which is defined at the top of the main.c file. The first half of the array is set up with the values 0x080, followed by 16 0x220 values. Recall that when the 0x200 bit is set, the value designates a character to print at the present cursor location on the display, and when it is clear, the value is a command for the display to execute. The first seventeen values sent to the display will move the cursor to the first character of the first line, and then print 16 space characters (0x20). The next seventeen characters of the array will do the same thing for the second line.

Modify **dma_display1()** to be like **nondma_display1()** except that, instead of calling **data()**, it takes each input character (**s[x]**), adds (or 'OR's) 0x200 with it, and writes it into **dispmem[x+1]**. It should pad out the remainder of the 16 characters with 0x220 in the same way that **nondma_display1()** calls **data(' ')**. Remember that **dispmem[0]** is already set to the command to move the cursor to the beginning of line 1. You should not overwrite that word.

The subroutine should then initialize the appropriate DMA channel with a request to copy 17 16-bit values from **dispmem** to SPI2_DR. Refer to the examples in the lecture notes to remember how to configure the proper channel of DMA controller 1. (There is one DMA controller with five different

channels. You should know how to choose the right channel.) Set the direction of data transfer from memory to peripheral. The memory address should be incremented after each transfer, but the peripheral address should not be incremented. You should set the priority to "low". Do not turn on circular transfers, interrupts, or memory-to-memory flags. You should also modify SPI channel 2 so that a DMA request is made whenever the transmitter buffer is empty. Finally, you should enable the DMA channel.

Once you complete this step, uncomment only the **step3()** call in **`main()`**.

Demonstrate your completed subroutine to your TA.

6.4 Using Circular DMA

Although step 6.3 was useful as a first attempt at using DMA, it had several shortcomings. First, it required almost as many writes to the I/O registers to set up the DMA transfer as it would have to actually send the 17 words to the SPI port. Second, although the DMA transfer wrote values to the SPI port no faster than it could accept them, there was no check in **`dma_display1()`** to detect if a previous DMA transfer was in operation. If the **`step3()`** subroutine had not waited between updates, it would have restarted the DMA transfer in the middle of a transfer already in progress. (You can try removing the **`nano_wait()`** from **`step3()`** to see what happens.) For other types of devices, this might have disastrous consequences.

For this step, you should modify **`dma_spi_init_lcd()`** so that it does everything that **`spi_init_lcd()`** did. (You may call one from the other.) Then it should also set up the appropriate DMA channel just like in **`dma_display1()`**. This time, set up the DMA transfer count to be 34–the number of entries in the entire **`dispmem`** array. It should also configure the DMA operation to be circular. Once enabled, the DMA operation will continually transfer the 34 words of **`dispmem`** to the display, filling it with spaces.

You should then modify **`circdma_display1()`** and **`circdma_display2()`** so that they copy and convert characters from the input string into the **`dispmem`** array just as you did in step 6.3. This time, you do not need to initiate a DMA transfer. It is already running. The **`circdma_display1()`** subroutine should copy and convert characters from **`s[x]`** to **`dispmem[x+1]`**, and the **`circdma_display2()`** subroutine should copy and convert characters from **`s[x]`** to **`dispmem[x+18]`**. Now these subroutines can be called at any time without waiting. The most recently written strings will soon be copied by the DMA controller to the display as fast as the SPI channel will permit them.

When you've finished your subroutines, uncomment only the **`step4()`** call in **`main()`**. Demonstrate your work to your TA.

6.5 Observing and Testing SPI Clock Rates

Set up the oscilloscope at your lab station so that the probes are connected to the NSS, SCK, and MOSI pins of the SPI channel as you did before. Set the scope to trigger on the falling edge of NSS. i.e. the start of a word transmission. Hold the reset button on your development board as you put the scope in "single" mode. When you release the reset button, you should catch the first initialization command sent to the display (0x038). What is the clock rate of the SPI channel?

Press the "Serial" button on your oscilloscope to invoke the mode that analyzes and displays the serial data stream in a high-level human-readable form. Configure the scope to tell it to analyze SPI protocol, and tell it which probes are connected to which pins. Capture and display a few words being sent over the SPI channel. Can you understand the values?

Change **`spi_init_lcd()`** to have a different value for the BR field in the SPI2_CR register to increase the SPI clock frequency. Find the highest frequency at which your display still works correctly. Once you find the lowest frequency at which it does not work, you may need to power off and reconnect your development board to make the display work again at a yet lower frequency.

Demonstrate your results to your TA.

6.6 Updating the display with an interrupt

Complete the functions **`init_tim2()`** so that it sets up a periodic interrupt that occurs ten times per second. Then write an interrupt service routine to go along with it that acknowledges the interrupt and calls the **`clock()`** function. Test these functions by uncommenting only the **`step6()`** function in **`main()`**.

Once this step of the lab is completed, you will have multiple asynchronous operations running without CPU loops to wait or check for device readiness. The CPU will spend most of its time stopped in the WFI instruction. Consider how you may use these techniques for your mini-project.

Demonstrate your work to your TA. Your display should show how long you have been waiting for your TA to check your work. (Although if you secretly inflate the hours, minutes, and seconds the counter starts at, who will know?)

7.0 Submit Your Code

When you finish your experiments, submit your main.c file.