f2022-public-labs

Lab 6: SPI and DMA

Step	Associated Steps (and checkoff criteria)	Points
1	2. Bit-Banging (setup_bb, bb_write_bit, bb_write_halfword, 0x79 word via SPI on scope)	25
2	3. SPI 7-segment (TIM15, SPI2, uncomment SPI_LEDS and test SPI2 on 7-seg with keypad)	25
3	4. SPI DMA 7-segment (same as 2, check that SPI_DMA_LEDS is uncommented)	15
4	5. SPI1 OLED (shows “Hello again, [their login]”)	10
5	6. SPI1 DMA OLED (uncomment SPI_OLED_DMA shows “Hello again, [their login]”)	25
6	7. Play the game	0
7	Total	100

1: Introduction

The Serial Peripheral Interface (SPI) is a widely-used method for communicating with digital devices with an economy of wires and connections. It is possible to control such devices by “bit-banging” the protocol using GPIO, but your microcontroller has high-level 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 display devices.

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 device 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 device responds to operations when its slave select pin ($\bar{SS}$ 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.

1.1: Instructional Objectives (100 points total)

- To replicate Serial Peripheral Interface via software emulation.
- To understand the Serial Peripheral Interface format
- To use and observe an SPI device
- To use DMA to automatically transfer data to an SPI device

1.2: Hardware Background

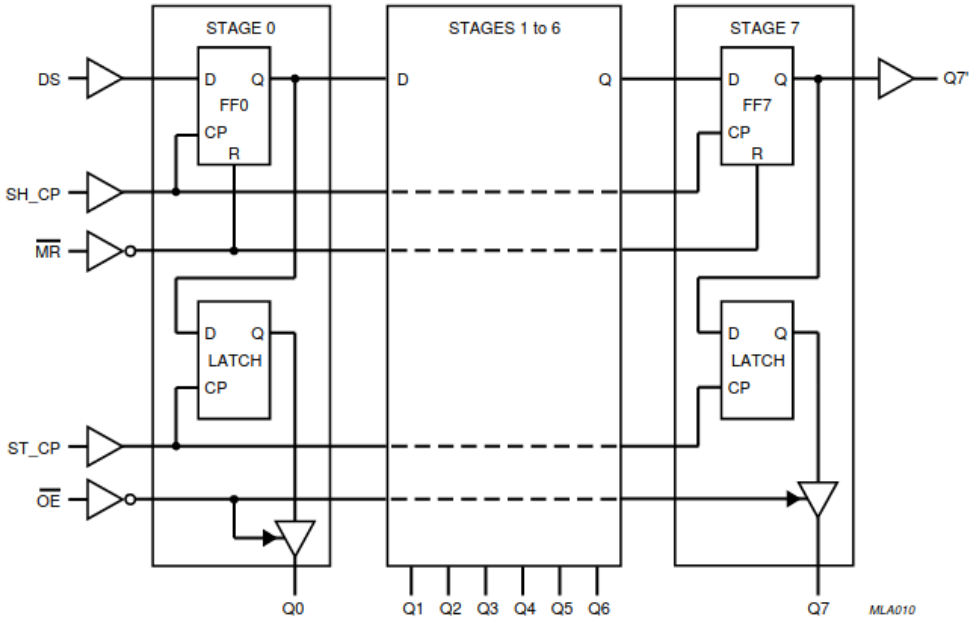
1.2.1: Shift Registers and 7-Segment Display

In Lab 7, you built an eight-character display out of multiplexed 7-segment LED displays. Since then, you have been using it through a parallel interface — you need to output 11 bits on Port B at the same time in order to display a character at a particular position. To reduce the number of STM32F091RC pins needed to drive your display, it is possible to use external shift registers as a *serial interface* for it.

Since each 74HC595 shift register in your lab kit only provides 8 output pins, you will need to cascade two together to effectively build a 16-bit shift register. Connect the serial input of one 74HC595 shift reigster to PB15 (MOSI). Connect the serial input of the other shift register to the serial output (Q7') of the first. Tie both shift register clock inputs (SH_CP) together, and connect them to PB13 (SCK). The outputs of the shift registers replace your previous Port B connections.

> Note: You may find it useful to place the 74HC595 shift registers on your breadboard upside-down, so the outputs of the shift registers are facing the inputs of the decoder and sink driver.

The output pins of each 74HC595 shift register are gated by an internal storage register. In order to update outputs, the storage register must be clocked (by asserting ST_CP). Tie both storage register clock inputs together, and connect them to PB12 (NSS). With SPI, you can simply use NSS as the storage register clock. Since NSS will be deassarted (set HIGH) at the end of every message in pulse mode, the shift registers will output new data.



1.2.2: SPI Protocol for OLED LCD Display

A controlling computer uses three pins to send data to the SOC1602A LCD. First, for any communication to take place, the CS pin must be asserted 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 device for all SPI protocol operations, and we can use automatic NSS protocol to control the display. Data is sent to the display on its SDI pin, and it is “clocked in” on the rising edge of the signal on the SCL pin. Each transfer must be terminated by deasserting (setting high) the CS pin.

Most SPI devices use a 4-, 8-, or 16-bit word size. The SOC1602A 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 SOC1602A implements an old and well-known LCD protocol. You may see it in many other two-line LCD modules. There are many commands that can be used to implement complex operations on the SOC1602A 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 7 of the SOC1602A datasheet lists the set of possible commands that can be sent to the display. Pay special attention to the column labeled “Max Execution Time”. Regardless of how fast data is sent to the display, some operations take significant time to complete. The sender must not start a new command before the previous one has finished. On the next page, details of the instruction format are given. Page 20 lists the initialization sequence (under the header LNITIALIATION SWUENCE[*]) that must be used to prepare the display for use. Until each step is properly completed, the display will not show anything. The greatest problem that students have with new hardware (other than poor documentation) is finding the patience to carefully implement each step of the initialization sequence.

The operations to be done are as follows:

- Wait 1ms for the display power to stabilize.
- **Function set**: The reason for issuing this command first is to set the data length for 8-bit operation. This is set by the DL bit in the command description for the 8-bit Function Set operation:
  - `0 0 1 DL 1 0 FT1 FT0`
  - To set the data length to 8-bit, we use DL=1. The FT[1:0] bits select a font. We’ll select 0 0 to use the English/Japanese font. The 8-bit command will be 00111000 or 0x38.
  - We cannot check the BUSY flag, because we did not connect the MISO pin to the display. Instead, we will simply wait long enough that the display can be guaranteed to finish the command. This command will complete in 600µs, at most. In practice, it will finish much faster.
- **Display OFF**: The recommendation is to turn the display off with the 8-bit command
  - `0 0 0 0 1 D C B`
  - where D enables the display output, C set the cursor to be visible, and B set the cursor to blink. Turn the display off with the 8-bit code 0x08.
- **Display Clear**: Clear the display. Note that this command requires **2ms** to complete. Delay for that long before continuing. The 8-bit command is 0x01.
- **Entry Mode Set**: Set the entry mode to move the cursor right after each new character is displayed without shifting the display. This is done with the command:
  - `0 0 0 0 0 1 D S`
  - This time, we will set D=1 and leave C=0, B=0. The 8-bit command is 0x0c.

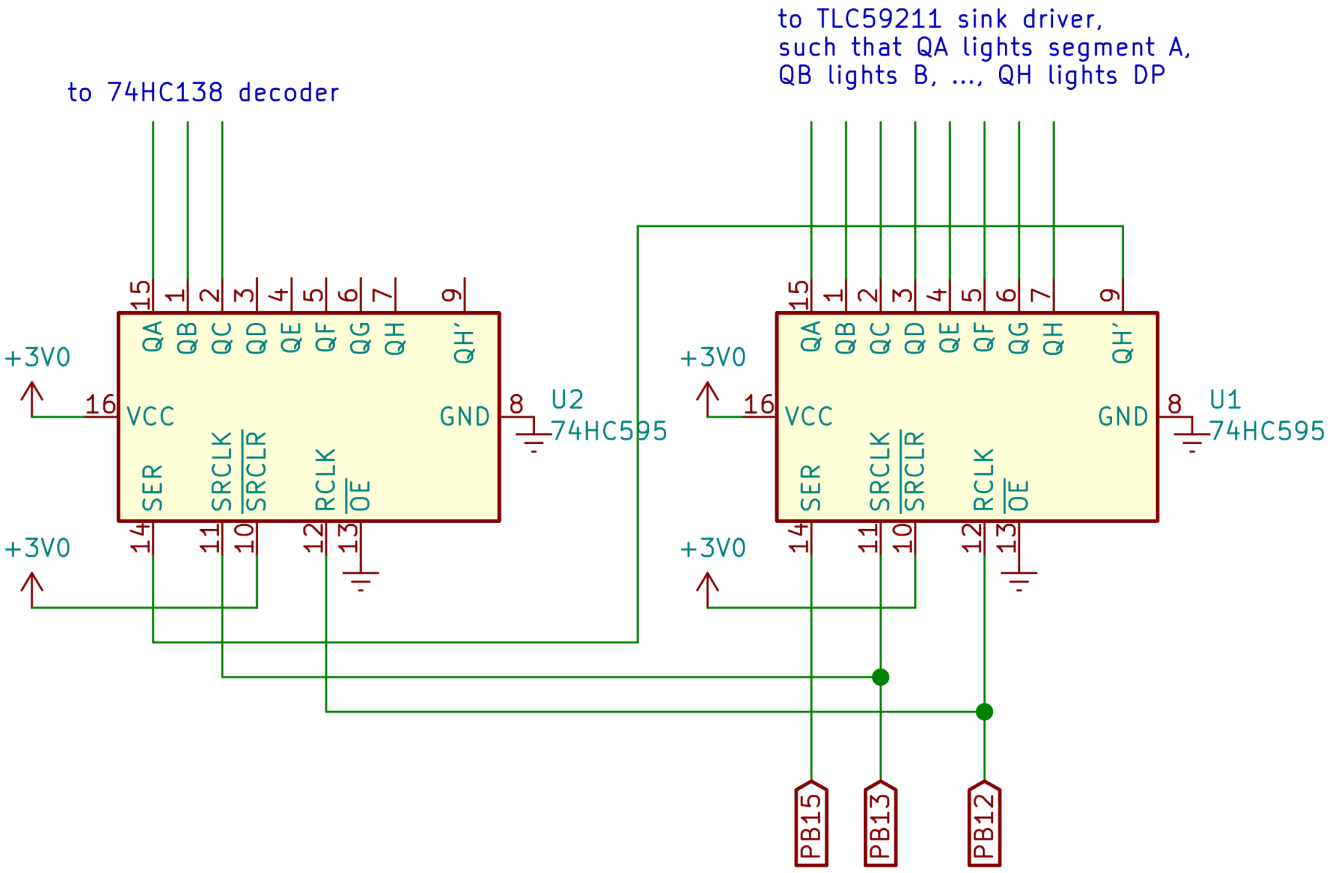
After these initialization steps are complete, the LCD is ready to display characters starting in the upper left corner. Data can be sent with a 10-bit SPI transfer where the first bit is a 1. For instance, the 10-bit word 10 0100 0001 (0x241) would tell the LCD to display the character ‘A’ at the current 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';
```

To understand why 0x41 is the same thing as ‘A’, you should consult the ASCII manual.

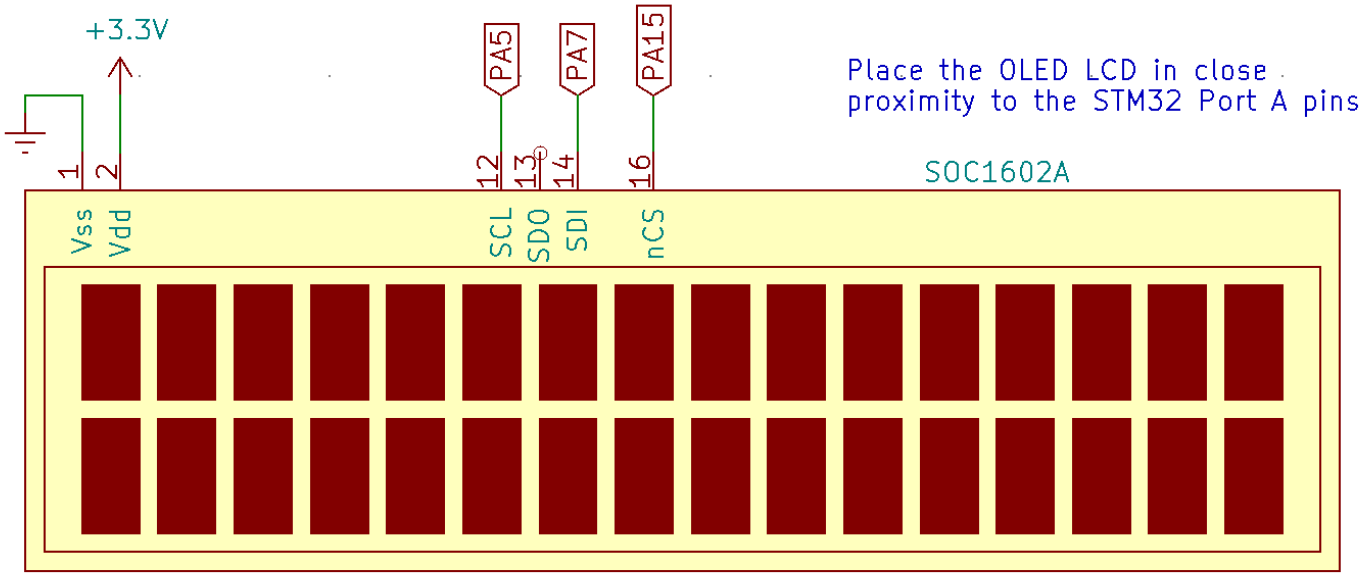
1.3: OLED Hardware Configuration

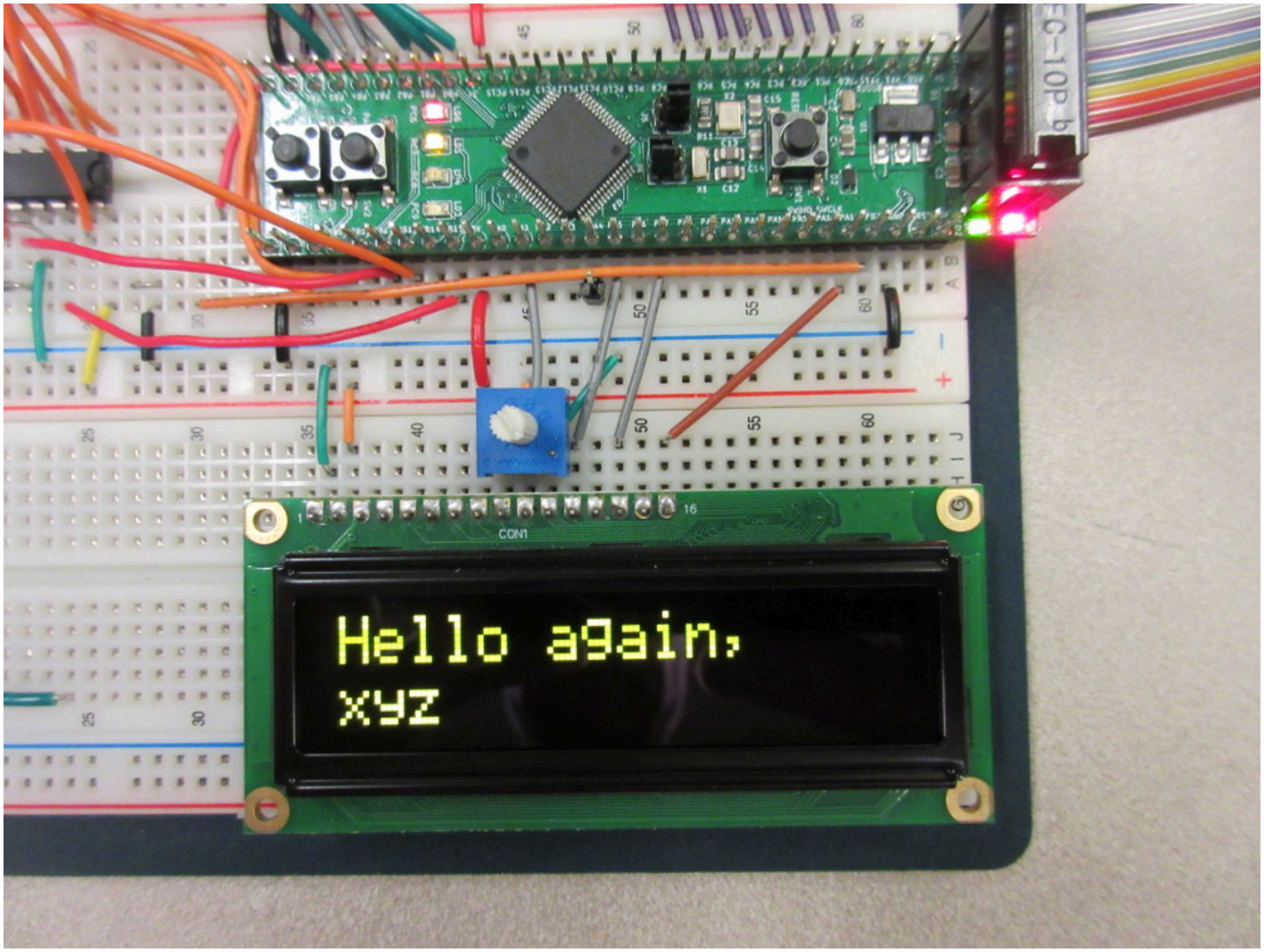
The schematic for this setup is shown below.



> **NOTE:** These are intended to replace the GPIO pins on your development board. You can treat PB0-PB7 as QA-QH on the rightmost 595 chip, and then PB8-PB10 as QA-QC on the leftomost 595 chip.

Page 4 of the datasheet for the SOC1602A OLED LCD display describes the pins for the serial interfaces. Like many SPI devices, the documented pin names differ from the canonical description of the SPI protocol. Pin 12 (SCL) is the SPI clock. Pin 14 (SDI) is the MOSI signal. Pin 16 (/CS) is a "negated chip select", which is connected to NSS. Figure 3 describes the connection to the STM32F091 development board.





2: Software Emulation Using Bit-Banging (25 Points total)

For this experiment, you will write the subroutines to write to the shift registers to drive the 7-segment LED displays and to initialize and write to the SOC1602A OLED LCD display through the SPI interface and using DMA.

Download the template folder from lab website and import it into SystemWorkbench.

Several subroutines you write will be copied directly from those you wrote in labs 6 and 7. Be sure that you implemented them correctly.

2.1: Driving an SPI interface by “bit-banging”

The SPI interface is simple enough that it can be driven by setting individual GPIO pins high and low — a process known as bit-banging. This is a common method for using SPI with most microcontrollers because no specialized hardware is needed to do so. For our first implementation, we will bit bang the SPI protocol for the shift registers connected to the 7 segment LED array.

2.2: `setup_bb()`

Write a C subroutine named setup_bb() that configures GPIO Port B for bit-banging the 7 segment LED displays. To do so, set these pins in their respective ways:

- PB12 (Represents NSS)
- PB13 (Represents SCK)
- PB15 (Represents MOSI)

for general purpose output (not an alternate function). Initialize the ODR so that NSS is high and SCK is low. It does not matter what MOSI is set to.

2.3: `small_delay()`

A C subroutine named `small_delay()` that calls the `nano_wait()` subroutine is provided for you in `main.c`. When you are starting out bit-banging an interface, it is helpful to have a uniform small delay that can be made arbitrarily large. The parameter for `nano_wait()` is number of nanoseconds to spend spinning in a loop. Having `small_delay()` allows you to always call `nano_wait()` with the same value. If things do not work, it helps to slow down all elements of the protocol. You can make the value very large so that you can debug it. We start out with a large value like 50000000 (50 ms). In practice, you would gradually reduce it to see what works, but you will find that, when driving the 7-segment LED array with the shift registers, you will not need any delay at all. Once the circuitry and software is working you can comment out the `nano_wait()` call in small delay.

Consider what the SPI protocol does. The delays are to be inserted between transitions of NSS-MOSI-SCK-MOSI-SCK-...-NSS. The SPI interface will certainly work at extremely low speeds. When starting to develop any hardware interface, it is helpful to be able to see things happening in slow-motion. Once it works, increase the speed.

2.4: `bb_write_bit()`

Write a C subroutine named `bb_write_bit()` that accepts a single integer parameter, which should always be either 0 or non-zero, and implements the following pseudocode:

```
void bb_write_bit(int out)
{
  // Set MOSI to 0 or 1 based on out
  small_delay();
  // Set SCK to 1
  small_delay();
  // Set SCK to 0
}
```

2.5: `bb_write_halfword()`

Write a C subroutine named `bb_write_halfword()` that accepts a single integer parameter and implements the following pseudocode:

> **NOTE:** You do not have to do this sequentially. You could do it in a loop form.
>
> ```
> void bb_write_halfword(int message)
> {
>   // Set NSS to 0
>   // Call bb_write_bit() for bit 15
>   // Call bb_write_bit() for bit 14
>   ...
>   // Call bb_write_bit() for bit 0
>   // Set NSS to 1
> }
> ```

> **NOTE:** If you're new at this, remember that you can use the `>>` operator to shift values to the right by an arbitrary amount. Then use the `&` operator to AND the result with a 1 to isolate one bit. If you don't feel like expressing this with a loop, you may make sixteen separate calls to `bb_write_bit()` in the proper sequence.

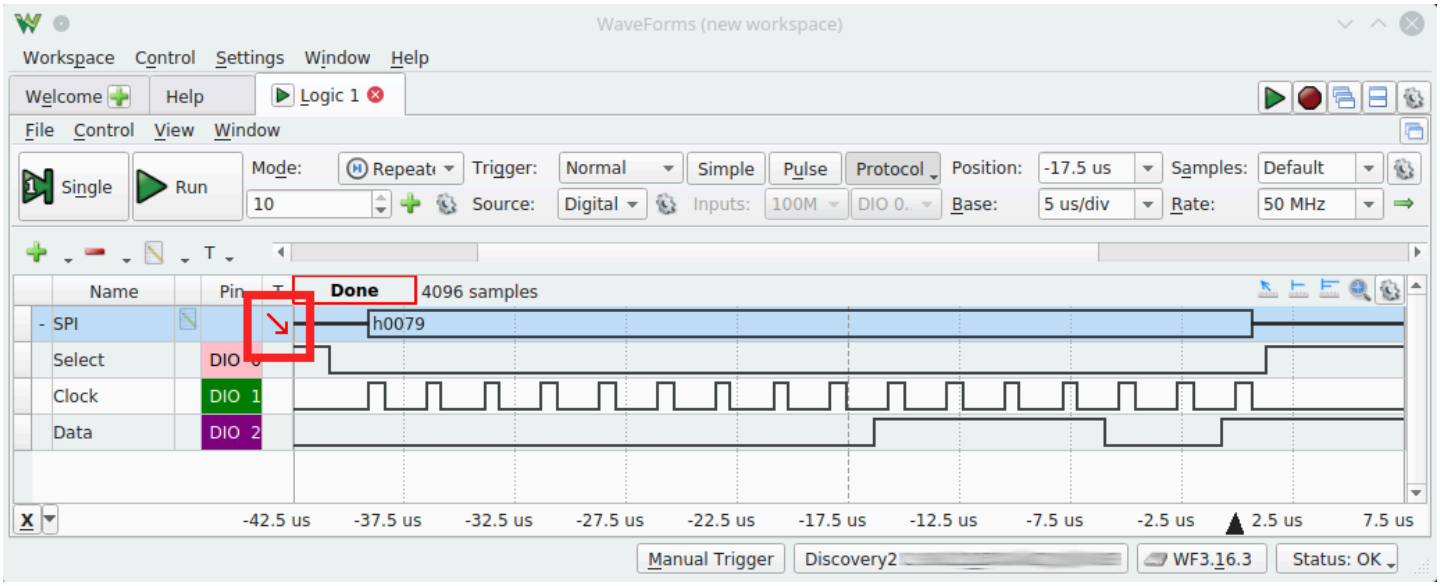
2.6: Demonstrate Bit Banging

Uncomment the `#define BIT_BANG` stanza in `main.c`. This will invoke your `setup_bb()` subroutine to configure the pins as outputs and then call your `bb_write_halfword()` repeatedly to show the entire array.

The display should say “ECE 362”, though it will iterate through each display element very slowly. Make sure that each digit displays correctly. Then reduce the number of nanoseconds that small_delay waits until the display is smooth. <!— Comment out the `nano_wait()` call and use your AD2 to capture a trace of the SPI protocol. To do so,

- Invoke the “Logic Tool” and add a new SPI bus in the signal list.
  - Use DIO 0 for the “Select” (NSS) signal.
  - Use DIO 1 for the “Clock” (SCK) signal.
  - Use DIO 2 for the “Data” (MOSI) signal.
- Click the box in the “SPI” row in the “T” column (see the thick red box in the picture below.) to set a value-based trigger. Set the trigger as follows:
  - Trigger: Value
  - Value: `h0079`
  - Place: Any
- Click the “Single” button to capture the transaction

While your program is running, capture a single trace of the protocol transaction that writes the “E” pattern to digit 0 of the display. It should look like the image below.



–>

Have a TA check you off for this step (TA Instructions: Confirm that the bit-bang subroutines are being called in `main()`. Check that display flashes one digit at a time. Ask the student to comment the `nano_wait()` call in `small_delay()` and confirm that the display is smooth.)

2.6.1: Debug Bit Banging

If your display is not working, check:

- Make sure that each pin is set to output mode on the debugger (0x01 per each pin)
- Make sure that you’re using the correct ODR (or BSRR) pins. You can check this by stepping through each line in your code with the debugger and seeing what pins change on the ODR.
- Set up your AD2 in a similar fashion as the next task. Do not use the value trigger, instead clock on the box to the right of DIO 1 to initialize a simple trigger. Zoom in/out until you can see the individual codes being sent out on each line. Is the NSS pin being deasserted when data is being sent, and reasserted after the frame completes? Is the SCK pin sending out a clock signal properly? Is the MOSI line outputting the expected data frames? If any of these are not true, then chances are the ODR is not being set correctly. There is a slight chance that your for loop is not passing the correct data, in which case you will need to step into the function with the debugger and look at the variable tab to find what variables are being passed in.

3: Hardware-Based Serial Peripheral Interface (SPI) (25 points total)

The STM32 has two SPI channels that do the work of the subroutines you just wrote in a hardware format instead of a software format, which saves valuable processing time. Implement the following subroutines to initialize and use the SPI2 interface.

3.1: Background

You should now comment out `BIT_BANG` stanza and and uncomment the lines below it for the `SPI_LEDS` stanza.

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 $\bar{SS}$ 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. You'll see this again in lab 11) 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 SOC1602A OLED LCD 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.

To have a more concise definition, here are the important registers and bit fields for this lab listed out: `SPIx->CR1:`

- `SPE:` The enable bit for the SPI peripheral.
- `BR[2:0]:` Baud rate selection field.
  `SPIx->CR2:`
- `DS[3:0]:` The data size field.
- `TXEIE:` Transmitter empty interrupt enable.
- `RXNEIE:` Receiver not empty interrupt enable.
- `TXDMAEN:` Transmitter DMA request enable.
- `RXDMAEN:` Receiver DMA request enable.

> **WARNING:** As mentioned above, if you set the DS[3:0] field to 0x000 (i.e. clear it out, reset, etc.), it will default back to its normal value.

3.2: `init_tim15()` and DMA setup/enable functions (10 points)

Copy: - `init_tim15()` - `setup_dma()` - `enable_dma()`

from lab 8 or lab 9. You must make one change: `setup_dma()` should configure the DMA channel to write to `SPI2->DR` instead of `GPIOB->ODR`.

Also copy the `TIM7` setup (`init_tim7()`) and ISR function to `main.c` to read the keypad.

3.3: `init_spi2()` (15 points)

Write a C subroutine named `init_spi2()` that initializes the SPI2 subsystem and connects its NSS, SCK, and MOSI signals to pins PB12, PB13, and PB15, respectively.

This subroutine should first configure these pins, and set them to use the alternate functions to be used by SPI. Remember to enable GPIOB clock in RCC.

The subroutine should then configure the SPI2 channel as follows:

- Ensure that the `CR1_SPE` bit is clear. Many of the bits set in the control registers require that the SPI channel is not enabled.
- Set the baud rate as low as possible (maximum divisor for BR).
- Configure the interface for a 16-bit word size.
- Configure the SPI channel to be in "master mode".
- Set the SS Output enable bit and enable NSSP.
- Set the `TXDMAEN` bit to enable DMA transfers on transmit buffer empty
- Enable the SPI channel.

Consider what you've achieved here...

In labs 8 and 9, you used DMA to copy 16-bit words into an 11-bit GPIO ODR to drive the 7-segment displays. Now you've moved the 7-segment driver to a synchronous serial interface. Since there is a built-in peripheral that serializes bits, you can now do the same thing, with the same software. The only things you had to change were the initialization code to set up the SPI peripheral instead of parallel GPIO lines, and the output target of the DMA channel.

3.4: Demonstrate SPI2 and 7-Segment Display

Have a TA check you off for this section (TA Instructions: Check that the correct functions are commented/uncommented in main and that the 7-segment display shows ECE 362, and that pressing keys on the keypad shows up on the same display.).

3.4.1: Debugging the SPI2 Channel

If your display is not working, check these items inside of the debugger:

- Are you setting the respective pins to Alternate Function mode? If you check inside of the debugger, each used pin should read 0x10.
- Because we are setting alternate functions in these pins, the AFRH (AFR[1]) should be reading the alternate function that can be used for these pins. These can be found in the STM32F0 datasheet, not the family reference manual.
- Are you turning on the SPI2 RCC clock? Every peripheral in the STM32F0 has a clock associated with it in the RCC. Assume that these peripherals will not work unless you turn their clock on.
- Are you turning off the SPE bit before configuration and turning it back on after?
- Are your CR1 and CR2 registers coming out to expected values? Make sure to put a breakpoint after your code, run the debugger to that breakpoint, and check the values in those registers. CR1 initializes to 0x0000 and CR2 initializes to 0x0700, so most of the things that are turned on are bits that you turn on in your code. If they are coming out different than what you expect, check through your code to make sure you are correctly setting and clearing bits.

4: Trigger DMA with SPI_TX (15 points)

An SPI peripheral can be configured to trigger the DMA channel all by itself. No timer is needed! It just so happens that the SPI2 transmitter triggers the same DMA channel that Timer 15 triggers. (See the entry in Table 32 for SPI2_TX.) The only additional work to do is to configure the SPI peripheral to trigger the DMA. The subroutines to do this are provided for you. They are named `spi2_setup_dma()` and `spi2_enable_dma()`. They call the code you wrote in `setup_dma()` and `enable_dma()`.

Consider what you've achieved here...

You now have a system that can automatically transfer 16-bit chunks from an 8-entry array to the LED displays. Instead of needing a timer to trigger the DMA channel, the SPI peripheral does that automatically. As soon as one word is transferred, it requests the next word. The SPI system can run fast enough that words can be transferred 1500000 times per second. That would make the digits change faster than the 74HC138 could settle, and the letters would blur together. By setting the SPI clock rate as low as possible, 187.5 kHz, it means that each 16-bit word is delivered 187500/16 = 11719 times per second. You may notice that this makes the digits a little dimmer than they were when driven by the timer.

4.1 Demonstrate Trigger DMA with SPI_TX

Comment the `SPI_LEDS` stanza and uncomment the `SPI_LEDS_DMA` stanza. **Have a TA check you off for this section** (TA Instructions: Run this in the System Workbench debugger and confirm that Timer 15 is not initialized.)

4.2 Debugging the DMA

This task is a very similar task to what we've done in the past with the course. You can copy and paste on of the previous DMA setup code blocks that you've written, change the channel to the one needed for SPI2TX, and update the CMAR, CPAR, and CNDTR to the new needed values for the SPI DR, msg array, and msg array size.

5 Using SPI to drive the OLED LCD (10 points total)

Once you have some experience configuring an SPI peripheral for one purpose, it is easy to set up another. We will now use the SPI1 peripheral to drive the LCD OLED display. This will not interfere with the operation of the SPI2 peripheral. You can use them both, simultaneously, for different devices.

5.1 `init_spi1()`

Write a C subroutine named `init_spi1()` to configure the SPI1 peripheral. The configuration for SPI1 is similar to SPI2 from before with a few key differences:

- Configure NSS, SCK, MISO and MOSI signals of SPI1 to pins PA15, PA5, PA6, and PA7, respectively.
- Configure the SPI register for 10-bit data size.
- Enable it.

> **WARNING:** When you're setting the data size register, you need to SET the bits you want, before you RESET the bits you don't want, in order to specify the 10-bit size. This has to do with a weird quirk with how the chip sets its default values that we can't seem to get around.

> **WARNING:** Similarly with last lab, if you configure Port A's AFR incorectly, you can turn the programming pin off.

5.2 `spi_cmd()`

A C subroutine named `spi_cmd()` is provided for you. It accepts a single integer parameter and does the following:

- Waits until the `SPI_SR_TXE` bit is set.
- Copies the parameter to the `SPI_DR` .

That's all you need to do to write 10 bits of data to the SPI channel. The hardware does all the rest.

5.3 `spi_data()`

A C subroutine named `spi_data()` is provided for you. It accepts a single integer parameter, and does the same thing as spi_cmd(), except that it ORs the value 0x200 with the parameter before it copies it to the SPI_DR. This will set the RS bit to 1 for the 10-bit word sent. For instance, if you call the subroutine with the argument 0x41, it should send the 10-bit value 0x241 to the SPI_DR. This will perform a character write to the OLED LCD.

5.4 `spi1_init_oled()`

A C subroutine named `spi1_init_oled()` is provided for you. It performs each operation of the OLED LCD initialization sequence:

- Use `nano_wait()` to wait 1 ms for the display to power up and stabilize.
- `cmd(0x38);` // set for 8-bit operation
- `cmd(0x08);` // turn display off
- `cmd(0x01);` // clear display
- Use `nano_wait()` to wait 2 ms for the display to clear.

- `cmd(0x06);` // set the display to scroll
- `cmd(0x02);` // move the cursor to the home position
- `cmd(0x0c);` // turn the display on

5.5 `spi1_display1()`

A C subroutine named `spi1_display1()` is provided for you. It accepts a `const char *` parameter (also known as a string) and does the following:

- `cmd(0x02);` // move the cursor to the home position
- Call `data()` for each non-NUL character of the string.

5.6 `spi1_display2()`

A C subroutine named `spi1_display2()` is provided for you. It accepts a `const char *` parameter (also known as a string) and does the following:

- `cmd(0xc0);` // move the cursor to the lower row (offset 0x40)
- Call `data()` for each non-NUL character of the string.

The display hardware allows for scroll buffers for each line, so the beginning of the second line is actually position 64 (0x40). That offset is combined with the "Set DDRAM address" command (0x80) to position the cursor.

5.7 Demonstrate the SPI OLED Display (10 points)

At this point, you should be able to comment and previous stanzas and uncomment the `SPI_OLED` stanza. It uses the `init_spi1()` you wrote along with the support code to initialize and write things to the OLED display.

(You can either add an empty infinite for loop, or comment out `game` in `main.c` to keep the display from being overwritten.)

Have a TA check you off for this part (TA Instructions: the OLED display should display "Hello again,\n[Their login]")

5.7.1 Debugging the SPI1 Channel

If your OLED isnt working right away, check these things:

- Are you setting your GPIOA pins to alternate function mode (0x10)? A common problem here is accidentally editing PA13 and PA14. If you edit these, it will make your debugger stop working. If your debugger suddenly is saying "device not detected," then you are accidentally changing PA13 and PA14. Fix your code, and hold the reset button while you're programming it.
- Are you turning on the RCC clock to SPI1?
- Are you correctly setting the data size to 10 bits? In an earlier help section, it's mentioned that CR2 intitializes to 0x0700. This means you can't just OR in 0x0900. Further, you cannot clear out 0x0f00 because it returns the SPI channel to a default state. You must do a set and clear operation to get the data size to the correct value for this case.
- If your code looks correct, your data size is set to ten bits, and it looks like everything is working on the AD2, you may need to reset your OLED display. These are pretty cheap displays that aren't very smart, so it's easy to confuse them and make them quit working. This can be fixed by unplugging its wiring and plugging it back in, or simply holding down the reset button, removing anything providing power to the circuit, then putting it back in and releasing the reset button.
- I've noticed that there are some edge cases where the OLED will not work with this task, but it will work with the next task. This is only happening with a small amount of students, so don't immediately assume it's your problem. Anyways, I'm chalking this specific problem up to the quality of the OLED displays. If you're confident that your SPI1 bus is working correctly, the output on the AD2 is correct, and your wiring is correct, skip to the next task and it might start working. If it still does not work after the next task, something went wrong that you missed here, and you'll need to look over this section again.

6 Trigger SPI1 DMA (25 points total)

After going through the initialization process for the OLED display, it is able to receive new character data and cursor placement commands at the speed of SPI. In this step, we will configure SPI1 for DMA. Instead of using the `spi1_display1()` and `spi1_display2()` subroutines to place the cursor and write characters, they will be continually copied from a circular buffer named `display[]`. It consists of 34 16-bit entries. Element 0 of the array holds the 10-bit command to set the display cursor at position 0 (the beginning of the top line) of the display. This is the "home command" for the display. The next 16 entries are characters that are copied into the first line of the display. Element 17 of the array holds the 10-bit command to set the cursor position to the beginning of the second line. It is followed by 16 more characters. (Remember that each character must be sent with the 0x200 prefix.)

6.1 `spi1_setup_dma()` (10 points)

Write a subroutine named `spi1_setup_dma()`. It should circularly copy the 16-bit entries from the `display[]` array to the `SPI1->DR` address. You should also configure the SPI1 device to trigger a DMA operation when the transmitter is empty.

Which DMA channel should you use for this step? You know how to determine this.

6.2 `spi1_enable_dma()` (15 points)

Write a subroutine named `spi1_enable_dma()`. The only thing it should do is enable the DMA channel for operation.

The array is set up, and circular DMA transfer is triggered by `SPI1_TX` to write it to the `SPI1->DR` address. Thereafter, the display is effectively "memory-mapped". As characters are written to the array values, they are quickly copied to the display. This allows you to easily create very complicated patterns and animations on the display.

An example of subroutines to update the `display[]` array is provided in the `support.c` file. Two subroutines named `spi1_dma_display1()` and `spi1_dma_display2()` do the work of converting a string of at most 16 characters to 16-bit entries at the right location in the `display[]` array. These are used for the game at the end of the lab experiment.

6.3 Demonstrate your work

Uncomment the `SPI_OLED_DMA` stanza and comment all other stanzas. You should see the message encoded into the `display[]` array. (TA instructions: Once this works, try commenting all stanzas so that the game is invoked.)

6.3.1 Debugging the DMA

This task is a very similar task to what we've done in the past with the course. You can copy and paste on of the previous DMA setup code blocks that you've written, change the channel to the one needed for SPI1TX, and update the CMAR, CPAR, and CNDTR to the new needed values for the SPI DR, display array, and display array size.

7. Play the game

If you have properly implemented all of your subroutines, you should be able to comment all of the test stanzas and play the game provided for you. This game provides several examples:

- It starts a free-running timer counter (TIM17) while it waits on the player to press a button. Once a button is pressed, the value of the counter is read and used as a seed for the random number generator. In doing so, each game will be different.
- This is an example of how to wait for key events and update multiple display devices (the 7-segment displays and the OLED display).
- All of the game logic is encoded into the Timer 17 ISR.
- The `ARPE` bit of the timer is set. This allows us to safely modify the ARR during the operation of the game. As the game progresses, the action gets faster.
- A critical section is set up where all interrupts are temporarily disabled when a key is pressed. In this way, the display array can be safely updated rather than have two things modify it in an overlapping fashion.

It is an easy game to play. Use the A and B keys to move the '>' character on the left of the display. Use it to hit as many 'x' characters as you can. Each time you hit an 'x', you gain a point. Each time an 'x' goes by you, you lose a point. As your score gets higher, the

game goes faster. If you can reach 100 points, you win.