CS466 Lab 5 – SPI Slave Device.
Two-week lab, due Wednesday 4/9/2025

Use provided lab format on Blackboard

Notes:
- This lab will require two target Pico boards, I recommend that you work with a partner but it's not a requirement.
-

Overview:
- In this LAB you will be implementing a SPI slave device on one Pico board, you will run a master derived from lab4 program on the second Pico.
- I recommend that you start with your working Bit-Banged master from lab4.
- The contrived slave device interface is defined as a remote interface to the Pico board. Similar to the GPIO expander it will have a register interface and is able to interrupt the master.
- Your Master program will be quite trivial, the complexity of the slave will occupy your time. Get the master running a simple sequence and then use it to drive the slave
- Getting the slave top work at as high of a speed is part of the lab but keep the speeds slow at first paying attention to the protocol and implementation first.

Objective:
- Understand that a SPI slave device is more complex than an SPI master.
- Build a working interrupt driven system peripheral device.
- Detect and service the remote device in an interrupt.
- Implement an arbitrary embedded system behind a de-facto industry standard interface.
- Implement a register-based SPI interface, Similar in concept to the GPIO expander
- Implement an interrupt driven state-controlled system.
- All the slaves in the class should match the following specification. In this way any master should be able to operate any slave.
- Implement a reset solution using the microcontroller watchdog hardware so the device can recover if a firmware error occurs.

Slave Device Specification:

- The slave will use a 2 byte command sequence for all operations (contrast to the GPIO expander 3-byte protocol used in lab4)
- Slave will be setup to control a common cathode (common ground) RGB LED and two switches using GPIO Pins.
- Byte 1, is organized as 2, 4 bit fields.
  - Bits 4-7: Command
    - 0x0b is defined as a write
    - 0x0a is defined as a read
    - 0x0c is a special test command that will hang the slave in a while(1); infinite loop behavior.. We will send this when we want to test the slave watchdog.
    - All other patterns are reserved for future use
  - Bits 0-3: Register Address of a device 8 bit register
- Byte 2, is always register data either to be written in the case of a write or to be read in the case of read.
  - NOTE: Special consideration must be taken into account on a read operation to have correct MISO data reg-MSB on the data line when the master raises the clock line.

- This Slave will contain 3 registers.
    - LED_REG:
        - An 8 bit Read/Write register with the following bits defined
        - Address is 0x01
        - B7-B3 – unused, may be set to 1 or 0
        - B2 – Setting to 1 will illuminate the slave green LED. (0 ➔ 0ff)
        - B1 – Setting to 1 will illuminate the slave red LED. (0 ➔ 0ff)
        - B0 – Setting to 1 will illuminate the slave blue LED. (0 ➔ 0ff)
        - Data written to this register may be read back

    - SW_REG:
        - An 8 bit Read/Write register with the following bits defined
        - Address is 0x02
        - B3 indicates if SW2 caused the interrupt
        - B2 Indicates if SW1 caused the interrupt
        - B1 reflects current state of slave SW2 (depressed is 1)
        - B0 Reflects current state of slave SW1 (depressed is 1)

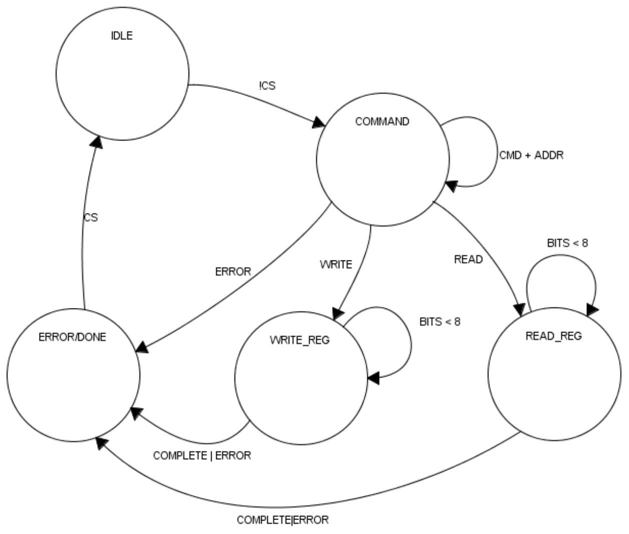
    - INT_REG:
        - An 8 bit Read/Write register with the following bits defined
        - Address is 0x03
        - B3 Enables SW2 Interrupt
        - B2 Enables SW1 Interrupt
        - B1 Acknowledges SW2 Interrupt
        - B0 Acknowledges SW1 Interrupt

Slave example operations:

- Turn on and off the green LED
    - Since the green LED is managed by bit-02 of the LED_REG. To turn the LED on you will write a 0x04 to LED_REG at address 01.
    - To turn the LED off write 0x00 to LED_REG.

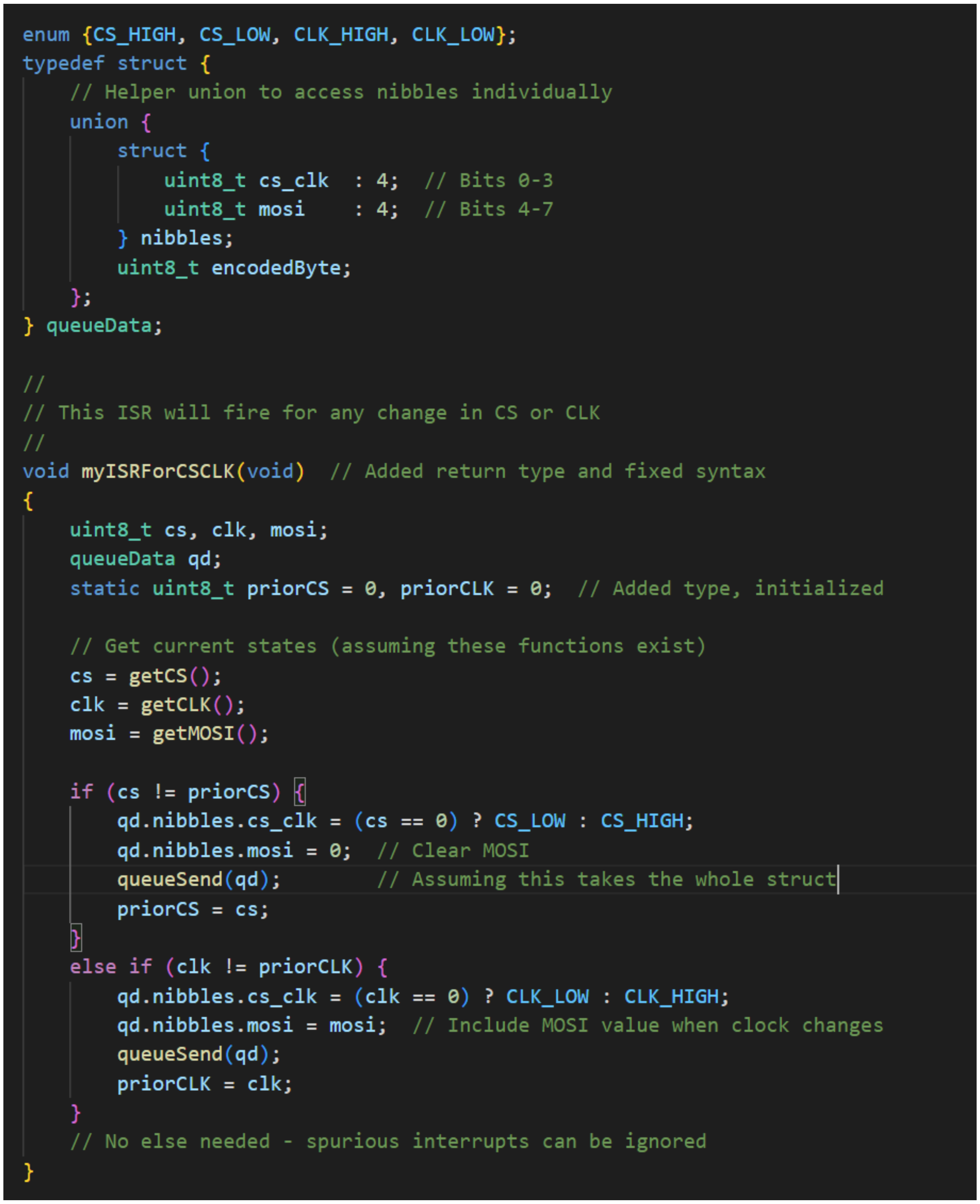
Prior to Lab:

- Using your prior labs you have working examples of interrupts, queues, an SPI master, all of the mechanical tools that are required to complete the lab
- I have covered briefly the notion of event driven programming in class. I strongly encourage you to follow that model here. Whenever operating a state machine via software it is usually a simple matter to navigate and change states using event.
- Work on a state machine that will allow your slave to operate.



- We know from lab4 that inbound SPI events can be boiled down to a small set of operations…
    - CS-HIGH
    - CS-LOW
    - CLK-HIGH
    - CLK-LOW
    - MOSI-HIGH
    - MOSI-LOW
- It would be possible to run the entire state machine in your ISR but I'd prefer that your ISR manufactures the data above and sends it in a queue to be processed by a state machine task.
- In another sense the 6 possible events are not completely independent. We generally only care about MOSI on a CLK low->High transition. I'd recommend that you setup your queue to transfer a byte on information and break the bytes into two nibbles that contain CS and CLK data in one nibble and the inbound data (MOSI) in the other. We do want to keep CLK-High and MOSI well correlated.

- You can use some ‘C’ foolery to keep this from being a bit masking/shifting exercise with bitslice strucures..   As in



```c
enum {CS_HIGH, CS_LOW, CLK_HIGH, CLK_LOW};
typedef struct {
    // Helper union to access nibbles individually
    union {
        struct {
            uint8_t cs_clk  : 4;  // Bits 0-3
            uint8_t mosi    : 4;  // Bits 4-7
        } nibbles;
        uint8_t encodedByte;
    };
} queueData;

//
// This ISR will fire for any change in CS or CLK
//
void myISRForCSCLK(void)  // Added return type and fixed syntax
{
    uint8_t cs, clk, mosi;
    queueData qd;
    static uint8_t priorCS = 0, priorCLK = 0;  // Added type, initialized

    // Get current states (assuming these functions exist)
    cs = getCS();
    clk = getCLK();
    mosi = getMOSI();

    if (cs != priorCS) {
        qd.nibbles.cs_clk = (cs == 0) ? CS_LOW : CS_HIGH;
        qd.nibbles.mosi = 0;  // Clear MOSI
        queueSend(qd);        // Assuming this takes the whole struct
        priorCS = cs;
    }
    else if (clk != priorCLK) {
        qd.nibbles.cs_clk = (clk == 0) ? CLK_LOW : CLK_HIGH;
        qd.nibbles.mosi = mosi;  // Include MOSI value when clock changes
        queueSend(qd);
        priorCLK = clk;
    }
    // No else needed - spurious interrupts can be ignored
}
```

Lab Work

1. ☐ Connect 4 SPI wires from your master to slave Pico boards. You can use the same master-pico connections as from lab4. Be careful because the slave pico will operate differently having 3 inputs (CS, CLK and MOSI) and a single output (MISO).

2. ☐ Write a Master program that toggles the green LED at 1 Hz in your heartbeat task. Add some delays in your SlaveWriteByte() and SlaveTransfer() so that there is at least 10 ms between all transitions.

3. ☐ Setup your slave board so that you have USB/Serial running and a green heartbeat light flashing for now. (You will disable the slave heartbeat later)

4. ☐ implementing your slave first as truly interrupt driven is difficult with the tools that we have. I suggest that you start with a polled version that can be converted to interrupt driven easily. Note that if you want to poll the SPI lines you need to poll significantly (say 10x) faster than the SPI clock signal.

a) At 1ms interval watch for changes on the 2 slave control lines. This is an order of magnitude faster than the throttled master and should work ok. Initially you may wish to start at a much slower speed.
b) Create a queue that will accept a state change as they are detected.
c) Detect CS_HIGH, CS_LOW, CLK_HIGH, CLK_LOW and stuff them in the queue when detected (only post changes to the queue)
   1. CS_HIGH = 0x00
   2. CS_LOW = 0x01
   3. CLK_HIGH = 0x03
   4. CLK_LOW = 0x04

   These should be all the transitions that you need to detect in a future interrupt handler and can stuff the same bytes in a queue.

5. ☐ Implement a processing thread that will block while listening to the queue.

6. ☐ Depending on the state that your slave is operating in you will have specific behavior. I strongly you implement the slave state diagram that I presented in class. It may not be the fastest but if we're all on the same page the lab will go better. I am mostly focusing on a system that you can design and debug incrementally

(Agile follows the debatable mantra 1-make it work, 2-make it right, 3-make it fast. Once you have a working model you can refactor at will to improve the design, purists argue that nobody ever gets to 'make it fast')

7. ☐ Send commands over your SPI connection to the slave board to echo the RED led operation there.
a) This is a large step (sorry). I suggest you have the master maintain this simple program until you get the slave device LED_REG functionality working.

8. ☐ Device a test to see how fast you can blink the Red LED, does this max frequency make sense? Measure the maximum frequency with a scope.

9. ☐ Make sure that your LED_REG commands work for all 3 LED's. Verify that you can write and read each of the 4 registers.

10. ☐ Change your slave so that the CLK and CS lines trigger an interrupt on the slave. In the interrupt, determine what line has transitioned and send the transition to the queue using the xQueueSendFromISR() command. Stop running your polling task..
Your slave should still be working but now with much better speed capability.

11. ☐ Will it run faster, than in step 8? Hint, ifdef all the slave printf commands out, they are very slow. How fast will it run now? This is the downstream bandwidth of our system.

12. ☐ Using lab2 as an example, Implement the operation of the SW_REG register. My recommendation is to generate interrupts on the slave when the buttons are pressed and update the appropriate register[SW_REG] bits when the interrupts occur. Note that you will need to detect both rising and falling edges of the signal to accurately represent the current state in the address array.

13. ☐ Write a test on your Master to poll the slave button states. Verify that you can poll the states for both buttons.

14. ☐ Similar to lab4, we need to generate an interrupt request to the master Pico board when a button is pressed.

a) When the slave button interrupts fire you will need to check the bits in the _register[INT_REG] array to see if the Master has enabled interrupts.
b) If so, then set the interrupt request line to the master via a GPIO line.
c) Update B2 and B3 of SW_REG accordingly

15. ☐ The Master will require an ISR be attached to the Interrupt request line and when it fires the master will acknowledge the slave interrupt by writing to the INT_REG register on the slave.

16. ☐ What is the latency of our connected system? By that I mean how long does it take from the time a button is pressed until the Master ISR get's context? You can use a logic analyzer or a scope with a couple GPIO's to answer the question..

17. ☐ Use the Pico SDK to register a watchdog interrupt and potential system reset if the slave device does not keep resetting the watchdog timer. The watchdog timer will be reset in the _heartbeatTask().

18. ☐ I Write up your lab using the lab format provided on Angel. Include your program as a fixed spaced (`I recommend Lucida Console`) addendum to your lab. I will cut points for proportionally spaced code pasted in the end of the lab. (2-3 pages)