Contents

1 Multiplexing pins

In this lab, we'll practice multiplexing GPIO pins using a shift register and using an I2C multiplexer.

1.1 Notes

- In this lab, you will create some breadboard circuits with exposed pins and wires. Please be especially careful not to accidentally create connections that shouldn't be connected (e.g. short circuits). Also, check your work carefully before connecting any breadboard circuit to a board, to avoid damaging the board.
- Read each subsection of this lab manual in its entirety before you start following the instructions in it. Some instructions are modified by explanations that come afterwards.
- Although you may work with a partner, this collaboration is limited to discussion. Your partner is not allowed to construct or modify your circuit, log in to your Pi, or run commands or write code on your Pi. Similarly, you are not allowed to do these things for your partner. (You *are* encouraged to collaborate by screen-sharing or showing video of your circuit to debug and discuss problems together.)
- For your lab report, you must submit data, code, and screenshots from your own experiment. You are not allowed to use your lab partner's data, code, or screenshots.
- For any question in the lab report that is marked "Individual work", you should *not* collaborate with your lab partner or anyone else (even via discussion). You can use your notes, the lab manual, or the lecture slides and video to help you answer these questions.

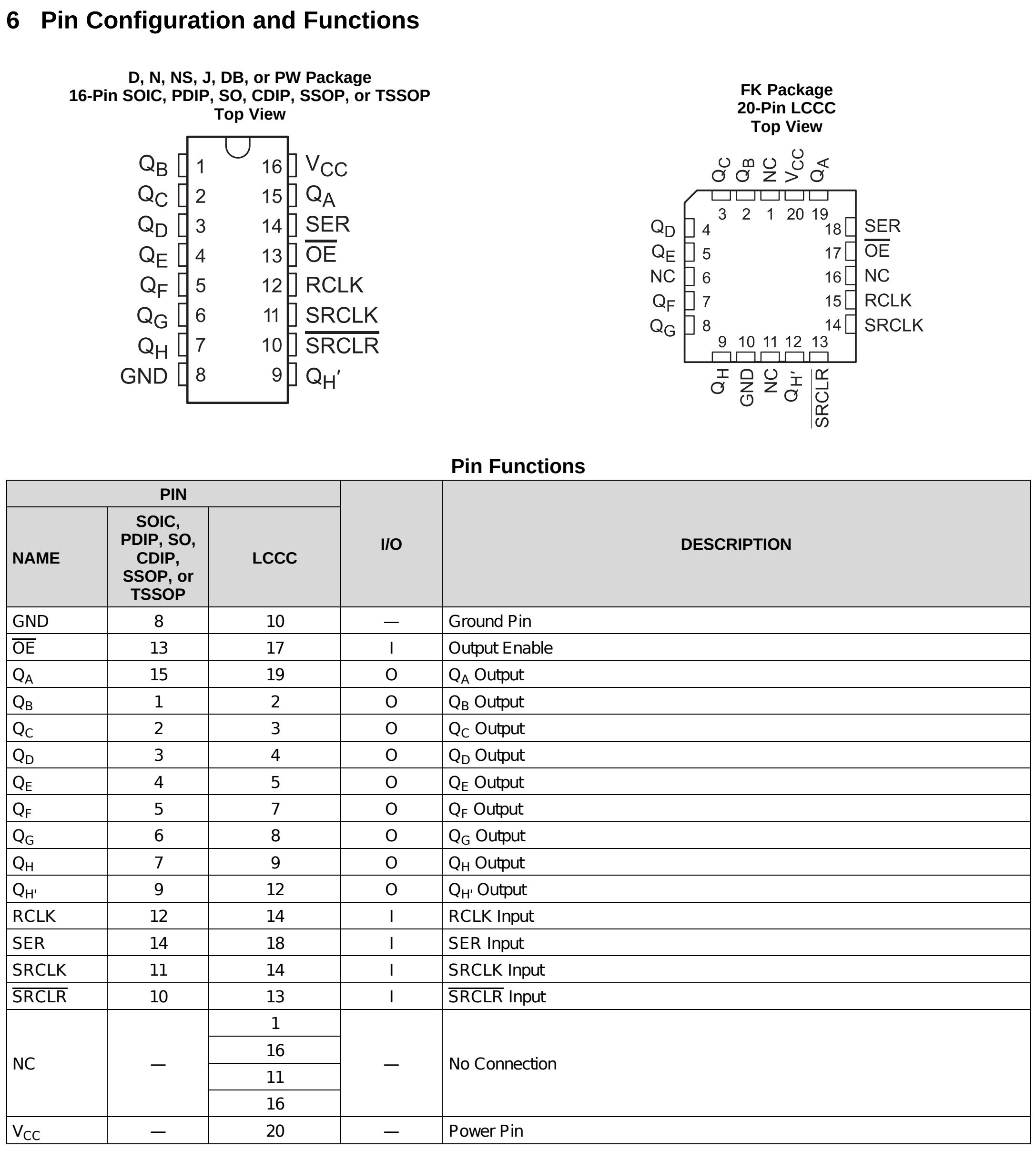
1.2 Parts

In this experiment, we will use an ADC to read an analog input. You will need:

- A Pi, SD card, and power supply
- Breadboard and jumper cables
- Digital multimeter
- SN74HC595N shift register
- 8x 2mm LEDs
- 8x 470Ω resistor
- MCP23008 I2C pin multiplexer
- 3x push button switches
- Other assorted resistors

1.3 Shift register

Place your shift register into a breadboard with a “ravine” in the middle, so that the pins on either side are not connected. Use the “Pin Configuration and Functions” section of the datasheet to identify the important pins.



6 Pin Configuration and Functions

D, N, NS, J, DB, or PW Package
16-Pin SOIC, PDIP, SO, CDIP, SSOP, or TSSOP
Top View
Q_B 1
Q_C 2
Q_D 3
Q_E 4
Q_F 5
Q_G 6
Q_H 7
GND 8
16 V_CC
15 Q_A
14 SER
13 OE
12 RCLK
11 SRCLK
10 SRCLR
9 Q_H′
FK Package
20-Pin LCCC
Top View
Q_C 3
Q_B 2
NC 1
V_CC 20
Q_A 19
Q_D 4
Q_E 5
NC 6
Q_F 7
Q_G 8
18 SER
17 OE
16 NC
15 RCLK
14 SRCLK
9 Q_H
10 GND
11 NC
12 Q_H′
13 SRCLR

Pin Functions

PIN			I/O	DESCRIPTION
NAME	SOIC, PDIP, SO, CDIP, SSOP, or TSSOP	LCCC		
GND	8	10	—	Ground Pin
$\overline{OE}$	13	17	I	Output Enable
Q_A	15	19	O	Q_A Output
Q_B	1	2	O	Q_B Output
Q_C	2	3	O	Q_C Output
Q_D	3	4	O	Q_D Output
Q_E	4	5	O	Q_E Output
Q_F	5	7	O	Q_F Output
Q_G	6	8	O	Q_G Output
Q_H	7	9	O	Q_H Output
$Q_{H'}$	9	12	O	$Q_{H'}$ Output
RCLK	12	14	I	RCLK Input
SER	14	18	I	SER Input
SRCLK	11	14	I	SRCLK Input
$\overline{SRCLR}$	10	13	I	$\overline{SRCLR}$ Input
NC	—	1	—	No Connection
		16		
		11		
		16		
V_{CC}	—	20	—	Power Pin

Figure 1: Shift register pins and functions.

Connect your shift register as described below:

- First, put the shift register in the breadboard, and set up the LEDs with their current-limiting resistors. Connect the QA-QH outputs of the shift register to the LEDs, in order of breadboard position.
- Then, connect the VDD pin of the shift register to GND and the VCC pin to 3.3V.
- The OE (output enable) pin on the shift register, which is active LOW, should be tied to ground to permanently enable the outputs.
- The SRCLR (shift register clear) pin on the shift register, which is also active LOW, is tied to the HIGH voltage level so that it is permanently *not* active.
- The SI/SER (serial input) pin on the shift register is connected to BCM 10 on the Pi.
- The RCLK pin on the shift register is connected to BCM 8 on the Pi.
- The SRCLK pin on the shift register is connected to BCM 11 on the Pi.

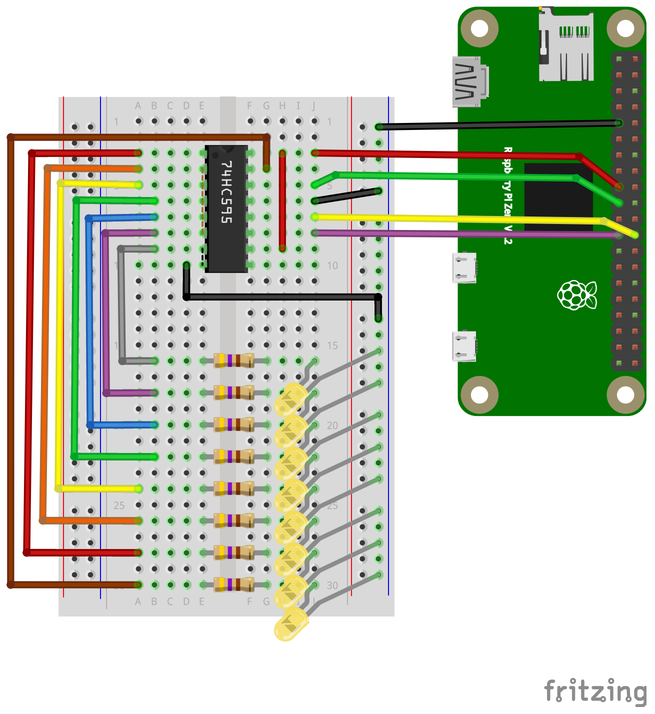


Figure 2: Shift register for control of LED array on Pi.

The basic operation of the shift register is described by "Figure 4. Logic Diagram (Positive Logic)" and "Table 1 - Function Table" on page 12 and 13 of the datasheet. The device is an 8-bit shift register, connected to an 8-bit storage register, which is connected to the 8 output pins.

- When the SRCLK line is pulsed, the bit on the SER line moves into the first stage of the shift register, and every other stage is shifted to the next stage.
- When the RCLK line is pulse, the contents of the shift register are copied to a storage register, which is connected to the output pins. Now, all of the outputs are HIGH or LOW depending on the contents of the register.

This logic is implemented in the following Python script, `shift-register-gpio.py` (also available on NYU Classes). It which will send the byte `0b10000000` on the serial input line of the shift register, and pulse first SRCLK and then RCLK after each bit. When you run the script, you should see the high bit move along the LED array from the first output to the last.

```
import RPi.GPIO as GPIO
import time

SDI   = 10  # BCM10 - physical pin 19
RCLK  = 8   # BCM8 - physical pin 24
SRCLK = 11  # BCM11- physical pin 23

CLK_TIME = 0.05

def setup():
  GPIO.setwarnings(False)
  GPIO.setmode(GPIO.BCM)   # BCM pin numbering
  GPIO.setup(SDI, GPIO.OUT)
  GPIO.setup(RCLK, GPIO.OUT)
  GPIO.setup(SRCLK, GPIO.OUT)
  GPIO.output(SDI, GPIO.LOW)
  GPIO.output(RCLK, GPIO.LOW)
  GPIO.output(SRCLK, GPIO.LOW)

def pulse_clk(clk):
  GPIO.output(clk, GPIO.LOW)
  time.sleep(CLK_TIME)
  GPIO.output(clk, GPIO.HIGH)
  time.sleep(CLK_TIME)

def send_byte(byte):
  GPIO.output(SDI,GPIO.LOW)
  GPIO.output(RCLK,GPIO.LOW)
  bitarray = [int(b) for b in format(byte, '08b')]
  for bit in bitarray:
    GPIO.output(SDI, bit)
    # when you pulse SRCLK, bits in SR move one stage over
    pulse_clk(SRCLK)
    # when you pulse RCLK, shift register contents are copied
    # to storage register, which is connected to outputs -QAQH
    pulse_clk(RCLK)
  GPIO.output(RCLK,GPIO.HIGH)
  GPIO.output(SDI,GPIO.LOW)

if __name__ == '__main__':
  setup()
  send_byte(0b10000000)
  # sleep, just so that you have a chance to see final output
  time.sleep(10)
  GPIO.cleanup()
```

Edit the line

```
send_byte(0b10000000)
```

in the Python script to send other patterns to the LED array, and observe the effect. Modify the Python script to have it send the ASCII 8-bit equivalent of the first letter of your Net ID to the LED array. Take a photograph of the LED array and a screenshot of the `piscope` output as you send this byte.

In this script, we pulsed RCLK every time we shifted the stages, so that the contents of the shift register are copied to the storage register (i.e. appear on the output) each time they are shifted. In practice, if we were using a shift register to control an LED array, we probably wouldn't want to do this. If we want to light up the last LED, we don't want to first light up every other LED before the HIGH bit reaches the last output! Instead of pulsing RCLK each time we pulse SRCLK, we would pull RCLK LOW at the beginning, send an entire byte to the shift register (defining the state of each output), and only then would we set RCLK HIGH.

Edit the script to work as described above (i.e. don't pulse RCLK after every bit - only after the entire byte has been loaded into the shift register).

Then, try running the script again, and observe how only the final output - and not the intermediate stages - appears on the LED array.

Modify the Python script to have it send the ASCII 8-bit equivalent of the first letter of your Net ID to the LED array. Take a photograph of the LED array and a screenshot of the `piscope` output as you send this byte to the shift register.

Lab report (individual work): Show your `piscope` screenshots from when you sent the first letter of your Net ID to the shift register, for *both* versions of the script - the version with RCLK pulsed after each bit, and the one with RCLK going high only at the end of the byte. Annotate the serial data line on the `piscope` screenshot to label each bit on the serial input line. Also, at each point in your screenshot where a CLK line goes from LOW to HIGH, annotate it to indicate which line of Table 1 is being applied.

Lab report (individual work): Fill in a table showing the shift register state (0 or 1) after each SER bit + SRCLK pulse when you send the first letter of your Net ID. (This will will be similar to the one on slide 16 of the lecture slides.)

Bit	QA	QB	QC	QD	QE	QF	QG	QH
0 (MSB)								
1								
2								
3								
4								
5								
6								
7 (LSB)								

Lab report (individual work): Show your photograph of the LED array displaying the first letter of your Net ID. Annotate the photograph - next to each LED, indicate the shift register output it is connected to (QA, QB, QC...) and also indicate whether it is on or off in the photograph.

After modifying the Python script to pulse RCLK only at the end of the byte, you may have looked at the `piscope` output and though to yourself, "This looks just like an SPI transaction, with the SRCLK line acting as clock, a serial data out line, and the RCLK line looks like a chip select line that is pulled low before the transaction begins and pulled high at the end of the transaction". In fact, the shift register is an SPI device, and the script we just used was a "bit banged" SPI implementation, where we manually toggled the GPIO pins to implement SPI, instead of using the build-in hardware SPI.

Alternatively, we can use the hardware SPI to send a byte to the LED array.

Since the script we just ran reconfigured the SPI pins to work in OUTPUT mode, we first have to set them back to ALT mode to use the hardware SPI device. We'll do this with the `pigs` utility in the `pigpio` library.

If `pigpiod` is not already running, then run

```
sudo pigpiod
```

Then, run

```
pigs m 8 w  # pin8 - ce0 - to write mode
pigs m 9 0  # pin9 - miso - to alt0
pigs m 10 0 # pin10 - mosi - to alt0
pigs m 11 0 # pin11 - sclk- to alt0
```

to set the pin modes. You can then use `gpio readall` and verify that the SPI pins are in ALT0 mode.

Finally, send a byte to the LED array using the hardware SPI device with the `shift-register-spi.py` script (also available on NYU Classes):

```
import spidev
import time

spi = spidev.SpiDev()
spi.open(0,0)    # SPI Port 0, Chip Select 0
spi.max_speed_hz = 7629
spi.xfer([0xF0])
```

Watch the SPI transfer in `piscope` as you use this script to send the first character in your Net ID to the LED array. Take a screenshot of `piscope` for your lab report and a photograph of the LED array.

Lab report (individual work): Show the `piscope` screenshot from the SPI transfer. Annotate the `piscope` screenshot and indicate the exact instant in time at which:

- Each bit is loaded into the first shift register stage.
- The output pins should show the byte that you sent.

1.4 I2C pin multiplexer

In this exercise, you'll use the I2C pin multiplexer to control multiple inputs and outputs using the (shared) I2C bus.

Place the pin multiplexer (MCP23008) in your breaboard. Refer to the datasheet to identify the pins. Make sure you look at the diagram for the MCP23008, not the other multiplexer varieties also described in the same datasheet.

Then,

- Connect VSS to GND, VDD to 3.3V.
- Connect the SCL and SDA pins to the I2C bus on your Pi.
- The RESET pin, which is active low, should be tied to 3.3V so that the IC is *not* reset.
- Each of the address pins A2, A1, A0, should be tied to either GND *or* 3.3V, so that the address of the IC is `0x20`. To determine the I2C address given the state of the A2, A1, A0 pins, refer to Figure 1-2, I2C control byte format, in the datasheet.

Before you connect any inputs and outputs, test your I2C connection. Use

```
i2cdetect -y 1 0x20 0x27
```

to scan the I2C bus and see if there are any devices connected in the address range 0x20 to 0x27. Make sure your MCP23008 appears on address 0x20.

For this exercise, you will implement a tally counter. It should have:

- three buttons - one button to increment the count, one button to decrement the count, and one button to reset the count.
- four bits of output. Use four of the 2mm LEDs, with current-limiting resistors, to display the current count.

Connect these inputs and outputs as follows:

- Connect the HIGH side of each LED to GP4, GP5, GP6, and GP7, in order of their position on the breadboard. Then, put the LOW side of each LED in series with a current-limiting resistor to GND.
- For each push button switch, use a multimeter to identify which pins are internally connected when the switch is not actuated. Then, connect one pin of each switch to GP0, GP1, and GP2, respectively, and the other pin of each switch to GND.

To configure the MCP23008, we'll mainly use two registers: the `IODIR` register and the `GPPU` register. Refer to "Table 1-2 Register Addresses", "Table 1-3 Configuration and Control Registers", section 1.6.1, and section 1.6.7 for details.

Lab report (individual work): What byte should be in the `IODIR` register for each of the pins connected to the LEDs to be in OUTPUT mode, each of the pins connected to the switches to be in INPUT mode, and all unused pins to be in INPUT mode? Explain, using relevant parts of the datasheet to support your answer.

Lab report (individual work): What byte should be in the `GPPU` register for each of the pins connected to the switches to have the internal pull-up enabled, and for the internal pull-up to be disabled for all other pins? Explain, using relevant parts of the datasheet to support your answer.

To use the MCP23008, we'll use the `smbus` library in Python, which allows us to read and write to the I2C bus. We'll mainly use two functions:

- `write_byte_data` accepts three arguments: the address of the I2C peripheral, the address of the register we want to write to, and the byte we want to write to that register.
- `read_byte_data` accepts two arguments: the address of the I2C peripheral, and the address of the register we want to read from.

A demo script that shows you how to read from the MCP23008 input pins and write to the MCP23008 output pins is provided for you. Carefully study this script, then try to run it.

The `demo-mcp23008.py` script is also available on NYU Classes:

```
import time
import sys
import smbus

ADDR = 0x20  # Address of peripheral on I2C bus
IODIR = 0x00 # IO direction configuration register
GPPU = 0x06  # Pull-up configuration register
GPIO = 0x09  # GPIO register

i2c = smbus.SMBus(1)
```

```
def setAllInput():
    i2c.write_byte_data(ADDR, IODIR, 0xFF)

def setAllOutput():
    i2c.write_byte_data(ADDR, IODIR, 0x00)

def getAllDir():
    return i2c.read_byte_data(ADDR, IODIR)

def getAllPullup():
    return i2c.read_byte_data(ADDR, GPPU)

def setPinDir(pin, isInput, pullup = False):

    directionReg = i2c.read_byte_data(ADDR, IODIR)

    if isInput:
        directionReg |= (1 << pin)
    else:
        directionReg = directionReg & ~(1<<pin)
    i2c.write_byte_data(ADDR, IODIR, directionReg)

    if isInput:
        puReg = i2c.read_byte_data(ADDR, GPPU)
        if pullup:
            puReg = puReg | (1<<pin)
        else:
            puReg = puReg & ~(1<<pin)
        i2c.write_byte_data(ADDR, GPPU, puReg)

def readAll():
    return i2c.read_byte_data(ADDR, GPIO)

def readPin(pin):
    gpioReg = i2c.read_byte_data(ADDR, GPIO)
    return (gpioReg >> pin) & 0x01

def setPin(pin, state):
    gpioReg = i2c.read_byte_data(ADDR, GPIO)
    if state:
        gpioReg = gpioReg | (1 << pin)
    else:
        gpioReg = gpioReg & ~(1<<pin)
    i2c.write_byte_data(ADDR, GPIO, gpioReg)

if __name__ == '__main__':

    setAllInput()
    setPinDir(4, 0)
    setPinDir(0, 1, pullup = True)

    print("Pin modes (0=output, 1=input):")
    print(format(getAllDir(), "010b"))
    print("Built-in pull-up (0=disabled, 1=enabled):")
    print(format(getAllPullup(), "010b"))

    print("Set GP4 HIGH")
    setPin(4, 1)
    d = readPin(4)
    print(d)
    time.sleep(0.5)
    print("Set GP4 LOW")
    setPin(4, 0)
    d = readPin(4)
    print(d)

    print("Read input on GP0")
    d = readPin(0)
    print(d)
```

This script provides some basic "library" functions for reading from or writing to entire registers on the MCP23008, and for reading and writing the configuration or state of a single pin.

However, to implement your tally counter, it will be convenient to be able to:

- write to all four LED outputs at once, and
- read from all three switch inputs at once,

without a separate I2C transaction for each pin.

To realize this, you will add two additional functions, one called `setPinArr` and one called `readPinArr`.

The function `setPinArr` should accept two arrays, each of length 8, as arguments: the first argument is the pin array and the second argument is the state array.

Each position in the pin array corresponds to the GPIO pins GP0-GP7. Set the value in the pin array to 0 if you do not want to set the pin value, and to 1 if you do want to set the pin value. Then, for the pins where you set a 1 in the pin array, the corresponding value in the state array indicates what state to set the pin to.

For example, if you call

```
setPinArr([0,0,0,0,1,1,1,1], [0,0,0,0,0,0,1,1])
```

the LEDs on GP6 and GP7 should be on, and the LEDs on GP4 and GP5 should be off. The GP0, GP1, GP2, and GP3 pins should not be affected.

The `setPinArr` function should call `i2c_write_byte_data` exactly once - it should not use multiple I2C writes.

Implement your `setPinArr` function, then test it. Use `piscope` to verify that it works - show that it sets the value of multiple pins with only one I2C write transaction.

Then, implement a `readPinArr` function that accepts one argument, a pin array, and returns an array that gives the values of those pins. This function should use only one I2C read transaction. Test your function, and make sure that it works.

Lab report: Upload your `setPinArr` function, and explain it. Then, call your `setPinArr` function, and use `piscope` to capture all I2C transactions that are triggered by your function call. You should have one I2C read and one I2C write transaction. Show an annotated `piscope` screenshot, in which you identify each transaction.

Lab report: Upload your `readPinArr` function, and explain it. Show a `piscope` screenshot to verify that it uses only one I2C read transaction.

Finally, implement the tally counter with the 4-bit output and the three switch inputs, as described above. Your counter should have an increment button, a decrement button, and a reset button, and should display the current count on the 4-bit LED output.

Lab report: Upload your tally counter implementation. Your code should be clean, well-organized, efficient, and commented.
