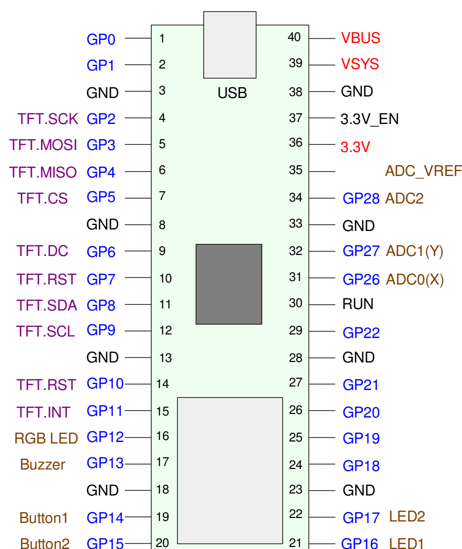
7. Serial I/O

Machine & Time Library, LED Cube



Introduction:

So far, we've been using the input / output pins in a parallel fashion: each I/O pin is connected to a different LED or a different button. There is nothing wrong, per say, in doing so: the Pi-Pico has 26 I/O pins available on the breakout boards, and you don't get a refund if some pins are not used. However, parallel I/O is not the intended use of the I/O pins on a Pi-Pico

One of the reasons to avoid parallel I/O is that many of the pins have other functions:

- GP0 and GP1 can be used as a serial port. If you want to read a GPS sensor or set up a wireless link to another processor, these pins are tied up for serial communications.
- GP 2..11 are going to be used by the graphics display on the Pico Breakout Board.
- Pins 13..17 are used by the buzzer, buttons, and LEDs on the Pico Breakout Board
- Pins 26..28 are used for analog inputs - such as reading the position of a joystick

The net results is there aren't a lot of unused pins on the Pi-Pico chip.

Fortunately, we don't need a lot of pins to talk to a large number of binary inputs and/or outputs. With a serial interface - something we'll talk about in this lecture - we'll be able to read and write to an essentially unlimited number of binary devices using just three wires.

SPI Communications

The two main forms of serial communications are SPI and I2C. Both forms are similar, with SPI originating with Motorola and becoming a defacto standard while I2C came from Phillips Semiconductors.

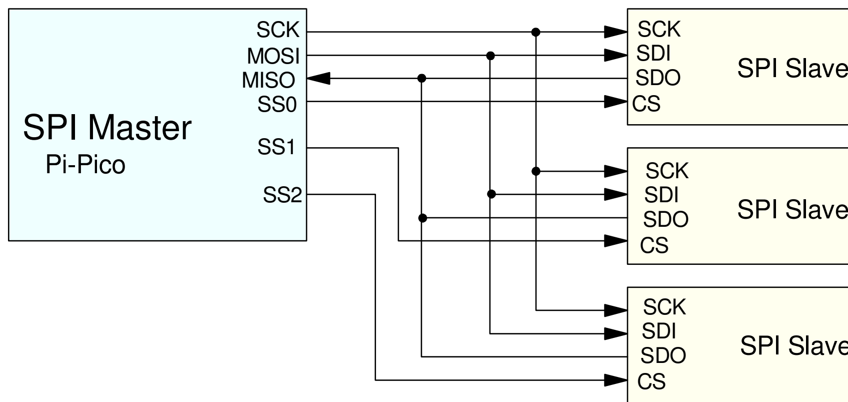
SPI uses a 3 or 4 wire interface, is capable of full-duplex communications (you can send and receive data at the same time), and capable of 40M bps[1] communications (maybe more?). I2C uses only 2 wires, but is limited to 400k bps. Both are supported by the Pi-Pico.

[1] bps: bits per second

The four wires used in SPI communications (along with a common ground) are:

- SCK: The clock line
- MOSI: Master Out, Slave In (write bus)
- MISO: Master In, Slave Out (read bus), and
- CS: Chip Select

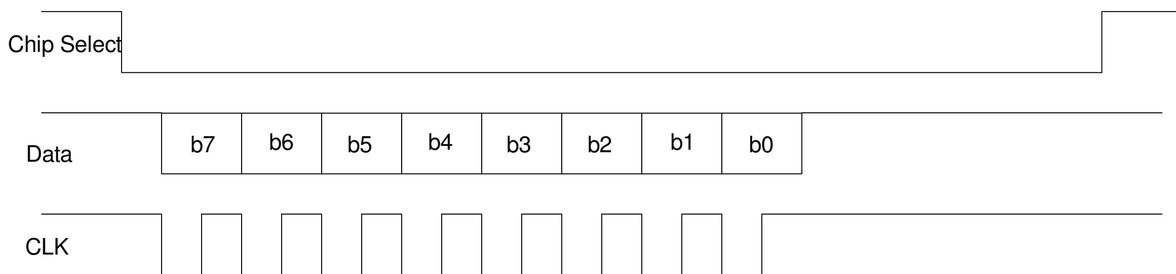
Note that the SCK, MOSI, and MISO lines can be shared by multiple devices: only the chip-select line needs to be separate (so the SPI master can specify which SPI slave the message is for.) Furthermore, if data only goes one way, the MISO line can be eliminated - leaving 3-wire communications. This allows a large number of devices to be addressed using just a few wires on the Pi-Pico



With SPI communications, SCK, MOSI, and MISO can be shared among your devices

Typically, an SPI message proceeds as follows:

- Chip Select goes low to start a message
- Bits are sent out on the MOSI line, one by one,
- Bits are read on the MISO line, one by one, and
- Each bit is synchronized by a clock line (sent by the master)
- At the end of the message, Chip Select goes high

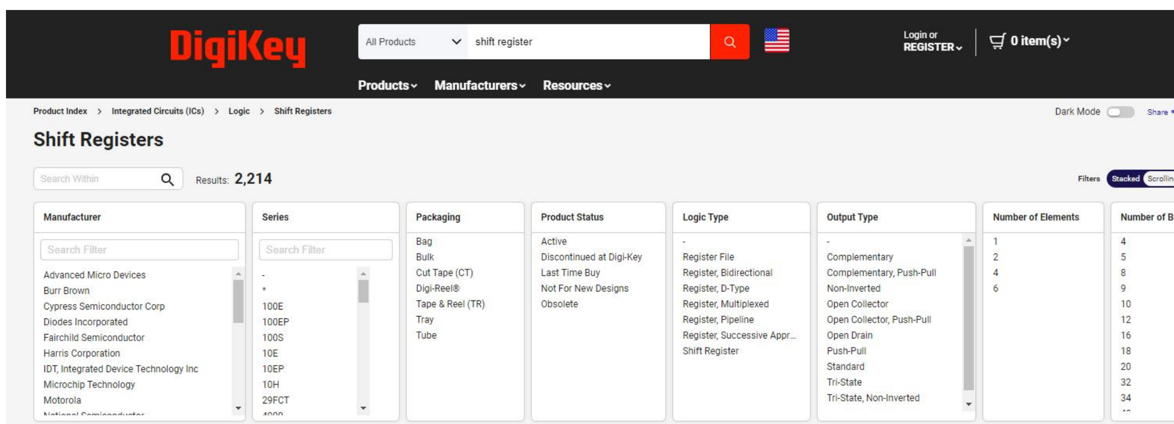


Typical signals on an SPI bus with 8-bits of data

In theory, there's no limit to how many bits can be sent in one message. This allows you to read from and write to a large number of binary I/O pins using just a few pins on the Pico.

Serial Input: 74HC594

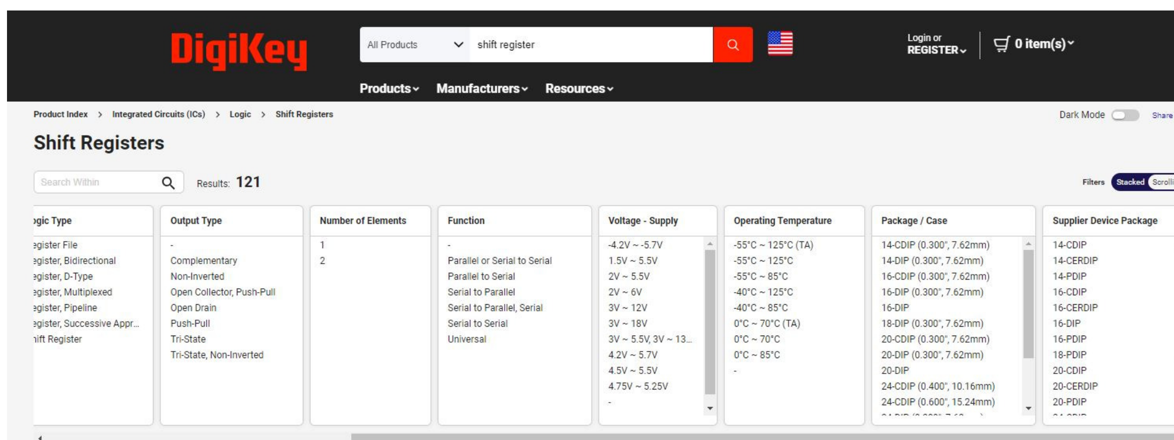
Starting out, lets look at having a Pi-Pico read eight binary inputs just a 3-wire SPI interface (since this is a read operation, the MOSI line isn't needed). If you search Digikey using the term *shift register*, you'll get 2214 hits (as of April 1, 2024).



Narrow the search as

- In-Stock
- Active
- 8-bits
- Through Hole

and you're down to 121 hits.



Narrow to *Parallel to Serial* (serial input) and you now have a manageable number of options. One that looks promising is a 74LS165. Select this one and pull up the datasheets

Page #1 of the data sheets tell you

- This device operated at up to 35MHz, and
- It gives the wiring diagram,

FAIRCHILD
SEMICONDUCTOR™

April 1998

DM74LS165
8-Bit Parallel In/Serial Output Shift Registers

General Description

This device is an 8-bit serial shift register which shifts data in the direction of Q_A toward Q_H when clocked. Parallel-in access is made available by eight individual direct data inputs, which are enabled by a low level at the shift/load input. These registers also feature gated clock inputs and complementary outputs from the eighth bit.

Clocking is accomplished through a 2-input NOR gate, permitting one input to be used as a clock-inhibit function. Holding either of the clock inputs high inhibits clocking, and holding either clock input low with the load input high enables the other clock input. The clock-inhibit input should be changed to the high level only while the clock input is high. Parallel loading is inhibited as long as the load input is high. Data at the parallel inputs are loaded directly into the register on a high-to-low transition of the shift/load input, regardless of the logic levels on the clock, clock inhibit, or serial inputs.

Features

- Complementary outputs
- Direct overriding (data) inputs
- Gated clock inputs
- Parallel-to-serial data conversion
- Typical frequency 35 MHz
- Typical power dissipation 105 mW

Connection Diagram

Dual-In-Line Package

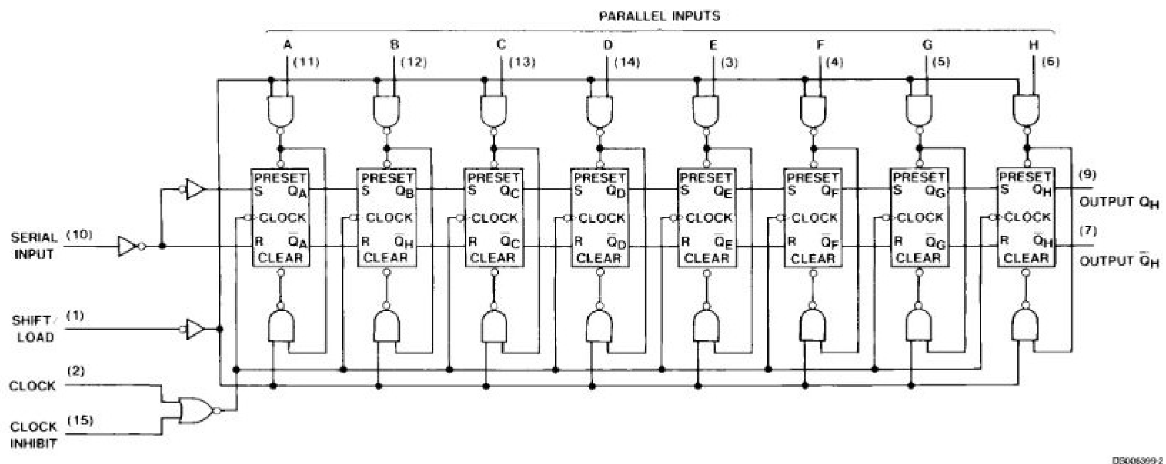
V_{CC} 16, CLOCK INHIBIT 15, PARALLEL INPUTS D 14, C 13, B 12, A 11, SERIAL INPUT 10, OUTPUT Q_H 9

SHIFT/LOAD 1, CLOCK 2, PARALLEL INPUTS E 3, F 4, G 5, H 6, OUTPUT $\overline{Q}_H$ 7, GND 8

DS006399-1

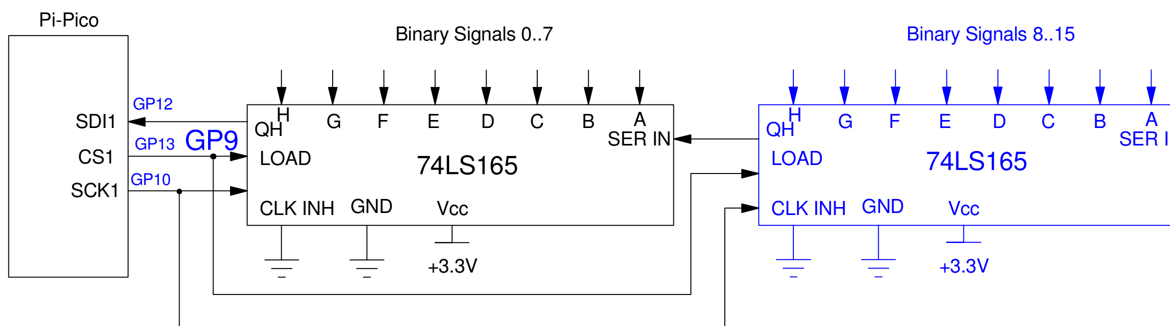
Order Number DM54LS165J, DM54LS165W, DM74LS165WM or DM74LS165N
See Package Number J16A, M16B, N16E or W16A

Page #6 shows you the schematic of the device, along with a timing diagram on page #5.



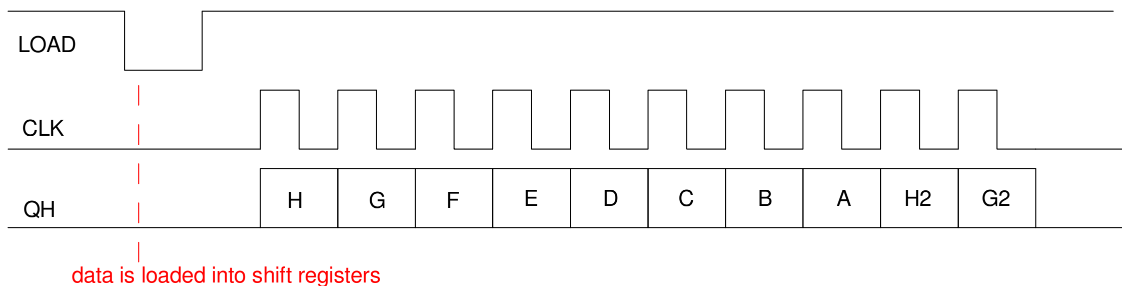
Schematics for a 74LS165 Shift Register

This is enough to set up the hardware



Wiring for a 74LS165: Parallel to Series IC

In terms of software, the following diagram on page #5 basically tells you how to write the code to read from this device.



Timing Diagram from Page 5 of data sheets

To read 16 binary inputs

- Pull LOAD low then high
- Pulse the clock high then low. The data on the QH line is valid on the falling edge of the clock.
- Read in the first bit (H),
- Pulse the clock high then low
- Shift the data left and read the next bit (G)
- repeat 16x to read 16 bits

Bit-Banging: This algorithm can be placed into a subroutine in Python

- Each bit is held high or low for 100ms so that you can see the data being shifted in.
- Since a LS165 can operate up to 35MHz, these wait times could be reduced to 1us without any problems.

This is called *bit banging*: you manually set and clear bits one by one.

The net result is you can

- Read 8 bits of data in 18us (if you change the sleep times to 1us)
- Read 16 bits of data in 34us

All while using just two wires.

```
from machine import Pin
from time import sleep_ms, sleep_us

CLK = Pin(10, Pin.OUT)
DIN = Pin(11, Pin.IN, Pin.PULL_UP)
LATCH = Pin(9, Pin.OUT)

def HC165():
    LATCH.value(1)
    CLK.value(0)
    sleep_ms(100)
    LATCH.value(0)
    sleep_ms(100)
    LATCH.value(1)
    # data is latched - now shift it in
    X = 0
    for i in range(0,8):
        CLK.value(1)
        sleep_ms(100)
        X = (X << 1) + DIN.value()
        CLK.value(0)
        sleep_ms(100)
        print(i, X)
    return(X)

while(1):
    Y = HC165()
    print(Y)
```

Bit-banging has some advantages:

- You have complete control of each signal
- You can use any I/O pins for the SPI communications

There *are* alternatives, however.

SPI communications has become a defacto standard, so not surprisingly, there are Python routines to do this for you as well as hardware on the Pi-Pico specifically designed for SPI communications.

To set up a SPI port in Python, the function SPI in *machine* is used:

```
from machine import Pin, SPI

spi = SPI(1,
baudrate=1000,polarity=0,phase=0,bits=8,sck=10,mosi=11,miso=12)

rxdata = spi.read(2, 0x42)
```

- *baud rate* sets the speed of the SPI communications (up to 30MHz for the LS165)
- *bits* tells you how many bits per message (8 or 16 for this example(
- *sck, mosi, miso* are the pins used for the SPI communications interface.
- *spi.read(2, 0x1234)* reads in two bytes while sending out 0x1234 on the MOSI line

(The MOSI line isn't used in this example - but could be used to drive a 74HC594 in the next section).

A full program which reads the SPI port using the SPI routine is as follows:

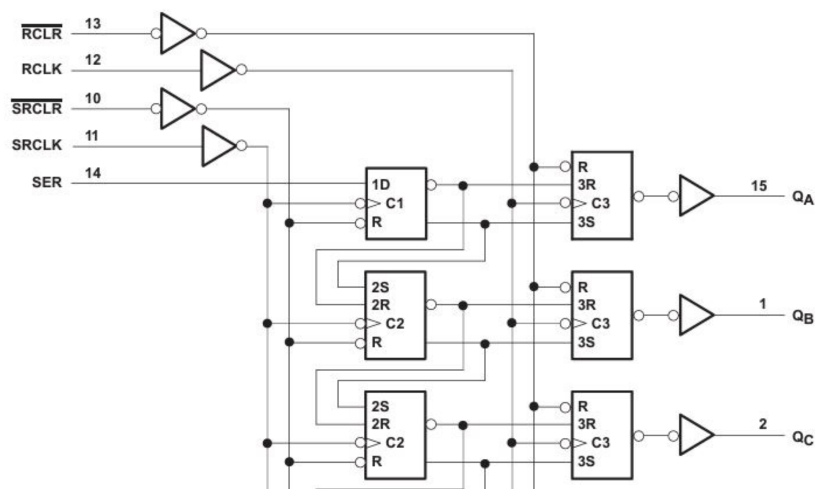
```
from machine import Pin
from time import sleep, sleep_ms, sleep_us

spi = SPI(1, baudrate=10_000_000, polarity=0, phase=0, bits=8,
sck=10, mosi=11, miso=12)

Button = Pin(20, Pin.IN, Pin.PULL_UP)
LATCH = Pin(13, Pin.OUT)

def LS165():
    LATCH.value(1)
    sleep_us(1)
    LATCH.value(0)
    sleep_us(1)
    LATCH.value(1)
    # data is latched - now shift it in
    rxdata = spi.read(2, 0x42)
    return(rxdata)


while(1):
    Y = LS165()
    print(Y)
    sleep(0.1)
```

At 10MHz, it will take only 3.6us to shift in 16 bits of data

- 2us to pulse the latch high then low
- 1.6us to shift in 16 bits of data

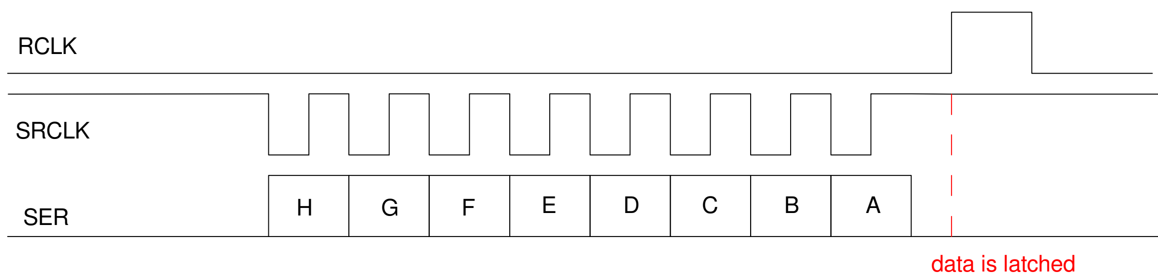
Serial Output: 74HC594

You can also do serial output with a Pi-Pico. To do so, first find a serial-in, parallel-out shift register. A 74HC594 is one such candidate.



74HC594 Serial-In, Parallel Out Shift Register

The corresponding timing diagram looks like this:

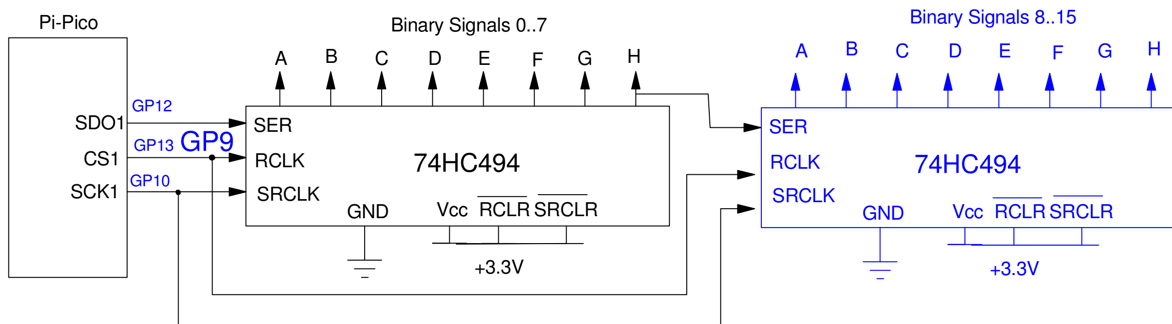


Translating....

- Start with RCLK = 0 and SRCLK = 1
- Send the first bit to the SER line (MOSI) and pulse the clock low then high
- Send the following bits to the SER line, pulsing the clock each time
- When done, pulse RCLK high then low to latch the outputs of the shift register to the outputs

At that point, the output pins are ready.

In terms of hardware, two 74HC594's could be used to output 16 bits using just 3 pins on the Pi-Pico:



Setting up 16 binary outputs using two 74HC594 series-in, parallel-out shift registgers.

Note:

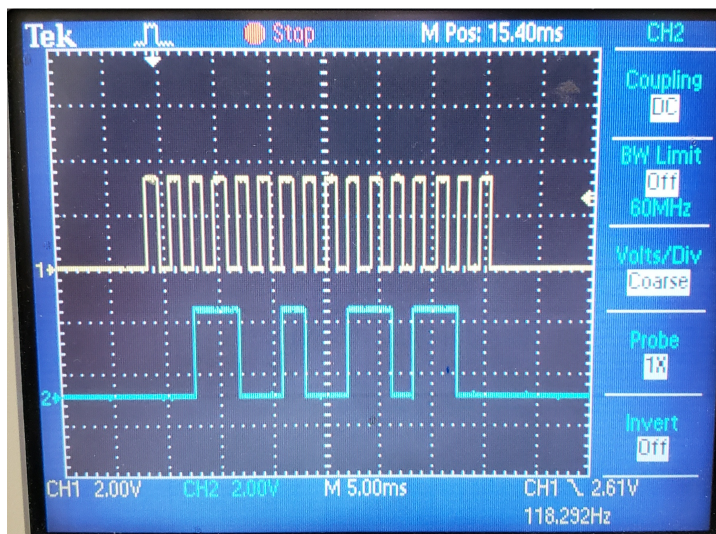
- There's no limit to how many shift registers you can cascade.
- You could use 5V for the 74HC494 shift registers. They don't send any signals to the Pi-Pico, so the 5V won't damage anything (serial inputs are high impedance)

The following code is one way to implement this using bit-banging:

```
from machine import Pin
from time import sleep_ms, sleep_us

CLK = Pin(10, Pin.OUT)
DOUT = Pin(11, Pin.OUT)
LATCH = Pin(13, Pin.OUT)

def HC594(X):
    LATCH.value(0)
    CLK.value(1)
    sleep_ms(1)
    for i in range(0,8):
        if(X & (0x80 >> i)):
            DOUT.value(1)
        else:
            DOUT.value(0)
        CLK.value(0)
        sleep_ms(1)
        CLK.value(1)
        sleep_ms(1)
    LATCH.value(1)
    DOUT.value(0)
    sleep_ms(1)
    LATCH.value(0)

x = 0
while(1):
    x = (x + 1) & 0xFF
    HC594(x)
    sleep_ms(100)
```

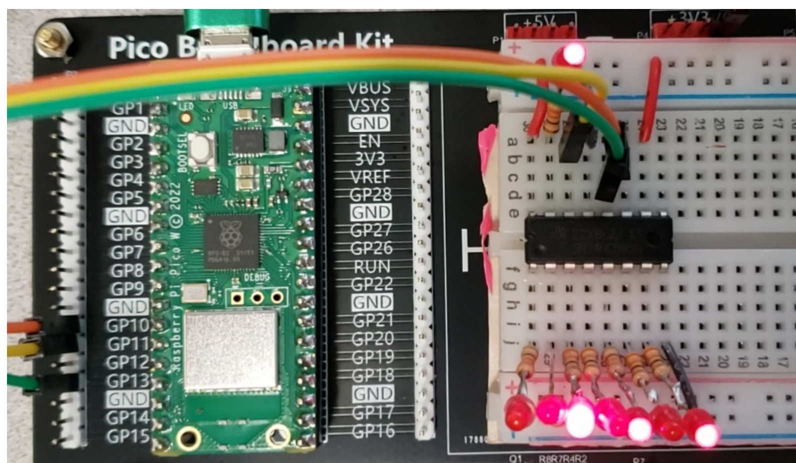
Using ticks_us(), the time it takes to execute the HC594() routine using bit-banging is 18,619us.

If you look at the CLK and DATA lines on an oscilloscope, you can see the data being sent out:



Oscilloscope showing the CLK line (yellow) and DATA line (blue).

If you connect LEDs to the output pins of the 74HC594, you can see the data lines as well:



LEDs connected to a 74HC594 showing which pins are 1 and 0
The data is HGFE DCBA = 1010 1001

Similarly, the same thing can be done using the built-in SPI functions in Python

```
from machine import Pin, SPI
from time import sleep_ms, sleep_us, ticks_us

spi = SPI(1, baudrate=10_000_000, polarity=0, phase=0, bits=8,
sck=10, mosi=11, miso=12)
LATCH = Pin(13, Pin.OUT)

def HC594(X):
    LATCH.value(0)
    Y = bytearray([X])
    spi.write(Y)
    LATCH.value(1)
    sleep_us(1)
    LATCH.value(0)

x = 0
while(1):
    x = (x + 1) & 0xFF
    t0 = ticks_us()
    HC594(x)
    t1 = ticks_us()
    print(t1 - t0)
    sleep_ms(10)
```

```
140
139
140
```

(note: code in blue is added to measure the execution time)

Note that using the built-in SPI funciton is *much* more efficient:

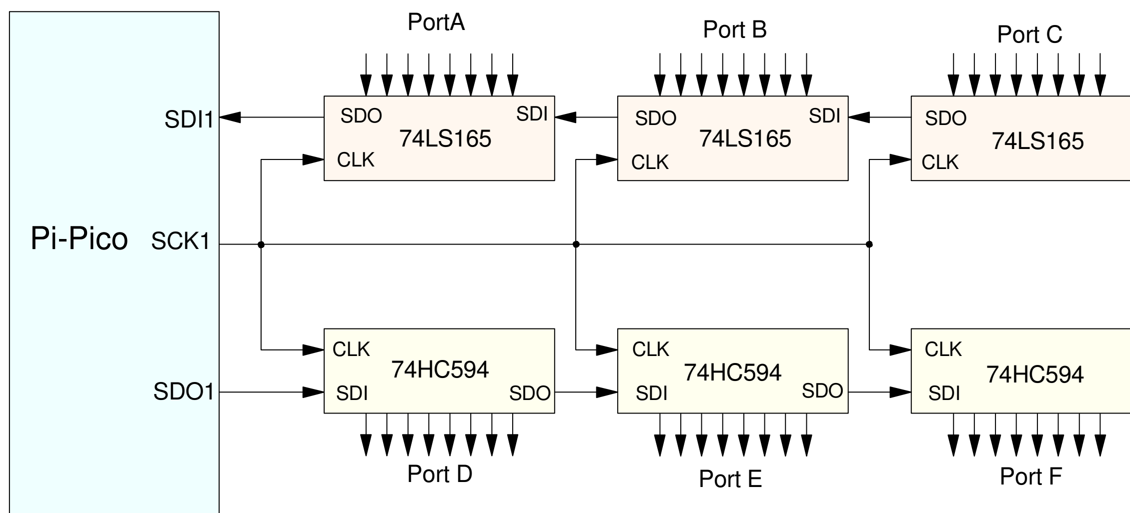
Bit-Banging	SPI
18,619us	140us

The SPI port is a little trickier to figure out. It's worth it though - using the SPI port sends data to the HC594 chip more than 100x faster then bit-banging.

Also note that this is why the Pi-Pico doesn't have Ports like other processors. The assumption is you're going to be using serial I/O to drive your devices. As such, Ports don't really make sense as far as the I/O pins on the Pico go. The actual ports will be the shift register inputs and outputs.

In addition, you can read the input ports at the same time you're writing to the output ports

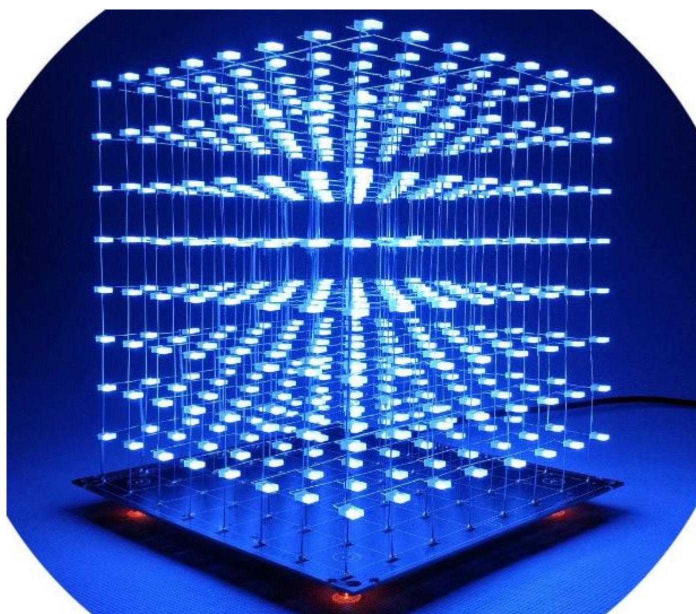
- SDI (MISO) reads the parallel-in, series out shift registers (74LS165)
- SDO (MOSI) writes to the series-in, parallel out shift registers (74HC594)
- All using the same clock line



A Pi-Pico can read and write to multiple shift registers at once - creating an almost unlimited number of I/O lines

Fun with Series Outputs: LED Cube

Finally, to illustrate how useful shift registers are, let's look at the design of an 8x8x8 LED cube



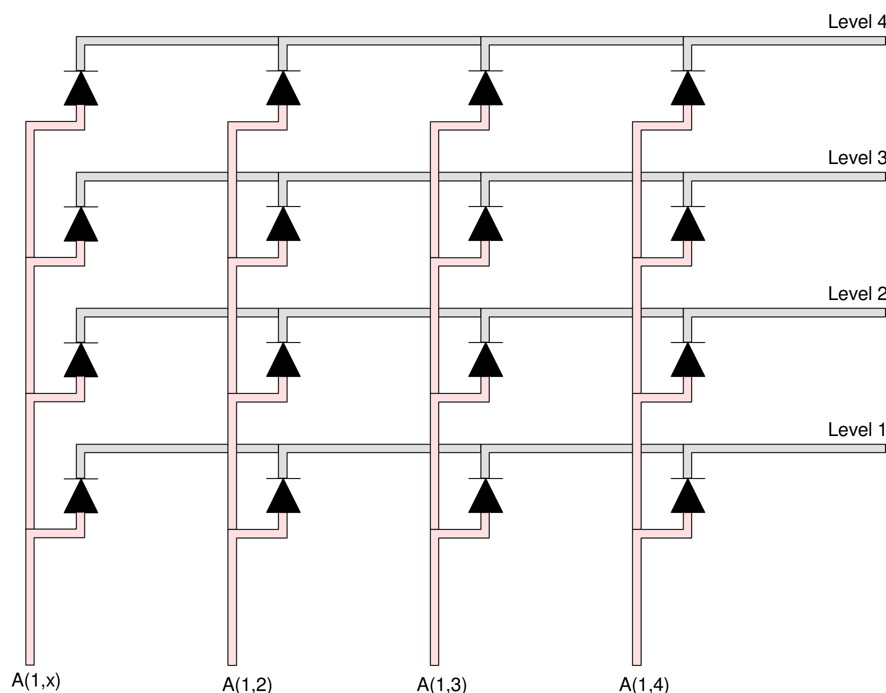
8x8x8 LED cube - kit available from Amazon for about $45

An 8x8x8 LED cube contains 256 LEDs, each able to be turned on and off from your microcontroller (Arduino, PIC, Pi-Pico, etc.) Considering that a Pi-Pico only has 26 I/O pins, it's clear that you can use a separate wire for each LED. Not to mention that connecting 256 LED would be a wiring nightmare.

To give an idea of how an 8x8x8 LED cube works, let's look at a smaller version: a 4x4x4 LED cube.

When you build a cube, you start with a 4x4 array of LEDs with

- The anodes (+) connected together, going up and down
- The cathodes (-) connected together going left to right (the floor), and



Start: Create four 4x4 resistor arrays with the cathodes shorted together (creating levels) and the anodes shorted together (top to bottom)

Once you have four of these,

- place them side by side creating a 4x4x4 array of diodes.
- Short the levels together (all diodes on Level 4 have their cathodes shorted, etc)

Note that soldering the cathodes together provides strength for the cube.

- Only one wire connecting the slices is necessary (bend the wires sticking out to the right over so they meet and can be soldered to the next slice.
- Optionally, you can add a spacing wires connecting the cathodes at each LED if you like. Electrically, it doesn't make a difference (one shorting wire is sufficient). Mechanically, the more cross braces you include, the stronger your LED cube will be.

.

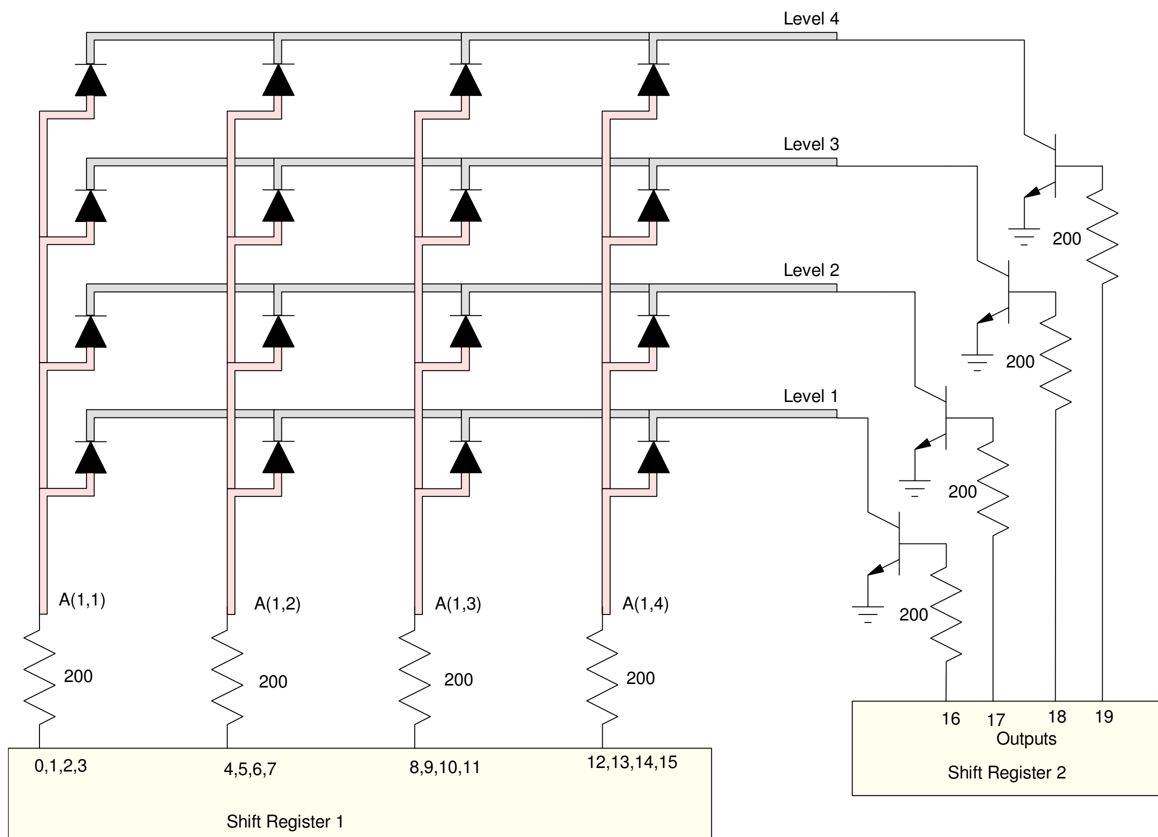
To power the LED cube, each anode is a separate output of a shift register. Connect these pins to the shift register with a 200 Ohm resistor to set the current to (assuming the shift registers are powered with 5V)

$$I = \left(\frac{5V - 3.0V}{200\Omega}\right) = 10mA$$

To tie the cathode to ground, use an NPN transistor

- If all 16 LEDs are on at a given level, the net current on the cathode is 160mA - more than a shift register can handle. A transistor amplifies this current.

For a 4x4 array, this requires 20 outputs (three shift registers).



Connections for driving a 4x4x4 LED cube.
Three more slices would be used (not shown), each slice connected to the next four pins on Shift Register 1

The way the program works is

- You first turn on Level 4 (SR2 = 0001)
  - SR1 defines which LEDs are on and off on level 4
  - Pause 2ms
- You then turn on Level 3 (SR2 = 0010)
  - SR1 defines which LEDs are on and off on level 3
  - Pause 2ms
- You then turn on Level 2 (SR2 = 0100)
  - SR1 defines which LEDs are on and off on level 2
  - Pause 2ms
- You finally turn on Level 1 (SR2 = 1000)
  - SR1 defines which LEDs are on and off on level 1
  - Pause 2ms
- and repeat

The net result is

- Each LED has a 25% duty cycle when on (8 levels would be a 12.5% duty cycle)
- The cycle time is 8ms (125Hz) for a 4x4x4 array, 62.5Hz for an 8x8x8 array
- You can selectively turn on each LED independently

Summary:

If you're only driving a few items, there's nothing wrong with using parallel I/O with a Pi-Pico chip. If you add some shift registers in your design, however, the number of I/O pins is almost unlimited.