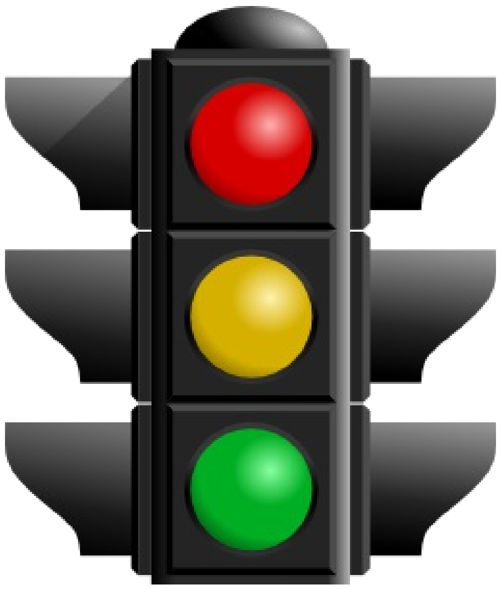
# Raspberry Pi Traffic Light Using an I/O Port Expander and a 2-Input NOR Gate

Project Manual



Imagination Station Science & History Museum

Brian Grawburg

**Version 3a**
**December 2014**

## Acknowledgements

This project was adapted from an original project by Bob Rathbone of Bob Rathbone Computer Consultancy (bobrathbone.com. August 16th, 2013)

Various articles from *The MagPi* magazine

Microchip Technology Inc., data sheet for the MCP23017/MCP23S17 chip

Texas Instruments, data sheet for CD54AC02, CD74AC02 Quadruple 2-input Positive-NOR gates

The Triangle Embedded Devices Group (www.triembed.org)

This project manual is part of a week-end class at the Imagination Station Science & History Museum that continued previous camps and programs related to the Raspberry Pi. For further information, or to purchase the Traffic Light printed circuit board, please contact:

Brian Grawburg
bgrawburg@imaginescience.org
224 Nash St.
Wilson, NC 27893

## Table of Contents

## Overview of the Project

A typical traffic signal at a major intersection has the usual red-yellow-green lights and a left-turn signal also with red-yellow-green lights/arrows. Although many intersections are asymmetrical as regards turning lanes and timing, for this project we'll limit the number to two symmetrical directions – North (representing north-south) and East (representing east-west).

Like a real traffic signal, the Python code has to turn on the red for one direction while the green and then yellow are on for the opposite direction; then both reds will be on for a short time simultaneously until the cycle starts over.

Most intersections with a separate left-turn lane don't usually activate the left-turn lights if there are no cars in the lanes. For this project momentary push-buttons will simulate a car in the lane to activate both left-turn lanes.

Python code to test the correct LED hookup and run the basic north-south and east-west light sequence and the two left-turn options is included in the full version of the project manual. This should be enough to get you started on other light combinations.

The Raspberry Pi is connected to an MCP23017 I$^2$C (usually pronounced I-squared-C) I/O (Input/Output) port expander and a CD74AC02 NOR gate (both are explained further on in the manual.)

Although the most direct way to connect a Raspberry Pi to external devices, either input types such as sensors or output types such as relays or LEDs, is by attaching jumpers directly to the Pi's GPIO pins, using a port expander offers several advantages. It provides additional I/O ports (the Model B Raspberry Pi has only 8 ports[1]) and protects the Pi from possible damage by isolating attached devices from the Pi.

The MCP23017s could be fitted to a breadboard and then wired back to the Pi's GPIO pins with jumpers, however I decided to use a pre-fabricated board to make the connections much easier. In addition, the board can be reused for a variety of projects.

In addition to the port expander a second chip, a CD74AC02 Quadruple 2-input NOR gate, is used to enable coding the “left turn” option. This chip and the left-turn push buttons will

1 The Model B+, which became available July 2014, has 17 GPIO pins.

be on the breadboard.

## What's Needed For The Project?

You'll need to supply your own Raspberry Pi Model B or B+. If you are using a Model B+ you'll need to use the GPIO pin adapter included with the kits.

### 1. The Traffic light PCB Kit: Available Only from the Museum

| | |
|---|---|
| 1 | Traffic light LED printed circuit board |
| 12 | 5mm LEDs (4 red, 4 yellow, 4 green) |
| 12 | 470Ω resistors |
| 5 | Jumper pin connectors for the PCB |

### 2. Breadboard and Accessories

| | |
|---|---|
| 1 | Solderless half-breadboard |
| 1 | CD74AC02 Quadruple 2-input Positive-NOR Gate & socket |
| 2 | Mini push-buttons |
| 4 | 3-wire jumper cables |
| 1 | 40 pin to 26 pin GPIO adapter for using a Model B+ with the project |

### 3. *Protect Your Pi* board kit (http://mypishop.com)

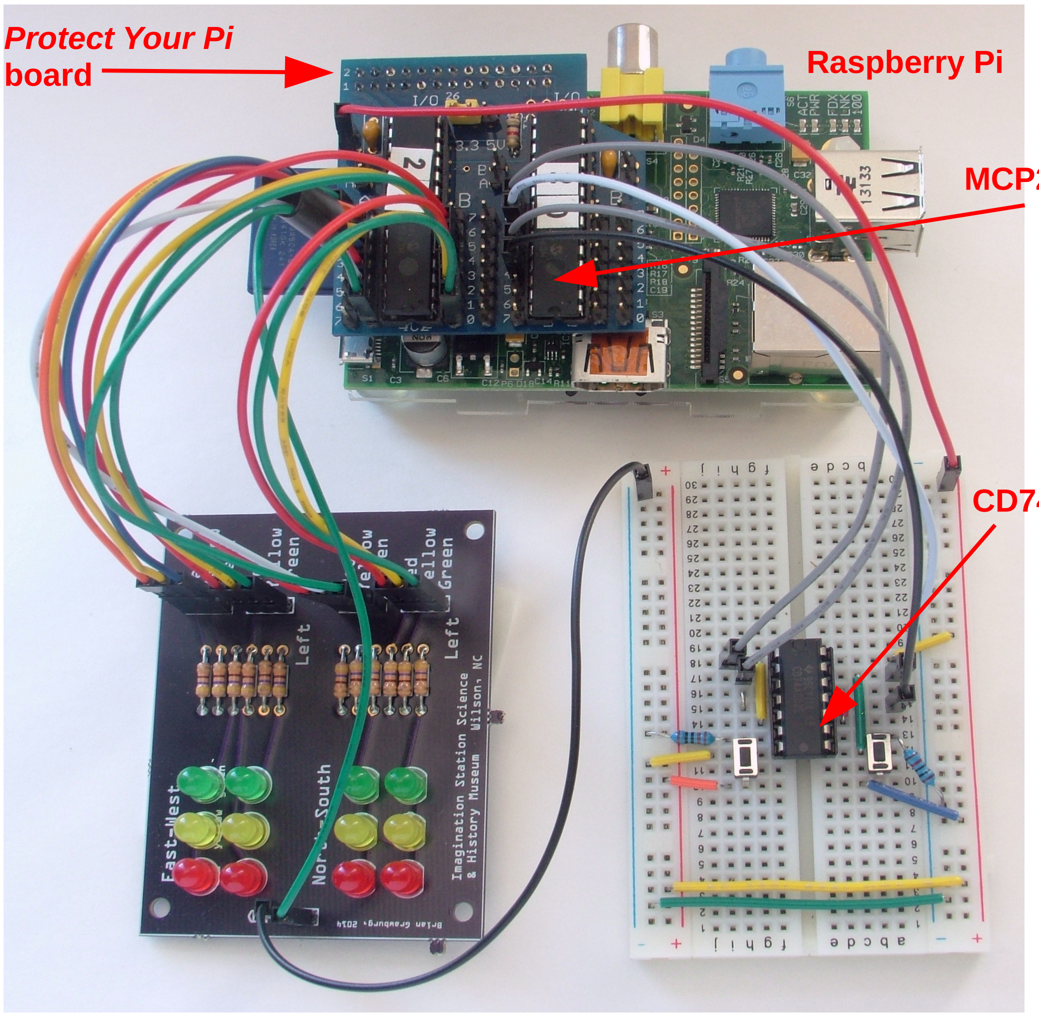
| | |
|---|---|
| 1 | A PCB board with two MCP23017 chips that gives 32 ports. |

### 4. Optional *Bridge Your Pi* also from MyPiShop

*In addition to hardware you'll also need a basic understanding of how to use the Raspberry Pi and how to write Python code.*

## The “Purpose” of the Project

This project has been designed to help students and hobbyists get started creating projects and applications using an **I²C** interface I/O expander such as the MCP23017 connected to the GPIO interface on the Raspberry Pi. It also introduces a logic-gate chip, the CD74AC02.



# The MCP23017 16-Bit Port Expander

# Using the I$^2$C Interface

## About the MCP23017

I²C, created by Philips Semiconductors and commonly written as 'I2C', stands for Inter-Integrated Circuit and allows communication of data between I2C devices over two wires.

The MCP23017 is a 16-bit general purpose I/O expander with serial interface. The chip has two banks designated as A and B. Each bank is 8 bits (1 byte). As helpful as increasing the number of available ports is, the primary reason for using an expander board is to isolate the RPi from devices and eliminate the possibility of damaging the Pi by improperly connecting an output into an input, especially if it's greater than 3.3 volts DC. The Pi costs $35 to replace, an MCP23017 chip costs less than $2.00. The 3.3 volts output from the GPIO pins are **control** signals and they really aren't meant to drive anything. The MCP23017, on the other hand, can drive devices (within reason, of course).

As a general rule the maximum current draw from the GPIO pins on the Pi at 3.3 V should not exceed 50 mA. The maximum from any one GPIO pin is 16 mA. The MCP23017 itself has maximum current draw per chip pin of 25 mA: but because of how it is actually connected to the Pi it does not violate the 16 mA current limit on the Pi.

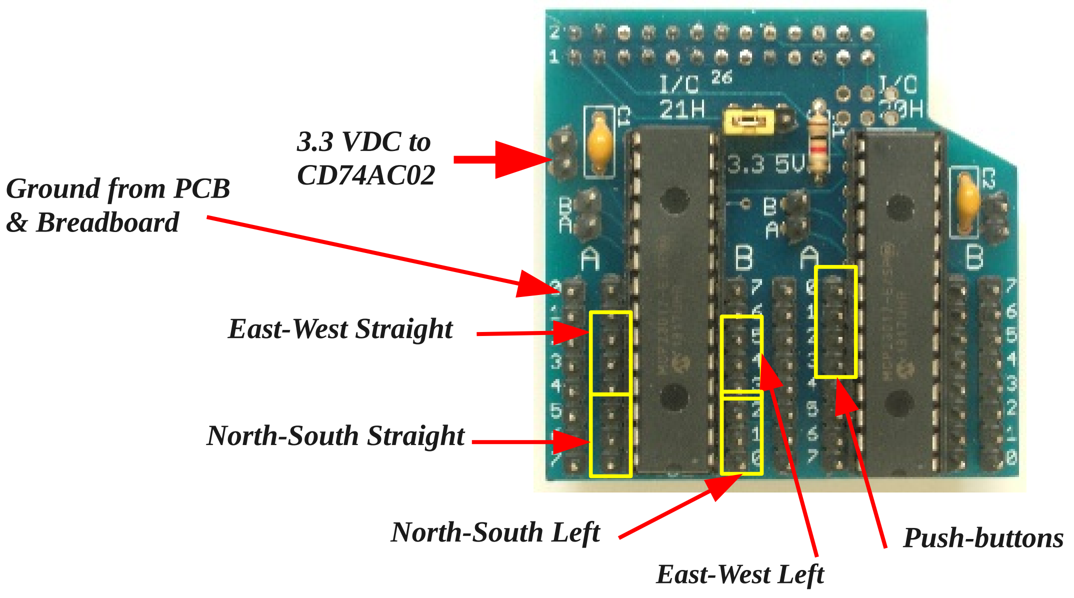
When writing the Python code for this project I used the register address naming convention employed by Microchip Technologies in their DS21952B data sheet for the MCP23017/MCP23S17.

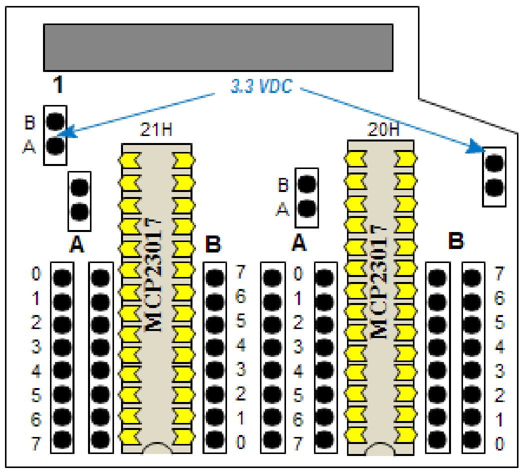
| **Register Address** | **Hex Address [Bank = 0]** | **Access to** |
|---|---|---|
| IODIRA | 00 (0x00 in the Python code[a]) | Will be used to indicate the direction, input (1) or output (0), of the data I/O for each bit for bank A |
| IODIRB | 01 (0x1 in the Python code) | Will be used to indicate the direction, input (1) or output (0), of the data I/O for each bit for bank B |
| GPIOA | 12 (0x12 in the Python code) | The GPIO data ports for bank A |
| GPIOB | 13 (0x13 in the Python code) | The GPIO data ports for bank B |
| OLATA | 14 (0x14 in the Python code) | Output latches for bank A |
| OLATB | 15 (0x15 in the Python code) | Output latches for bank B |

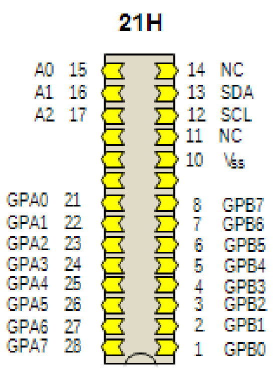
[a] The 0x indicates to Python that this is a hexadecimal value

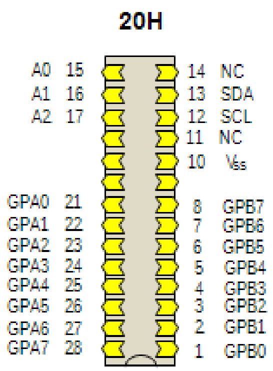
## Connecting the MCP23017 to the Raspberry Pi Using The *Protect Your Pi* Board

This project requires a minimum of 16 ports; 14 outputs and 2 inputs. The board can be constructed as a shield to attach directly to the Pi (as shown on page 6) or attached with an external ribbon. If space is a limiting factor the shield option is a good idea. However, I prefer to attach the boards to the Pi using a ribbon cable while I work on the code and component layout.

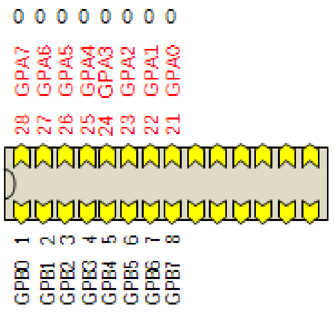




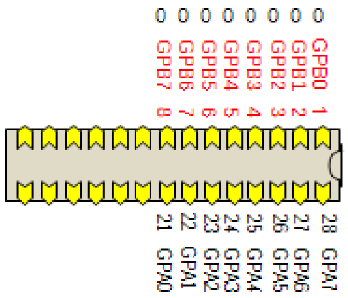




# MCP23017 Pin Layout


0 0 0 0 0 0 0 0
GPA7 GPA6 GPA5 GPA4 GPA3 GPA2 GPA1 GPA0
28 27 26 25 24 23 22 21
1 2 3 4 5 6 7 8
GPB0 GPB1 GPB2 GPB3 GPB4 GPB5 GPB6 GPB7

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Chip GPA__** | *7* | *6* | *5* | *4* | *3* | *2* | *1* | *0* |
| **Binary Value** | *128* | *64* | *32* | *16* | *8* | *4* | *2* | *1* |
| **Bank A Pin Number** | *28* | *27* | *26* | *25* | *24* | *23* | *22* | *21* |


0 0 0 0 0 0 0 0
GPB7 GPB6 GPB5 GPB4 GPB3 GPB2 GPB1 GPB0
8 7 6 5 4 3 2 1
21 22 23 24 25 26 27 28
GPA0 GPA1 GPA2 GPA3 GPA4 GPA5 GPA6 GPA7

I've rotated the chip 180º to better show the direction of the binary code

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Chip GPB__** | *7* | *6* | *5* | *4* | *3* | *2* | *1* | *0* |
| **Binary Value** | *128* | *64* | *32* | *16* | *8* | *4* | *2* | *1* |
| **Bank B Pin Number** | *8* | *7* | *6* | *5* | *4* | *3* | *2* | *1* |

## Traffic Light Binary to Hexadecimal Work Sheet

### 21H Bank A

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *Chip GPA__* | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** | |
| *Chip Pin* | **28** | **27** | **26** | **25** | **24** | **23** | **22** | **21** | **Hex** |
| *Binary Count* | **128** | **64** | **32** | **16** | **8** | **4** | **2** | **1** | |
| **Straight LEDs** | **N_Green** | **N_Amber** | **N_Red** | **E_Green** | **E_Amber** | **E_Red** | | | |
| North Green, East Red | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0x84 |
| North Amber, East Red | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0x44 |
| East Green, North Red | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0x30 |
| East Amber, North Red | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0x28 |
| East Red, North Red | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0x24 |
| | | | | | | | | | |

## 21H Bank B

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| *Chip GPB__* | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |  |
| *Chip Pin* | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **Hex** |
| *Binary Count* | **128** | **64** | **32** | **16** | **8** | **4** | **2** | **1** |  |
|  |  |  | **E_Red** | **E_Amber** | **E_Green** | **N_Red** | **N_Amber** | **N_Green** |  |

## Left Turn LEDs

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| Both Lefts Green | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0x09 |
| Both Lefts Amber | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0x12 |
| Both Lefts Red | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0x24 |
| East Left Red: North Left Green |  |  |  |  |  |  |  |  |  |
| East Left Red: North Left Amber |  |  |  |  |  |  |  |  |  |
| East Left Green: North Left Red |  |  |  |  |  |  |  |  |  |
| East Left Amber: North Left Red |  |  |  |  |  |  |  |  |  |

## 20H Bank A

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| *Chip GPA_* | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| *Chip Pin* | **28** | **27** | **26** | **25** | **24** | **23** | **22** | **21** |
| *Binary Count* | **128** | **64** | **32** | **16** | **8** | **4** | **2** | **1** |

## Left Turn Buttons

|  |  |  |  |  |  |  |  |  | **Hex** | **Decimal** |
|---|---|---|---|---|---|---|---|---|---|---|
| East-West Push-button Input | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x80 | 128 |
| North-South Push-button Input | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40 | 64 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| East-West Push-button Output | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0x10 | 16 |
| North-South Push-button Output | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0x08 | 8 |
|  | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0x18 | 24 |

## Installing or Updating the SMBUS Libraries

Before you can use the MCP23017, or any I2C chip, you must install the SMBUS libraries on your Pi and update a required configuration file.

**1. SMBus support may already be installed on your Pi.  To check and update enter the following commands in a terminal window.**

```
sudo apt-get install python-smbus
sudo apt-get install i2c-tools
```

**2. Because the I$^2$C drivers are disabled by default you must make a change in a configuration file.  Open a terminal window and type:**

```
sudo nano /etc/modprobe.d/raspi-blacklist.conf
```

```
blacklist  i2c-bcm2708
blacklist  spi-bcm2708
```

**Edit the file by adding a hash "## as shown below and save it by typing Ctrl-O, then Crtl-X to close.**

```
# blacklist  i2c-bcm2708
# blacklist  spi-bcm2708
```

**3. Open a terminal window, and edit the modules file as shown below by typing:**

```
sudo nano /etc/modules
```

```
# /etc/modules: kernel modules to load at boot time
# This file contains the names of kernel modules that should be loaded
# at boot time, one per line. Lines beginning with "#" are ignored.
# Parameters can be specified after the module name.

snd-bcm2835
i2c-bcm2708
i2c-dev
```

*Add these two lines*

**Type Ctrl-O to save the file and Ctrl-X to close it. Reboot the Pi.**

**4. Finally, add the 'pi' user to the i2c group by entering this command from the terminal window.**

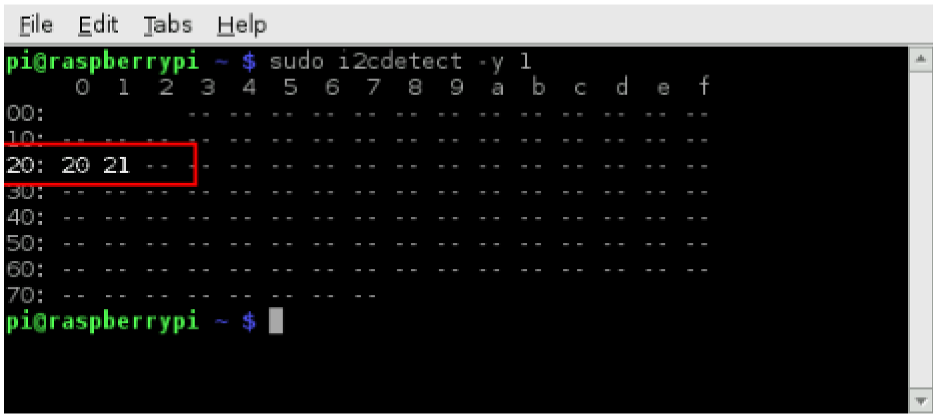
```
sudo adduser pi i2c
```

**Reboot the Raspberry Pi**

**5. Connect the MCP23017 board to the Raspberry Pi, either by attaching it directly to the GPIO pins or with a ribbon cable. If you're using a cable make sure you have the #1 pins correctly connected.**

**Use the i2c tools to display the address of the MCP23017. Type this command in the terminal window.**

```
sudo i2cdetect -y 1
```

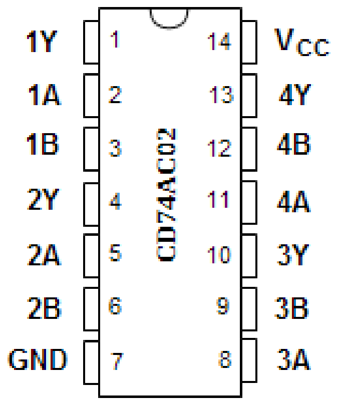


```
pi@raspberrypi ~ $ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 21 -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
pi@raspberrypi ~ $
```

The addresses for the two chips on the PCB are 20 and 21. If there are no numbers you need to check your connections.

**If there are no numbers, SHUT DOWN THE PI. DO NOT UNPLUG OR PLUG-IN THE CABLE WITH THE PI TURNED ON.**

# The CD74AC02 Quadruple 2-input NOR Gate[2,3]



2 The TI data sheet describes this chip as Positive-NOR
3 If you're unfamiliar with logic gate symbols see: http://electronicsclub.info/gates.htm

## About the Chip

One of the more frustrating aspects of this project was trying to incorporate the left-turn option into the Python code. No doubt there was a way to do it but I couldn't figure it out (I saw one possible way, but it required a lot of code). Here's where being part of a user group can really make a difference. I'm active in the Triangle Embedded Devices group (triembed.org) and posted a series of emails to the group; got a great answer that I'm using here.

**The "problem"**. As the lights are cycling through I want to push a button to simulate a car pulling into the left turn-only lane which activates something to tell the program "there's a car that wants to turn so when you're finished with the normal cycle run the left turn cycle." Back in 2010/2011 I took a year-long course at a local community college on programming PLCs (Programmable Logic Controllers). One of the basic functions is latching an output, meaning that when it is set high it would remain high until specifically set low by the program. That's what I was trying to do within the Python code.

A couple of solutions were put forth from the TriEmbed group but I really liked the suggestion to use a NOR gate as a flip flop (don't worry, I'll explain what that is). I was given full instructions for wiring it and it worked great. But once I got it working I had an intermittent glitch that I'll explain in a moment.

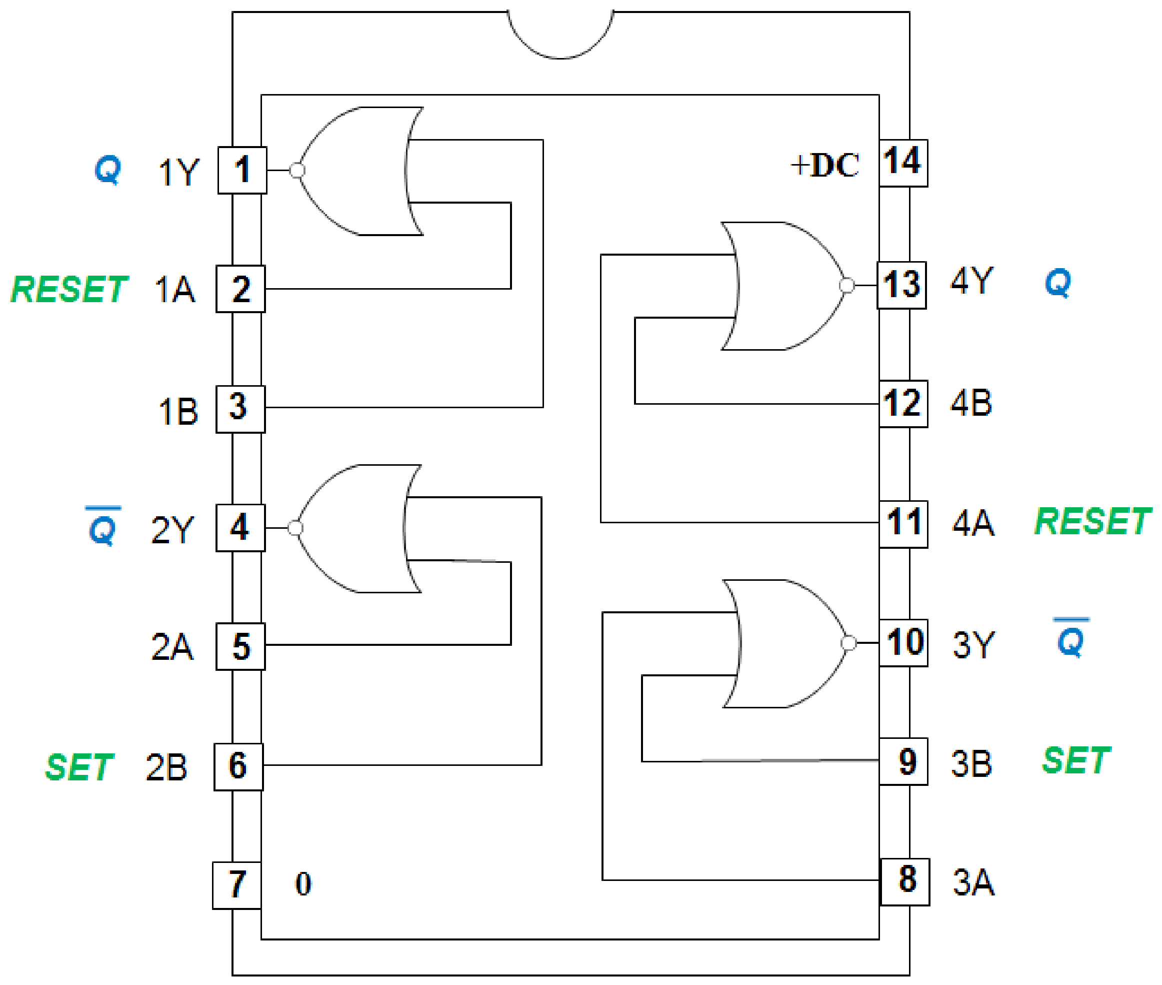
What exactly is a Quadruple 2-input NOR gate? If you're going to do much experimenting I suggest you get a copy of *Electronics All-In-One For Dummies* by Doug Lowe.[4] I'll be quoting from this book on page 17.

So, what was my "intermittent glitch"? When I started the program it ran fine for several cycles, then I noticed that the left-turn option periodically started without my pushing the button. It took only a second to realize that input to the CD74AC was floating[5] and I needed a resistor between pin 6 (2B) and ground. I put in a 10k resistor and the floating stopped.

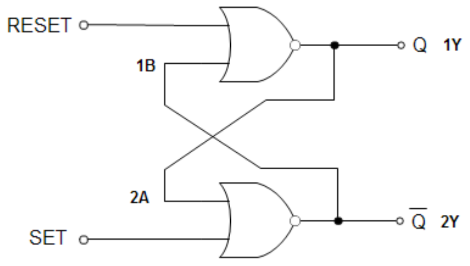
4 See this page for some diagrams of NOR gates - http://www.electronics-tutorials.ws/sequential/seq_1.html.
5 See the Appendix C, "Pull-Up and Pull-Down Resistors"

## CD74AC02 Quadruple 2-Input NOR Gate Chip Layout

The CD74AC02 contains four independent 2-input NOR gates as diagrammed below (NOR is an abbreviation for NOT OR, two options in logic construction).



## Making The Latching Set-Reset Flip Flop



A *latch* is a logic circuit that has two inputs and one output. One of the inputs is called the *SET input*; the other is called the *RESET input*. Latch circuits can be either active-high or active-low. The difference is determined by whether the operation of the latch circuit is triggered by HIGH or LOW signals on the inputs.

◆ **Active-high circuit:** Both inputs are normally tied to ground (LOW) and the latch is triggered by a momentary HIGH signal on either of the inputs.

◆ **Active-low circuit:** Both inputs are normally HIGH, and the latch is triggered by a momentary LOW signal on either input.

In an active-high latch, both the SET and RESET inputs are connected to ground [by a pull-down resistor]. When the SET input goes HIGH, the output also goes HIGH. When the set input returns to LOW, however, the output remains HIGH. The output of the active-high latch stays HIGH until the RESET input goes HIGH. Then, the output returns to LOW and will go HIGH again only when the SET input is triggered once more.

In other words, the latch remembers that the SET input has been activated. If the SET input goes HIGH for even a moment, the output goes HIGH and stays HIGH, even after the SET input returns to LOW. The output returns to LOW only when the RESET input goes HIGH.

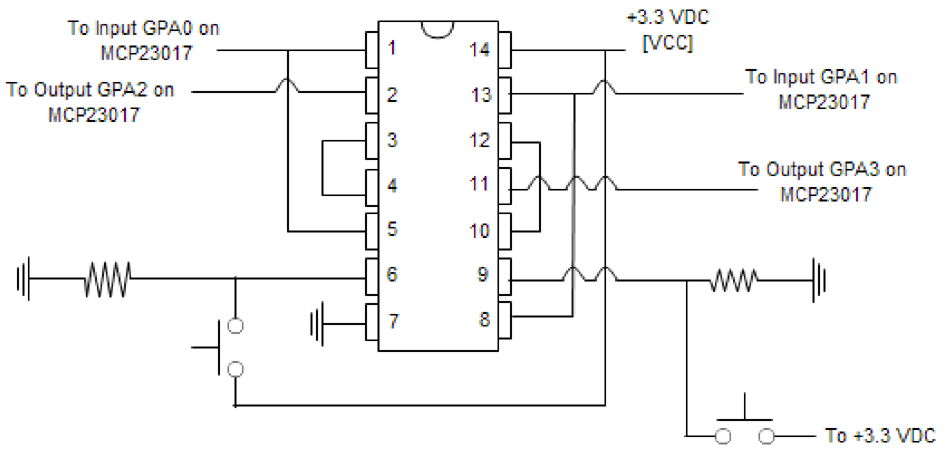
Note that most latch circuits actually have a second output that is simply the first output inverted. In other words, whenever the first output is HIGH, the second output is LOW, and vice versa. These outputs are usually referred to as *Q* and $\overline{Q}$.

The notation is usually pronounced either "bar Q" or "Q bar," though some people pronounce it "not Q." The horizontal bar symbol over a label is a common logical shorthand for inversion. That is, $\overline{Q}$ is the inverse of Q.[6,7]

## Wiring The Chip

6 *Electronics for Dummies*, page 583-584. See also pages 500 - 517

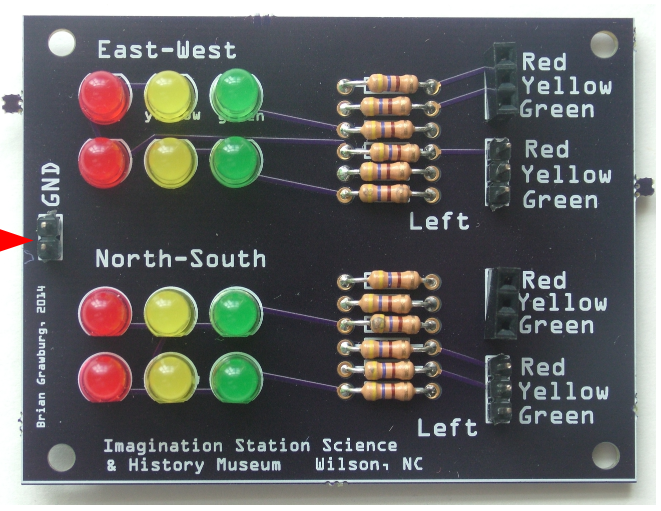
7 See this site for a good explanation: http://www.hobbyprojects.com/flip_flop/NOR_gate_flip-flop.html


To Input GPA0 on MCP23017
To Output GPA2 on MCP23017
+3.3 VDC [VCC]
To Input GPA1 on MCP23017
To Output GPA3 on MCP23017
To +3.3 VDC
1
2
3
4
5
6
7
14
13
12
11
10
9
8

This is a schematic showing how the chip is to be wired on the breadboard.

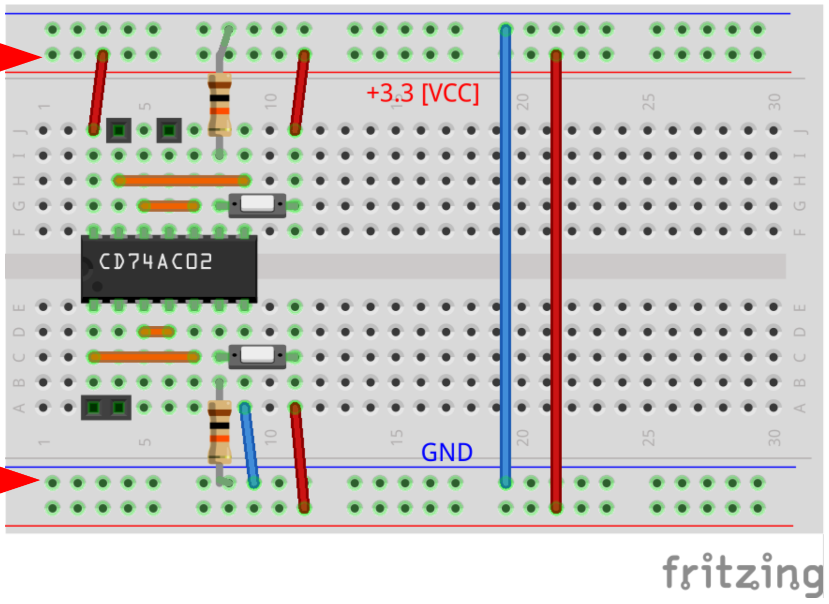
## The Bi-directional Traffic Light PCB & Breadboard

*To Ground on Protect Your Pi board*



*From +3.3 V on Protect Your Pi board*

*From Ground on PYP board*



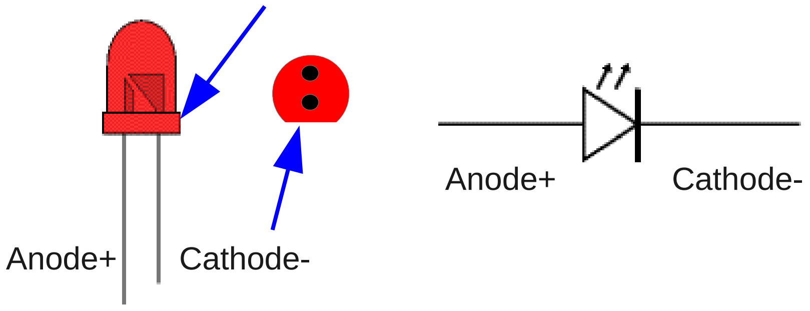
This shows how the breadboard needs to be put together. It looks a bit different than the actual board on page 6. This is the layout you should follow.

## Wiring LEDs

Unlike incandescent bulbs LEDs have polarity and so must be wired correctly to work. In addition some form of current limiting device, usually a resistor, must be connected to the LED.

> Limiting current into an LED is very important. An LED behaves very differently to a resistor in circuit. Resistors behave linearly according to Ohm's law: V = IR. For example, increase the voltage across a resistor, the current will increase proportionally, as long as the resistor's value stays the same. Simple enough. LEDs do not behave in this way. They behave as a diode with a characteristic I-V curve that is different than a resistor.
>
> For example, there is a specification for diodes called the characteristic (or recommended) forward voltage (usually between 1.5-4V for LEDs[8]). You must reach the characteristic forward voltage to turn 'on' the diode or LED, but as you exceed the characteristic forward voltage, the LED's resistance quickly drops off. Therefore, the LED will begin to draw a bunch of current and in some cases, burn out. A resistor is used in series with the LED to keep the current at a specific level called the characteristic (or recommended) forward current.
>
> https://www.sparkfun.com/tutorials/219
>
> See also: http://electronicsclub.info/leds.htm#calculate
>
> http://www.ohmslawcalculator.com/led_resistor_calculator.php

Usually the longer leg of the LED is the anode. Often there will be a flat side on the bottom of the LED on the cathode side.



There are two ways to connect an LED to an I/O line: current sourcing and current sinking.

In the **current sourcing** configuration the current-limiting resistor is connected between the LED anode and the I/O pin. The cathode of the LED connects to ground. When the I/O pin is low (at 0V) the LED is off, when the I/O pin is high (3.3 VDC for this project) the pin is the

8 Typically red, green, and yellow have 2.1 forward voltage and current limitation of 20 mA. Blue, white, violet, and "bright" LEDs have a forward voltage of 3.2 and can tolerate a current of 25 to 30 mA.

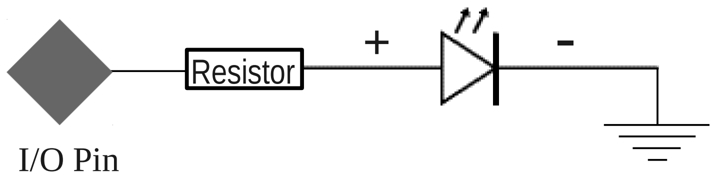
source of current and so the LED lights.

With the **current sinking** configuration the resistor is connected between the LED cathode and the I/O pin. When the I/O is high no current flows and so the LED is off. The pin must be switched low to sink current and turn the LED on.

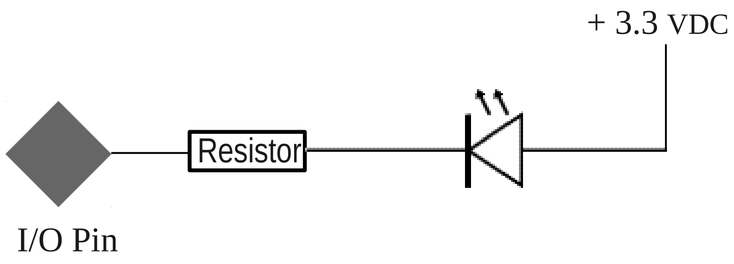
Both configurations are acceptable, however it seems that most projects I've seen use the current sourcing configuration, i.e. current-limiting resistor between the I/O pin and the LED anode.

To protect the RPi or the MCP23017 a minimum 220Ω resistor should be used – I have settled on 470Ω resistors for most applications. Yes, the LED is just bit dimmer with the larger resistor, but since I'm not using them to light-up something the differences is immaterial.

*Current Sourcing Configuration*



*Current Sinking Configuration*



Sometimes it's important to know just how much current several LEDs will draw. Remember, the maximum draw for the Raspberry Pi from the 3.3 V pin is 50 mA.  Using Ohm's Law we can calculate the current draw for a typical red LED using a 470Ω resistor:

$$I=\frac{V}{R} \qquad I=\frac{3.3-2.1}{470} \qquad I=0.0025\,A \qquad I=2.5\,mA$$

Based upon this data we can safety light four LEDs and only draw 10 mAs.

# Appendix A

## Program Listings

Three Python programs are listed. The first introduces the basic Python and MCP23017 required code segments that flash each of the twelve LEDs in sequence. The second (A.2) is the full bi-directional code written just to make the lights work properly; it is functional, but could be improved. The third (A.3) is an improved version.

The Python code may be entered in the IDLE (not the IDLE 3) program or in Geany.

## A.1 Flashing The Lights – traffic_test.py

```
#!/usr/bin/ python
# This program is for the bidirectional traffic light project from Brian Grawburg.
# As of this writing (8/2014) only Python 2.7 supports smbus. Trying to run  using Python 3 or later will result in an error

import smbus
import sys
import os
import time

bus = smbus.SMBus(1)

address1  = 0x21                 # address of MCP23017
IODIRA_1          = 0x00                 # Pin direction register, bank A
OLATA_1           = 0x14                 # Register for bank A output latches
GPIOA_1           = 0x12                 # Register for inputs
IODIRB_1          = 0x01
OLATB_1           = 0x15
GPIOB_1           = 0x13

bus.write_byte_data(address1,IODIRA_1,0x00)  # Set all to outputs
bus.write_byte_data(address1,IODIRB_1,0x00)

#Set all bank A & B pins low
bus.write_byte_data(address1,OLATA_1,0x00)                # 0b00000000
bus.write_byte_data(address1,OLATB_1,0x00)

while True:
#test all lights
        bus.write_byte_data(address1,OLATA_1,0x80)                #0b10000000
        print "North green: ", bus.read_byte_data(address1,GPIOA_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATA_1,0x40)                #0b01000000
        print "North amber: ",bus.read_byte_data(address1,GPIOA_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATA_1,0x20)                #0b00100000
        print "North red: ",bus.read_byte_data(address1,GPIOA_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATA_1,0x10)                #0b00010000
        print "East green: ",bus.read_byte_data(address1,GPIOA_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATA_1,0x8)     #0b00001000
        print "East amber: ",bus.read_byte_data(address1,GPIOA_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATA_1,0x4)     #0b00000100
        print "East red: ",bus.read_byte_data(address1,GPIOA_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATB_1,0x1)
        print "North Left red: ",bus.read_byte_data(address1,GPIOB_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATB_1,0x02)
        print "North Left amber: ",bus.read_byte_data(address1,GPIOB_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATB_1,0x04)
        print "North Left red: ",bus.read_byte_data(address1,GPIOB_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATB_1,0x8)
        print "East Left green: ",bus.read_byte_data(address1,GPIOB_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATB_1,0x10)
        print "East Left amber: ",bus.read_byte_data(address1,GPIOB_1)
        time.sleep(.5)

        bus.write_byte_data(address1,OLATB_1,0x20)
        print "East Left red: ",bus.read_byte_data(address1,GPIOB_1)
        time.sleep(.5)

#stop process
        bus.write_byte_data(address1,OLATA_1,0x00)          #0b00000000
        bus.write_byte_data(address1,OLATB_1,0x00)
        print "End:"
        sys.exit()
```

## A.2 Bidirectional operating light traffic_3.py - First Try Version

```
#!/usr/bin/python
# This program is for the bidirectional traffic light project from Brian  Grawburg.
# As of this writing (8/2014) only Python 2.7 supports smbus. Trying to run it using Python 3 or later will result in an error

import smbus
import sys
import os
import time

bus = smbus.SMBus(1)

address1        = 0x21          # address of  left MCP23017 on MyPiShop board
IODIRA_1        = 0x00          # Pin direction register, bank A
OLATA_1         = 0x14          # Register for bank A output latches
GPIOA_1         = 0x12          # Register for inputs
IODIRB_1        = 0x01
OLATB_1         = 0x15
GPIOB_1         = 0x13
address2        = 0x20          # address of right MCP23017
IODIRA_2        = 0x00
OLATA_2         = 0x14
GPIOA_2         = 0x12

light_on = 5            # number of seconds each light is on
both_red = 1            # number of seconds both reds are on

bus.write_byte_data(address1,IODIRA_1,0x00)  # Set all to outputs
bus.write_byte_data(address1,IODIRB_1,0x00)  # Set all to outputs
bus.write_byte_data(address2,IODIRA_2,0x07)  # Set 1, 2, and 3 as input, others as as outputs

#Set all bank A, chip 21H as low
bus.write_byte_data(address1,OLATA_1,0x00)              # 0b00000000
#Set all bank B, chip 21H as low
bus.write_byte_data(address1,OLATB_1,0x00)              # 0b00000000

def cycle_1():
        # Straight East Green, North Red, left turns Red
        bus.write_byte_data(address1,OLATA_1,0x30)                      #0b00110000
        bus.write_byte_data(address1,OLATB_1,0x24)                      #0b00100100
        time.sleep(light_on)

def cycle_2():
        # Straight East Amber, North Red
        bus.write_byte_data(address1,OLATA_1,0x28)                      #0b00101000
        bus.write_byte_data(address1,OLATB_1,0x24)                      #0b00100100
        time.sleep(light_on)

def cycle_3():
        # East & North Red, including left turn lanes
        bus.write_byte_data(address1,OLATA_1,0x24)                      #0b00100100
        bus.write_byte_data(address1,OLATB_1,0x24)                      #0b00100100
        time.sleep(both_red)

def cycle_4():
        # North Green, East Red
        bus.write_byte_data(address1,OLATA_1,0x84)                      #0b10000100
        bus.write_byte_data(address1,OLATB_1,0x24)                      #0b00100100
        time.sleep(light_on)
```

```
def cycle_5():
        # North Amber, East Red
        bus.write_byte_data(address1,OLATA_1,0x44)            #0b01000100
        bus.write_byte_data(address1,OLATB_1,0x24)            #0b00100100
        time.sleep(light_on)

def cycle_6():
        # Both left turns Green, straights Red
        bus.write_byte_data(address1,OLATA_1,0x24)            #0b00100100
        bus.write_byte_data(address1,OLATB_1,0x09)            #0b00001001
        time.sleep(4)

def cycle_7():
        # Both lefts Amber
        bus.write_byte_data(address1,OLATA_1,0x24)            #0b00100100
        bus.write_byte_data(address1,OLATB_1,0x12)            #0b00010010
        time.sleep(4)

try:
        while True:
                buttonPressed=bus.read_byte_data(address2,GPIOA_2)
                #print buttonPressed
                if buttonPressed == 0:
                        cycle_1()
                        cycle_2()
                        cycle_3()
                        cycle_4()
                        cycle_5()
                        cycle_3()
                else:
                        #print bus.read_byte_data(address2,GPIOA_2)
                        cycle_6()
                        cycle_7()
                        cycle_3()
                        #reset the flip-flop
                        bus.write_byte_data(address2,OLATA_2,0x18)
                        bus.write_byte_data(address2,OLATA_2,0x00)    # clears all

except KeyboardInterrupt:
        bus.write_byte_data(address1,OLATA_1,0x00)
        bus.write_byte_data(address1,OLATB_1,0x00)
        bus.write_byte_data(address2,OLATA_2,0x00)
```

## A.3 Improved Python Code – new_traffic.py

```
#!/usr/share/python
# This program is for the bidirectional traffic light project from Brian  Grawburg.
# As of this writing (8/2014) only Python 2.7 supports smbus. Trying to run it using Python 3 or later will result in an error

import smbus
import sys
import os
import time

bus = smbus.SMBus(1)

address1        = 0x21
address2        = 0x20
IODIRA          = 0x00
IODIRB          = 0x01
GPIOA           = 0x12
GPIOB           = 0x13
OLATA           = 0x14
OLATB           = 0x15

light_on = 5    # Straight lights
both_red = 1
left_on = 4     # Left lights

io_setting = [
                [0,0,0],                                # Reset: all off, no delay
                [0x30,0x24,light_on],   # Straight East Green, North Red, L-turns Red
                [0x28,0x24,light_on],   # Straight East Amber, North Red
                [0x24,0x24,both_red],   # East & North Red, L-turns Red
                [0x84,0x24,light_on],   # North Green, East Red
                [0x44,0x24,light_on],   # North Amber, East Red
                [0x24,0x09,left_on],    # Both L-turns Green, Straights Red
                [0x24,0x12,left_on]     # Both L-turns Amber
                ]

def cycle(index):
        bus.write_byte_data(address1,OLATA,io_setting[index][0])
        bus.write_byte_data(address1,OLATB,io_setting[index][1])
        time.sleep(io_setting[index][2])

bus.write_byte_data(address1,IODIRA,0x00)
bus.write_byte_data(address1,IODIRB,0x00)
bus.write_byte_data(address2,IODIRA,0x03)

cycle(0)

bus.write_byte_data(address2,OLATA,0x00)

try:
        while True:
                buttonPressed=bus.read_byte_data(address2,GPIOA)
                if buttonPressed == 0:
                        cycle(1); cycle(2);
                        cycle(3); cycle(4);
                        cycle(5); cycle(3)
                else:
                        cycle(6); cycle(7); cycle(3)
                bus.write_byte_data(address2,OLATA,0x18)
                bus.write_byte_data(address2,OLATA,0x00)

except KeyboardInterrupt:
    cycle(0)
    bus.write_byte_data(address2,OLATA,0x00)
```

# Appendix B

## Pull-Up and Pull Down Resistors

## Pull-Up and Pull-Down Resistors

[The following was written by Darren Grant and was part of an article in Issue 2, June 2012 of *The MagPi* magazine. In this article he connected resistor R3 to the 3.3 VDC rail whereas I have chosen to connect the resistor to the ground rail.]

### Digital Logic States

As mentioned earlier computers see the world as a series of zeros and ones. The Raspberry Pi uses 3.3V CMOS logic meaning that a binary 1 is created by applying a voltage of 3.3V to an input. When the voltage is present it is referred to as a logic HIGH state. Alternatively when the GPIO pin is connected to 0V a binary 0 or logic LOW state is created. In our switch example pressing the switch creates a LOW state by connecting the IO port to 0V. *Brian's Note: Because I have connected the resistor to the ground rail the unpressed value of the pin is LOW. Pressing the button sends the pin HIGH.*[9]

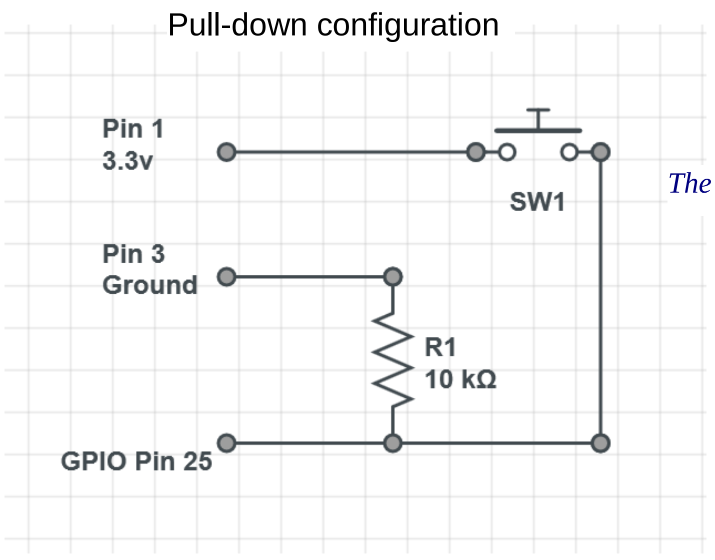
Digital logic in fact has three possible states known as Tri-State logic. We have already discussed HIGH (+3.3V) and LOW (0V) states but we also have a third state that is called FLOATING. Basically floating means the state is not clearly defined, attempting to determine the status of a connection that is floating may result in unpredictable results. You might be thinking that if the pin is not connected to a voltage it must be low: the problem is we can not guarantee that we will get a reliable LOW signal by simply relying on the absence of a voltage on the pin.

To illustrate the problem, imagine that we hold a piece of ribbon between two gate posts, the two gate posts would be connected by the ribbon. Now if we let one end of the ribbon go and allow it to fall to the ground the two posts are no longer connected. But what happens when the wind blows? The ribbon will flap around where sometimes it will be in contact with the ground and other times it will touch or even become entangled with the other post. To prevent the ribbon moving in the wind we would tie it either to the ground or the post so it wouldn't move unless we wanted it to. We need to do the same thing with a logic circuit, to avoid unpredictable behavior we tie the connection to either

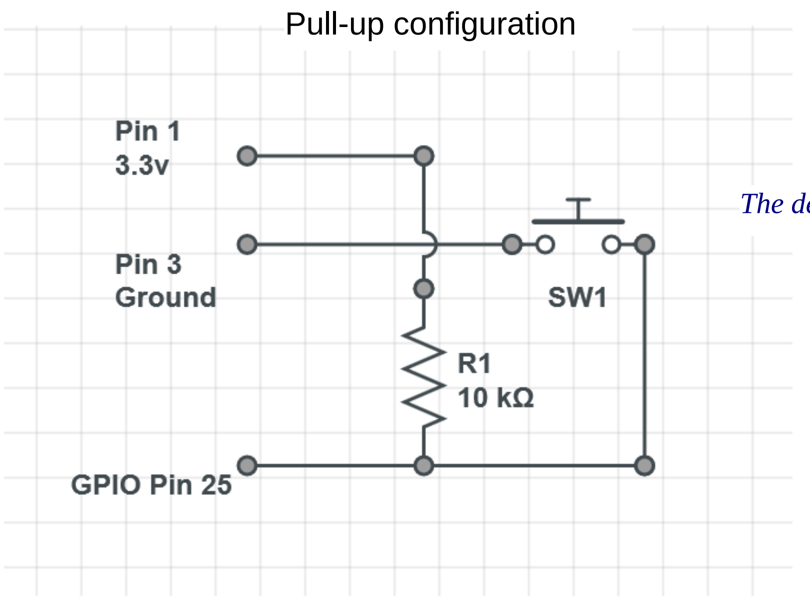
9 I learned how to write code for programming devices such as this project by taking a Programmable Logic Controller (PLC) course. The usual procedure with a PLC is to set the pin low (0) when it is not activated and high (1) when it is considered 'on'.

3.3V or 0V using what is known as a pull-up or pull-down resistor that effectively creates a default state. In this experiment we have used a pull-up resistor so the IO port will always be connected to +3.3V making it the default state. In our example when reading the state of the IO pin it will always be high until we press the switch that will change it to low. *Brian's Note: Because I have connected the resistor to the ground rail the default state of the pin is LOW. Pressing the button sends the pin HIGH.*

Keith Peters, www.bit-101.com, created some easily understood drawings that illustrate pull-down and pull-up resistors.



*The default state of GPIO25 is low.*



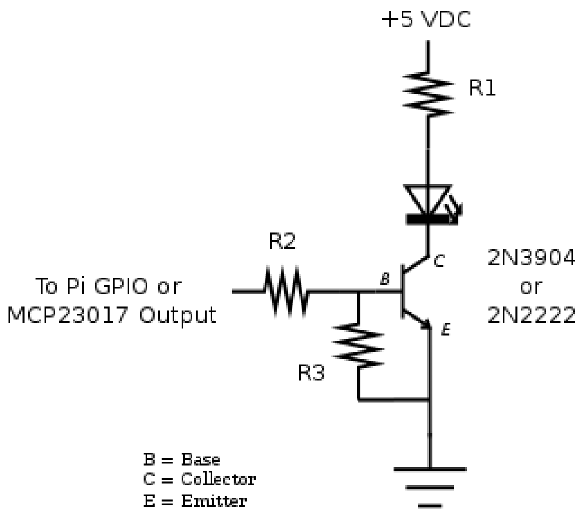
*The default state of GPIO25 is high.*

# Appendix C

## Using an External Power Supply to Light the LEDs in Conjunction with Transistors

Although the Pi and the MCP23017 can safely handle turning on four LEDs simultaneously, at least when suitable resistors are used and are powered by +3.3 VDC, there may be a circumstance when it is desirable to use an external +5 VDC to power the LEDs. In this case transistors should be used in conjunction with the MCP23017 to fully isolate both the Pi and the MCP from the power supply.

If an MCP23017 is used the same Python code can run the new set-up. If the transistors are connected directly to the GPIO pins the code will need to be altered.



The schematic shows how one LED and transistor would be set-up.

R1 = 220Ω

R2 = 3.3kΩ

R3 = 33kΩ

# Appendix D

## A Quick Refresher on

## High, Low, Binary, and Hexadecimal

## High Or Low

In the world of digital electronics there are only two states of existence: you're either ON or OFF. For computers and microprocessors, whose language is binary (1s and 0s), that becomes HIGH or LOW. A binary one usually means a positive voltage (either +3.3 VDC or +5 VDC) is being output and a binary zero means that a 0 voltage is being output. The Raspberry Pi outputs +3.3. Now here comes the tricky part. If a microprocessor/computer sends an attached device a HIGH signal (a digital 1) does that turn the device on or off?

Several years ago I took an 18 month course at a local community college on Programmable Logical Controllers (PLCs). These devices are what control much of the automation equipment in industrial plants, and, *it is what runs the museum's **Race The Wild** exhibit.* PLC programming usually sets the output to an attached device such as an electric motor starter as HIGH when it is to be considered ON. That is my approach to coding. However, it is not uncommon to see Python code written for the Raspberry Pi that considers LOW as the state to indicate an attached device is ON. To me, that seems counter-intuitive and so I don't typically use that method. Therefore, in this project when the output is HIGH (a digital 1) the attached device will be ON.

## Binary: Bits and Bytes

**http://www.fayewilliams.com/2014/02/18/binary-numbers-explained/**

**http://php.about.com/od/programingglossary/qt/binary.htm**

## Hexadecimal

**http://www.binaryhexconverter.com/hex-to-binary-converter**

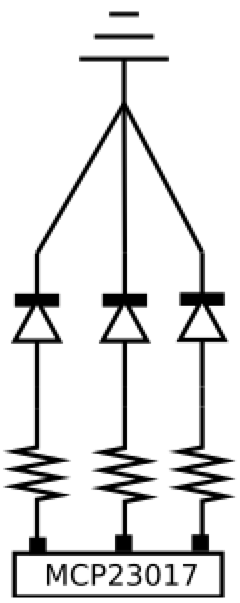
**http://www.vlaurie.com/computers2/Articles/hexed.htm**

**http://thestarman.pcministry.com/asm/hexawhat.html**

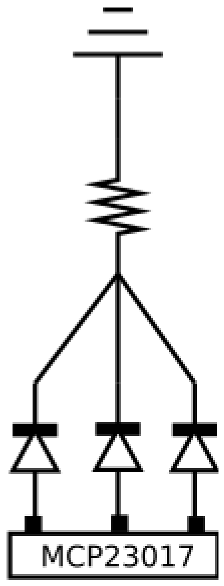
# Appendix E

## A Bad Alternative Way to Connect the LEDs To The MCP23017

The LEDs on the Traffic Light PCB are wired such that each individual LED has it's own resistor between the output pin of the MCP23017 and the LED. In most applications this would be entirely appropriate and would be required if at anytime all three LEDs were turned on simultaneously [Option 1]. However, if only one LED in a segment could ever be turned on at one time a single resistor is all that would be needed [Option 2].



**Option 1**



**Option 2**

**Option 2 is NOT a good way to set up the LEDs; it is bad design. I show it here only to warn you NOT to even consider it.**