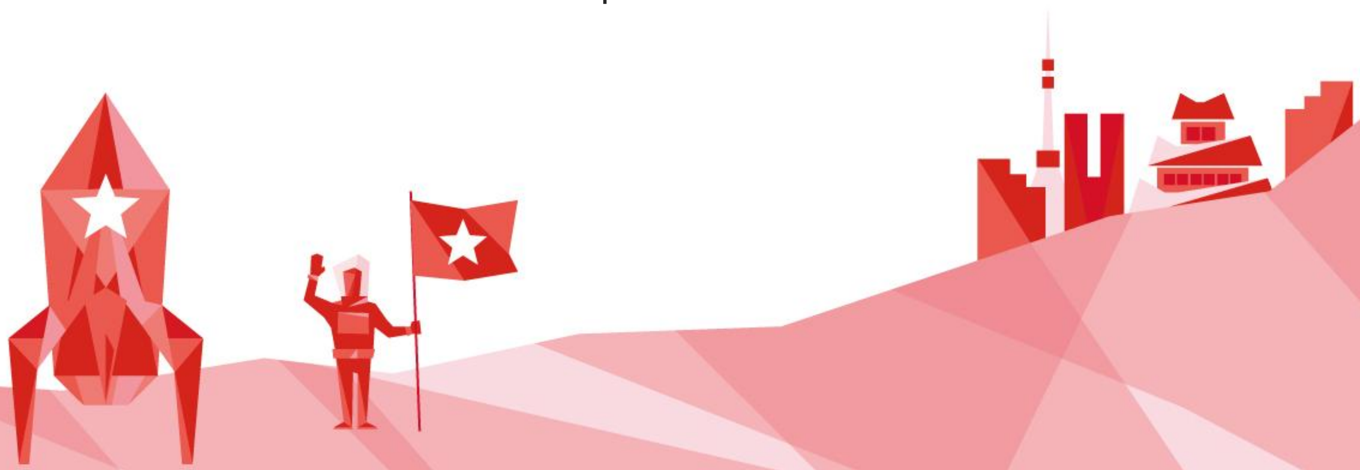
Controlling Hardware with Python

PyZurich Meetup 2015-10-29

Danilo Bargen (@dbrgn), Webrepublic AG

Webrepublic

- Leading Digital Marketing agency in Switzerland
- Owner-managed and independent
- Established in 2009
- Based in Zurich and Lausanne
- Portfolio of 120+ national and international brands
- Full coverage of digital performance path
- Sparring partner for ambitious organizations
- Own software development team



Me

- Software Engineer
- Mostly interested in Python and Rust
- Bought a Raspberry Pi the day it became available
- Founded a Hackerspace in Rapperswil in 2013 (coredump.ch)
- NOT a hardware or electronics expert! =)

Twitter: @dbrgn
Blog: blog.dbrgn.ch

Agenda

1. Linux, Python and Hardware
2. The Raspberry Pi 2 Hardware
3. Electronics Crashcourse
4. Example: Simple Circuit
5. Input
6. Example: Using the RPLCD Library
7. RPLCD Implementation Details

About this talk

- I'll try to keep the language as simple as possible.
- Target audience: Python developers (maybe of the webdev flavor) that have no or little experience with hardware.
- Correct me if something is wrong, but...
- ...simplifications are being used on purpose. This isn't a lecture.

ETA: 60–90 minutes

1: Linux, Python and Hardware

A complicated relationship.

Controlling Hardware with Code

- Usually C/C++ or Assembly is being used to control hardware
- Realtime performance / exact timing is often important
- Deterministic runtimes: Knowing how long a CPU cycle takes

Why Linux?

- A regular Linux kernel does not guarantee timing
- The Linux kernel can be configured to guarantee specific response times
- The Raspbian kernel is not realtime
- (I won't get into the details of what defines "realtime" =))

Why Python?

- Python is a high-level garbage collected language
- Not terribly well suited for controlling hardware
- No timing guarantees due to GC pauses

Then why Linux + Python?

- Turns out that timing is not always that important
- Python is easy to learn
- Python is easy to use
- Good to get started with hardware

2: The Raspberry Pi Hardware

A great platform for n00b hardware hackers.

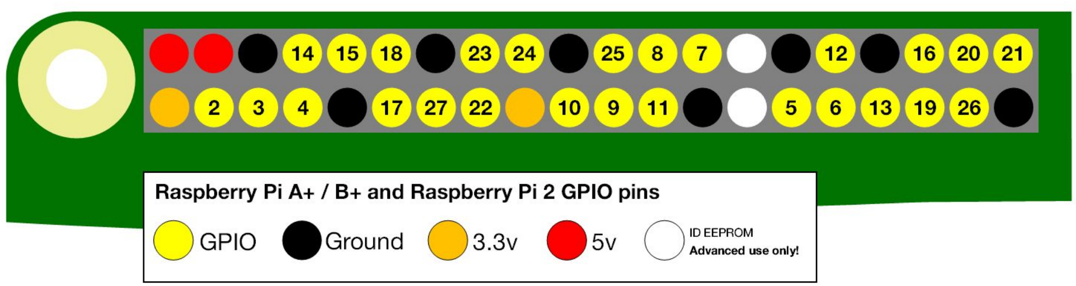
The Raspberry Pi 2

- 900 MHz ARM Cortex-A7 CPU
- 1 GiB RAM
- 4 USB Ports
- 40 GPIO Pins
- HDMI / Ethernet / Audio Jack / Composite Video / Camera Interface / Display Interface / MicroSD
- Serial communication: UART / I^2C / SPI
- Other stuff I haven't covered here

UART, I^2C, ARM, GPIO, WTF?

- GPIO stands for General Purpose Input / Output
- Pins to communicate with external devices
- This is what they look like:

- Pin numbering:



Public Service Announcement

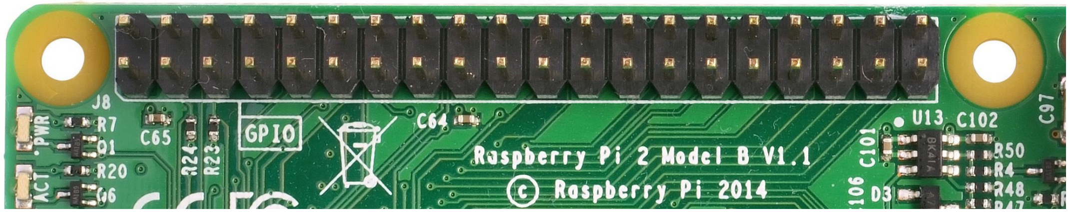
- Complicated abbreviations make everything sound hard
- Most stuff is actually easy
- **Never** think “this is too hard for me”!
- Here are some (simplified) translations:
  - GPIO: “Wires sticking out of the hardware that can be set to 5V or 0V”
  - UART: “Two wires for sending and receiving”
  - Bus: “A cable with many devices on it”
  - SPI: “Like UART with support for multiple devices and faster”
  - Syscalls: The Linux kernel API
  - Driver: “An API client that sends 1’s and 0’s through a wire”
  - Kernel driver: “A driver that is a pain to debug”
  - Interrupt: “A high-priority callback”

Public Service Announcement

NEVER think
“this is too hard for me”!

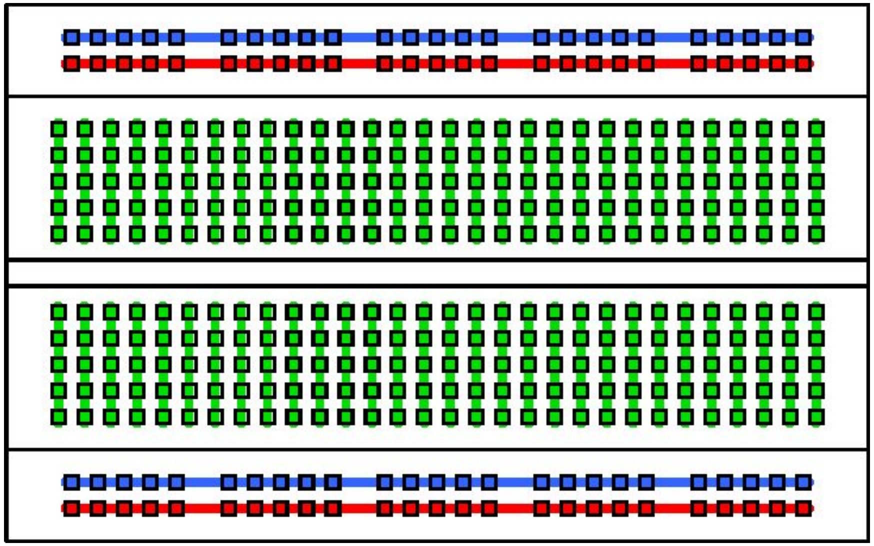
Back to the GPIO

- You can stick cables into these pins.
- Make sure you use the right pin.
- You're responsible for the wiring! Avoid short circuits.
- Using a breadboard helps.



What is a "breadboard"?

- A breadboard helps you to connect wires.
- This is how it works:



What is a "short" or "shorting"?

- A "short" is short for "short circuit"
- This means that you connect a voltage source (e.g. the 5V pin) with the ground pin without having anything in between that uses some of the current.
- The "something in between" could be a resistor or a LED
- Don't do it!

https://www.youtube.com/watch?v=PqyUtQv1WoQ

Important facts about the GPIO pins

- You need to configure the pins as either input- or output-pins
- They use 3.3V internally, so don't feed them 5V!
- Maximum current draw per GPIO pin is 16 mA.
- Maximum current draw for all GPIO pins is 50 mA.

http://elinux.org/RPi_Low-level_peripherals

http://raspberrypi.stackexchange.com/a/9299/6793

- What is a "mA"?

3: Electronics Crashcourse

The essentials you need to know.

The Water Analogy

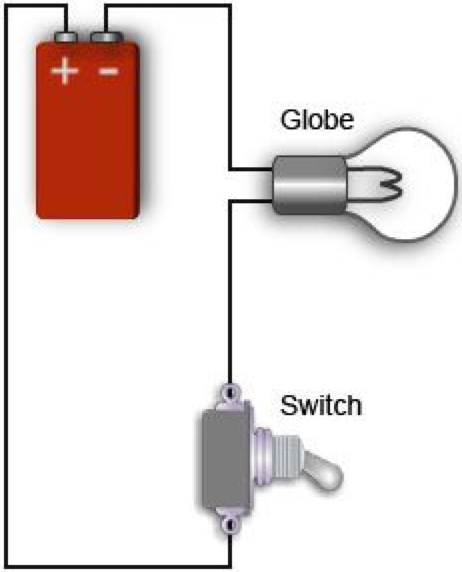


Figure 1: Electrical

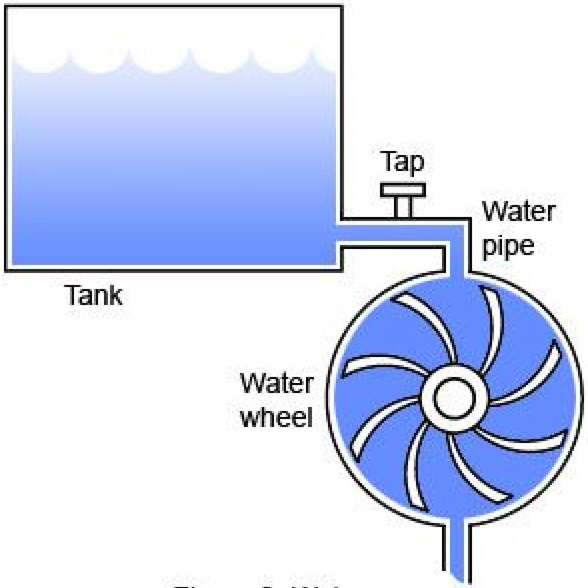
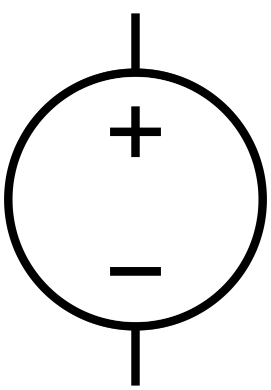


Figure 2: Water

What is current?

- Movement of electrons through a conductor
- Electrons are negatively charged
- Electrons move from one side of a power source to the other side.
- Measured in Amperes (A, Amps), symbol is I
- Analogue to the amount of water in a pipe

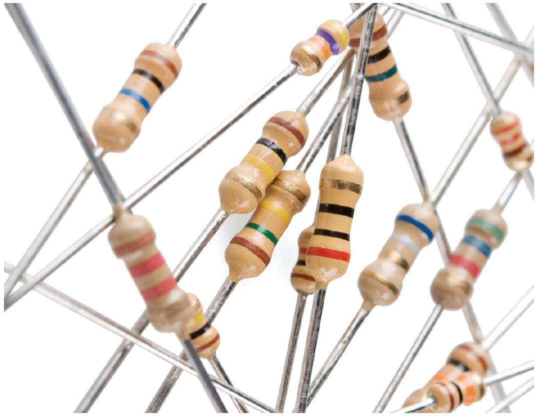


What is voltage?

- Electronic potential difference between two points
- Analogue to the pressure in a pipe
- Measured in Volts (V, Voltage), symbol is U
- An AA battery has 1.3–1.5 V
- The Swiss electricity grid uses ~230V

What is resistance (R)?

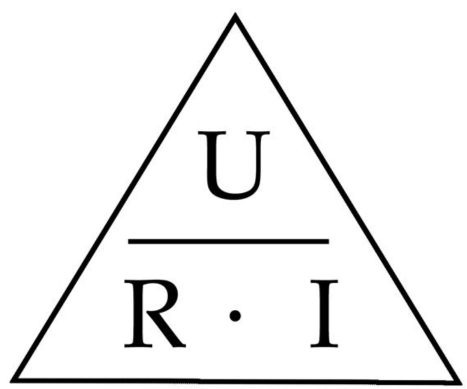
- Something that hinders the flow of electricity
- Measured in Ohms (Ω), symbol is R
- A resistor or an LED has some resistance
- An open switch has infinite resistance



Ohm's Law

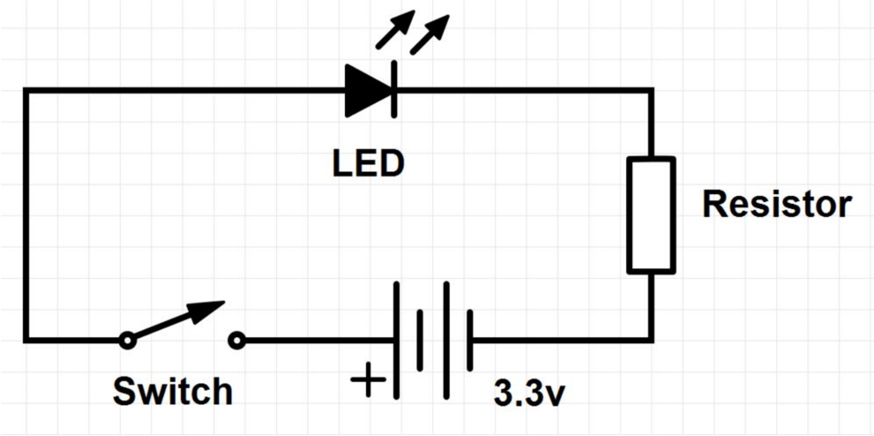
$$R = \frac{U}{I} \quad \Leftrightarrow \quad U = R \cdot I \quad \Leftrightarrow \quad I = \frac{U}{R}$$

- The most important formula you need to know.
- R is resistance, U is voltage, I is current
- Example: If you increase the resistance but still want the same current flow, then you need to increase the voltage. If voltage stays the same, the current decreases.



Circuits

- For electricity to flow, a circuit always needs to be closed.
- For a simple circuit, that's easy.



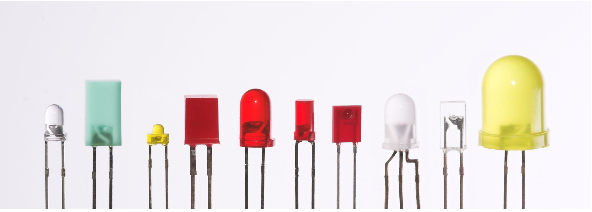
- For multiple connected circuits, that's also easy!

  You just need Kirchhoff's circuit laws. Google them!

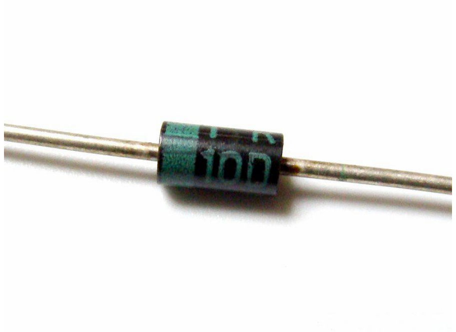
Some electrical components

Resistors

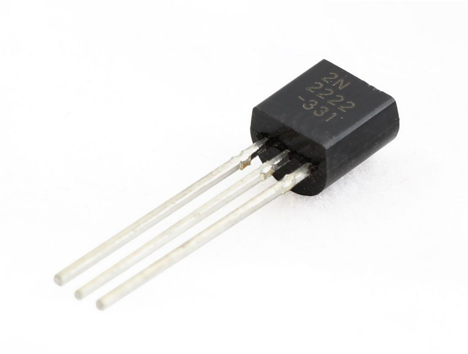
Capacitors



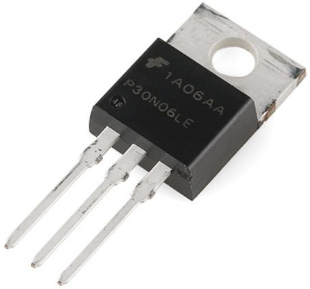
LEDs



Diodes



Transistors



MOSFETs

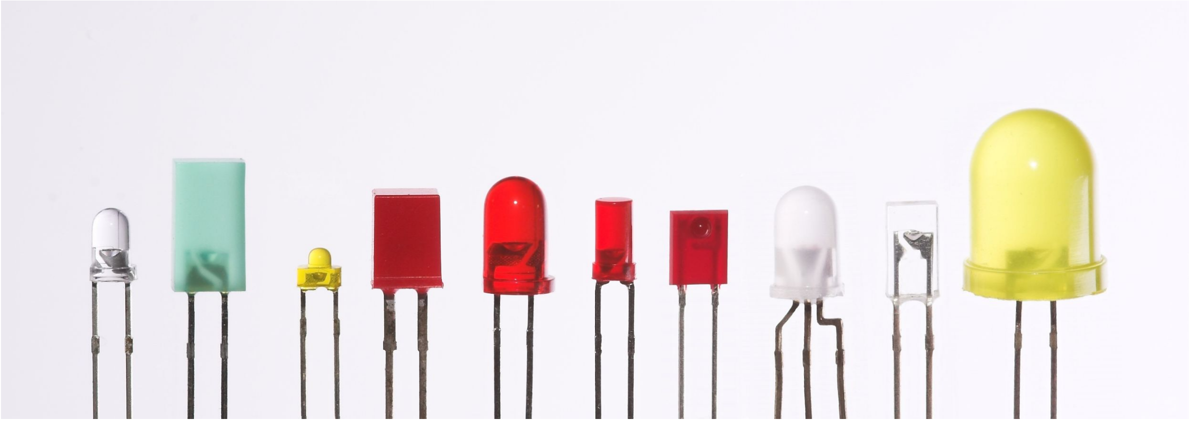
Resistors

- Provide resistance
- Measured in Ohms (Ω)
- Color coded

Capacitors

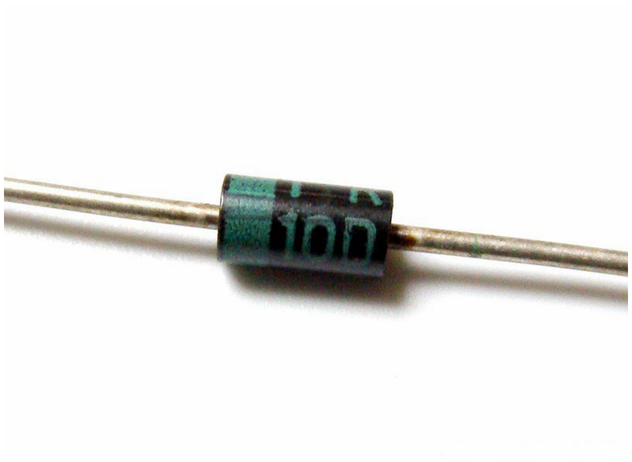
- Think of them as a small battery that can be charged and discharged very quickly
- Measured in Farad (F)

LEDs



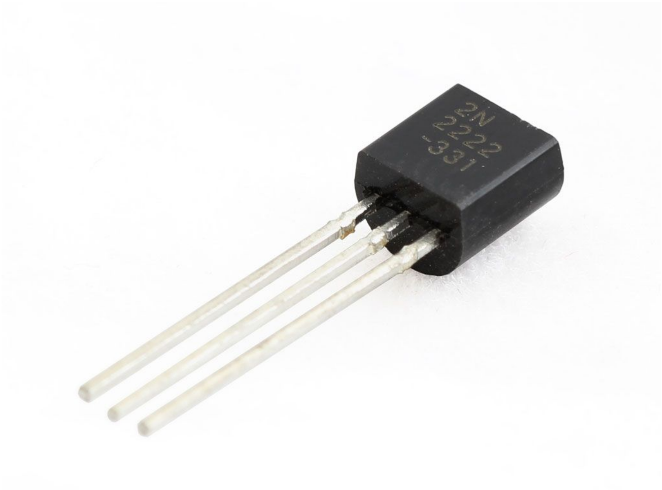
- Need no introduction
- Only allow current to flow in one direction!
- Legs are called “anode” (+, long leg) and “cathode” (-, short leg)

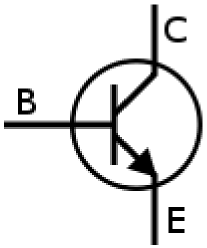
Diodes



- Allow current to flow only in one direction
- Like a valve
- A LED is a special version of a diode

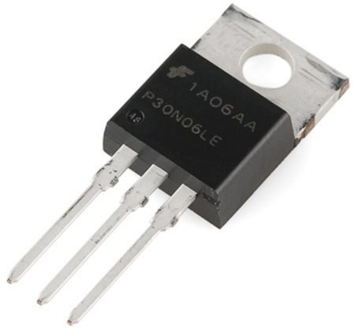
Transistors





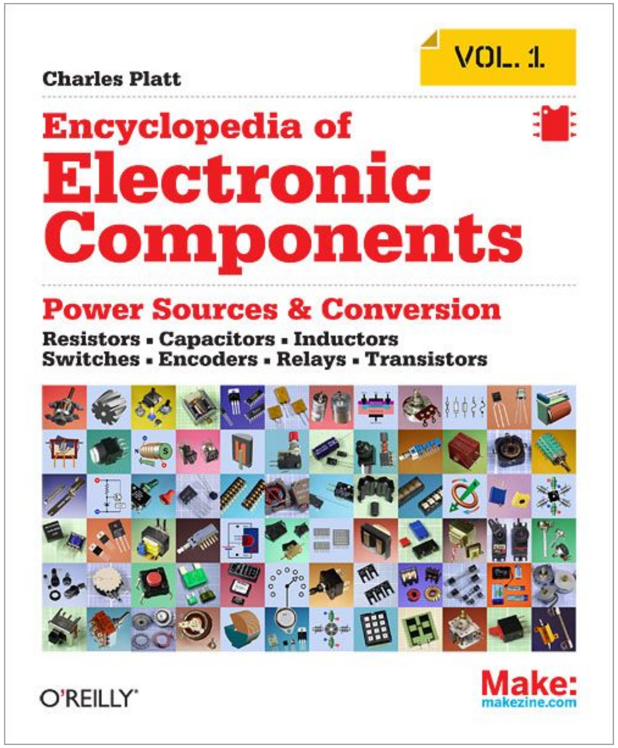
- Kind of important for computers :)
- Think of them like an electrical switch
- If you feed enough current to the base B, the current flows freely from the collector C to the emitter E (for a N-channel BJT transistor). There are also other variants.
- Can also be used as amplifiers.

MOSFETs



- A special type of transistor (metal–oxide–semiconductor field-effect transistor).
- Needs voltage instead of current at the base (called "gate")
- Can be used to switch high-power devices with low-power microcontrollers

More components



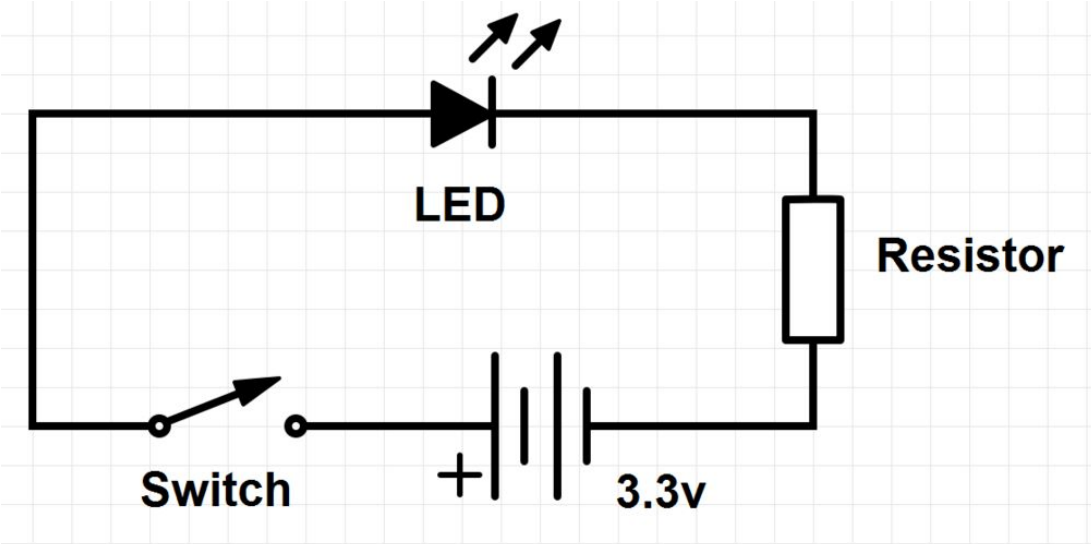
http://shop.oreilly.com/product/0636920026105.do

4: Example: Simple Circuit

Hello world!

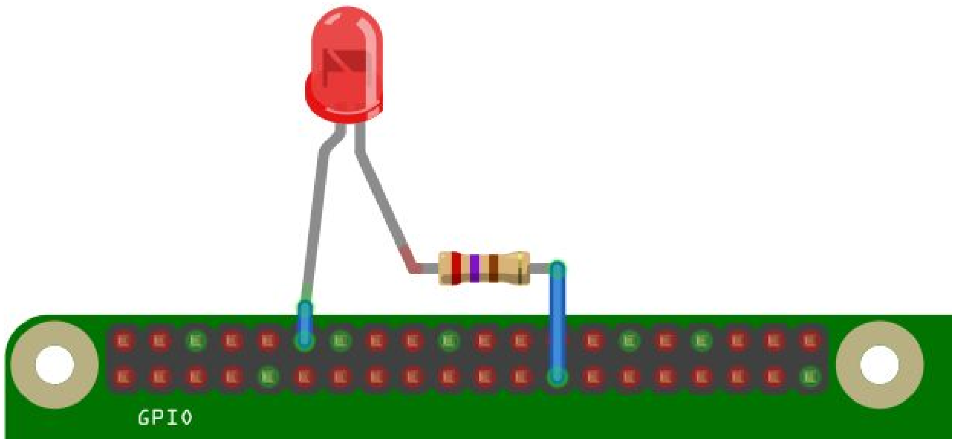
Let's blink an LED

- The "Hello World" of electronics and microcontrollers.



Connect to GPIO pins

- Add a resistor to avoid frying your GPIO pins
- Circuit goes from a GPIO pin to GND (0V)



Controlling the GPIO pins

By setting the GPIO pin to HIGH (3.3 V) we can turn the LED on.

```
import RPi.GPIO as GPIO
led = 18
GPIO.setup(led, GPIO.OUT)
GPIO.output(led, 1)
```

Blinking the LED

You can use a regular loop to toggle the LED every second.

```
import RPi.GPIO as GPIO
import time
led = 18
GPIO.setup(led, GPIO.OUT)
state = 1
while True:
    GPIO.output(led, state)
    state ^= 1
    time.sleep(1)
```

Don't forget to clean up

If you want to be a good citizen™, clean up after every program to make pins available again to other scripts.

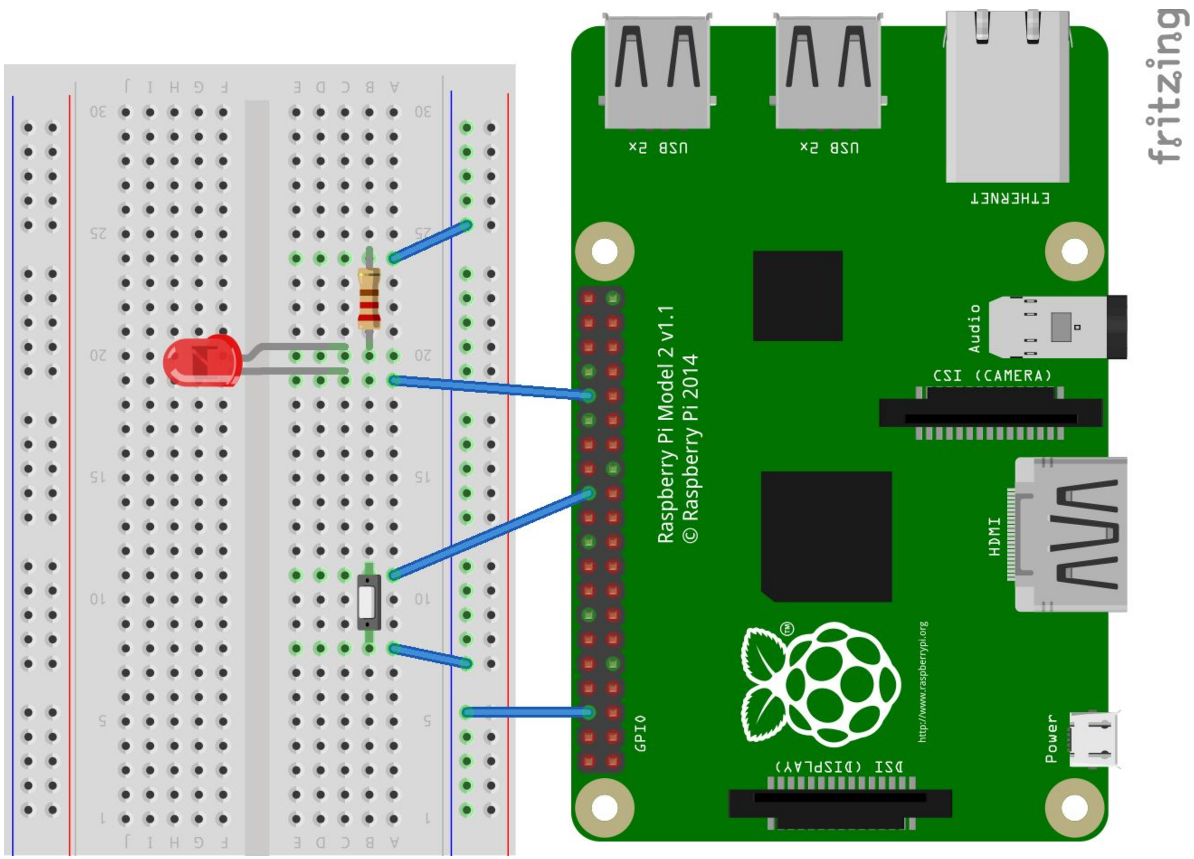
```
try:
    main_loop()
except Exception:
    GPIO.cleanup()
```

5: Input

Let's look at reading input values, debuncing and interrupts.

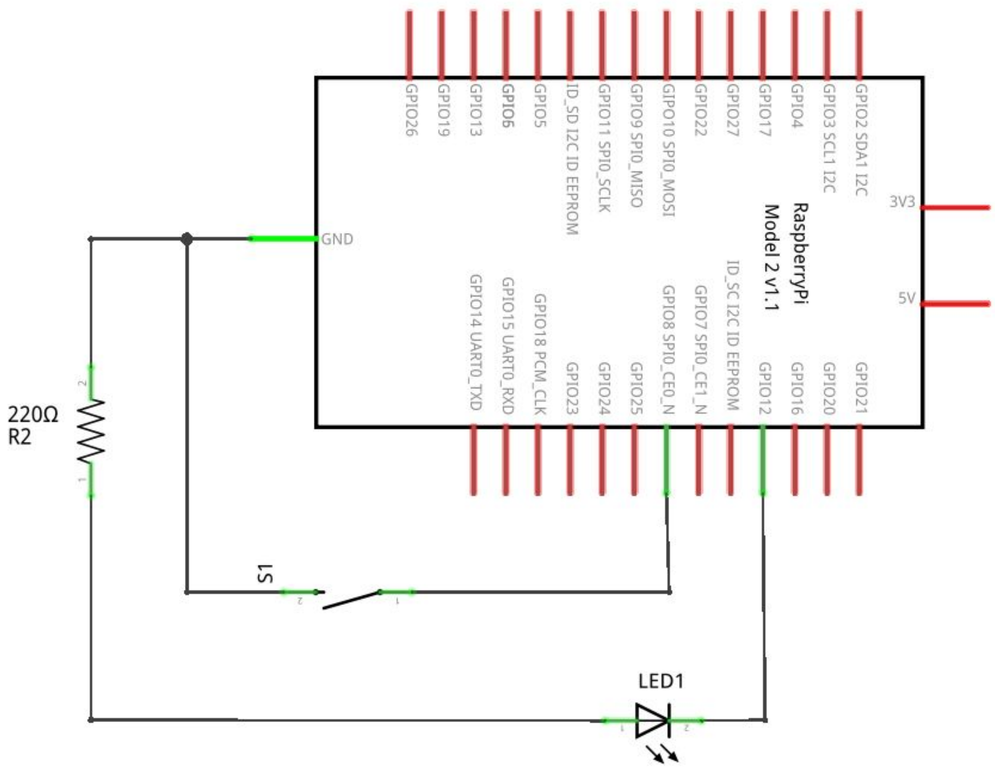
Reading Input

Let's read the state of a button and turn on the LED accordingly.



The Schematic

You should learn to read schematic diagrams! =)



Reading Input

We can set a GPIO pin to **INPUT mode**.

If we don't push the button, the GPIO pin "floats". It is neither always HIGH nor always LOW, it has an **undefined state** that may be affected by static electricity.

We can enable internal **pull-up resistors** to make the pin HIGH by default.

```
button = 8
GPIO.setup(button, GPIO.IN, GPIO.PUD_UP)
```

Reading Input

Now that the GPIO pin is configured, we can read the current input value.

```
value = GPIO.input(button)
if value:
    print(“GPIO pin is HIGH”)
else:
    print(“GPIO pin is LOW”)
```

Reading Button State

Remember that the pin is high by default. The button pulls it to LOW.

```
button_pressed = not GPIO.input(button)
if button_pressed:
    print("Button pressed")
else:
    print("Button not pressed")
```

Turning on the LED

We can now turn on the LED depending on the button state.

```
button_pressed = not GPIO.input(button)
GPIO.output(led, button_pressed)
```

Triggering events

We could also poll the button to trigger events.

```
was_pressed = 0
while True:
    button_pressed = not GPIO.input(button)
    if button_pressed and not was_pressed:
        toggle_led()
    was_pressed = button_pressed
```

Polling?

- If you use a busy-loop like in our example, our CPU load will be very high.
- If you're a webdev you know that polling sucks.
- Instead, you want to wait for an event or register a callback.
- Turns out, we can! In hardware-land, events are called interrupts.

Waiting for events

The wait_for_edge method blocks until an event occurs.

```
while True:
    GPIO.wait_for_edge(button, GPIO.FALLING)
    toggle_led()
```

Y U NO CALLBACK?

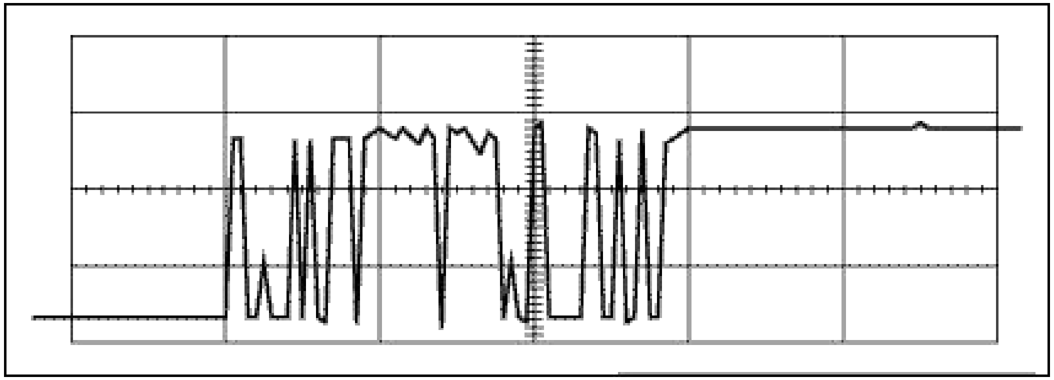
We can also use threaded callbacks (aka "interrupt handlers" or "interrupt service routines"):

```
def callback(channel):
    print('Button pushed on GPIO %s!" % channel)
    toggle_led()

GPIO.add_event_detect(button, GPIO.FALLING,
                                  callback=callback)
```

Bugs, bugs everywhere!

- If you actually implement this code, you will notice that the LED toggling is buggy.
- Sometimes it turns on properly, sometimes it flickers, or it stays off.
- The reason is physical switch bouncing:



Software debouncing

Simple software debouncing is pretty straightforward:

```
was_pressed = 0
while True:
    button_pressed = not GPIO.input(button)
    if button_pressed and not was_pressed:
        time.sleep(0.2)
        still_pressed = not GPIO.input(button)
        if still_pressed:
            toggle_led()
    was_pressed = button_pressed
```

We like free stuff

We can also get this for free though:

```
def callback(channel):
    print('Button pushed on GPIO %s!" % channel)
    toggle_led()

GPIO.add_event_detect(button, GPIO.FALLING,
                      callback=callback,
                      bouncetime=200)
```

Get creative!

Now go and code some more useful stuff with this:

```
twitter_pin = 2; cat_pin = 3

def callback(channel):
    if channel == twitter_pin:
        tweet('The button was pressed!')
    elif channel == cat_pin:
        food_dispenser.dispense(1)

pins = [twitter_pin, cat_pin]
GPIO.add_event_detect(pins, GPIO.FALLING,
                callback=callback, bouncetime=200)
```

6: Example: Using RPLCD

A library for writing to HD44780 character LCDs.

What is a character LCD?

- A char LCD is a simple display that can display pixel characters.
- Usually 8x5 pixel characters.

What is “HD44780”?

- A char LCD controlling chip by Hitachi
- The most widely used character LCD controller
- Many compatible controllers not by Hitachi

What is RPLCD?

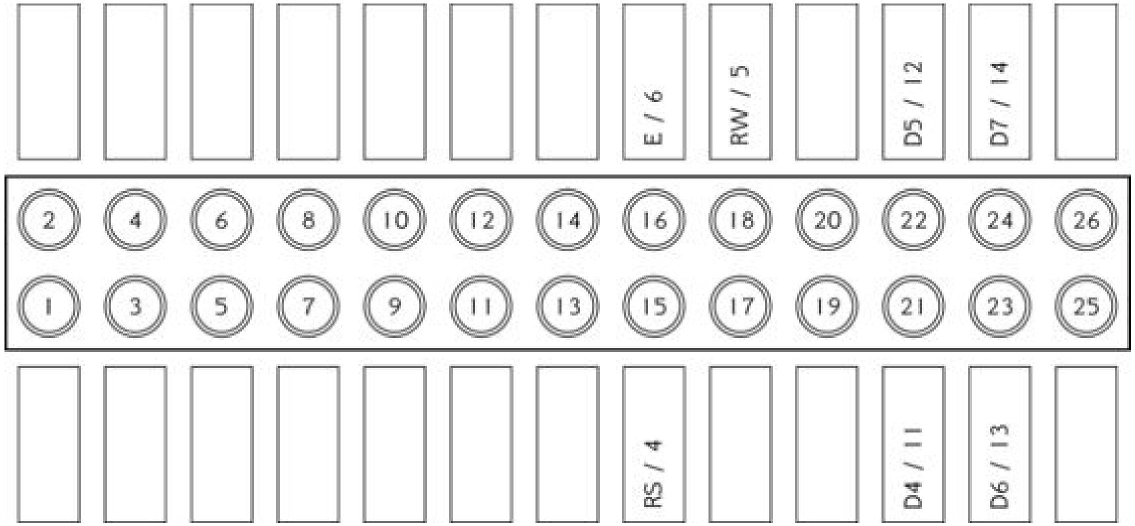
- A Python library I wrote in 2013 to control HD44780 displays.
- Idiomatic Python 2 / 3
- Properties instead of getters / setters
- Simple test suite (with human interaction)
- Caching: Only write characters if they changed
- Support for custom characters
- No external dependencies
- MIT licensed

https://github.com/dbrgn/RPLCD

https://pypi.python.org/pypi/RPLCD/

Wiring

- Wiring is configurable
- LCD can run both in 4 bit and in 8 bit mode
- Here's the default wiring for 4 bit mode:



See also: https://learn.adafruit.com/character-lcds/wiring-a-character-lcd

Usage example

```
$ sudo pip install RPLCD
$ sudo python3
>>> from RPLCD import CharLCD
>>> lcd = CharLCD()
>>> lcd.write_string('Raspberry Pi HD44780')
>>> lcd.cursor_pos = (2, 0)
>>> lcd.write_string(
...     'http://github.com/\n\rdbrgn/RPLCD')
```

Context managers

```
from datetime import date
import time
from RPLCD import CharLCD, cleared

lcd = CharLCD()

while True:
    with cleared(lcd):
        today = date.today().isoformat()
        lcd.write(today)
    time.sleep(1)
```

Properties

```
from RPLCD import CharLCD, Alignment, CursorMode

lcd = CharLCD()

lcd.display_enabled = True
lcd.cursor_pos = (0, len(“Python”))
lcd.cursor_mode = CursorMode.blink
lcd.text_align_mode = Alignment.right
lcd.write(“nohtyP”)
```

Other stuff

- You can build additional functionality on top of the library.
- For example scrolling text: https://blog.dbrgn.ch/2014/4/20/scrolling-text-with-rplcd/
- See https://youtu.be/49RkQeiVTGU
- Communication over I^2C (uses less wires than the parallel wiring we used) will probably be added in the future.

7: RPLCD Implementation Details

This looks complicated, but is it?

The guts

- The implementation is actually quite easy. I needed to learn reading datasheets though.
- The low level part works like this:
  a. Output either 0 (instruction) or 1 (data) to the RS pin to specify whether you're gonna send a command or data.
  b. If in 8 bit mode, output the 8 bits of the character or the command to GPIO pins D0-D7.
  c. Else, if in 4 bit mode, output the lower part of the character or the command to GPIO pins D0-D3.
  d. Toggle the "enable" pin for at least 37 μs (according to datasheet)
  e. If in 4 bit mode, GOTO c and output the upper part of the byte.
- Rest is implementing all commands as high level functions.

```python
def _send(self, value, mode):
    """Send the specified value to the display with automatic 4bit / 8bit
    selection. The rs_mode is either ``RS_DATA`` or ``RS_INSTRUCTION``."""

    # Choose instruction or data mode
    GPIO.output(self.pins.rs, mode)

    # If the RW pin is used, set it to low in order to write.
    if self.pins.rw is not None:
        GPIO.output(self.pins.rw, 0)

    # Write data out in chunks of 4 or 8 bit
    if self.data_bus_mode == LCD_8BITMODE:
        self._write8bits(value)
    else:
        self._write4bits(value >> 4)
        self._write4bits(value)
```

```python
def _write4bits(self, value):
    """Write 4 bits of data into the data bus."""
    for i in range(4):
        bit = (value >> i) & 0x01
        GPIO.output(self.pins[i + 7], bit)
    self._pulse_enable()

def _write8bits(self, value):
    """Write 8 bits of data into the data bus."""
    for i in range(8):
        bit = (value >> i) & 0x01
        GPIO.output(self.pins[i + 3], bit)
    self._pulse_enable()
```

```python
def _pulse_enable(self):
    """Pulse the `enable` flag to process data."""
    GPIO.output(self.pins.e, 0)
    usleep(1)
    GPIO.output(self.pins.e, 1)
    usleep(1)
    GPIO.output(self.pins.e, 0)
    usleep(100)  # commands need > 37us to settle
```

How to implement lcd.clear()?

Table 6 Instructions

Instruction	Code: RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (max) (when f_{cp} or f_{OSC} is 270 kHz)
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 μs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 μs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 μs
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 μs
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 μs
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 μs

How to implement lcd.clear()?

Table 6 Instructions

Instruction	Code										Description	Execution Time (max) (when f_{cp} or f_{OSC} is 270 kHz)
	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	

```
# Commands
LCD_CLEARDISPLAY = 0x01
LCD_RETURNHOME = 0x02
LCD_ENTRYMODESET = 0x04
LCD_DISPLAYCONTROL = 0x08
LCD_CURSORSHIFT = 0x10
LCD_FUNCTIONSET = 0x20
LCD_SETCGRAMADDR = 0x40
LCD_SETDDRAMADDR = 0x80
```

How to implement lcd.clear()?

```
def command(self, value):
    """Send a raw command to the LCD."""
    self._send(value, RS_INSTRUCTION)


def clear(self):
    """Overwrite display with blank characters and reset cursor position."""
    self.command(LCD_CLEARDISPLAY)
    self._cursor_pos = (0, 0)
    self._content = [[0x20] * self.lcd.cols for _ in range(self.lcd.rows)]
    msleep(2)
```

web republic

It's possible!

Thank you.

Slides will be available here:

https://speakerdeck.com/dbrgn