IoT-ALE: Demystifying MCUs with Arduino

Nova (aka Justin) King

SCaLE 17x - March 2019
FIGURE 1. The Sunbeam Radiant Control toaster.
Arduino

- **ATMEGA328**
- **Key parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU type</td>
<td>8-bit AVR</td>
</tr>
<tr>
<td>Performance</td>
<td>20 MIPS at 20 MHz[^2]</td>
</tr>
<tr>
<td>Flash memory</td>
<td>32 kB</td>
</tr>
<tr>
<td>SRAM</td>
<td>2 kB</td>
</tr>
<tr>
<td>EEPROM</td>
<td>1 kB</td>
</tr>
<tr>
<td>Maximum operating frequency</td>
<td>20 MHz</td>
</tr>
<tr>
<td>Number of touch channels</td>
<td>16</td>
</tr>
<tr>
<td>Hardware QTouch Acquisition</td>
<td>No</td>
</tr>
<tr>
<td>Maximum I/O pins</td>
<td>23</td>
</tr>
<tr>
<td>External interrupts</td>
<td>2</td>
</tr>
<tr>
<td>USB Interface</td>
<td>No</td>
</tr>
<tr>
<td>USB Speed</td>
<td>–</td>
</tr>
</tbody>
</table>
ESP8266

- Processor: L106 32-bit RISC microprocessor core based on the Tensilica Xtensa Diamond Standard 106Micro running at 80 MHz\[^5\]

- Memory:
  - 32 KiB instruction RAM
  - 32 KiB instruction cache RAM
  - 80 KiB user-data RAM
  - 16 KiB ETS system-data RAM

- External QSPI flash: up to 16 MiB is supported (512 KiB to 4 MiB typically included)

- IEEE 802.11 b/g/n Wi-Fi
  - Integrated TR switch, balun, LNA, power amplifier and matching network
  - WEP or WPA/WPA2 authentication, or open networks

- 16 GPIO pins

- SPI

- I²C (software implementation)\[^6\]

- I²S interfaces with DMA (sharing pins with GPIO)

- UART on dedicated pins, plus a transmit-only UART can be enabled on GPIO2

- 10-bit ADC (successive approximation ADC)
ESP32

- Processors:
  - CPU: Xtensa dual-core (or single-core) 32-bit LX6 microprocessor, operating at 160 or 240 MHz and performing at up to 600 DMIPS
  - Ultra low power (ULP) co-processor
- Memory: 520 KiB SRAM
- Wireless connectivity:
  - Wi-Fi: 802.11 b/g/n
  - Bluetooth: v4.2 BR/EDR and BLE
- Peripheral interfaces:
  - 12-bit SAR ADC up to 18 channels
  - 2 × 8-bit DACs
  - 10 × touch sensors (capacitive sensing GPIOs)
  - Temperature sensor
  - 4 × SPI
  - 2 × I²S interfaces
  - 2 × I²C interfaces
  - 3 × UART
ESP32 Con’t

- **SD/SDIO/CE-ATA/MMC/eMMC** host controller
- SDIO/SPI slave controller
- **Ethernet** MAC interface with dedicated DMA and [IEEE 1588 Precision Time Protocol](https://en.wikipedia.org/wiki/IEEE_1588) support
- **CAN bus** 2.0
- Infrared remote controller (TX/RX, up to 8 channels)
- Motor **PWM**
- LED **PWM** (up to 16 channels)
- **Hall effect sensor**
- Ultra low power analog pre-amplifier


ESP32 Con’t

- Security:
  - IEEE 802.11 standard security features all supported, including WFA, WPA/WPA2 and WAPI
  - Secure boot
  - Flash encryption
  - 1024-bit OTP, up to 768-bit for customers
  - Cryptographic hardware acceleration: AES, SHA-2, RSA, elliptic curve cryptography (ECC), random number generator (RNG)

- Power management:
  - Internal low-dropout regulator
  - Individual power domain for RTC
  - 5uA deep sleep current
  - Wake up from GPIO interrupt, timer, ADC measurements, capacitive touch sensor interrupt
Your devices and networking

Hybrid solution: Local access + cloud access
Your devices and networking

Cloud only access, no local network
Your devices and networking

No access because you forgot to install the wifi firmware :)

CO₂ meter based on a Pocketbeagle SBC

Hardware used:
- Pocketbeagle SBC - €25
- Adafruit CCS811 Air Quality Sensor - €20
- Adafruit Monochrome 1.12" display - €5
Labs

- **IDE/Board Setup**
  - Install Python if needed
  - Install Arduino IDE
  - Install ESP32 board interface

- **Blinky**
  - Open and upload to board

- **WiFi**
  - Open from examples menu
  - Upload

- **Sensors**
  - Install library from library manager
  - Open example
  - Modify example to work with the current board
IDE Setup

  - `sudo chmod 666 /dev/ttyUSB0` if it won’t upload


Setting up the ESP32 board drivers

(https://dl.espressif.com/dl/packageEsp32_index.json)
Blinky

- File > Open
- Open the Blinky file in the Blinky folder
- Upload the program to the board
- `sudo chmod 666 /dev/ttyUSB0`
  if it won't upload
```c
#define LED_INDICATOR 0

void setup() {
    pinMode(LED_INDICATOR, OUTPUT);
}

void loop() {
    digitalWrite(LED_INDICATOR, HIGH);
    delay(500);
    digitalWrite(LED_INDICATOR, LOW);
    delay(500);
}
```
Wifi Scan
Wifi Monitor (Tools > Serial Monitor)

Change to 115200 baud
Library Manager (for sensors)
Sensors Library Installation
Sensors Library Example
Modifying it to Work

```cpp
// Choose between Arduino Boards and the NodeMCU board (ESP8266)
// Arduino boards have predefined SDA and SCL pins for the I2C communication
// For NodeMCU boards, we need to assign two pins for the SDA and SCL pins
// The Wire.begin() function allows you to define which pins you want to use
// It works like this: Wire.begin([SDA],[SCL])

// Wire.begin(); // Use this function for Arduino

Wire.begin(21, 22); // Use this for NodeMCU board

// For the NodeMCU V3 board:
// GPIO0 = Pin D3 = SDA
// GPIO2 = Pin D4 = SCL
```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature in Celsius:</td>
<td>20.97</td>
</tr>
<tr>
<td>Humidity in %:</td>
<td>39.11</td>
</tr>
<tr>
<td>Pressure in hPa:</td>
<td>1019.05</td>
</tr>
<tr>
<td>Altitude in Meters:</td>
<td>-50.26</td>
</tr>
<tr>
<td>Illuminance in Lux:</td>
<td>2248.00</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Duration in Seconds:</td>
<td>214.05</td>
</tr>
<tr>
<td>Temperature in Celsius:</td>
<td>20.97</td>
</tr>
<tr>
<td>Humidity in %:</td>
<td>39.11</td>
</tr>
<tr>
<td>Pressure in hPa:</td>
<td>1019.05</td>
</tr>
<tr>
<td>Altitude in Meters:</td>
<td>-50.26</td>
</tr>
<tr>
<td>Illuminance in Lux:</td>
<td>2251.00</td>
</tr>
</tbody>
</table>
Examples I Used:

github.com/chromenova/sensornodeexamples
IoT-ALE:
Discovering Tiny Snakes

IoT development without the need to compile
(mostly)

John ‘Warthog9’ Hawley

SCaLE 17x - March 2019
Quick: MicroPython vs. CircuitPython?
Why is this different?
Why is this different?

- Quick, iterative, development
- Most of the advantages of Python
- 0 to blinking LED very quick
- Mostly no need to compile anything
- Lots of default functionality, and upip (library / package management!)
Why is this possible?

- Same reason IoT is becoming ubiquitous
  - MCUs & CPUs are getting more powerful, and cheaper
- ESP32 on the SensorNode cost $5.10 to place on the board.
  - Dual Core
  - Wifi (802.11b/g/n up to 150Mbps 2.4GHz)
  - Bluetooth (v4.2 BR/EDR & BLE)
  - 4MB of flash
  - 520KB RAM
- There’s lots of competition in this space
Flashing MicroPython:

With the VM:
- Select the VM, plug in SensorNode
  - Should cause it to attach to the VM, if it’s not VM -> Removable Devices and attach it
- Helper script (specific to this tutorial)
  
  `flash_sensornode.sh`
  - Sets Serial port (usually /dev/ttyUSB0)
  - Fully erases the flash on the ESP32
    - `esptool.py --chip esp32 --port "${USBPORT}" erase_flash`
  - Flashes MicroPython
    - `esptool.py --chip esp32 --port "${USBPORT}" --baud 460800
      write_flash -z 0x1000 "${flash_file}"`

Without the VM:
- Serial Drivers
  - Linux: Driver in Most Distros
  - Windows / Mac:
    Install Silicon Mechanics CP2104
- Download / Install esptool
  - This requires Python
  - Linux:
    distro packages are available
  - Windows / Mac:
    use pypi to install
- Download MicroPython & Upload it to the board
  - [http://micropython.org/download#esp32](http://micropython.org/download#esp32)
  - `esptool.py --chip esp32 --port /dev/ttyUSB0 erase_flash &&
    esptool.py --chip esp32 --port
    /dev/ttyUSB0 write_flash -z 0x1000
    <path to micropython .bin>`
Make Sure the SensorNode is ‘on’

Helpful tip:
If there’s a flashing light on the board it’s on (it’s the charging indicator light). If it’s solid, it’s off.

The switch is on the side with the USB port:
- Down = On
- Up = Off
Breaking down the flash commands

```
esptool.py \ 
  --chip esp32 \ # Identifies which chip variant we are dealing with
  --port /dev/ttyUSB0 \ # Identifies which port the serial device is on
  erase_flash \ # Erases the flash area of the chip
&& 
esptool.py \ # (not including the boot loader area)
  --chip esp32 \ # Identifies which chip variant we are dealing with
  --port /dev/ttyUSB0 \ # Identifies which port the serial device is on
  write_flash \ # Indicates to write to the flash chip
  -z 0x1000 \ # Indicates WHERE on the flash chip to write to
  <path to micropython .bin> # What to flash to the chip
```
What this should look like:

```
[root@tutorial-base ~]# dmesg | tail -n 8
[...
[86344.904683] cp210x 2-2.1:1.0: cp210x converter detected
[86344.915286] usb 2-2.1: cp210x converter now attached to ttyUSB0
[root@tutorial-base ~]# ./flash_sensornode.sh
Flash File: esp32-20190214-v1.10-98-g4daee3170.bin
esptool.py v2.7-dev
Serial port /dev/ttyUSB0
Connecting.......
Chip is ESP32D0WDQ6 (revision 1)
Features: WiFi, BT, Dual Core, Coding Scheme None
MAC: 30:ae:a4:86:c7:64
Uploading stub...
Running stub...
Stub running...
Erasing flash (this may take a while)...
Chip erase completed successfully in 4.4s
Hard resetting via RTS pin...
```
Open up the serial console

- Minicom:
  - minicom -D /dev/ttyUSB0 --baudrate 115200
    (to exit <ctrl>c-q)

- Screen:
  - screen /dev/ttyUSB0 115200n8
    (to exit <ctrl>c-A \)

- Windows: use PuTTY
Reset the board
On the serial console...

rst:0x1 (POWERON_RESET), boot:0x13 (SPI_FAST_FLASH_BOOT)
configspi: 0, SPIWP:0xee
clk_drv:0x00, q_drv:0x00, d_drv:0x00, cs0_drv:0x00, hd_drv:0x00, wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018, len:4
load:0x3fff001c, len:5060
load:0x40078000, len:8788
ho 0 tail 12 room 4
load:0x40080400, len:6772
entry 0x40081610
I (428) cpu_start: Pro cpu up.
I (428) cpu_start: Application information:
I (428) cpu_start: Compile time: 12:32:34
I (430) cpu_start: Compile date: Feb 14 2019
I (436) cpu_start: ESP-IDF: v3.3-beta1-268-g5c88c5996
I (442) cpu_start: Single core mode
I (447) heap_init: Initializing. RAM available for dynamic allocation:
I (454) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (460) heap_init: At 3FFB92B0 len 00026D50 (155 KiB): DRAM
I (466) heap_init: At 3FFE0440 len 0001FBC0 (126 KiB): D/IRAM
I (472) heap_init: At 40078000 len 00008000 (32 KiB): IRAM
I (479) heap_init: At 40092834 len 0000D7CC (53 KiB): IRAM
I (485) cpu_start: Pro cpu start user code
I (55) cpu_start: Starting scheduler on PRO CPU.
OSError: [Errno 2] ENOENT
MicroPython v1.10-98-g4daee3170 on 2019-02-14; ESP32 module with ESP32
Type "help()" for more information.
>>>

ets Jun 8 2016 00:22:57
Quick *Hello World!*

```python
>>> print("Hello World!")
Hello World!
>>> 
```
Now to Blink an LED!

```python
>>> import machine
>>> led_pin = machine.Pin(0, machine.Pin.OUT)
>>> led_pin.on()
>>> led_pin.off()
```

**Note:** You’ll quickly find the `on()` turns the LED off, and `off()` turns the LED on. To “Fix”

```python
>>> led = machine.Signal( led_pin, invert=True)
>>> led.off()
>>> led.on()
```
Some interesting things to note

● boot.py
  - executed on every start, good for setting up the board
    (good place for wifi settings for example)

● main.py
  - Run after boot.py, think of it like the autoexec.bat

● It’s possible to upload more files to the board
  - Ampy - [https://github.com/adafruit/ampy](https://github.com/adafruit/ampy)

● Tab completion works in the repl prompt

● <ctrl>+e at the repl prompt puts you into “paste” mode
Disconnect From Serial before trying file transfers!

- **Minicom:**
  - to exit: `<ctrl>c-q`

- **Screen:**
  - to exit: `<ctrl>c-A \ y`

- **Putty:**
  - Hit the X and close the application
Where to go from here

**Setup Wifi in client mode**

- ampy --port /dev/ttyUSB0 get boot.py | tee boot.py
  
  # This file is executed on every boot (including wake-boot from deepsleep)
  #import esp
  #esp.osdebug(None)
  #import webrepl
  #webrepl.start()

- Add to boot.py:
  
  # This file is executed on every boot (including wake-boot from deepsleep)
  #import esp
  #esp.osdebug(None)
  #import webrepl
  #webrepl.start()
  import network
  sta = network.WLAN(network.STA_IF)
  sta.active(True)
  sta.connect("ALE", "Penguins")

- ampy --port /dev/ttyUSB0 put boot.py
Re-connect to Serial and check:

- >>> sta.ifconfig()
  ('192.168.123.456', '255.255.255.0', '192.168.123.1', '192.168.123.1')
- >>> sta.status()
  1010
- >>> sta.isconnected()
  True

- >>> import socket
  >>> addr_info = socket.getaddrinfo("towel.blinkenlights.nl", 23)
  >>> addr = addr_info[0][-1]
  >>> s = socket.socket()
  >>> s.connect(addr)
  >>> while True:
    ...         data = s.recv(500)
    ...         print(str(data, 'utf8'), end='')
    ...

<ctrl>+c will stop the while loop
One more thing to note, but not try here...

- Access Point Mode (can be used with client mode at the same time, albeit slowly)
  ■ >>> ap = network.WLAN(network.AP_IF)
  >>> ap.active(True)
  >>> #ap.config(essid="network-name", authmode=network.AUTH_WPA_WPA2_PSK,
  password="abcdabcdabcd")
  ■ Can be added to boot.py, same as the client information
Links to more resources

- https://github.com/unreproducible/tinysnakes
  (note: most of the ideas are the same, the boards ARE different)
- https://boneskull.com/micropython-on-esp32-part-1/
- https://www.cnx-software.com/2017/10/16/esp32-micropython-tutorials/

- Any questions before you start this on your own?

John ‘Warthog9’ Hawley | warthog9@eaglescrag.net | @warty9
IoT-ALE: Reading Sensor Data with I2C

Jon Mason

SCaLE 17x - March 2019
I2C Some background

- Released in 1982
- Bus Protocol
- Devices use addresses
- 2-pins needed
  - Clock (scl)
  - Data (sda)
- SCL & SDA pulled-up against voltage to ship (Vdd)
  - Level shifting can be complicated to get right
- Upwards of 3.4Mbps
  - More realistically: ~1Mbps
  - Most devices communicate in Kbps
- SMBus is derived from, but not identical to, I2C
  - Devices may claim to be one or the other not really consistent
What this looks like on the bus:

1. Data transfer is initiated with a start bit (S) signaled by SDA being pulled low while SCL stays high.
2. SCL is pulled low, and SDA sets the first data bit level while keeping SCL low (during blue bar time).
3. The data are sampled (received) when SCL rises for the first bit (B1). For a bit to be valid, SDA must not change between a rising edge of SCL and the subsequent falling edge (the entire green bar time).
4. This process repeats, SDA transitioning while SCL is low, and the data being read while SCL is high (B2, ...Bn).
5. The final bit is followed by a clock pulse, during which SDA is pulled low in preparation for the stop bit.
6. A stop bit (P) is signaled when SCL rises, followed by SDA rising.
Addresses

- Device specific implementation
- Some devices only provide a single address
  - One device per-bus
- Address on the bus “needs” to be unique
- 7-bit address normal
  - 128 Devices normally
  - 10-bit exists, very little uses it
  - 10-bit gives you 1008 devices (reserved addresses)
I2C is **NOT** discoverable, detection is not guaranteed

- Random probing can cause systems to crash - you are warned
SensorNode has 2 x I2C devices:

**BME280**
- Temperature
- Humidity
- Relative Pressure

**TSL2591**
- Full Spectrum Light Sensor
- IR Spectrum Light Sensor
Figuring out Address - See the Schematic(s)


BME280: TSL2591

<table>
<thead>
<tr>
<th>Ordering Code</th>
<th>Address</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSL25911FN</td>
<td>0x29</td>
<td>$i^2C V_{bus} = V_{DD}$ Interface</td>
</tr>
<tr>
<td>TSL25913FN*</td>
<td>0x29</td>
<td>$i^2C V_{bus} = 1.8V$</td>
</tr>
</tbody>
</table>
Figuring out MCU pins

BMEx80 - Temp, Humidity
BME680 - Temp, Humidity, Pressure, Particle

SJ4 controls the lowest bit of the I2C address—can bet 0=1110110
0=1110111 Open for SPI 3-wire mode

ESP3212 MODULE + AUTORESET

IO0/A2_1
IO2/A2_2
IO4/A2_0
IO5
IO12/A2_5
IO13/A2_4
IO14/A2_6
IO15/A2_3
IO16
IO17
IO18
IO19
IO20
IO21
IO22
IO23

25 GPIO0
24 GPIO2
26 A5 IO4
29 SCK
14 IO12_A11
16 IO13_A12
13 IO14_A6
23 IO15_A8
27 IO16
28 IO17
30 MOSI
31 MISO
33 SDA
36 SCL
37 IO23
0 A1 DAC1

TSL2591
Light Sensor
Time to read some data

1. Exit screen

2. Upload the following using ampy:
   # ampy --port /dev/ttyUSB0 put sensornode-stuff/src/bme280.py
   # ampy --port /dev/ttyUSB0 put sensornode-stuff/src/tsl2591.py
   # ampy --port /dev/ttyUSB0 put sensornode-stuff/src/usmbus
   ○ Note the last one is a directory

3. Open up the serial port again
Confirm file upload

```python
>>> import os
>>> os.listdir()
['boot.py', 'bme280.py', 'tsl2591.py', 'usmbus']
```
BME280 - Environment Sensor

```python
>>> from machine import Pin, I2C
>>> import machine
>>> import bme280

>>> pin_i2c_scl     = 22
>>> pin_i2c_sda     = 21

>>> bme280_address  = 0x77

>>> sensor_i2c = I2C( scl=Pin(pin_i2c_scl), sda=Pin(pin_i2c_sda) )

>>> bme = bme280.BME280( i2c=sensor_i2c, address=bme280_address )

>>> bme.values
('26.84C', '1015.59hPa', '17.71%')
```
TSL2591 - Light Sensor

```python
>>> import tsl2591
>>> tsl = tsl2591.Tsl2591()
>>> tsl.get_full_luminosity()
(58, 14)
```
Places to find more information on I2C:

- [https://i2c.info/](https://i2c.info/)
- [https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BME280-DS002.pdf](https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BME280-DS002.pdf)
IoT-ALE:
Reading and Writing to SPI SDcards

Nisha Kumar

SCaLE 17x - March 2019
SPI Background

- Not a hard defined standard like I2C
  - Ubiquitous despite no hard standard
  - Data on the bus is effectively device unique
  - Quad SPI can add 2 more data lines, uncommonly used

- Faster than I2C
  - Possible to go >10Mbps

- Duplex communications
  - Master Out Slave In (MOSI)
  - Master In Slave Out (MISO)

- Hardwired device selection
Where this gets messy...

- While fast, it’s not easy to implement
- Chip select lines can get very expensive, very quickly
- Some devices need more than the minimum 4* wires

* Minimum is based on duplex operation, some devices are write or read only and you only need 3 wires then
SPI Screens, cases in point as “odd”

E-link
- SPI Like Interface
- Busy pin
- Reset pin
- Data/Command (DC) pin
- Write-only device (MOSI)
- 8-pins (including Vcc & GND)

OLED Screen
- SPI Like interface
- Write-only device (MOSI)
- Reset pin
- Data/Command (DC) pin
- 7-pins (including Vcc & GND)
Normal SPI Device

BME280 (SPI mode)
- CSB - Chip Select
- SCL - Clock
- SDA - MOSI (serial data in)
- SDO - MISO (serial data out)
- GND - Ground
- VCC - Power
SDcards and SPI

- SDcards have two basic modes:
  - SD mode
  - SPI mode

- SPI mode disadvantages:
  - Slower transfers (no parallel data)
  - ‘U’ modes aren’t supported

- SPI mode advantages:
  - Easier to implement
  - Less hardware needed
  - Simpler interface
Hardware vs. Software Implementation

Hardware:
- 4 SPI Busses
  - SPI0 - typically dedicated to Flash
  - SPI1 - tied to same pins as SPI0
  - HSPI (SPI2)
    - CS: 15
    - SCLK: 14
    - MISO: 12
    - MOSI: 13
    - QUADWP: 2
    - QUADHD: 4
  - VSPI (SPI3)
    - CS: 5
    - SCLK: 18
    - MISO: 19
    - MOSI: 23
    - QUADWP: 22
    - QUADHD: 21

Software
- Any pins will do
- Bitbanged in software / timers
- SensorNode uses:
  - CS: 15
  - SCLK: 14
  - MISO: 12
  - MOSI: 13
  - QUADWP: -
  - QUADHD: -
Wiring up an SDcard to an MCU

Micro SD Card Cage

SD & MMC

GND
GND1
VSS
VDD
DAT2
DAT1
CS
SCLK
DATA IN
DATA OUT
CARD DETECT1
CARD DETECT
3.3V

MT1
MT2
1
2
3
4
5
6
7

ESP-WROOM-32

7 CS
6 SCK
5 MOSI
4 MISO
3 VDD
2 VSS
1 VDD

Pin connections:
1. CS (MISO) -> D19
2. VDD -> D0
3. VSS -> GND
4. SCLK -> D18
5. MOSI -> D14
6. MISO -> D15
7. CS -> IO5
Prep work for using the SDcard

1. Exit screen

2. Upload the following using ampy:
   # ampy --port /dev/ttyUSB0 put sensornode-stuff/src/sdcard.py

3. Open up the serial port again
Lets look at some code - Setup the SPI Interface

Software (use this on SensorNode)

```python
>>> from machine import Pin, SPI
>>> cs = Pin(15, Pin.OUT)
>>> mosi = Pin(13, Pin.OUT)
>>> miso = Pin(12, Pin.IN)
>>> sck = Pin(14, Pin.OUT)
>>> spi_bus = SPI(sck = sck,
                mosi = mosi, miso = miso)
```

Hardware (for comparison only)

```python
>>> from machine import Pin, SPI
>>> cs = Pin(15, Pin.OUT)
>>> spi_bus = SPI(2)
```
Adding the SD card to the mix

1. Plug in the SD card
   ○ SD Card is on the back behind the buttons

2. Add the following:

```python
>>> import sdcard
>>> sd = sdcard.SDCard(spi_bus, cs)
```

What this looks like, without the SD card in place:
```python
>>> sd = sdcard.SDCard(spi_bus, cs)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sdcard.py", line 54, in __init__
  File "sdcard.py", line 82, in init_card
OSError: no SD card
```
Mounting the SDCard

- You mount it to the filesystem like Unix / Linux
- ```
  >>> import os
  >>> os.mount(sd, '/sd')
  >>> os.listdir('/')
  ['sd', 'boot.py', 'bme280.py', 'sdcard.py', 'tsl2591.py', 'usmbus']
  >>> os.listdir('/sd')
  ['MISC', 'DCIM', 'old']
  ```

Contents here will likely be empty unless you've Put things on the card already
Reading & Writing to the SD card

```python
>>> f = open("/sd/demofile.txt", "a")
>>> f.write("Hello World!")
12
>>> f.close()
>>> f = open("/sd/demofile.txt", "r")
>>> f.read()
'Hello World!'
>>> 
```
IoT-ALE:
Connecting to the Internet MQTT

putting the I in IoT

John ‘Warthog9’ Hawley

SCaLE 17x - March 2019
Let us lay some ground works…
What most “home” networks look like:

Firewall

Main Network

Wireless Guest
More Groundwork: IoT devices
Typical ways devices connect to the Internet

- Through a Gateway:
  - Bluetooth
  - Z-wave
  - 802.11.6
  - Zigbee
  - IR
  - Smoke Signals
  - Carrier Pigeons

- Directly:
  - Wifi
  - Ethernet

- Using:
  - IPv4
  - IPv6
Let's come back to this for a minute to talk about IPv4 vs. IPv6.
Local Access vs. Remote Access

● IPv4 - Local
  ○ Direct Access
  ○ Straight Forward
  ○ Mostly ubiquitous support

● IPv4 - Remote
  ○ NAT traversal
  ○ Punching holes in firewalls
  ○ Port Forwarding
  ○ UPNP
  ○ Cloud reverse proxies

● IPv6 - Local
  ○ Direct Access
  ○ Straight Forward
  ○ Getting more ubiquitous but not there

● IPv6 - Remote
  ○ Direct Access
  ○ Punching holes in firewalls
  ○ UPNP
  ○ Cloud based IP lookup (and/or reverse proxies)
Some general words of caution...

- Think about what you are using the Internet for
- Be mindful of where your services live
- Sometimes UX the user can use may make you less secure
- Always change the default passwords!
- Make it possible to do things without auto-discovery
- Don’t always assume you are on the same network as the device
- Upgrade schemes need to be done
Shifting gears & talk about how to talk to the devices

But the real advantage to IoT is the I - Internet!

Lots of good ways to do this…

- MQTT
- Liota
- AMQP
- STOMP
- RabbitMQ
- REST
- WAMP
- ZeroMQ
- Java Message Service (JMS)
- CoAP
- CLOUD!
- XMPP-IOT
- XMPP
- etc…..

https://xkcd.com/927/
- CC-BY-NC 2.5
Now lets talk about something to try

- MQTT - Mosquitto, MQTT broker, good for local passing of data
- Think of it as a message bus on the network
- Clients Subscribe to Topics that can be hierarchical, and listen to the Topic
  - /myhome/groundfloor/livingroom/temperature for example
  - You can listen at any level of the hierarchy, anything below your level will be filtered to you
  - Wildcards, +, are allowed /myhome/+/temperature
- Devices Publish data to topics
  - The data is freeform, the receiving end is expected to interpret it
Lets just try listening...

On your laptop/VM:

\[\text{yum install mosquitto}\]

\[\text{apt-get install mosquitto-clients}\]

\[\text{then}\]

\[\text{mosquitto_sub} \]
\[\quad -h 10.100.0.5 \]
\[\quad -t "pugnose/temp/core0" \]
\[\quad -u "ale" \]
\[\quad -P "Penguins" \]

**Expected output:**

+67.0°C

What’s running on “pugnose”:

\[\text{while} \quad [[ 1 ]]; \text{do} \]
\[\quad \text{mosquitto_pub} \]
\[\quad \quad -h 10.100.0.5 \]
\[\quad \quad -t "pugnose/temp/core0" \]
\[\quad \quad -m "$( \]
\[\quad \quad \quad \text{sensors} \quad | \]
\[\quad \quad \quad \text{grep} "Core 0" \quad | \]
\[\quad \quad \quad \text{tr} " " "\n" \quad | \]
\[\quad \quad \quad \text{grep} "°" \quad | \]
\[\quad \quad \quad \text{head} -n 1 \]
\[\quad \quad )" \]
\[\quad \quad -u "ale" \]
\[\quad \quad -P "Penguins"; \]
\[\quad \quad \text{sleep 10;}\]
\[\quad \text{done}\]


Listening from the IoT device (subscribing)

From the repl prompt:
```python
>>> from umqtt.simple import MQTTClient
>>> import socket
>>> import time
>>> from ubinascii import hexlify
>>> CLIENT_ID = hexlify(machine.unique_id())
>>> def sub_cb(topic, msg):
...     print((topic, msg))
...     ...
...     ...
...     ...
...     c.set_callback(sub_cb)
...     c = MQTTClient(CLIENT_ID,
...                     "10.100.0.5")
...     c.connect()
...     c.subscribe(b"topic/yourname")
```  
```python
>>> while True:
...     if True:
...         c.wait_msg()
...     else:
...         c.check_msg()
...         time.sleep(1)
...     ...
...     ...
...     ...
>>> c.disconnect()
```

From your VM / Laptop
```
mosquitto_pub \
    -h 10.100.0.5 \
    -t "topic/yourname" \
    -m "Hello YourName" \
    -u "ale" \ 
    -P "Penguins"
```
Publishing from the IoT device

From the repl prompt:

```python
>>> from umqtt.simple import MQTTClient
>>> import socket
>>> from ubinascii import hexlify
>>> CLIENT_ID = hexlify(machine.unique_id())
>>> c = MQTTClient(CLIENT_ID,
... "10.100.0.5")
>>> c.connect()
>>> c.publish(b"topic/yourname",
... b"hello from mpy")
>>> c.disconnect()
```

On your laptop/VM:

```shell
yum install mosquitto
```
```
apt-get install mosquitto-clients
```
```
them
```
```
mosquitto_sub \
-h 10.100.0.5 \
-t "topic/yourname" \
-u "ale" \
-P "Penguins"
```
from umqtt.simple import MQTTClient
from machine import Pin
from ubinascii import hexlify
import machine
import micropython

led = Pin(0, Pin.OUT, value=1)
SERVER = "10.100.0.5"
CLIENT_ID = hexlify(machine.unique_id())
TOPIC = b"topic/yourname"
state = 0
def sub_cb(topic, msg):
    global state
    print((topic, msg))
    if msg == b"on":
        led.value(0)
        state = 1
    elif msg == b"off":
        led.value(1)
        state = 0
    elif msg == b"toggle":
        led.value(state)
        state = 1 - state

def main(server=SERVER):
    c = MQTTClient(CLIENT_ID, server)
    c.set_callback(sub_cb)
    c.connect()
    c.subscribe(TOPIC)
    print("Connected %s, sub to %s topic" % (server, TOPIC))
    try:
        while 1:
            c.wait_msg()
    finally:
        c.disconnect()