I2C Communications

ECE 476 Advanced Embedded Systems

Jake Glower - Lecture #27

Please visit Bison Academy for corresponding
lecture notes, homework sets, and solutions

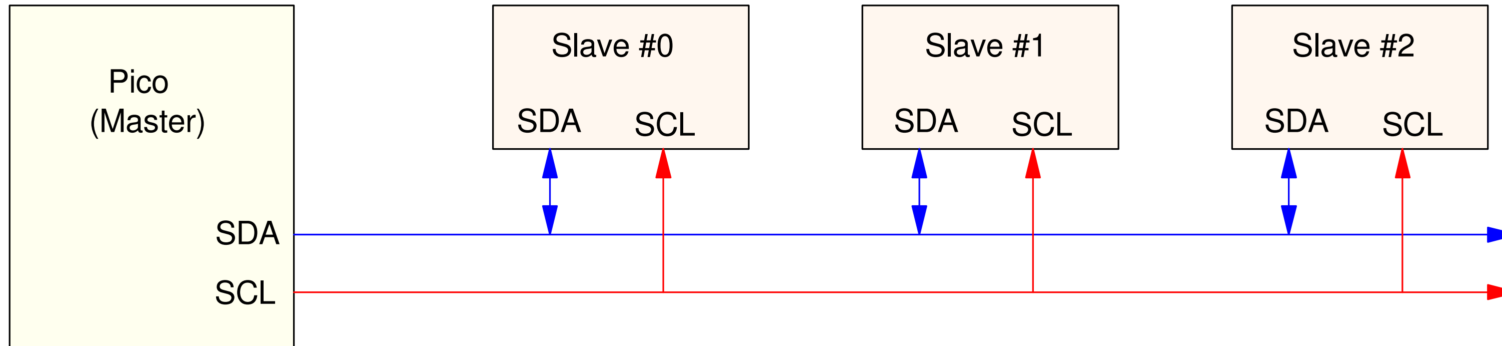
I2C Communications

Previously, we looked at SPI communications

- Four wire communication
  - CS, CLK, MISO, MOSI

I2C Communications is another for of serial communications

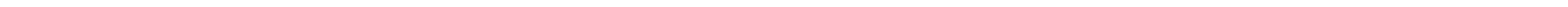
- Two wire communication
  - SDA: A bi-directional bus
  - SCL: The clock output from the bus master



I2C Functions

The machine library includes several I2C functions

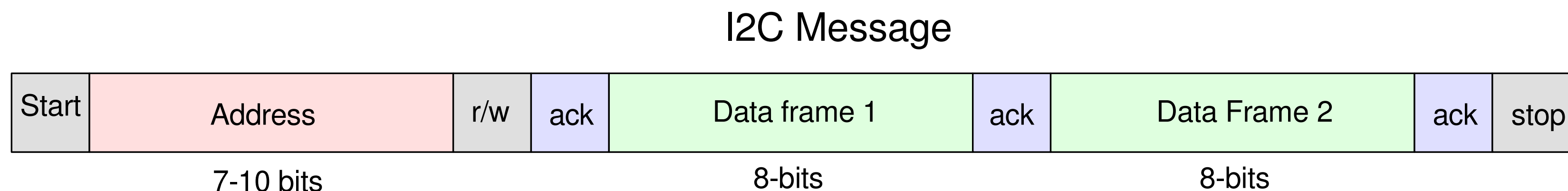
```
from machine import I2C

i2c = I2C(0)                      declare I2C as an object

i2c.scan()                        scan for I2C devices,
                                  returns 7-bit addresses

i2c.writeto(42, b'123')           Write three butes to device at address 42

i2c.readfrom(0x3a, 4)             Read four bytes from device at address 0x3a

I2C.start()                       Generate a START condition on the bus

I2C.stop()                        Generate a STOP condidion on the bus

i2c.writeto_mem(addr, reg, data)

i2c.readfrom_mem(addr, reg, bytes)
```



I2C Data Packets

With I2C communications, data is sent in packets which consist of

- A Start Condition:
  - SDA switches high to low before SCL switches from low to high
- An address: 7 or 10 bit sequence unique to each slave
- Read/Write Bit:
  - 0: Master talking to slave
  - 1: Master requests data from slave
- Data (data going to and from the master), and
- A Stop Condition:
  - SDA line switches from low to high after the SCL switches from low to high



I2C Communications Style

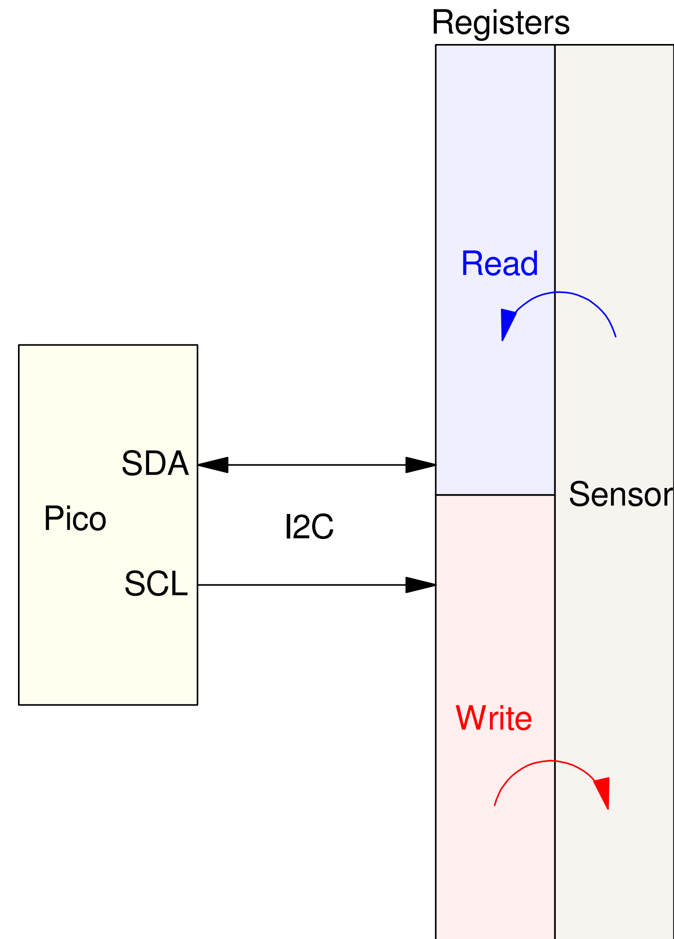
- I2C is a little different from SPI

SPI

- Send commands on MOSI
- Send data on MOSI
- Receive data on MISO

I2C

- Read from and write to registers
- Reading
  - Get constants such as calibration data
  - Get sensor readings
- Writing
  - Set the bandwidth
  - Set the sampling rate



I2C on a Pi-Pico

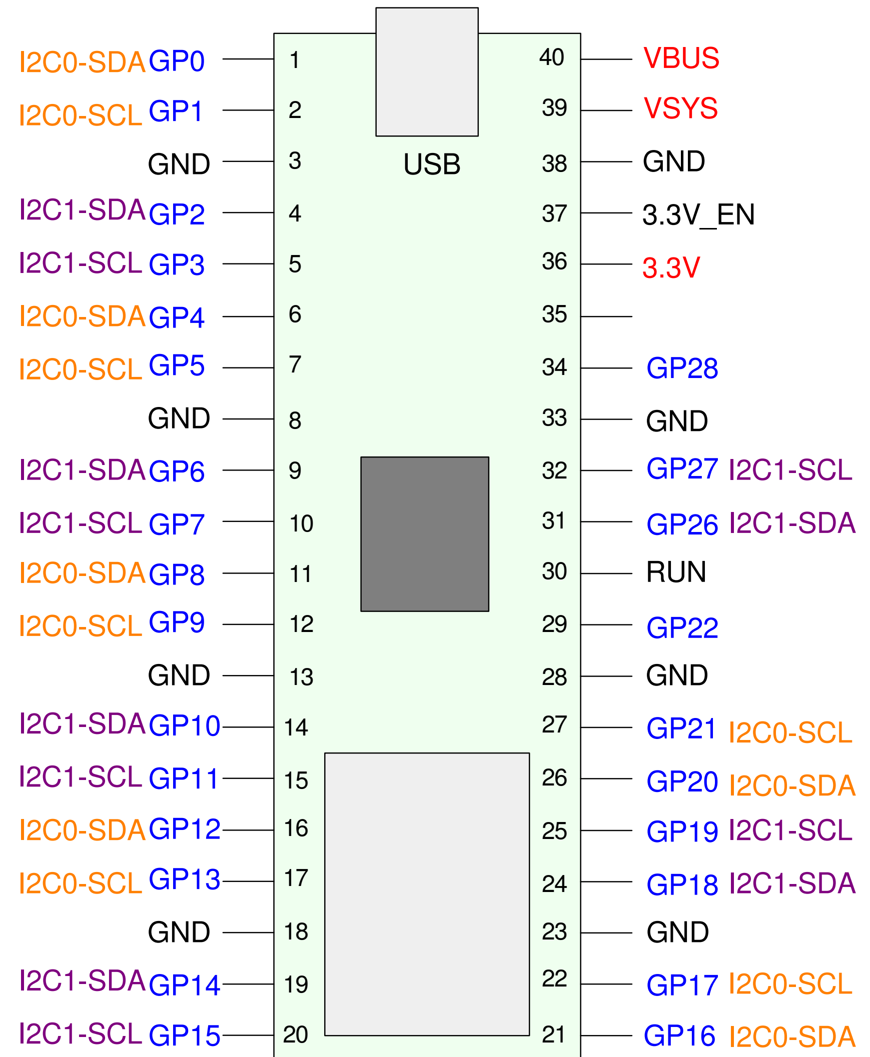
The Pi-Pico has two I2C busses

- I2C0 & I2C1

Your choice which pins are used

I2C0	
SDA	SCL
GP0	GP1
GP4	GP5
GP8	GP9
GP12	GP13
GP16	GP17
GP20	GP21

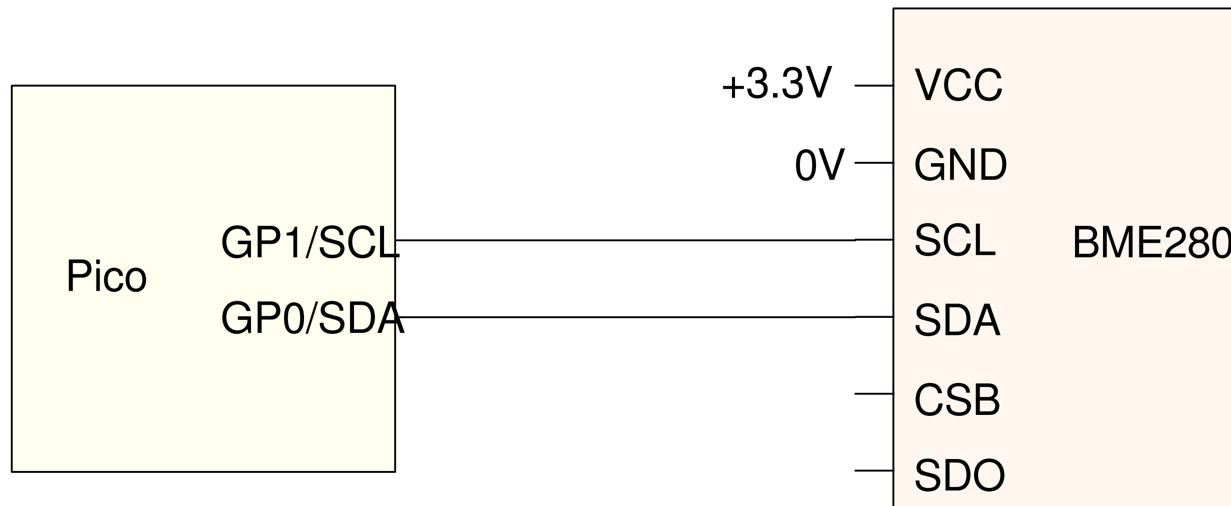
I2C1	
SDA	SCL
GP2	GP3
GP6	GP7
GP10	GP11
GP14	GP15
GP18	GP18
GP26	GP27



I2C Example: BME280

Only two wires are needed for I2C communications

- SDA: Serial Data
- SCL: Serial Clock
- (plus a common ground of course)



Identifying I2C Devices:

i2c.scan()

- Returns ID of all I2C devices on the bus
  - devices is an array
  - [id0, id1, id2, etc]
  - [] if nothing is connected
- Example: BME280
  - ID is 0x76

```
import machine

i2c = machine.I2C(0, scl=machine.Pin(1), sda=machine.Pin(0))

devices = i2c.scan()

if(devices):
    for d in devices:
        pring(hex(d))
```

Shell

```
MPY: soft reboot
0x76
```

Reading and Writing Registers

Useful routines

- Write 1+ bytes to an I2C device
  - addr = device ID (0x76)
  - register = what register you're writing to (starting address)
  - data = bytes to write (1+)
- Read 1+ bytes from an I2C device
  - nbytes = number of bytes to read

```
i2c = machine.I2C(0, scl=machine.Pin(1), sda=machine.Pin(0))

def reg_write(i2c, addr, reg, data):
    msg = bytearray()
    msg.append(data)
    i2c.writeto_mem(addr, reg, msg)

def reg_read(i2c, addr, reg, nbytes):
    data = i2c.readfrom_mem(addr, reg, nbytes)
    return data
```

BME280 Registers

With I2C, all communication goes through registers

The data sheets tell you

- The address of the registers
- Whether they are read or write
- What the register does

You kind of need this information...

BME280 Registers		
address	r/w	name
0xf5	w	config
0xf4	w	ctrl_meas
0xf3	r	status
0xf2	w	ctrl_hum
0xfe	r	humidity_7:0
0xfd	r	humidity_15:8
0xfc	r	temp_4:0
0xfb	r	temp_12:5
0xfa	r	temp_20:13
0xf9	r	pres_4:0
0xf8	r	pres_12:5
0xf7	r	pres_20:13

BME280 Registers: 0xF5 config

Writing to 0xF5 sets

- The standby time (low-power state) per sample
  - Total time = standby time plus the measurement time
  - Measurement time = 1ms x Number of oversamples
- the filter's pole
  - 1st-order digital filter: $\left(\frac{kz}{z-a}\right)$
- the type of communications:

config (0xF5)							
7	6	5	4	3	2	1	0
tstandby 000 = 0.5ms 001 = 62.5ms 010 = 125ms 011 = 250ms 100 = 500ms 101 = 1000ms			filter pole @ z = a 000: a = 0 (no filter) 001: a = 1/2 010: a = 3/4 011: a = 7/8 1xx: a = 15/16			comm 0 = I2C 1 = SPI	

BME280 Registers: 0xF4: ctrl_meas

Writing to 0xF4 sets

- the oversampling for temperature,
- the oversampling for pressure, and
- the operation mode:
  - sleep (no conversions)
  - forced (one conversion)
  - normal (constantly sampling at a rate determined by *tstandby*

ctrl_meas (0xF4)							
7	6	5	4	3	2	1	0
oversampling (temp) 0 = 0x 1 = 1x 2 = 2x 3 = 4x 4 = 8x 5+ = 16x			oversampling (pres) 0 = 0x 1 = 1x 2 = 2x 3 = 4x 4 = 8x 5+ = 16x			mode 00 = sleep 01 = forced 10 = forced 11 = normal	

BME280 Registers: 0xF2: ctrl_hum:

Writing to 0xF2 sets the oversampling rate for humidity

ctrl_meas (0xF4)							
7	6	5	4	3	2	1	0
					oversampling (hum) 0 = 0x 1 = 1x 2 = 2x 3 = 4x 4 = 8x 5+ = 16x		

BME280 Registers: 0xF3 status

Status tells you when the A/D conversion is complete

- bit 3 = 1: conversion complete
- bit 3 = 0: conversion in process

status (0xF3)							
7	6	5	4	3	2	1	0
				1 = done 0 = working			

20-Bit A/D Registers

The raw A/D results are read from 0xF7 yo 0xFE

- Conversion to relative humidity, degrees C, hPa requires calibration constants
  - Also stored in the registers
- The algorithm is fairly complicated
  - Given in the data sheets

Name	Memory Locations
Humidity raw data reading (16 bits)	0xFD : 0xFE
Temperature raw data reading (20 bits left justified)	0xFA : 0xFB : 0xFC
Pressure raw data reading (20 bits left justified)	0xF7 : 0xF8 : 0xF9

BME280 Temperature

- I2C communications example

Step 1: Set up the conversion registers

- config(0xf5) = 0x60
  - 500ms sampling rate
  - No filter
  - I2C communications
- ctrl_meas(0xf4) = 0xFF
  - 16x oversampling
  - normal operation

the set-up would be

```
# set up BME280
reg_write(i2c, addr, 0xf5, 0x60)
reg-write(i2c, addr, 0xf4, 0xff)
```

Step 2: Read the raw A/D reading

- Waiting until bit #3 of *Status* is one
  - meaning the A/D conversion is done
- Read the data at registers 0xFA : 0xFB : 0xFC
  - 20-bit integer
  - left justified

```
    while(ord(reg_read(i2c, addr, 0xf3, 1)) & 0x08):
         pass
     x0 = ord(reg_read(i2c, addr, 0xFA, 1))
     x1 = ord(reg_read(i2c, addr, 0xFA+1, 1))
     x2 = ord(reg_read(i2c, addr, 0xFA+2, 1))
     raw = ((x0 << 16) | (x1 << 8) | x2) >> 4
```

Step 3: Convert to degrees C

- Read in the calibration constants (T1, T2, T3)
- Convert the 20-bit integer to a degrees C (float)
- Procedure is given in the data sheets

```
def read_temp():
    while((ord(reg_read(i2c, addr, 0xf3, 1)) & 0x08) == 0):
        pass
    x0 = ord(reg_read(i2c, addr, 0xFA, 1))
    x1 = ord(reg_read(i2c, addr, 0xFA+1, 1))
    x2 = ord(reg_read(i2c, addr, 0xFA+2, 1))
    raw = ((x0 << 16) | (x1 << 8) | x2) >> 4
    x = raw - (T1<<4)
    ax2 = (x*x*T3) >> 34
    bx = x*T2 >> 14
    T = (ax2 + bx) / 5120
    return(T)
```

The resulting main routine is then pretty simple:

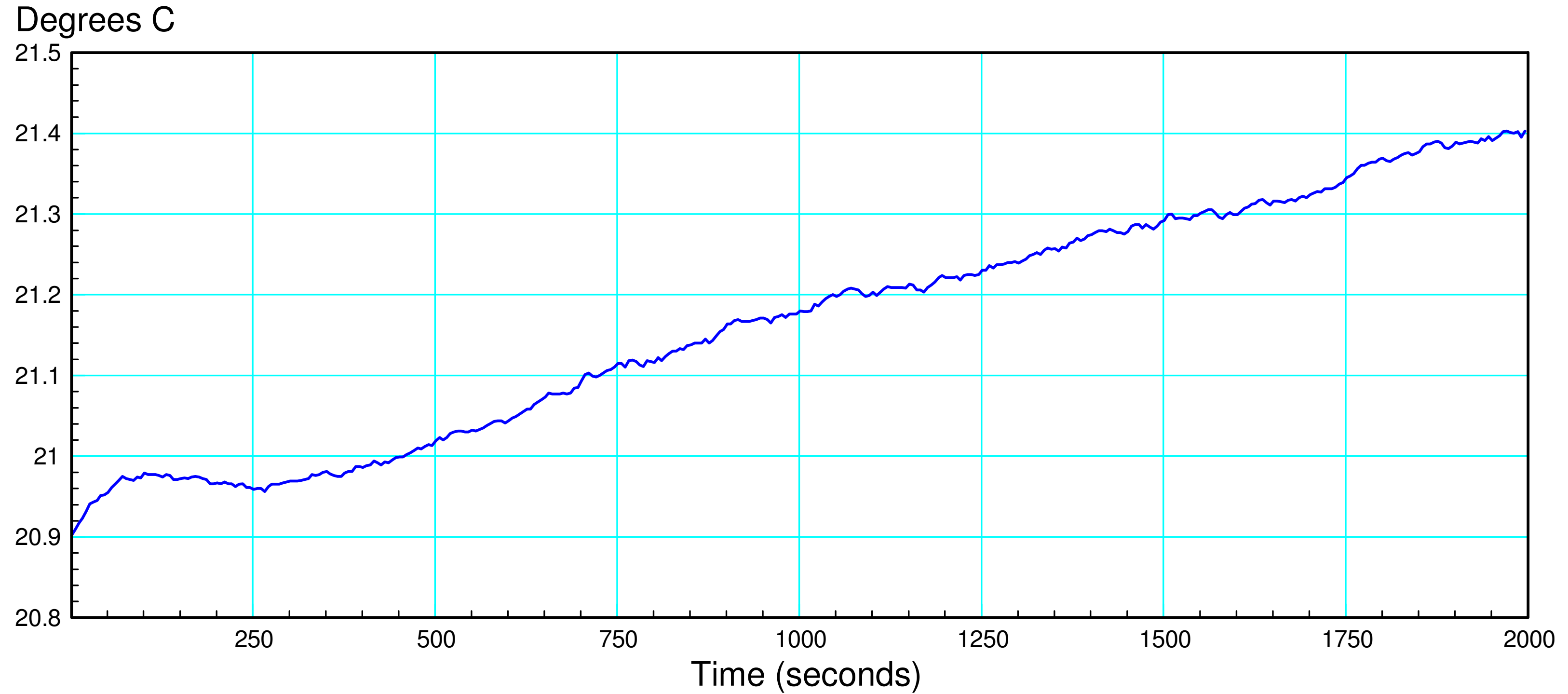
- Conversion time set to 500ms
- t measures the time since the main loop started
- Timing is a little off since *print()* also takes some time

```
t0 = time.ticks_ms()
for i in range(0,5):
    T = read_temp()
    t = time.ticks_ms()
    print((t-t0)/1000, T))
```

```
sec   degC
0.003 25.3166
0.534 25.68203
1.075 25.75625
1.606 25.78144
2.148 25.80117
```

Resulting Temperature vs. Time

- conversion time = 1000ms
  - max (20-bit) resolution
- 16x oversampling
- 2nd-order filter



BME280: Pressure

Similar algorithm for pressure

- Note: Calibration requires a temperature reading
- Algorithm given in data sheets

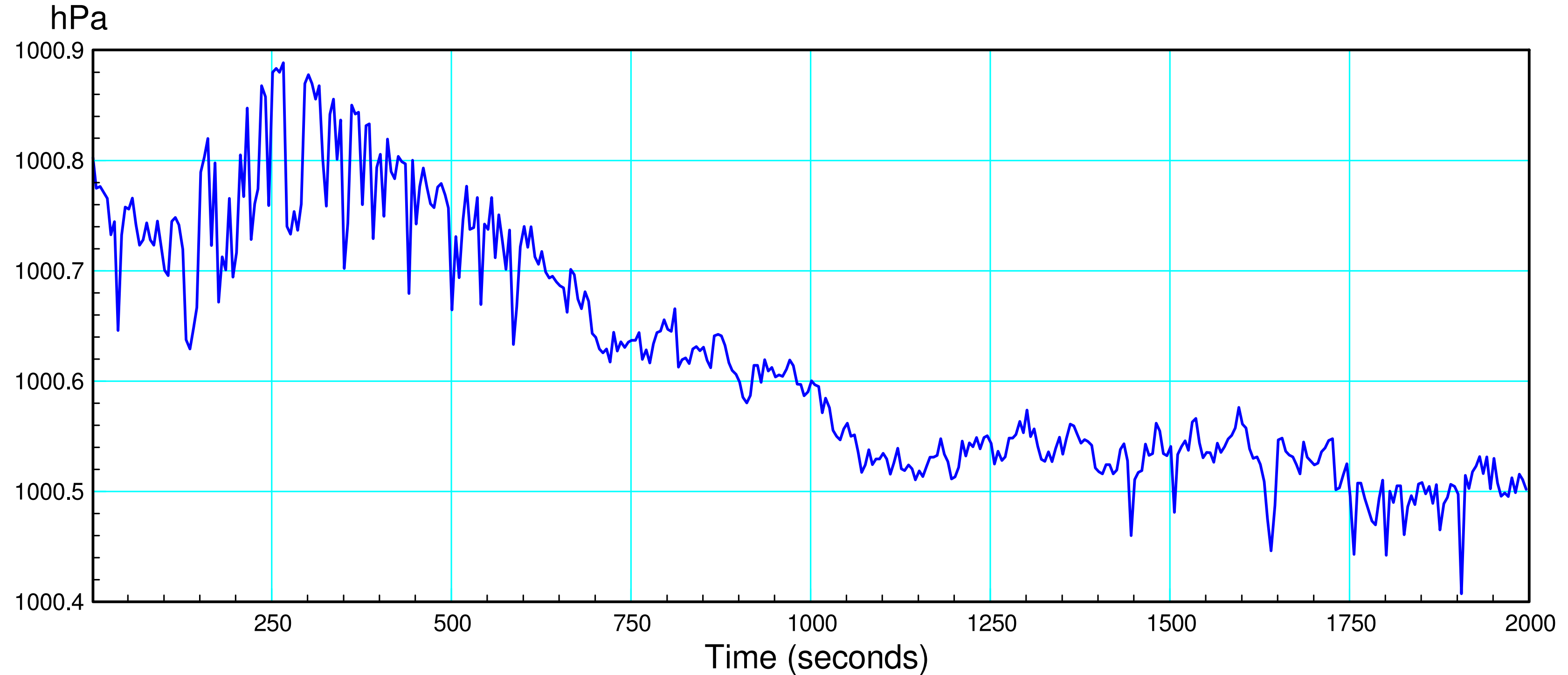
```
def read_pres(T):
    x0 = ord(reg_read(i2c, addr, 0xF7, 1))
    x1 = ord(reg_read(i2c, addr, 0xF7+1, 1))
    x2 = ord(reg_read(i2c, addr, 0xF7+2, 1))
    raw = ((x0 << 16) | (x1 << 8) | x2) >> 4
    t_fine = round(T*25600/5)

    var1 = t_fine - 128000
    var2 = var1 * var1 * P6
    var2 = var2 + ((var1 * P5) << 17)
    var2 = var2 + (P4 << 35)
    var1 = (((var1 * var1 * P3) >> 8) + ((var1 * P2) >> 12))
    var1 = (((1 << 47) + var1) * P1) >> 33
    if var1 == 0:
      return 0
    p = 1048576 - raw
    p = (((p << 31) - var2) * 3125) // var1
    var1 = (P9 * (p >> 13) * (p >> 13)) >> 25
    var2 = (P8 * p) >> 19
    pressure = ((p + var1 + var2) >> 8) + (P7 << 4)
    pressure = pressure / 25600
    return(pressure)
```

Pressure Readings vs. Time

- delay = 1000ms, 16x oversampling, 2nd-order filter

A little noisy

- Windy days caused by changes in air pressure
- Resolution is better than 0.1hPa



Measure the Height of AGHill

- Can you measure the height of a building using air pressure?

As you go from the basement to the 3rd floor

- Altitude goes up
- Air pressure goes down

Can you measure this with a BME280?

- Resolution < 0.1 hPa

What is the corresponding height of AGHill?

Measuring Pressure Change

Start in the basement

- Measure pressure every 1.0 second
- Save data to a file
- Pause 100 seconds

Go up the stairs to the 3rd floor

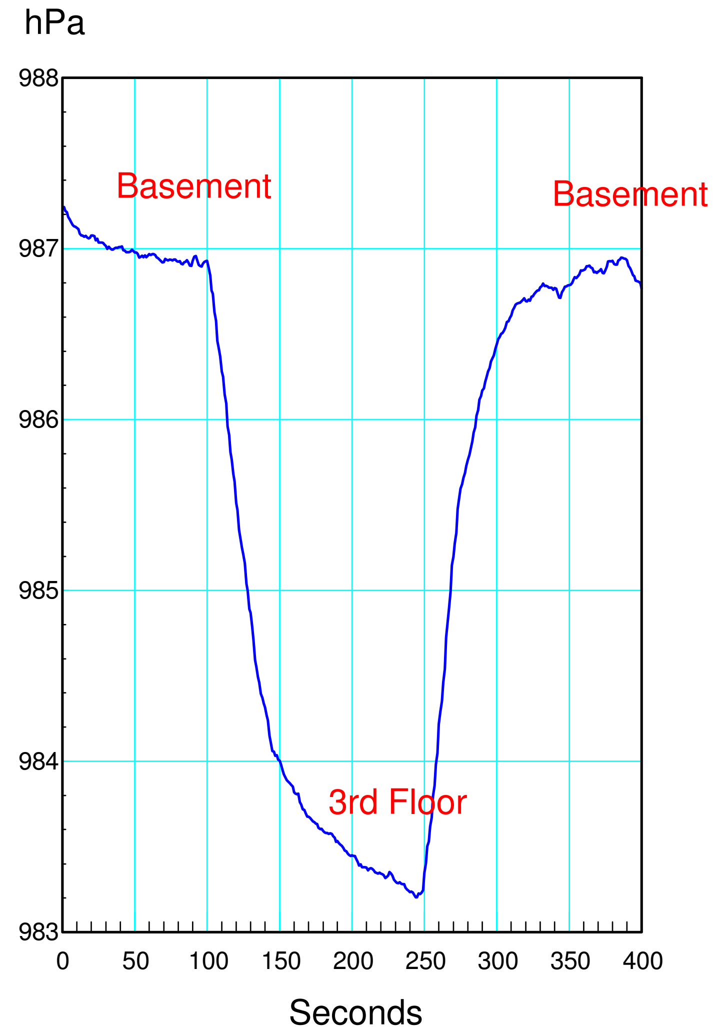
- Pause 100 seconds

Go back to the basement

- Pause 100 seconds

The net result is as expected:

- Higher elevation = lower pressure
- Change in pressure = 3.5 hPa



Computing Height

- https://www.omnicalculator.com/physics/air-pressure-at-altitude

From an on-line calculator

- 15m change in pressure = 1.9 hPa
- 7.89 m / hPa

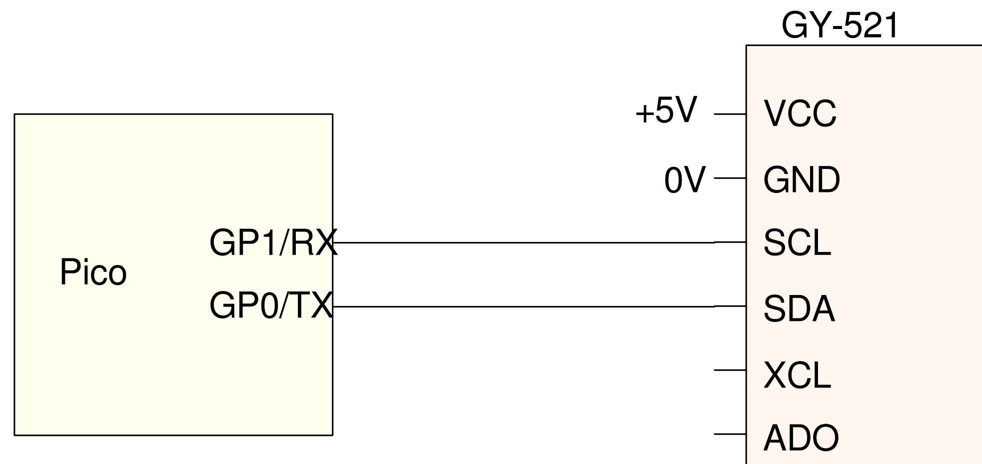
3.5 hPa = 27.6 meters

- Computed distance
- Basement to 3rd floor
- Seems kind of high

Height (m)	Air Pressure (hPa)
0	987.000
5	986.366
10	985.732
15	985.099

GY521: Accelerometer

Previously, we looked at reading a GY-521 sensor using a library. With the I2C bus, you can access the information directly. Assume the GY-521 is connected to pins 0 and 1



The register locations can be found using the data sheets

0x1A: CONFIG

0x1A CONFIG							
7	6	5	4	3	2	1	0
		EXT_SYNC_SET			DLP_CFG		

EXT_SYNC_SET allows you to synchronize sampling based upon temperature, gyro, or acceleration. Set to zero if not using.

DLP_CFG configures the digital low-pass filter

DLP_CFG	Bandwidth (Hz)	Delay (ms)
0	260	0
1	184	2
2	94	3
3	44	4.9
4	21	8.5
5	10	13.8
6	5	19
7	reserved	

0x1C: ACCEL_CONFIG This register allows you to set the range of the accelerometer

<table>
<tr><th colspan="8">0x1C: ACCEL_CONFIG</th></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>XA_ST</td><td>YA_ST</td><td>ZA_ST</td><td colspan="2">AFS_SEL</td><td></td><td></td><td></td></tr>
</table>

XA_ST enables the self-test of the accelerometer.

AFS_SEL sets the range

AFS_SEL	full-scale
0	+/- 2g
1	+/- 4g
2	+/- 8g
3	+/- 16g

0x6B: PWR_MGMT_1 This lets you set the poer mode and clock source

0x6B: PWR_MGMT_1							
7	6	5	4	3	2	1	0
DEVICE_RESET	SLEEP	CYCLE		TEMP_DIS	CLKSEL		

SLEEP:

- 0 Normal operation
- 1 places it in a low-power sleep state

CYCLE:

- 0 Normal operation
- 1 Cycle between sleep state and taking one sample.

TEMP_DIS

- 0 Normal operation
- 1 Disable temperature sensor

CLKSEL

- 0 Internal 8MHz
- 1 PLL with X axis gyroscope as reference
- 7 Stops the clock and keeps the timing generator in reset

The actual acceleration data is available from reading registers 0x3B to 0x40:

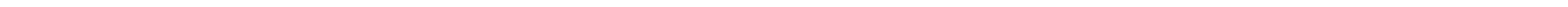
Addr (hex)	Register Name	R/W	
3B	ACCEL_XOUT_H	R	X acceleration: bits 15:8
3C	ACCEL_XOUT_L	R	X acceleration: bits 7:0
3D	ACCEL_YOUT_H	R	Y acceleration: bits 15:8
3E	ACCEL_YOUT_L	R	Y acceleration: bits 7:0
3F	ACCEL_ZOUT_H	R	Z acceleration: bits 15:8
40	ACCEL_ZOUT_L	R	Z acceleration: bits 7:0

Net Reslult: first set up the sensor. Assuming

- 260Hz bandwidth
- +/- 2g range
- 8MHz internal oscillator

The set-up code is

```
i2c = machine.I2C(0, scl=machine.Pin(1), sda=machine.Pin(0))

# Print out any addresses found
devices = i2c.scan()
if devices:
    for d in devices:
        print('I2C Device Found:',hex(d))

addr = devices[0]
print('Communicating with ', hex(addr))

# set bandwidth
reg_write(i2c, addr, 0x1a, 6)
# set range to +/- 2g
reg_write(i2c, addr, 0x1c, 0x00)
RANGE = 2
# set clock freq
reg_write(i2c, addr, 0x6b, 0)
```



The acceleration can then be read as

```
def accel_read(reg):
    x = reg_read(i2c, addr, reg, 2)
    y = (x[0] << 8) + x[1]
    if(y > 0x8000):
        y = y - 0x10000
    y = y / 0x8000
    return(y)

x = accel_read(0x3b) * RANGE
y = accel_read(0x3d) * RANGE
z = accel_read(0x3f) * RANGE
```

Drop Test

Measure distance an objet falls

Experiment:

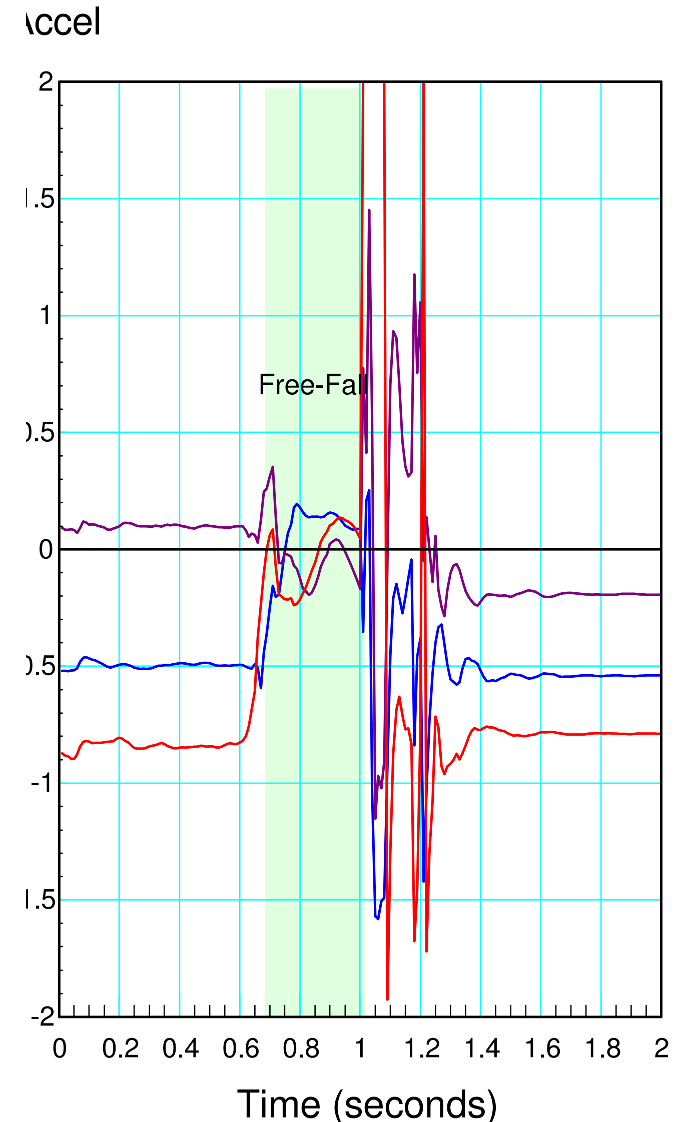
- Drop the sensor from 50cm
- Measure the acceleration every 10ms

Define "free-fall" as when |accel| < 0.5g

- Drop @ 0.61 sec
- Land @ 1.00 sec

Distance = 50.1 cm

Bandwidth	Time |accel| < 0.5g	Distance
5Hz	0.32 s	50.1 cm



How High Can I Jump?

Hold the sensor & jump

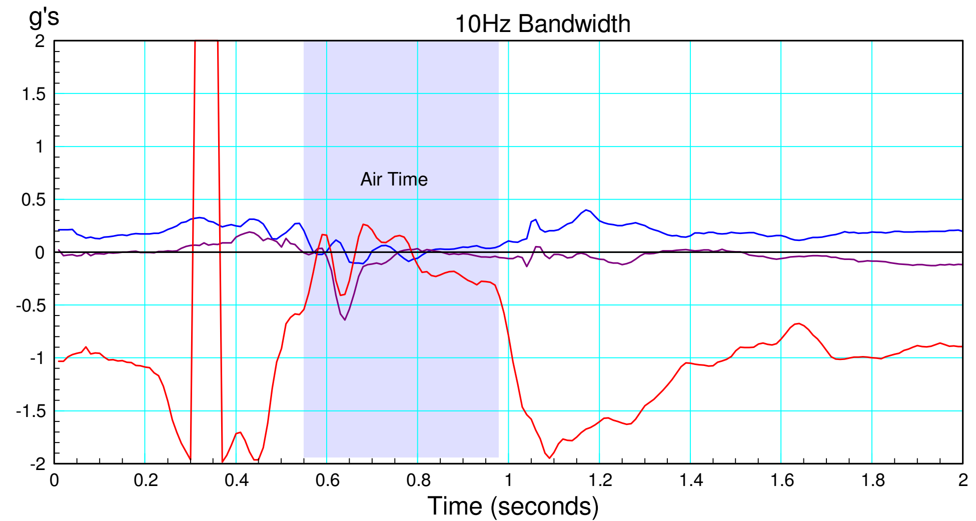
- With a bandwidth of 10Hz and
- With a bandwidth of 260Hz

Measure air time

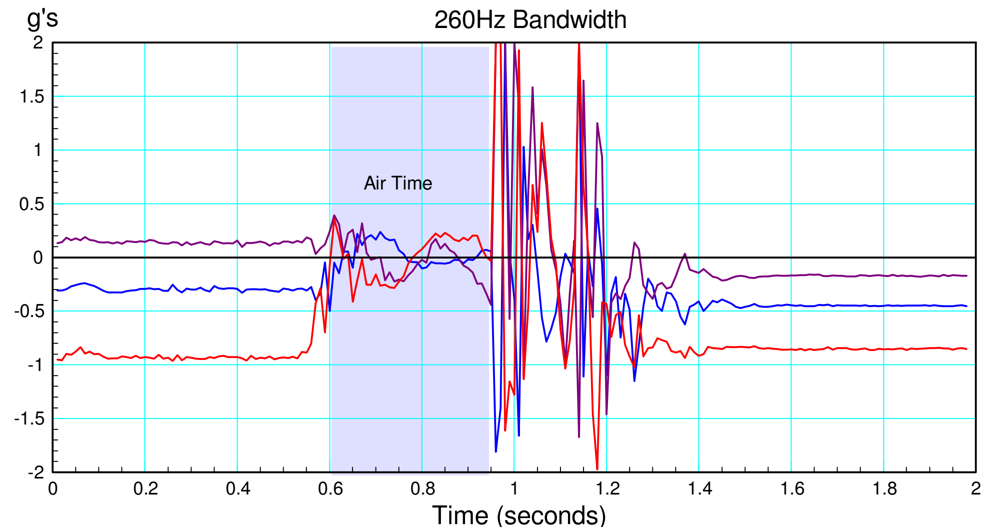
- Time when acceleration < 0.5g

Compute distance jumped

$$d = \frac{1}{2}a\left(\frac{t}{2}\right)^2 = \frac{1}{8}at^2$$



Bandwidth	Time |accel| < 0.5g	Distance
10Hz	0.33 s	13.34 cm
260Hz	0.40 s	19.6 cm



Summary:

I2C Communications is actually pretty easy with a Raspberry Pi-Pico.

With it, you're reading and writing to registers.

- Writing to registers
  - Sets the bandwidth
  - Sets the sampling rate
  - etc
- Reading registers
  - Allows you to read the sensor's data
  - Gives you calibration constants
  - etc

Data sheets are really helpful

- Tells you the address of the registers
- Tells you what the registers do
- Tells you what to write
- Tells you how to interprit what you read