27. I2C Communications

I2C Functions

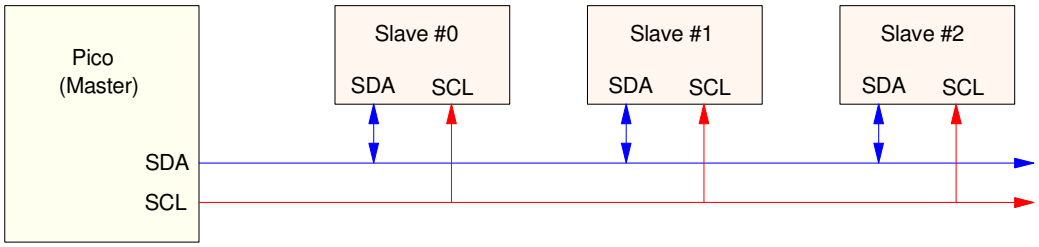
```
i2c = I2C(0)                        declare I2C as an object
i2c.scan()                          scan for I2C devices,
                                    returns 7-bit addresses
i2c.writeto(42, b'123')             Write three butes to device at address 42
i2c.readfrom(0x3a, 4)               Read 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 Communications

Previously, we looked at SPI communications for a Pi-Pico to talk to devices using a serial data interface. I2C Communications is another way to communicate with devices using a serial data bus.

With I2C, there are two data lines:

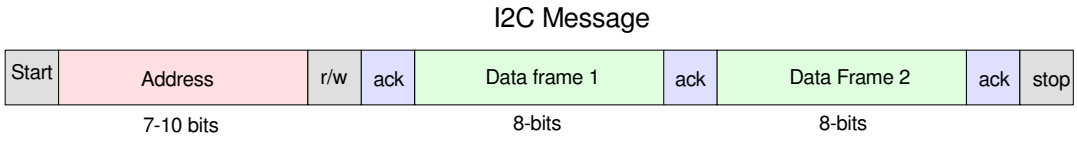
- SDA: A bi-directional bus where 8-bit data is sent between the bus master and the sensors
- SCL: The clock output from the bus master



I2C Communications: two wire communications

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



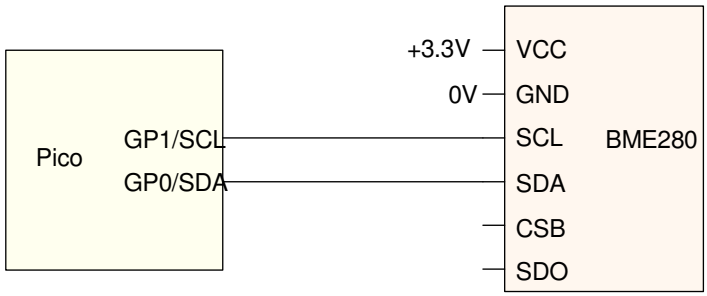
I2C Communications: Data is sent in packets

Communications with I2C is a little different that with SPI communications.

- With SPI communications, you send commands to the slave for what it is to do. Many commands are also followed by data.
- With I2C communications, you are reading and writing to registers. These registers control how the slave operates and relate to what data you are reading. More on this later

Wiring for a BME280

Only two wires are needed to communicate with I2C communications (plus a common ground of course)



Connections used for I2C communications with a BME280

Identifying I2C Devices:

Starting out, you need to identify the name of the device on the I2C bus. The following code will report all I2C devices connected to pins 0 (sda) and 1 (scl). The BME280 reads as device 0x76 on the I2C bus:

```
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
```

Scanning the I2C bus for devices. A BME280 (0x76) was found.

Registers: BME280

Next, assume a BME280 is placed on pins 0 and 1. With I2C communications, the way you send and receive data from the sensor is by writing and reading from registers (8-bit locations in memory). Code to read and write to registers is as follows.

```
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
```

I2C routines to write or read from registers

Before you read and write to registers, however, you need to know what each register location means. Form the data sheets, this can be found. For example, for a BME280 sensor, the registers mean the following:

Control 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 $\left(\frac{kz}{z-a}\right)$ with a pole at a:
- 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			000 = off 001: z = 1/2 010: z = 3/4 011: z = 7/8 1xx: z = 15/16			comm 0 = I2C 1 = SPI	

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	

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		

0xF3: Status: Status tells you when the A/D conversion is complete

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

20-Bit A/D Registers

Reading from the following registers gives the results from the 20-bit A/D. Note that this is the raw A/D reading. A fairly complicated algorithm along with some calibration constants (also stored in registers) converts these readings to relative humidity, degrees C, and hPa.

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

Sample Code: I2C Reading of Temperature

The procedure to do a temperature reading is as follows:

First, the configuration registers are set up. If you want to read temperature using

In order to read the temperature using

- 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)
```

Set-up for 250ms sampling rate, no filter, I2C, 16x oversampling, and normal operation

The raw A/D reading can be read in by

- Waiting until bit #3 of *Status* is one (meaning the A/D conversion is done), then
- Read the data at registers 0xFA : 0xFB : 0xFC

```
 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
```

The raw A/D reading is read into variable *raw* (a 20-bit A/D result)

Once read in, the raw reading is converted to degrees Celsius as follows where {T1, T2, and T3} are calibration constants also read from registers:

```
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)
```

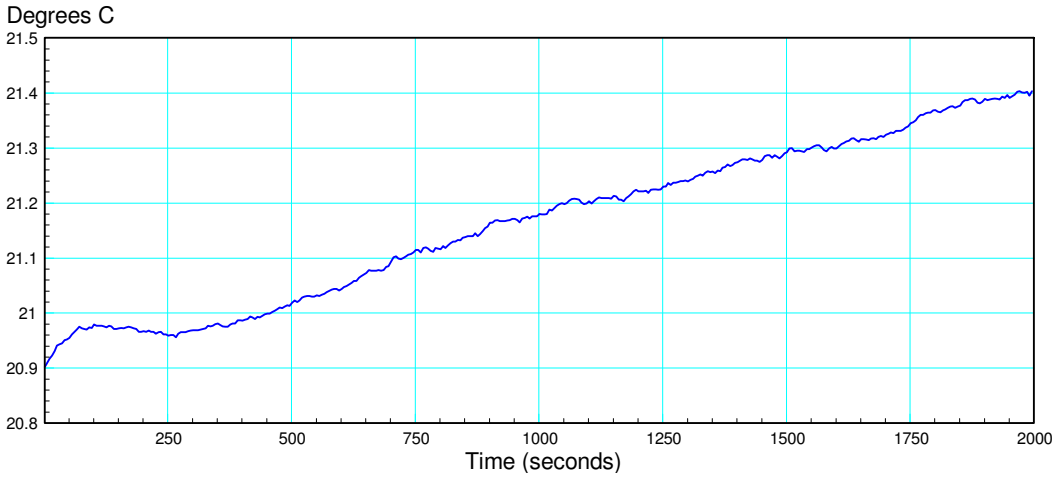
Reading the 20-bit A/D and converting the result to degrees C

The resulting main routine is then pretty simple:

```
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
```

Reading in the temperature calibration constants: T1, T2, and T3



Measured temperature with a BMP280 with delay = 1000ms, 16x oversampling, 2nd-order filter

BME280: Pressure

Similarly, pressure can be read by

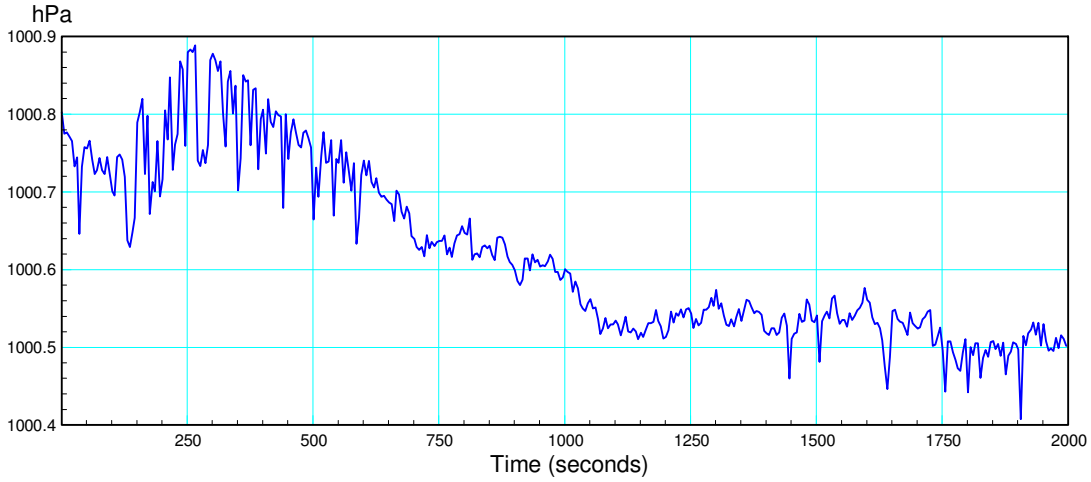
- Reading the raw pressure reading (blue code below), and
- Converting to hPa using calibration constants

Note that you need to know the temperature in the calibration equation

```
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)
```

Algorithm for reading in the pressure and returning in units of hPa



Pressure Reading from a BME280 Sensor

This brings up an interesting question: Can you measure the height of a building using a barometer (i.e. using a BME280 sensor)?

Using an on-line calculator, the air pressure at elevation should be:

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

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

With a resolutions of 0.1hPa, you *should* be able to measure the height of a building. So, let's measure the air pressure of AGHill on the NDSU campus.

AGHill Building - NDSU Campus

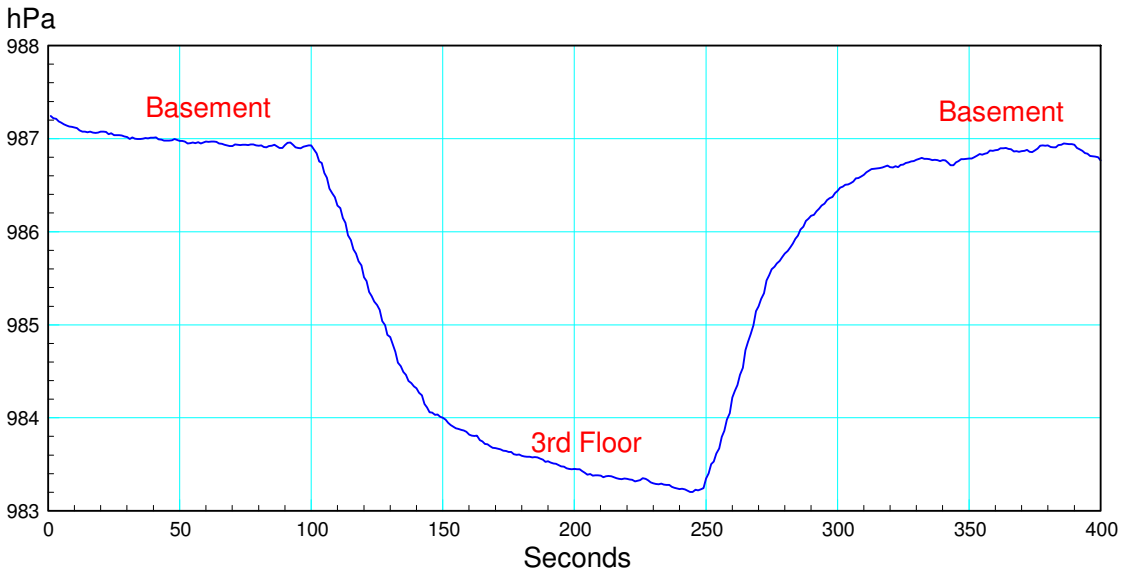
Starting out

- Pause in the basement for 100 seconds,
- Then go up the stairs to the third floor and pause for another 100 seconds,
- Then go back down the stairs to the basement in the basement and pause for another 100 seconds

The net result is as expected:

- As you go up the stairs, the air pressure drops
- As you go back down to the basement, the air pressure goes back up
- The air pressure drops as you go up as expected.

The total drop in air pressure from the basement to the 3rd floor is about 3.5hPa. This suggests a total distance of 35m from the basement to the 3rd floor - which seems kind of high. That's what the data says though.



Measured air pressure when traveling from the basement of AG Hill to the third floor.
As you climb higher, the air pressure drops.

Summary:

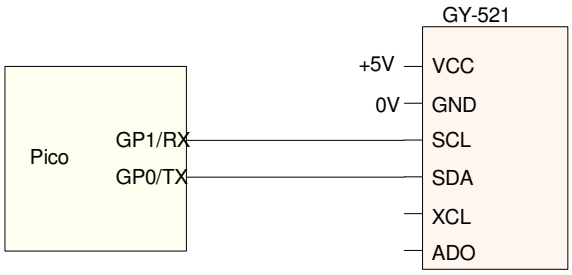
I2C Communications is actually pretty easy with a Raspberry Pi-Pico. With it, you're reading and writing to registers.

- Writing to registers allows you to set up how the sensor operates (bandwidth, sampling rate, etc.)
- Reading registers allows you to read the sensor's data or other information, such as calibration constants.

The main challenge is finding the data sheets so you know what registers to read from and what registers to write to.

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



<GY-521 connection>

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

0x1C: ACCEL_CONFIG							
7	6	5	4	3	2	1	0
XA_ST	YA_ST	ZA_ST	AFS_SEL				

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_DI S	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. Sleep time is set by LP_WAKE_CTRL (reg 108)

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

0x6C: PWR_MGMT_2 This allows you to set the frequency of wake-ups in Accelerometer Only Low Power mode

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

The net result is to read the acceleration, 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
```

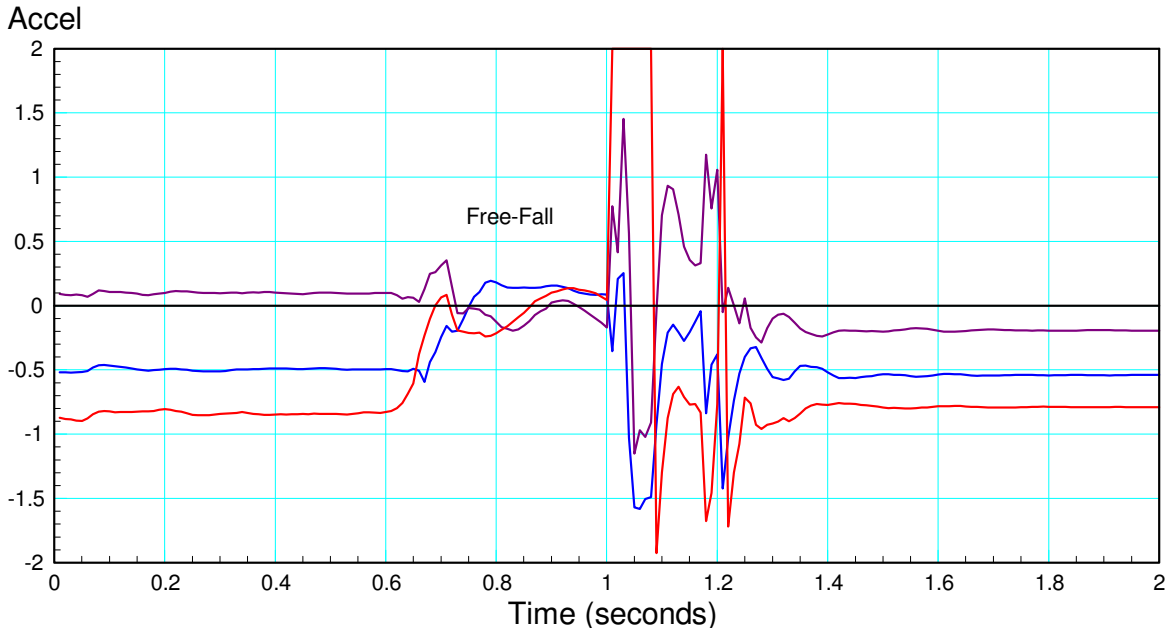
With this, you can measure your XYZ acceleration during freefall and while jumping. First, let's drop the GY521 sensor from a distance of about 50cm. While it's in free-fall, the acceleration should be zero.

Trying this with a sampling rate of 10ms and a bandwidth of 5Hz results in the following graph.

- The sensor is dropped at about 0.61 seconds (acceleration drops to zero)
- The sensor hits the ground at 1.00 seconds (acceleration jumps on impact)

Defining the time of freefall as when the net acceleration is less than 0.5g's (somewhat arbitrary) gives the measured distance of 50cm as expected.

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



Drop Test - 5Hz Bandwidth

Now that that works, let's measure how high I can jump. Holding the sensor while jumping

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

allows you to measure your air time. From this, the height of the jump can be computed as

$$d = \frac{1}{2}at^2$$

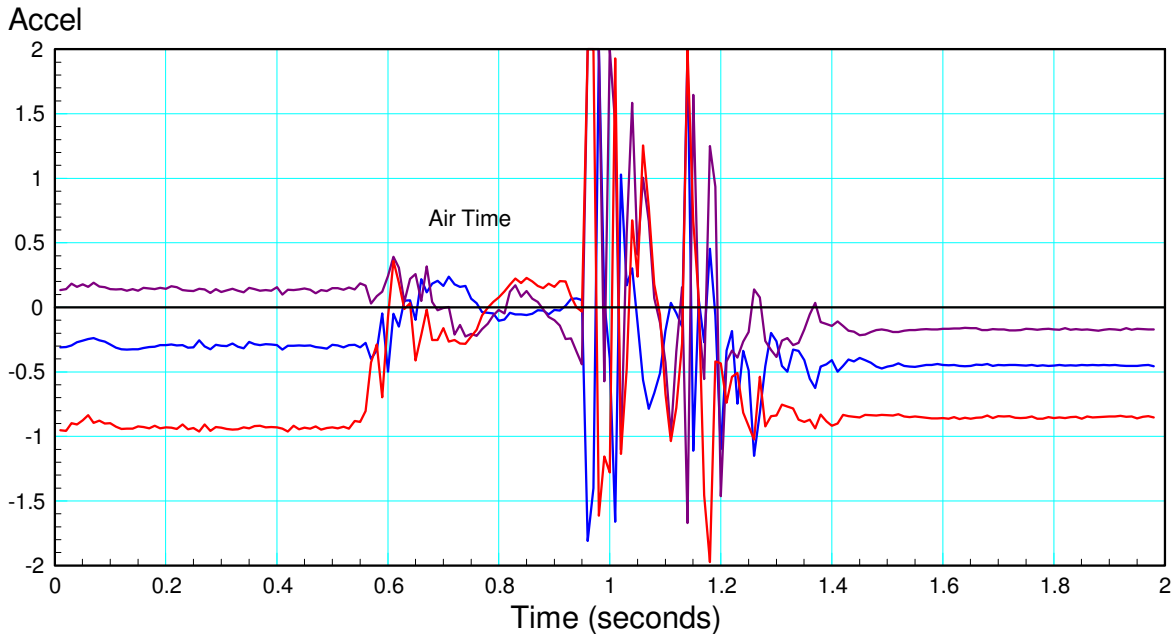
where t is the time from the apogee to the ground, or

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

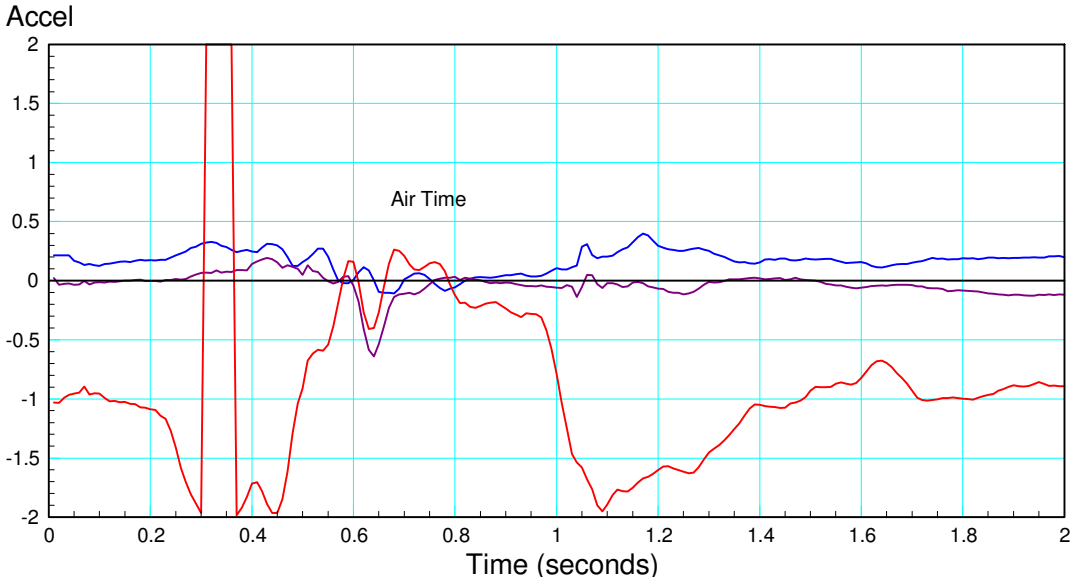
where t is the total time of the jump. Using the total time results in my vertical leap being measured as about 13 to 19cm. Not really impressive - it use to be more. Happens when you get old.

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

Measured air time with a GY521 accelerometer along with the corresponding height of the jump.



Jump: 260Hz Bandwidth. Air time is 400ms for a height of 19cm



Jump: 10Hz Bandwidth. Air time = 330ms for a height of 14cm