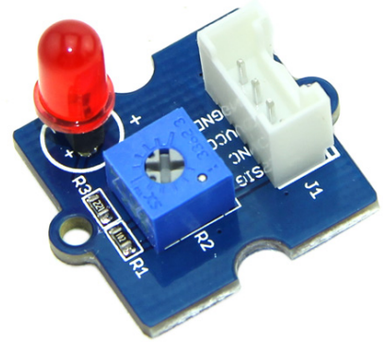
ELIXIRCONF™ 2017

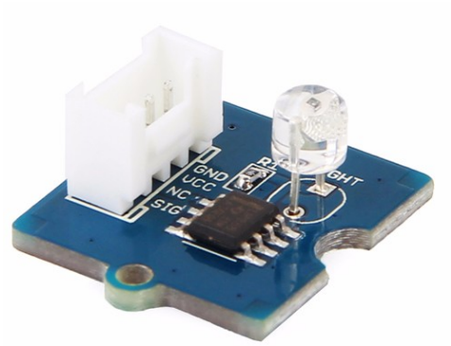
Nerves Training
Part 3: I2C

Motivating more complex inter-device communication

- Turning a wire on and off
  - Good for LEDs and buzzers
  - So low level!



- Measuring voltage
  - Conceptually simple
  - Inaccurate and noisy



Digital Buses at 10K

Serial Peripheral Bus - SPI

- Three wires for one-to-one communication*
- Simple and reliable - favorite for FPGA programmers and EEs
- Basis of SDCard communication
- Drawbacks
  - Low level transfer of bits (no flow control, error handling, etc)
  - Short distance

Inter-integrated Bus (I2C)

- Two wires for one-to-many device communication
- Fairly simple and widely supported
- Used to transfer monitor information to computers (See DDC pins on on VGA, DVI, and HDMI cables)
- Drawbacks
  - Slow - 100 or 400 kbps
  - Some interoperability/reliability issues

Universal Asynchronous Receiver/Transmitter (UART)

- Two wires for one-to-one or one-to-many device communication
- Well understood and reliable tech for low speed comms
- Serial ports (RS232)
- Drawbacks
  - Slow - 115200 bps very common, but Mbps possible
  - Large variation in upper layer protocols

1-wire

- One wire for one-to-many device communication
- Supports very low power devices with two connections
- Apple MagSafe and Dell power supplies
- Drawbacks
  - Really slow (<16.3 Kbps)

Controller Area Network (CAN)

- Twisted pair cable for many-to-many device communication
- Message-based networking w/ prioritization
- Cars and boats
- Drawbacks
  - Can become complex
  - Slow compared to Ethernet

Universal Serial Bus (USB)

- Two wires for one-to-one device communication
- Everywhere
- Drawbacks
  - High speed bus that needs extra TLC to get right
  - Needs hubs when you run out of ports

Ethernet

- 4 twisted pair cable for many-to-many device communication
- Insanely fast and runs IP
- Common for high end cameras
- Drawbacks
  - Expensive components (compared to other buses)

Common Embedded Digital Interfaces

Interface	Common use	Speed	Multiple devices?	Device notifications	Protocol
SPI	Connect to FPGAs, ADCs, and other sensors	Low Mbps	If Slave-Select wires are used	Interrupt wire needed	Depends on chip
I2C	Connect to sensors and other chips short distances	100 Kbps 400 Kbps	Yes	Interrupt wire needed	Mostly register-based
UART	Connect processors; GPS and modems	115200 to low Mbps	Depends	Yes	AT cmds; some specs
CAN	Connect processors in a vehicle or factory	125 Kbps to 1 Mbps	Yes	Yes	J1939, NMEA2k
1-Wire	Connect sensors	<16.3 Kbps	Yes	Polled	Defined by 1-Wire spec.
USB	Connect high speed devices or interface to PCs	1.5, 12, 480+ Mbps	USB hub required	Yes	HID, Mass storage, etc.
Ethernet	Interface with PCs and industrial equipment	100 Mbps+	Switch required	Yes	IP, GigE Vision, etc.

Choosing a bus

- Device availability usually picks the bus for you
- SPI for simple devices that stream lots of data
- I2C for hooking up control interfaces and low data rate devices
- UART for communicating with microcontrollers
- USB for non-trivial devices especially when Linux has built-in drivers

Primer on I2C

- One device (usually main CPU) is the bus master; everything else just responds to requests
- Each device has a hardcoded 7-bit address to identify it
- Transactions are usually less than 32 bytes
- Common protocol is to have devices expose a register-based interface
  - Writes are `<<register::size(8), value::binary>>`
  - Reads are a write to select the register followed by a read operation
  - Device auto-increments active register when not told so multi-byte reads and writes do the expected thing

Let’s integrate an I2C sensor

- Bosch BMP280
  - Barometric pressure and temperature
  - High enough resolution for altitude calculations

- A Grove sensor exists so it’s easy to connect
- Best of all, no drivers!*

Platforms supported (only for battery)

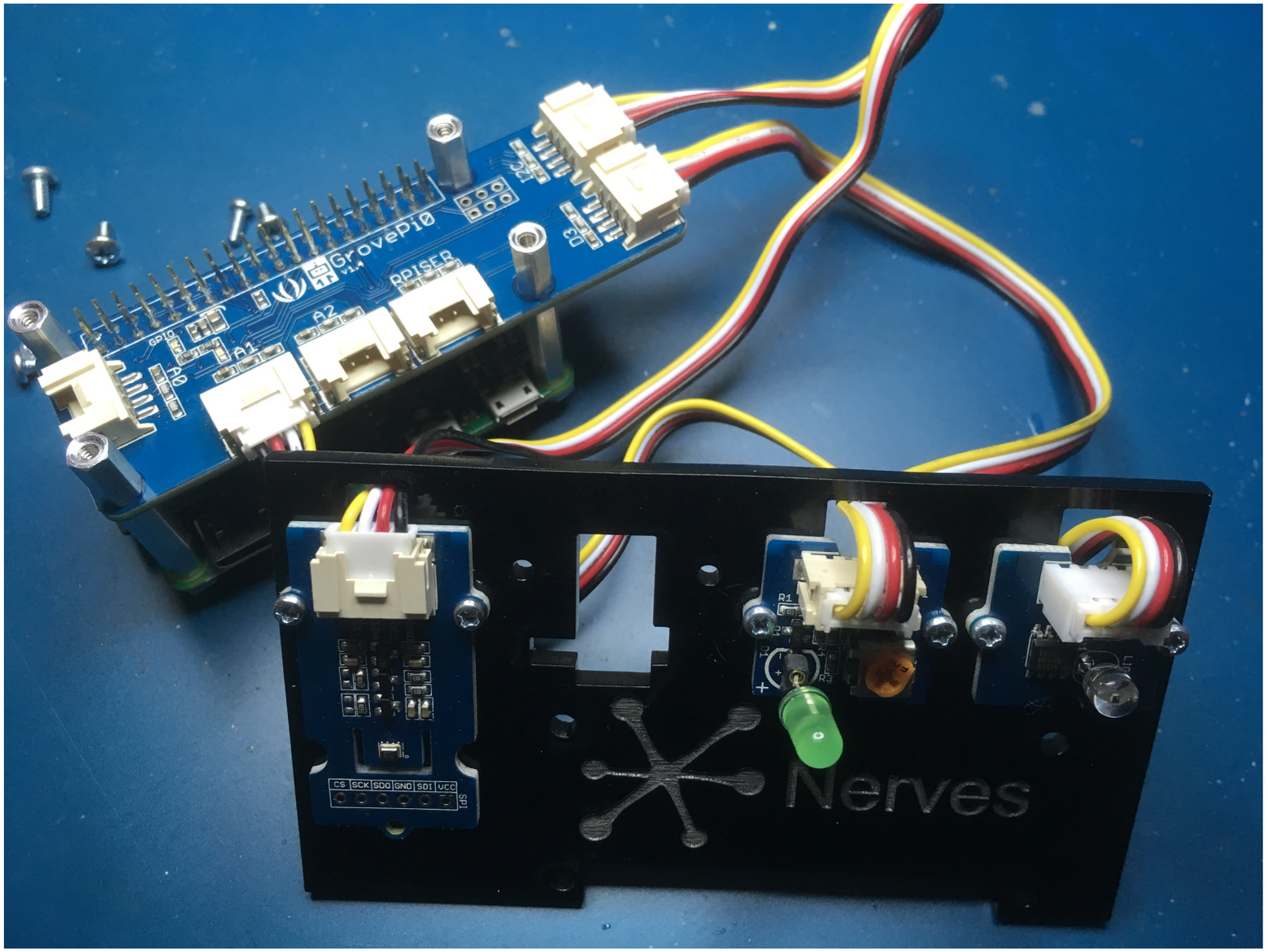
Platform	Seeeduino/Arduino	Rasberry Pi	Beaglebone	LinkIt ONE
Supported status	Supported	Not supported	Supported	Supported
Notes	If no version number is present for a specific platform, it means this product supports all versions within this platform.			

http://wiki.seeed.cc/Grove-Barometer_Sensor-BMP280/

Attach the barometer

- Grove - Barometer Sensor(BMP280)
- Two 2-56 3/8" screws
- Two 2-56 hex nuts

Attach the barometer



Connect to I2C port

Create a new nerves_init_gadget project

```
$ cd <workspace>
$ mix nerves.new barometer
$ cd barometer

add nerves_init_gadget to mix.exs and config.exs
copy ssh key config to config/config.exs
```

Skip typing

```
$ git clone \
https://bitbucket.org/fhunleth/nervestraining-barometer.git barometer
$ cd barometer
$ git checkout step1
```

ElixirALE.I2C

- device_names/0 - Enumerate available I2C buses
- detect_devices/1 - Scan an I2C bus for devices
- start_link/3 - Start the I2C bus GenServer
- read/2 - Read bytes from a device
- write/2 - Write bytes to a device
- write_read/3 - Combined write and read operation

https://hexdocs.pm/elixir_ale/ElixirALE.I2C.html#summary

Pull in elixir_ale

```
def deps(target) do
  [ system(target),
    {:bootloader, "~> 0.1"},
    {:nerves_runtime, "~> 0.4”},
    {:nerves_init_gadget, "~> 0.1”},
    {:elixir_ale, “~> 1.0”}
  ]
end
```

Docs: https://hexdocs.pm/elixir_ale/readme.html

Skip typing

```
$ git clone \
https://bitbucket.org/fhunleth/nervestraining-barometer.git barometer
$ cd barometer
$ git checkout step2
```

Try it out

```
$ mix firmware

$ mix firmware.push nerves.local
   -or-
$ ./upload.sh

$ picocom /dev/tty.usb*
```

Step 1: What's the device's I2C address?

- Check the datasheet
  - Google "Bosch BMP280" -> go to Documents & drivers and download it
  - octopart.com -> Search for BMP280
  - BST-BMP280-DS001-18.pdf (the last number is the rev)
- Hint: Look in section 5 of the datasheet

Datasheet

5.2 I²C Interface

The I²C slave interface is compatible with Philips I²C Specification version 2.1. For detailed timings refer to Table 27. All modes (standard, fast, high speed) are supported. SDA and SCL are not pure open-drain. Both pads contain ESD protection diodes to VDDIO and GND. As the devices does not perform clock stretching, the SCL structure is a high-Z input without drain capability.

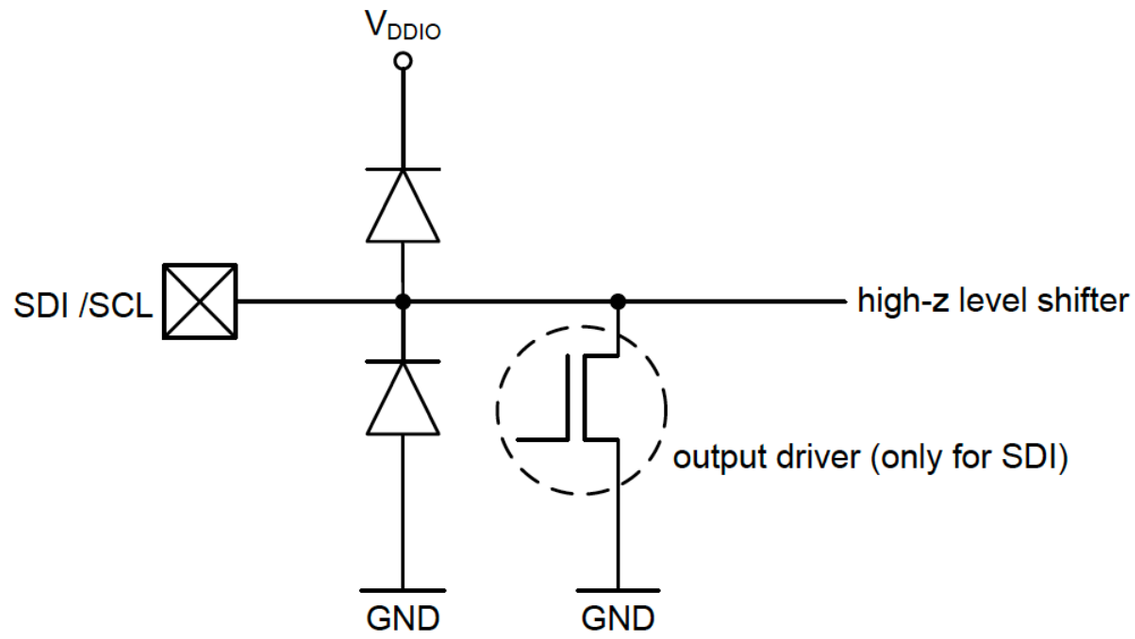
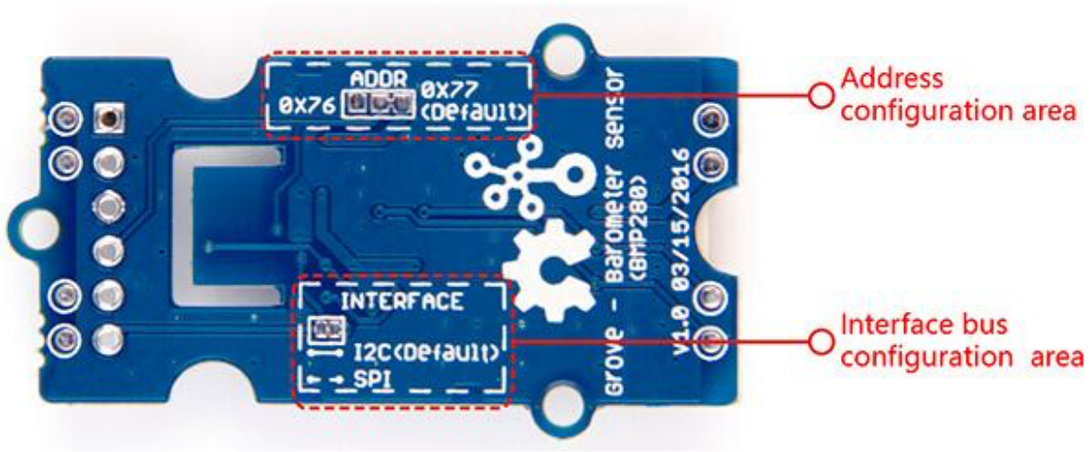


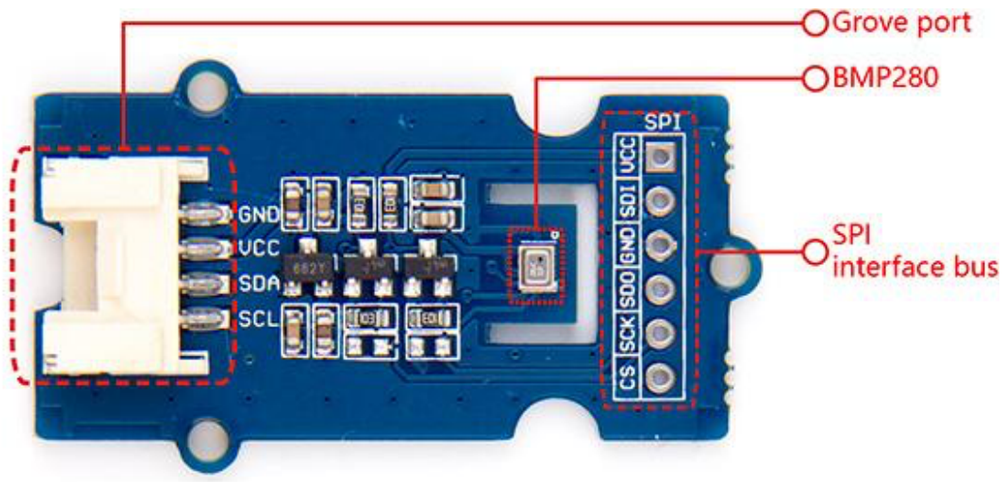
Figure 6: SDI/SCK ESD drawing

The 7-bit device address is 111011x. The 6 MSB bits are fixed. The last bit is changeable by SDO value and can be changed during operation. Connecting SDO to GND results in slave address 1110110 (0x76); connection it to V_{DDIO} results in slave address 1110111 (0x77), which is the same

Ok, so find the schematic

- Google “Grove BMP280 schematic”
- Sigh… it’s a battery kit schematic, that’s not helpful
- Or look at the back





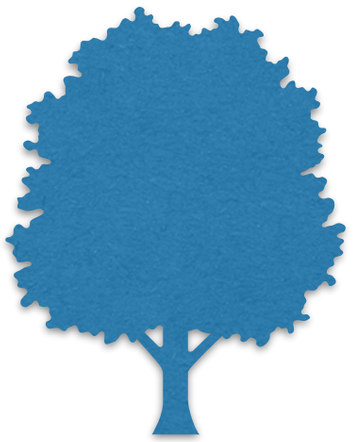
See if it shows up on the I2C bus

```
iex> alias ElixirALE.I2C
iex> h I2C.detect_devices()

Need to get the name of the I2C bus (called the device name)

iex> I2C.device_names()
[“i2c-1”]
iex> I2C.detect_devices("i2c-1")
[4, 199]
iex> I2C.detect_devices("i2c-1") |> IO.inspect(base: :hex)
[0x4, 0x77]

or

iex> I2C.detect_devices("i2c-1") |> hex
"[0x4, 0x77]"
```

Cheater way!

Where did “i2c-1” come from?

- "i2c" comes from Linux's i2c kernel module
- "-1" means the bus "1" as defined in the Linux kernel configuration (aka device tree)



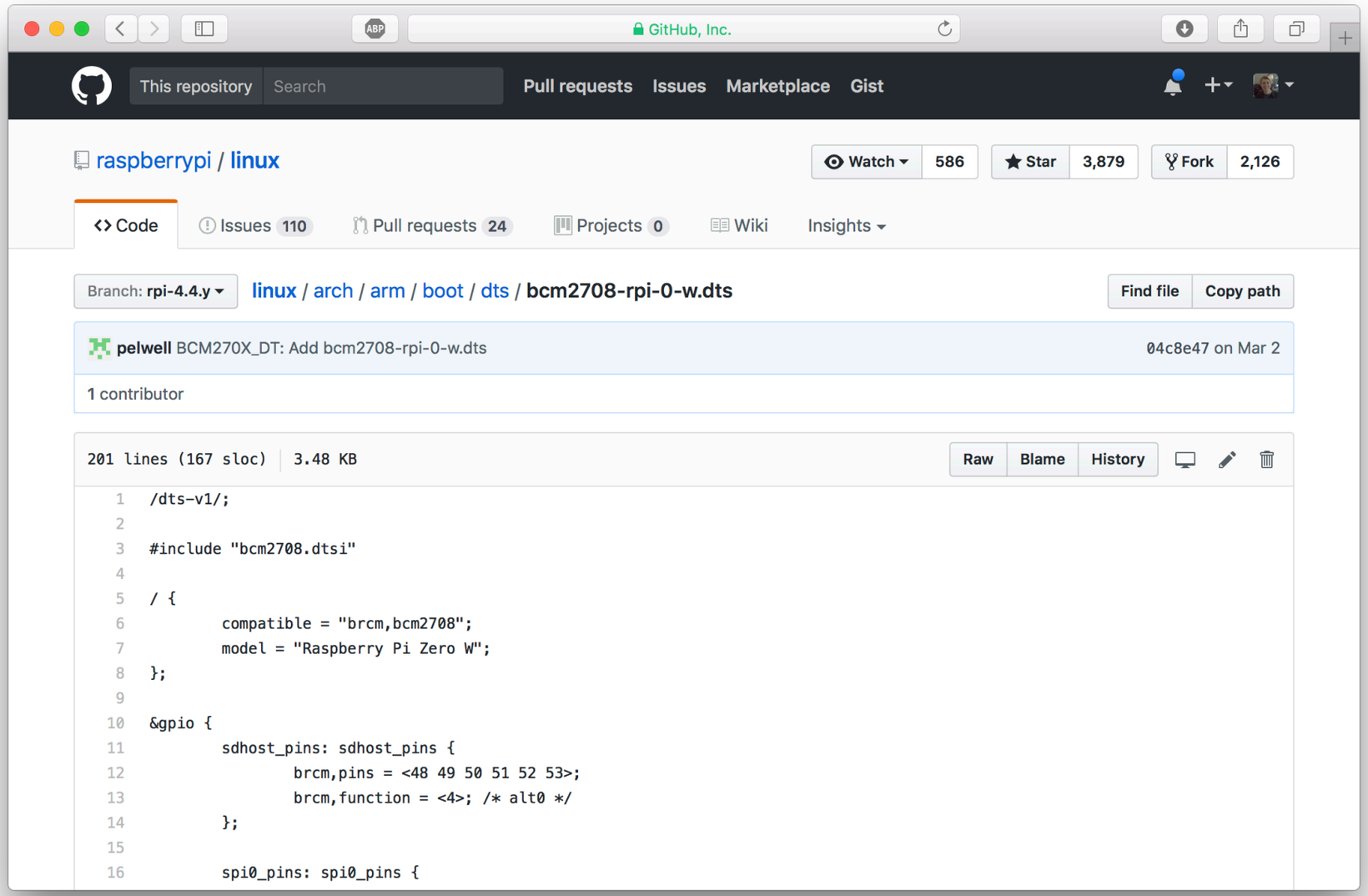
Raspberry Pi Bootloader

- Configured via config.txt
- https://www.raspberrypi.org/documentation/configuration/device-tree.md

3.3: BOARD-SPECIFIC LABELS AND PARAMETERS

Raspberry Pi boards have two I2C interfaces. These are nominally split: one for the ARM, and one for VideoCore (the "GPU"). On almost all models, `i2c1` belongs to the ARM and `i2c0` to VC, where it is used to control the camera and read the HAT EEPROM. However, there are two early revisions of the Model B that have those roles reversed.

Raspberry Pi Zero W DT



https://github.com/raspberrypi/linux/blob/rpi-4.4.y/arch/arm/boot/dts/bcm2708-rpi-0-w.dts

Raspberry Pi Zero I2C

```
124  &i2c0 {
125          pinctrl-names = "default";
126          pinctrl-0 = <&i2c0_pins>;
127          clock-frequency = <100000>;
128  };
129
130  &i2c1 {
131          pinctrl-names = "default";
132          pinctrl-0 = <&i2c1_pins>;
133          clock-frequency = <100000>;
134  };
135
136  &i2c2 {
137          clock-frequency = <100000>;
138  };
139
```

```
173  / {
174          __overrides__ {
175                  uart0 = <&uart0>,"status";
176                  uart0_clkrate = <&clk_uart0>,"clock-frequency:0";
177                  uart1 = <&uart1>,"status";
178                  i2s = <&i2s>,"status";
179                  spi = <&spi0>,"status";
180                  i2c0 = <&i2c0>,"status";
181                  i2c1 = <&i2c1>,"status";
182                  i2c2_iknowwhatimdoing = <&i2c2>,"status";
183                  i2c0_baudrate = <&i2c0>,"clock-frequency:0";
184                  i2c1_baudrate = <&i2c1>,"clock-frequency:0";
185                  i2c2_baudrate = <&i2c2>,"clock-frequency:0";
186                  core_freq = <&clk_core>,"clock-frequency:0";
187
```

If you want to learn more about device tree, search for “Device Tree for Dummies”

What to read and write?

Table 18: Memory map

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state
temp_xlsb	0xFC	temp_xlsb<7:4>				0	0	0	0	0x00
temp_lsb	0xFB	temp_lsb<7:0>								0x00
temp_msb	0xFA	temp_msb<7:0>								0x80
press_xlsb	0xF9	press_xlsb<7:4>				0	0	0	0	0x00
press_lsb	0xF8	press_lsb<7:0>								0x00
press_msb	0xF7	press_msb<7:0>								0x80
config	0xF5	t_sb[2:0]			filter[2:0]				spi3w_en[0]	0x00
ctrl_meas	0xF4	osrs_t[2:0]			osrs_p[2:0]			mode[1:0]		0x00
status	0xF3					measuring[0]			im_update[0]	0x00
reset	0xE0	reset[7:0]								0x00
id	0xD0	chip_id[7:0]								0x58
calib25...calib00	0xA1…0x88	calibration data								individual

Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Revision	Reset
Type:	do not write	read only	read / write	read only	read only	read only	write only

Binary patterns and integers

- ElixirALE.I2C reads and writes binaries so need to convert to and from integers and floats

- Examples:

```
iex> value = <<1,2,3,4>>
<<1, 2, 3, 4>>
iex> <<x::big-integer-size(32)>> = value; hex(x)
"0x1020304"
iex> <<x::little-integer-size(32)>> = value; hex(x)
"0x4030201"
iex> <<x::native-integer-size(32)>> = value; hex(x)
"0x4030201"
iex> <<x::integer-size(32)>> = value; hex(x)
"0x1020304"
iex> <<x::size(32)>> = value; hex(x)
"0x1020304"
```

If you forget, run this:

```
iex> h <<>>
```

Try it out

```
iex> alias ElixirALE.I2C
iex> {:ok, barometer} = I2C.start_link("i2c-1", 0x77)
{:ok, #PID<0.423.0>}
iex> <<pressure::size(24), temp::size(24)>> =
  I2C.write_read(barometer, <<0xf7>>, 6)
<<128, 0, 0, 128, 0, 0>>
iex> hex(pressure)
"0x800000"
iex> hex(temp)
"0x800000"
```

Not looking like a real temperature…

Back to the datasheet

- Anything look like it will turn temperature and pressure measurements on?

ctrl_meas

4.3.4 Register 0xF4 *"ctrl_meas"*

The *"ctrl_meas"* register sets the data acquisition options of the device.

Table 20: Register 0xF4 *"ctrl_meas"*

Register 0xF4 *"ctrl_meas"*	Name	Description
Bit 7, 6, 5	osrs_t[2:0]	Controls oversampling of temperature data. See chapter 3.3.2 for details.
Bit 4, 3, 2	osrs_p[2:0]	Controls oversampling of pressure data. See chapter 3.3.1 for details.
Bit 1, 0	mode[1:0]	Controls the power mode of the device. See chapter 3.6 for details.

<<osrs_t::size(3), osrs_p::size(3), mode::size(2)>>

mode

Table 10: *mode* settings

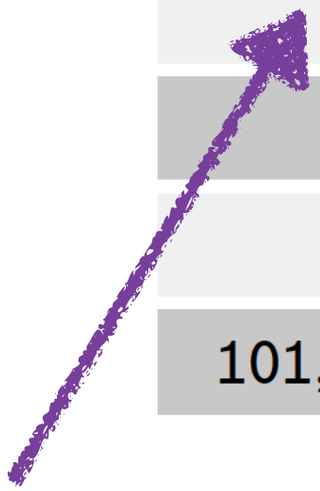
***mode*[1:0]**	**Mode**
00	Sleep mode
01 and 10	Forced mode
11	Normal mode

- TL;DR we want normal mode
- In normal mode, the sensor takes a measurement, then sleeps and then repeats
- Update rate is defined by another setting called *t_stby*
- Tradeoff between power and update rate (see Table 14 and 15)
- *mode* = 0b11 (3)

osrs_t

Table 5: *osrs_t* settings

***osrs_t*[2:0]**	**Temperature oversampling**	**Typical temperature resolution**
000	Skipped (output set to 0x80000)	–
001	×1	16 bit / 0.0050 °C
010	×2	17 bit / 0.0025 °C
011	×4	18 bit / 0.0012 °C
100	×8	19 bit / 0.0006 °C
101, 110, 111	×16	20 bit / 0.0003 °C



Recommended highest setting, so *osrs_t=0b010 (2)*

osrs_p

Table 4: *osrs_p* settings

Oversampling setting	Pressure oversampling	Typical pressure resolution	Recommended temperature oversampling
Pressure measurement skipped	Skipped (output set to 0x80000)	–	As needed
Ultra low power	×1	16 bit / 2.62 Pa	×1
Low power	×2	17 bit / 1.31 Pa	×1
Standard resolution	×4	18 bit / 0.66 Pa	×1
High resolution	×8	19 bit / 0.33 Pa	×1
Ultra high resolution	×16	20 bit / 0.16 Pa	×2

Why not?

Set *osrs_p=0b101 (5)*

Table 21: register settings *osrs_p*

osrs_p[2:0]	Pressure oversampling
000	Skipped (output set to 0x80000)
001	oversampling ×1
010	oversampling ×2
011	oversampling ×4
100	oversampling ×8
101, Others	oversampling ×16

Try it out

```
iex> mode=3
iex> osrs_t=2
iex> osrs_p=5
iex> <<osrs_t::size(3), osrs_p::size(3), mode::size(2)>>
"W"
iex> hex(v())
"<<0x57>>"
iex> I2C.write(barometer, <<0xf4, 0x57>>)
:ok
iex> <<pressure::size(24), temp::size(24)>> =
  I2C.write_read(barometer, <<0xf7>>, 6)
<<94, 197, 80, 127, 160, 128>>
iex> hex(pressure)
"0x5EC550"
iex> hex(temp)
“0x7FA080"
```


Looks promising! Hopefully you see something similar.

lib/barometer/application.ex

```elixir
defmodule Barometer.Application do
  use Application

  def start(_type, _args) do
    import Supervisor.Spec, warn: false

    children = [
      worker(ElixirALE.I2C, ["i2c-1", 0x77, [name: Barometer.I2C]]),
    ]

    opts = [strategy: :one_for_one, name: Barometer.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Skip typing

```
$ git checkout step3
```

lib/barometer.ex

```elixir
defmodule Barometer do
  alias ElixirALE.I2C

  def enable() do
    mode = 3  # normal
    osrs_t = 2 # x2 oversampling
    osrs_p = 5 # x16 oversampling
    ctrl_meas_register = 0xf4
    I2C.write(Barometer.I2C, <<ctrl_meas_register, osrs_t::size(3),
                               osrs_p::size(3), mode::size(2)>>
  end

  def raw_temp_and_pressure() do
    <<pressure::size(20), _::size(4), temp::size(20), _::size(4)>> =
      I2C.write_read(Barometer.I2C, <<0xf7>>, 6)
    {temp, pressure}
  end
end
```

Skip typing

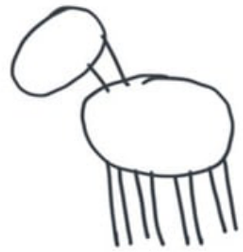
```
$ git checkout step3
```

Some cleanup - Ignore unused bits

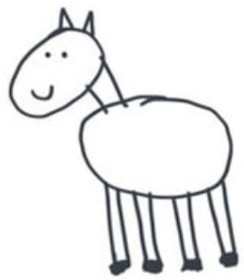
HOW TO:
DRAW A HORSE

BY VAN OKTOP

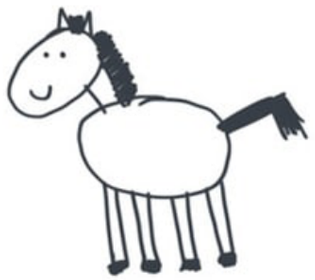
① DRAW 2 CIRCLES



② DRAW THE LEGS



③ DRAW THE FACE



④ DRAW THE HAIR

⑤ ADD SMALL DETAILS.

Pressure and Temperature formulas

Calculation of pressure and temperature for BMP280

Sample trimming values			
Register Address (LSB / MSB)	Name	Value	Type
0x88 / 0x89	dig_T1	27504	unsigned short
0x8A / 0x8B	dig_T2	26435	short
0x8C / 0x8D	dig_T3	-1000	short
0x8E / 0x8F	dig_P1	36477	unsigned short
0x90 / 0x91	dig_P2	-10685	short
0x92 / 0x93	dig_P3	3024	short
0x94 / 0x95	dig_P4	2855	short
0x96 / 0x97	dig_P5	140	short
0x98 / 0x99	dig_P6	-7	short
0x9A / 0x9B	dig_P7	15500	short
0x9C / 0x9D	dig_P8	-14600	short
0x9E / 0x9F	dig_P9	6000	short
0xA0 / 0xA1			

Sample measurement values			
Register Address (MSB / LSB / XLSB)	Name	Value	Type
0xF7 / 0xF8 / 0xF9[7:4]	UT [20 bit]	519888	signed long (*)
0xFA / 0xFB / 0xFC[7:4]	UP [20 bit]	415148	signed long (*)

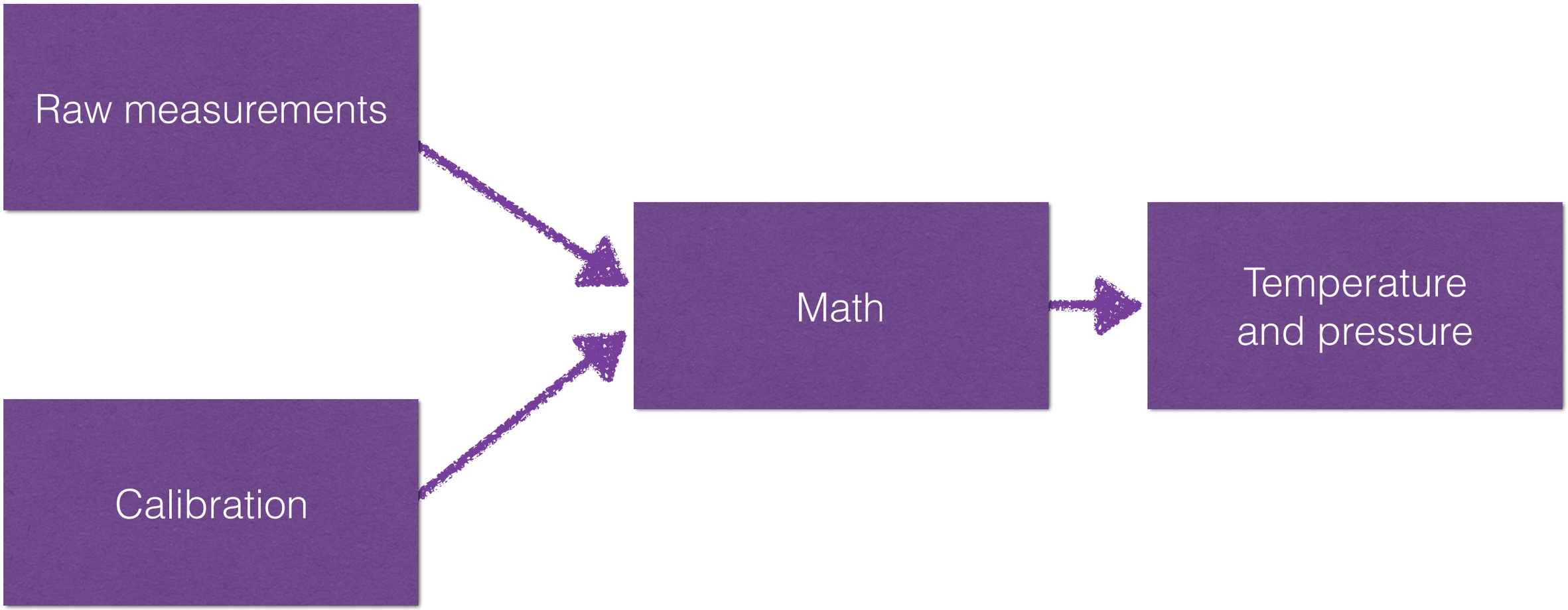
(*) Value is always positive, even though the compensation functions expect a signed integer as input

(*) Value is always positive, even though the compensation functions expect a signed integer as input

var1 =	128793,1787		var1 = (((double)adc_T)/16384.0 - ((double)dig_T1)/1024.0) * ((double)dig_T2);
var2 =	-370,8917052		var2 = ((((double)adc_T)/131072.0 - ((double)dig_T1)/8192.0) * (((double)adc_T)/131072.0 - ((double) dig_T1)/8192.0)) * ((double)dig_T3);
tfine =	128422		t_fine = (BMP280_S32_t)(var1 + var2);
T =	**25,08**	***Temperature [°C]***	T = (var1 + var2) / 5120.0;
integer result (**):	**2508**	***Temperature [1/100 °C]***	
var1 =	211,1435029		var1 = ((double)t_fine/2.0) - 64000.0;
var2 =	-9,523652701		var2 = var1 * var1 * ((double)dig_P6) / 32768.0;
var2 =	59110,65716		var2 = var2 + var1 * ((double)dig_P5) * 2.0;
var2 =	187120057,7		var2 = (var2/4.0)+(((double)dig_P4) * 65536.0);
var1 =	-4,302618389		var1 = (((double)dig_P3) * var1 * var1 / 524288.0 + ((double)dig_P2) * var1) / 524288.0;
var1 =	36472,21037		var1 = (1.0 + var1 / 32768.0)*((double)dig_P1);
p =	633428		p = 1048576.0 - (double)adc_P;
p =	100717,8456		p = (p - (var2 / 4096.0)) * 6250.0 / var1;
var1 =	28342,24444		var1 = ((double)dig_P9) * p * p / 2147483648.0;
var2 =	-44875,50492		var2 = p * ((double)dig_P8) / 32768.0;
p =	**100653,27**	***Pressure [Pa]***	p = p + (var1 + var2 + ((double)dig_P7)) / 16.0;
int32 result (**):	**100653**	***Pressure [Pa]***	
int64 result (**):	**25767236**	***Pressure [1/256 Pa]***	

(**) The actual result of the integer calculation may deviate slightly from the values shown here due to integer calculation rounding errors

TL;DR


Raw measurements
Calibration
Math
Temperature and pressure

lib/barometer.ex

Added formulas to convert raw samples.
Trust me that you don't want to retype these.

Skip typing

```
$ git checkout step4
```

Try it out

```
iex> Barometer.enable()
iex> report = Barometer.measure_all()
%{altitude: 15.832297389117103, pressure: 101134.97726988251,
  temperature: 24.288115969556383, units: :si}
iex> Barometer.celsius_to_fahrenheit(report.temperature)
75.7186087452015
iex> Barometer.pascals_to_inHg(report.pressure)
29.86514166710605
```

Going farther

- Update altitude calculation based on today's weather here
- Log measurements to a file
- Is this really accurate enough to measure small altitude differences?
- Integrate in with the grovepi library

Recap - How to support a new I2C sensor

- Download its datasheet
- Find its address in the datasheet
- Connect and verify that reading a register works
- Read through control register for power up/turn on sequences
- Poll sensor data registers

Troubleshooting

- Check for Python and C code samples
- Verify that the hardware works on Raspbian (if on an RPi)
- Connect a logic analyzer (e.g. Saleae Logic) to monitor I2C bus transactions
- See if another module has the same behavior

ELIXIRCONF™ 2017

Nerves Training
Part 3: I2C