BMP085 Barometric Pressure Sensor Quickstart

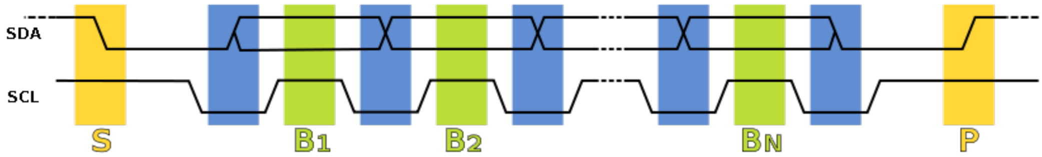
by Jimb0 | January 21, 2011 | *3 comments* Skill Level: ☆ Beginner

BMP085 Quickstart Guide

BMP085 Barometric Pressure Sensor Quickstart

Bosch's BMP085 is a rock-solid barometric pressure sensor. It features a measuring range of anywhere between 30,000 and 110,000 Pa. 'Pa' meaning the Pascal unit, which you'll probably more often see converted to hPa (hectoPascal), equal to 100 Pa, or kPa (kiloPascal), which is 1000 Pa. As a bonus the BMP085 also provides a **temperature** measurement, anywhere from 0 to 65 °C.

The BMP085 has a **digital** interface, I^{2}C to be specific. This means there may is a bit more overhead to get it talking to your microcontroller, but in return you get data that is much less susceptible to noise and other factors that may hamper an analog signal. I^{2}C is a synchronous two-wire interface, the first wire, SDA, transmits data, while a second wire, SCL, transmits a clock, which is used to keep track of the data. If you're using an Arduino to talk to the BMP085, the Wire library will conveniently take care of most of the work in communicating with the sensor.

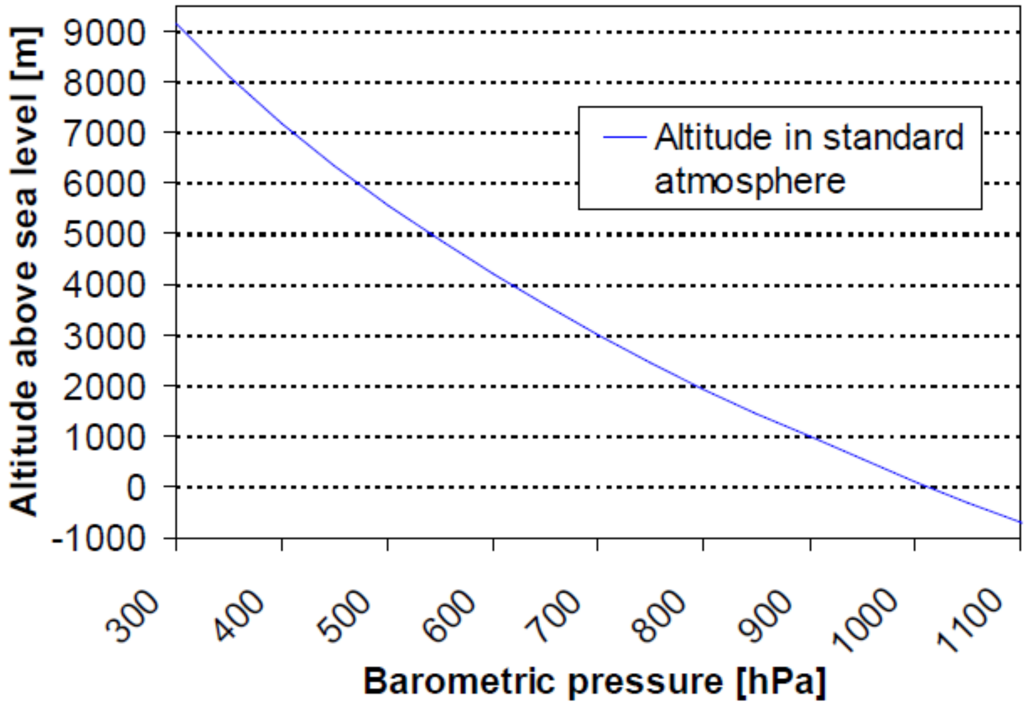


As with most SparkFun products, before you start toying around with something, it's always best to RTFM...err...read the fancy datasheet. The BMP085 datasheet covers everything you'd ever need to know about that little sensor on your breakout board.

Why do we care about barometric pressure?

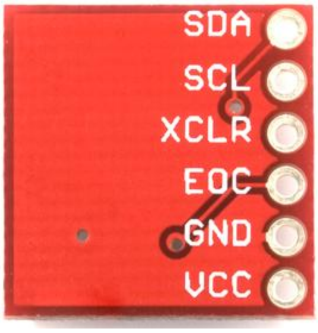
There are a lot of projects that can make use of barometric pressure data. Here's at least a couple reasons to care about pressure. You've probably heard weather reporters going on about low pressure systems and high pressure systems, that's because atmospheric pressure can be directly related to changes in **weather**. Low pressure typically means cloudier, rainier, snowier, and just generally uglier weather. While high pressure generally means clearer, sunnier weather. This means the BMP085 would be a perfect component for your Weather Prophet robot.

A second, widely applied use for pressure sensors is in **altimetry**. Barometric pressure has a measurable relationship with altitude, meaning you can use the BMP085 to deduce how high you (or maybe one of your robots) have climbed. At sea level air pressure is on average 1013 hPa, while here in Boulder, CO, at 5184 ft above sea level, average air pressure is about 831.4 hPa. The measuring limits of the BMP085 should allow you to measure pressure at elevations anywhere between -1640 to about 29,000 ft above sea level.



Hardware Explanation and Assembly

When you receive the BMP085 Breakout board, you're presented with a tiny 0.65 x 0.65” board with just 4 components, and six un-soldered holes. Your first task will be to **solder** *something* into those holes to get a good, solid electrical connection. What you solder into those holes is up to you. If I'm using a breadboard I like to stick some male headers in there, but simple wires are also a good option. However, if you do choose to go with wire, make sure to keep it short, I^{2}C is really only reliable at short distances (a few meters max).



Now, if you take a gander at the bottom side of the BMP085 Breakout you'll notice the labels of all six pins: 'SDA', 'SCL', 'XCLR', 'EOC', 'GND, and 'VCC.' **VCC** and **GND** are obviously the power pins. **SDA** and **SCL** are the I^{2}C communication lines. SDA being where the data is transmitted, and SCL is the clock that keeps track of that data. The last two pins, XCLR and EOC, are a couple extra functions of the BMP085. **XCLR** acts as a master reset. It's active-low, so if it's pulled to GND it will reset the BMP085 and set its registers to their default state. **EOC,** standing for "end of conversion", is a signal generated by the BMP085 that's triggered whenever a pressure or temperature conversion has finished. These last two pins are optional, and if you don't need to use them you can just leave them unconnected.

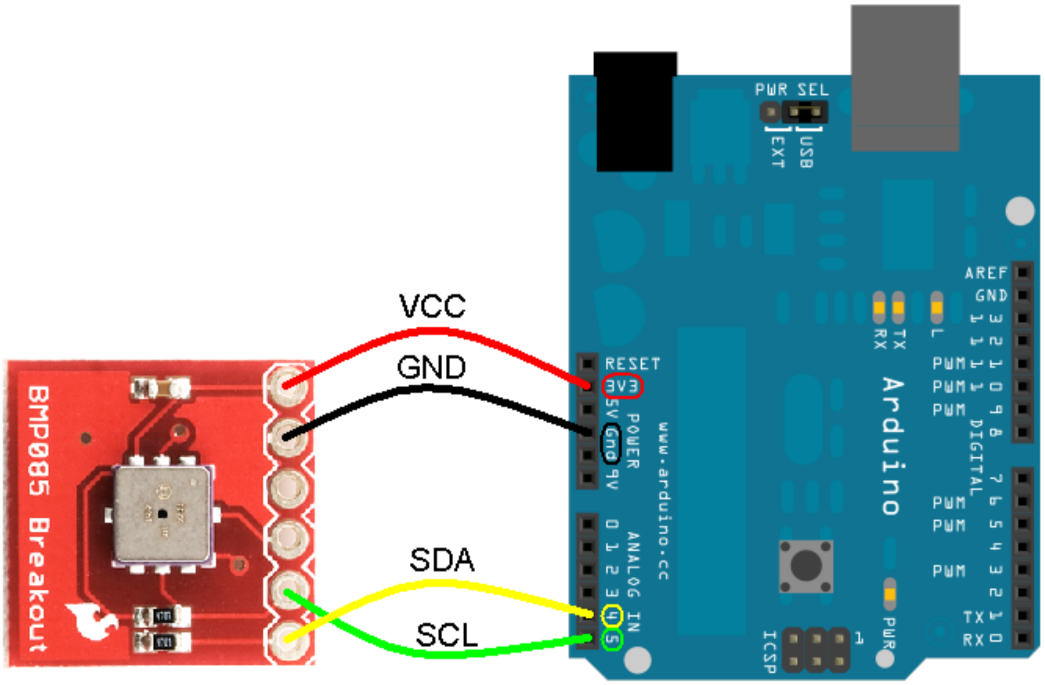
BMP085 Pin	Pin Function
VCC	Power (1.8V-3.6V)
GND	Ground

EOC	End of conversion output
XCLR	Master Clear (low-active)
SCL	Serial Clock I/O
SDA	Serial Data I/O

Pay special attention to how you power the BMP085. Its maximum supply voltage is 3.6V. So don't go feeding it 5V! If you've got a regular Arduino Uno, use the 3.3V header to supply power to the chip.

Getting pressure and temperature readings

Now that you've got the gist of the sensor, let's play! In this example we'll use an Arduino to initialize the BMP085, collect its pressure and temperature data, and display it on the serial output. First off, you'll need to connect it to the Arduino like so:

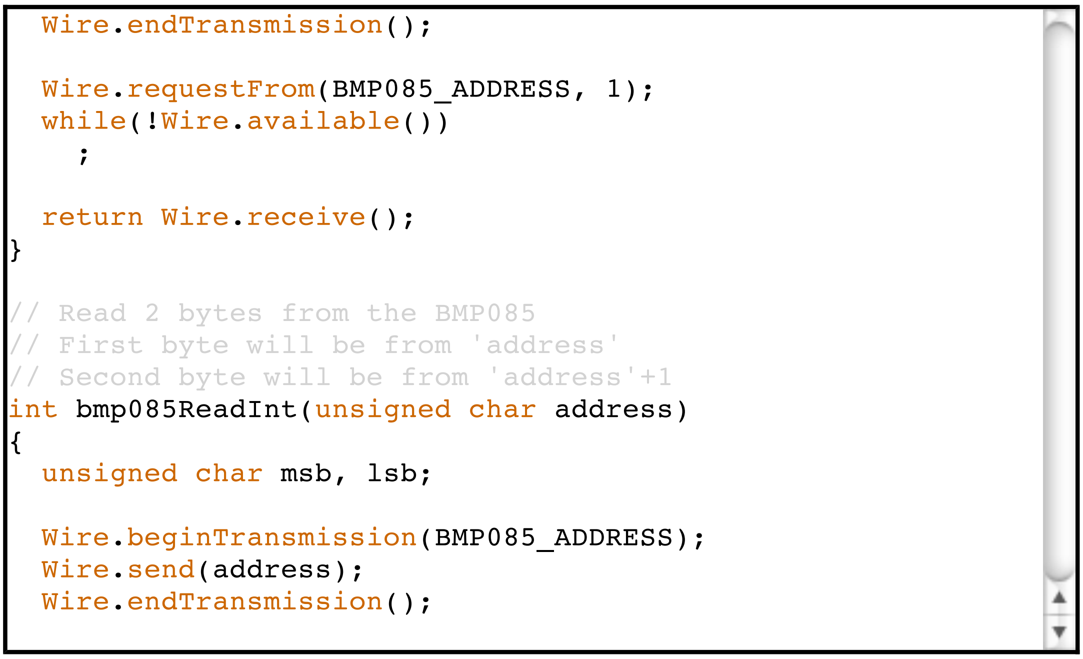


BMP085 Pin	Arduino Pin
VCC	3.3V
GND	GND
SCL	A5
SDA	A4

For now, we're going to ignore XCLR and EOC, it's safe to just leave them unconnected.

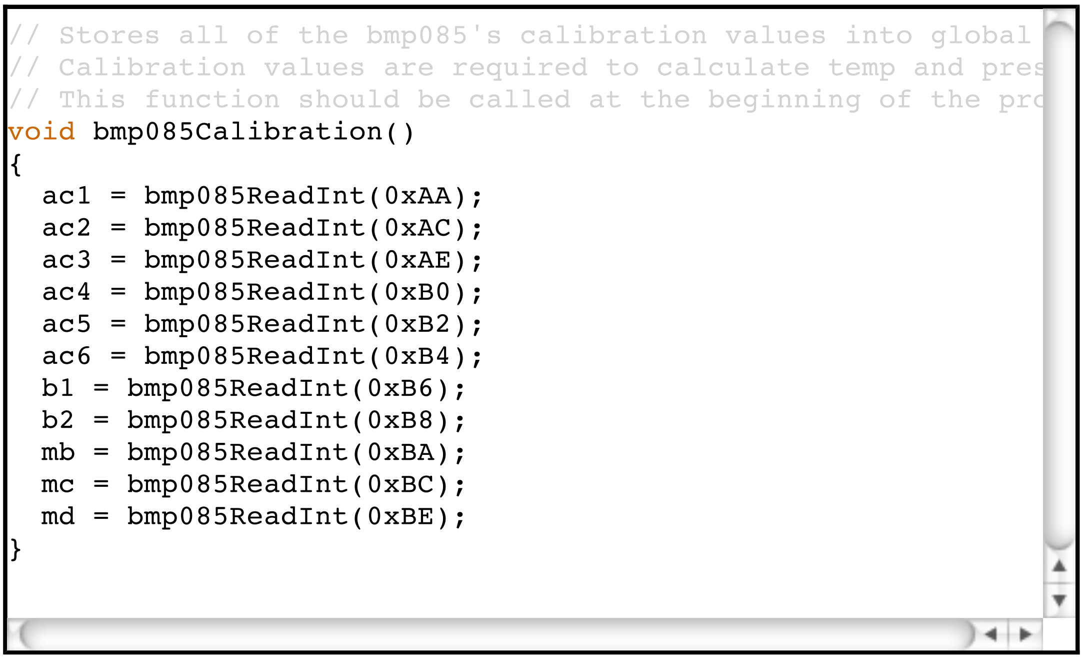
Now let's get to the nitty-gritty Arduino code. You can download the code here, or get ready to do some copy-pasta.

The first thing I like to do with digital sensors like this is make sure I can actually **communicate** with it. For many sensors this is half the battle. Let's first look at the functions used to read 8- and 16-bit values from the BMP085.



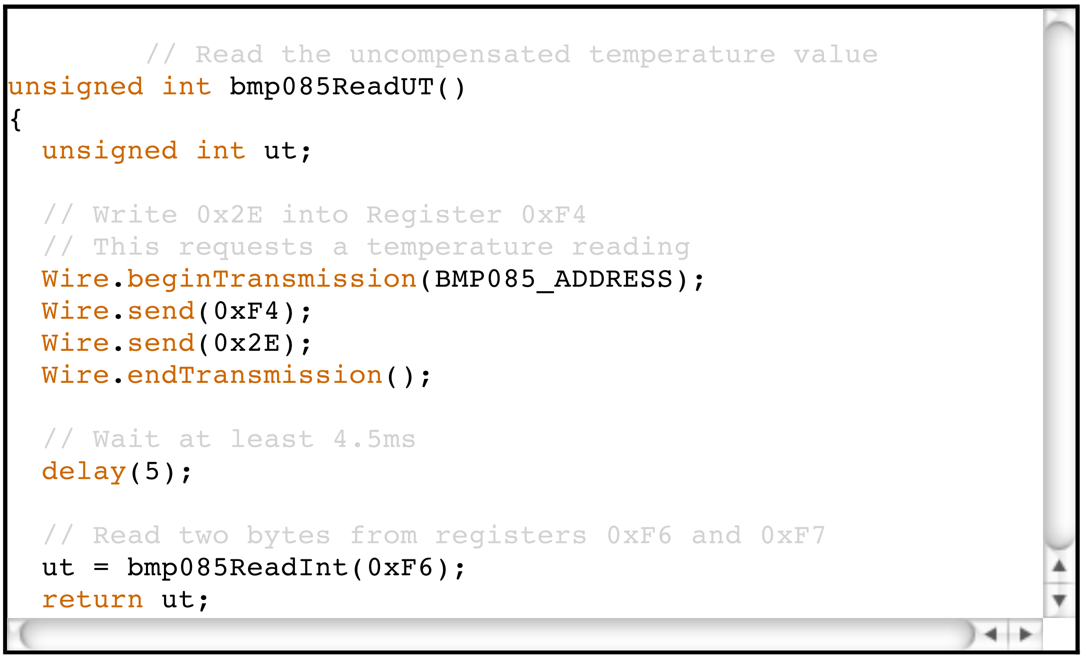
```
  Wire.endTransmission();

  Wire.requestFrom(BMP085_ADDRESS, 1);
  while(!Wire.available())
    ;

  return Wire.receive();
}

// Read 2 bytes from the BMP085
// First byte will be from 'address'
// Second byte will be from 'address'+1
int bmp085ReadInt(unsigned char address)
{
  unsigned char msb, lsb;

  Wire.beginTransmission(BMP085_ADDRESS);
  Wire.send(address);
  Wire.endTransmission();
```

Now, if you've glanced through the BMP085 datasheet, you may have noticed some very cryptic variables, and a lot of ugly math involving them. The BMP085 is calibrated in the factory, and left with a series of eleven 16-bit **calibration coefficients** stored in the device's EEPROM. These variables all play a small role in calculating the absolute pressure. They need to be read just once, at the start of the program, and stored for later use. We can use the I^{2}C functions above to read and store each of the calibration coefficients. We'll stick all of this in the setup() function, so it'll be called just once at the start of the program.



```
// Stores all of the bmp085's calibration values into global
// Calibration values are required to calculate temp and pres
// This function should be called at the beginning of the pro
void bmp085Calibration()
{
  ac1 = bmp085ReadInt(0xAA);
  ac2 = bmp085ReadInt(0xAC);
  ac3 = bmp085ReadInt(0xAE);
  ac4 = bmp085ReadInt(0xB0);
  ac5 = bmp085ReadInt(0xB2);
  ac6 = bmp085ReadInt(0xB4);
  b1 = bmp085ReadInt(0xB6);
  b2 = bmp085ReadInt(0xB8);
  mb = bmp085ReadInt(0xBA);
  mc = bmp085ReadInt(0xBC);
  md = bmp085ReadInt(0xBE);
}
```

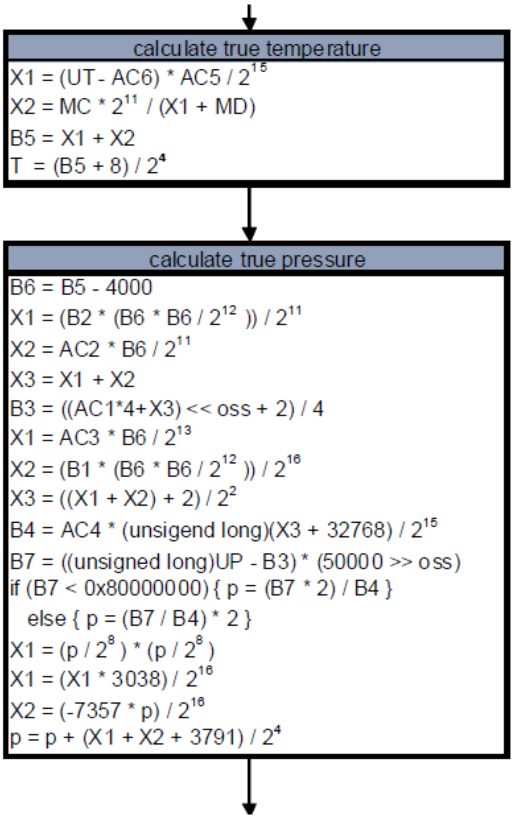
Once we've read the calibration values, we just need two more variables in order to calculate temperature and pressure. 'ut' and 'up' are the **uncompensated temperature** and **pressure** values, they're our starting point for finding the actual temperature and pressure. Every time we want to measure temperature or pressure, we need to first find out what these values are. The uncompensated temperature value is an unsigned 16-bit (int) number while 'up' is an unsigned 32-bit number (long).



```
          // Read the uncompensated temperature value
unsigned int bmp085ReadUT()
{
  unsigned int ut;

  // Write 0x2E into Register 0xF4
  // This requests a temperature reading
  Wire.beginTransmission(BMP085_ADDRESS);
  Wire.send(0xF4);
  Wire.send(0x2E);
  Wire.endTransmission();

  // Wait at least 4.5ms
  delay(5);

  // Read two bytes from registers 0xF6 and 0xF7
  ut = bmp085ReadInt(0xF6);
  return ut;
```

In those two functions, we're delaying to give the BMP085 enough time to finish its readings. The time for the delay is specified as the maximum conversion time in the datasheet. If you wanted to make it a bit more robust, you could read the **EOC** pin. While the conversion is still running, EOC will read LOW. Once finished it will go HIGH.

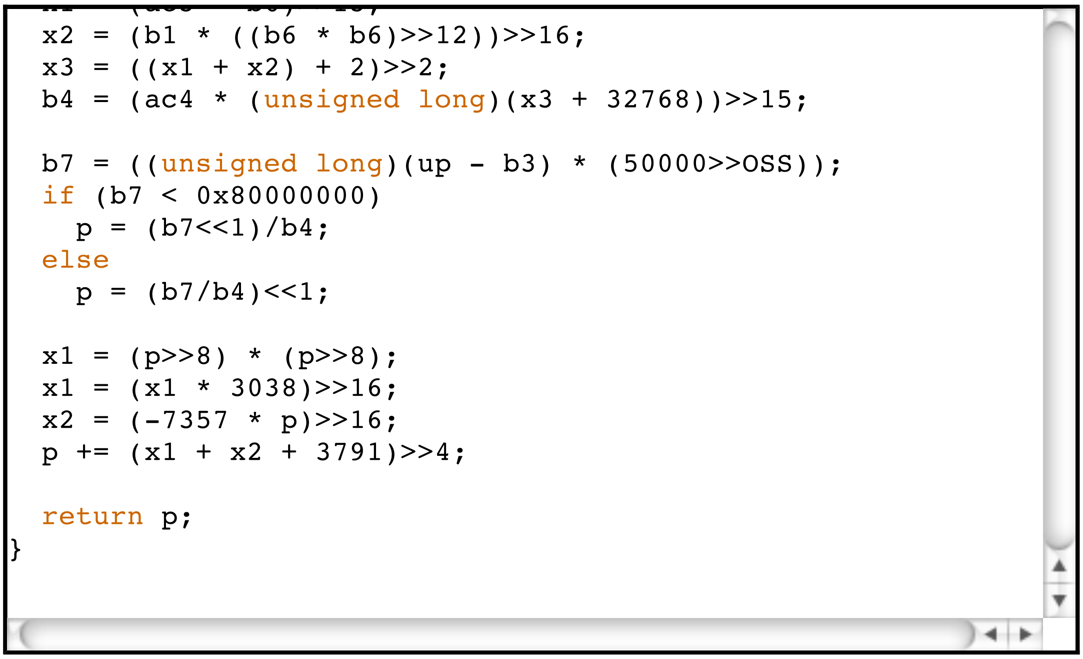
Now that we have all the variables we need, we need one last function that takes care of the following for us:



```
calculate true temperature
$X1 = (UT - AC6) * AC5 / 2^{15}$
$X2 = MC * 2^{11} / (X1 + MD)$
$B5 = X1 + X2$
$T = (B5 + 8) / 2^{4}$

calculate true pressure
$B6 = B5 - 4000$
$X1 = (B2 * (B6 * B6 / 2^{12})) / 2^{11}$
$X2 = AC2 * B6 / 2^{11}$
$X3 = X1 + X2$
$B3 = ((AC1*4+X3) << oss + 2) / 4$
$X1 = AC3 * B6 / 2^{13}$
$X2 = (B1 * (B6 * B6 / 2^{12})) / 2^{16}$
$X3 = ((X1 + X2) + 2) / 2^{2}$
$B4 = AC4 * (unsigend long)(X3 + 32768) / 2^{15}$
$B7 = ((unsigned long)UP - B3) * (50000 >> oss)$
if $(B7 < 0x80000000)$ { $p = (B7 * 2) / B4$ }
  else { $p = (B7 / B4) * 2$ }
$X1 = (p / 2^{8}) * (p / 2^{8})$
$X1 = (X1 * 3038) / 2^{16}$
$X2 = (-7357 * p) / 2^{16}$
$p = p + (X1 + X2 + 3791) / 2^{4}$
```

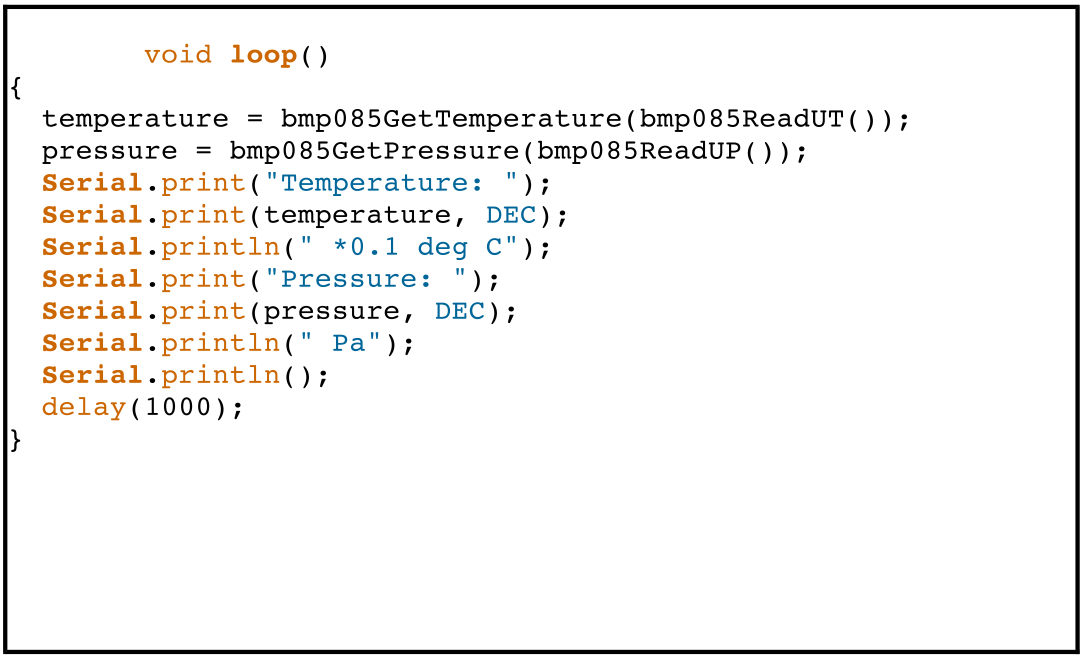
Ugh. Calculating the temperature isn't so bad, but, just...ugh, those pressure calculations. A couple quick notes: dividing by 2^N is the same thing as shifting a byte N bits to the right. Likewise, multiplying by 2^N, is the same thing as shifting it N bits to the left. We also have to be careful not to overflow any of the variables, which means paying close attention to the order of operations, and doing a little typecasting.

Here's the two functions we need to **calculate temperature** and **pressure**. Note that the variable *b5*, which is calculated in the *bmp085GetTemperature()* function, is also required to calculate pressure. This means you'll have to call *bmp085GetTemperature()*, before calling *bmp085GetPressure()*.



```
  x2 = (b1 * ((b6 * b6)>>12))>>16;
  x3 = ((x1 + x2) + 2)>>2;
  b4 = (ac4 * (unsigned long)(x3 + 32768))>>15;

  b7 = ((unsigned long)(up - b3) * (50000>>OSS));
  if (b7 < 0x80000000)
    p = (b7<<1)/b4;
  else
    p = (b7/b4)<<1;

  x1 = (p>>8) * (p>>8);
  x1 = (x1 * 3038)>>16;
  x2 = (-7357 * p)>>16;
  p += (x1 + x2 + 3791)>>4;

  return p;
}
```

And finally, we need to write something into the main loop() function so that our program actually does something. Let's just calculate the temperature and pressure, and spit it out over a serial line.

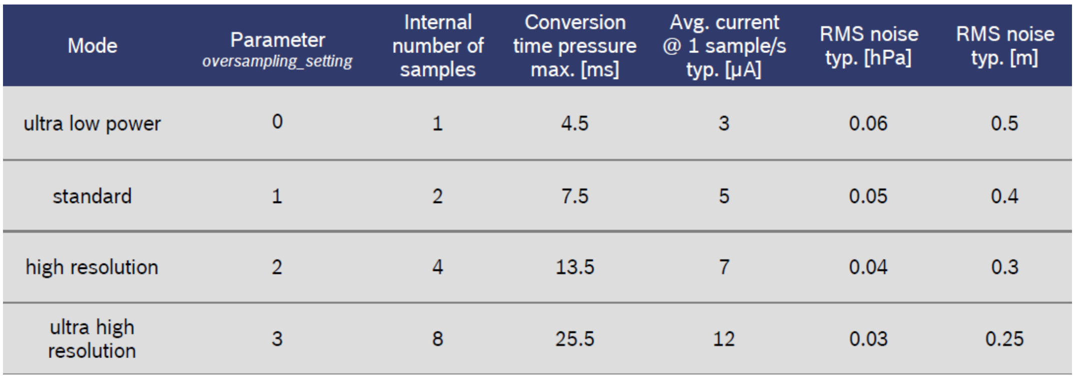


```
        void loop()
{
  temperature = bmp085GetTemperature(bmp085ReadUT());
  pressure = bmp085GetPressure(bmp085ReadUP());
  Serial.print("Temperature: ");
  Serial.print(temperature, DEC);
  Serial.println(" *0.1 deg C");
  Serial.print("Pressure: ");
  Serial.print(pressure, DEC);
  Serial.println(" Pa");
  Serial.println();
  delay(1000);
}
```

Send that over to your Arduino. Open up the serial monitor and check that the baud rate is set to 9600. Are you getting something reasonable? The data should be in units of 0.1°C for the temperature, and Pa for pressure. Here in my office I'm reading a comfy 258 (25.8°C) for temperature, and a pressure of 83523 Pa. If your elevation is lower than Boulder's ~5200ft, the measured pressure should read higher. Average pressure for those of you at sea level should be about 101325 Pa.

If the program slows to a halt before printing any readings, that usually means something's hooked up incorrectly. Make sure you didn't swap SDA and SCL!

You may have noticed a few of the calculations take into account an oversampling setting, *OSS*. OSS selects which mode the BMP085 operates in, and can be set to either 0, 1, 2, or 3. OSS determines how many samples the BMP085 will take before it sends over its uncompensated pressure reading. With OSS set to 0, the BMP085 will consume the least current. Setting OSS to 3 increases resolution, as it samples pressure eight times before producing a reading, this comes at a cost of more power usage. If you want to change OSS, just set it accordingly at the top of the program. Try changing OSS to 3, does the data become more stable?



Mode	Parameter *oversampling_setting*	Internal number of samples	Conversion time pressure max. [ms]	Avg. current @ 1 sample/s typ. [µA]	RMS noise typ. [hPa]	RMS noise typ. [m]
ultra low power	0	1	4.5	3	0.06	0.5
standard	1	2	7.5	5	0.05	0.4
high resolution	2	4	13.5	7	0.04	0.3
ultra high resolution	3	8	25.5	12	0.03	0.25

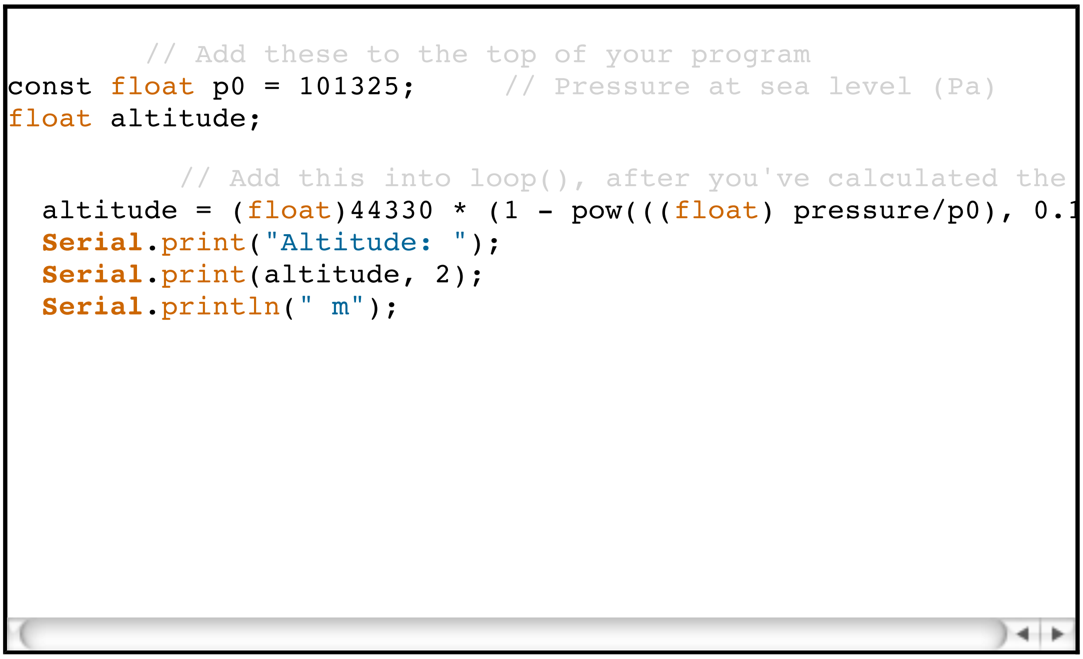
Applying the data: Altimetry and Meteorology

Knowing the pressure is all good fun, but what really matters is what we do with this shiny knowledge. As I mentioned above, pressure is most often used to either calculate altitude, or as an indicator of what the weather might be planning on doing. Let's make a altitude tracking weather forecaster!

We're really just one step away from calculating the altitude, all we need is the non-linear equation that converts pressure to altitude. This equation is conveniently in the BMP085's datasheet:

$$\text{altitude} = 44330 * \left(1 - \left(\frac{p}{p_0}\right)^{\frac{1}{5.255}}\right)$$

p_0 is the *average* pressure at sea level 101325 Pa, and p is the pressure that we measured. Note that this equation gives you altitude in units of *meters*. So, after we calculate pressure in our code, we just need to add these lines to calculate and print out the altitude:



```
        // Add these to the top of your program
const float p0 = 101325;     // Pressure at sea level (Pa)
float altitude;

          // Add this into loop(), after you've calculated the
  altitude = (float)44330 * (1 - pow(((float) pressure/p0), 0.1
  Serial.print("Altitude: ");
  Serial.print(altitude, 2);
  Serial.println(" m");
```

The altitude that you calculate might be off a tad, as the pressure will vary depending on the weather. Google Earth says the altitude here at SparkFun is about 1580 m (5183 ft), while at the pressure I'm measuring altitude calculates out to about 1600m (5249 ft). Close enough for ~~government~~ SparkFun work.

Now, let's add a little code to turn our Arduino into a simple weather forecaster. If our altitude is known, we can turn the above formula around to calculate expected pressure at a specific altitude.

$$p = p_0 * (1 - \frac{altitude}{44330})^{5.255}$$

You can use Google Earth, this website, or even a GPS module to find your altitude.

If your measured pressure is about equal to the expected pressure at your altitude, weather will probably be fair, if somewhat cloudy. Typically anything over about 250 Pa of the expected pressure will result in awesome, sunny weather, while anything 250 Pa below means rainy/snowy weather. Knowing all of that, here's some quick code to implement a simple weather forecaster:

```
        // Add these to the top of your program
const float currentAltitude = 1580.08; // current altitude in M
const float ePressure = p0 * pow((1-currentAltitude/44330), 5.2
float weatherDiff;

  // Add this into loop(), after you've calculated the pressure
  weatherDiff = pressure - ePressure;
  if(weatherDiff > 250)
    Serial.println("Sunny!");
  else if ((weatherDiff <= 250) || (weatherDiff >= -250))
    Serial.println("Partly Cloudy");
  else if (weatherDiff > -250)
    Serial.println("Rain :-(");
  Serial.println();
```

I'll stress that this is very *simple*. To get a better idea of what the weather's going to do, you need to look at how pressure is changing over time. Measuring the rate of change of pressure, you can get a much better idea of what the weather is going to do. For example, pressure changes of more than about -250 Pa per hour (Pa/h), result in unstable weather patterns like thunderstorms, while lower rates, between -50 and -250 Pa/h produce stable, though still rainy weather. Now that you know how to use the BMP085, maybe you can program your Arduino to out-forecast your local weatherman!

▼ Comments 3 comments

Login to post comments.

Microzoul | **April 12, 2011 at 7:57 AM** ★ **1**

what this line means b3 = ((((long)ac1)*4 + x3)2;

is it possible way to calculate altitude directly from UP value without calibration overhead thank you

bloater | **May 13, 2011 at 3:08 PM★ 1**

Excellent tutorial Jimbo - thanks ;)

What's your opinion on using i2c pullups connected to 5V with a 3V3 part such as the BMP085? Presumably this is out of spec and may stress the part. I went to the trouble of tweaking the Arduino 'Wire' library to avoid enabling the AVR pullups (it's a shame this is not an option in the library).

- Bernard ;)

Keebler | **June 7, 2011 at 9:30 AM★ 1**

Great tutorial... I'm concerned that the diagram has 5v SDA and SCL lines going to a 3.3v part without a logic level shifter. I don't doubt it will work but won't reduce the life of the part and could this affect accuracy?

Also, is there a way to disable the code "scroll view" windows? They don't print well (at all)... anyone know a workaround? Thanks!