Processing Analog Outputs from Environmental Sensors

David R. Brooks, © 2021

Arduinos like the UNO and its relatives include the ability to accept DC voltage inputs in the range from 0 V to the operating voltage of the board, 5 V for UNOs and many others. (Negative voltage inputs are never allowed.) Some Arduino boards, like the Pro Mini, are available in 5 V or 3.3 V versions. UNOs have 6 analog input pins, labeled A0 through A5; other boards may have more or less. The analog pins can be assigned to operate either in an "input" or "output" mode. For typical use with analog voltage inputs,

`pinMode(`*pin designation*`, INPUT);`

where the *pin designation* can be the pin identifier, A0 through A5, or a type `int` variable name assigned one of those pin values. The `pinMode()` statement is often missing from Arduino code because `INPUT` is the default mode for pins A0 through A5. When the analog input pins are used as output pins – a topic for a different discussion – you would specify `OUTPUT` as the mode. You can find an introduction to reading analog pins here: https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/.

The analog-to-digital-to-analog process involves using `analogRead(`*pin designation*`)` to digitize the voltage signal on an analog input pin and then, usually, converting that digitized value back to its analog equivalent. For some applications it may be sufficient just to work with the digitized integer value.

The A-to-D conversion on most Arduinos has 10-bit resolution, where an analog input voltage is digitized into 1024 possible integer values between 0 and 1023. Hence the resolution for the conversion for 5 V boards is 5/1024≈4.9 mV. On UNOs and other Arduinos like the Nano, Mini, and Mega, that use ATmega processors, it takes about 0.1 ms to read an analog input, or no more than about 10,000 reads per second; for environmental monitoring, reading analog pins at high speeds is almost never a consideration.

The default conversion of an analog input voltage in the 0-5 V range (no voltage outside this range should ever be applied) is:

(1) $V_{analog} = 5.0 \bullet V_{digitized}/1023$

Some online sources insist that the divisor in equation (1) should be 1024 and not 1023. However this is essentially a moot point because the 10-bit resolution produces in any case only a relatively coarse A-to-D-to-A representation of the input voltage over the 5 V range.

Equation (1) assumes that the board operating voltage is exactly 5 V. This may not be true. If the board is powered by a USB cable, the default reference voltage can be 5±0.25 V as allowed by the USB standard. Often the USB voltage from a computer is higher than 5.0 V, but in any case you need to measure that voltage to provide the correct value in equation (1). On a board I tried connected through a USB cable to my desktop computer, the measured voltage on the 5 V power pin was 5.15 V. Hence, the 5.0 in equation (1) should be replaced with 5.15. That is, in general, equation (1) should be replaced by:

(2) $V_{analog} = V_{ref} \bullet V_{digitized}/1023$

where V_{ref} is the voltage measured at the 5 V power pin.

If the board is powered from an external 9-12 VDC plug-in power supply through the 2.1 mm input jack, then that input voltage passes through an on-board regulator that should provide very

close to 5.00 V. You can also provide 9-12 VDC to an Arduino board's Vin pin, which also goes directly through the on-board 5 V regulator.

Finally, you can power a 5 V Arduino by applying a *regulated* 5 VDC source to the 5 V power pin; never apply any voltage higher than about 5.25 V to this pin, as you may destroy your Arduino board. That regulated voltage could be as low as about 4.75 V, but for a lower voltage your board may not work at all.

For many projects, you will write and test code with your Arduino connected to your computer, but then you will use your project away from your computer, with an external power supply. It's not very convenient to have to keep testing and replacing the measured reference voltage value used in equation (2) depending on the power source.

There's another way to determine how to do A-to-D-to-A conversions with 5 V Arduinos like the UNO. Recall equation (2), which assumes that the reference voltage – the voltage powering the board as measured on the 5 V power pin – is known. But suppose, instead, that V_{ref} is assumed to be unknown and V_{analog} is assumed to be known. Then:

(3) $V_{ref} = V_{analog} \bullet (1023/V_{digitized})$

For 5 V Arduinos, UNOs and many others, there is a 3.3 V power output pin. In Figure (1) you can see this pin labeled on the lower row of UNO pins, to the left of the 5V and two GND pins. (The Vin pin, mentioned above, is to the right of the two GND pins.) Regardless of how the 5 V board is powered, that input voltage will pass through an on-board 3.3 V regulator that *should* produce an output of precisely 3.30 V; that was true for several boards I tested, although you should still measure the voltage at the 3.3 V output pin just to be sure.

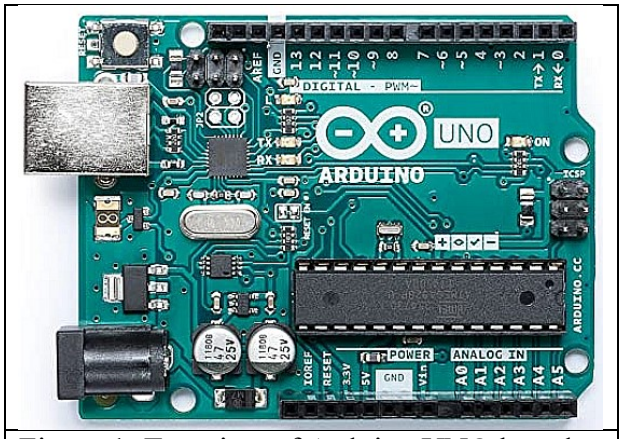


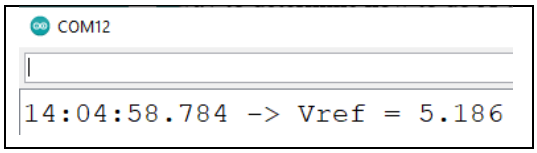
Figure 1. Top view of Arduino UNO board.

So, if you attach the 3.3 V power pin to one of the analog pins, and use equation (3), the calculated V_{ref} is what you would use in equation (2) for converting a digitized voltage back to its analog equivalent.

Sketch 1 shows how to use Equation (3).

Sketch 1. Calculating V_{ref} for a 5 V Arduino board.

```
/* VREF_test.ino, D. Brooks, September 2021
  Shows how to compute accurate reference voltage from the internal 3.3 V power
  output pin. The result may be different from the default value of 5.0,
  depending on how the board is powered.
*/
const int Vref_pin=A0;
int Vref_digitized;
float Vanalog=3.30,Vref;
void setup() {
  Serial.begin(9600);
  pinMode(Vref_pin,INPUT); // Optional, as this is the default state.
  Vref_digitized=analogRead(Vref_pin);Vref=Vanalog*1023./Vref_digitized;
  Serial.print("Vref = ");Serial.println(Vref,3);
}
void loop() {}
```



Here's some output from Sketch 1 when an UNO was powered through a USB cable from my computer. The calculated and reported V_{ref} value of 5.186 is very close to the 5.18 V measured with an inexpensive digital voltmeter, and it's the value that should be used, rather than the nominal value of 5.0, when equation (3) is used to process an input analog voltage. This code will give a different value for V_{ref} if the board is powered from a source other than a USB cable, or perhaps even with USB cables coming from different computers or others sources, but in any case you don't have to measure the voltage at the 5 V power pin every time you use a different power source. The only possible downside to this method is that it ties up one of the available analog input pins.

Other considerations for using analog input pins

Note that you may encounter problems when you're reading multiple analog input channels one right after the other. Arduino boards require a little time to "switch" from one pin to another. Unless your application has some very specific reason for reading analog values as quickly as possible, the recommended solution is to read each pin twice and overwrite the first value with the second. For a series of analog pin readings, separate each `analogRead()` by a delay of perhaps 10 or 20 milliseconds between each read, something like:

```
V1=analogRead(A0); delay(10); V1=analogRead(A0); delay(10);
V2=analogRead(A1); delay(10); V2=analogRead(A0); delay(10);
```

It's also often recommended that for some sensors that work in "noisy" or highly variable environments you should consider reading multiple values and averaging them. In that case you could read the analog pin N times with a short delay between reads (a few milliseconds?), sum the values, and then divide the total by N. Make sure that you don't sum so many values that you exceed the size limit on type int variables (32767). Also, when you divide an integer sum by N, you should force real number division rather than integer division by specifying `(float)N` instead of `N`.

In some cases when you are reading and averaging multiple values, it may be worthwhile to calculate both the maximum and minimum values during the averaging period. This can alert you to spurious sensor outputs.

You can also use the Arduino IDE's Serial Plotter option, accessed from the Tools menu, to look at a time series graph of sensor output. Sketch 2 is a version of Sketch 1 with code added to display values from a TMP35 sensor every half second. The x-axis values count the number of values displayed. I pushed the side of a glass of ice water against the sensor for a few seconds and then pinched the sensor between my fingers to heat it up again. There are no "spurious" values, but the graph does illustrate the coarse resolution of the A-to-D-to-A process.

Sketch 2. Displaying values from a TMP35 sensor.

```
/* VREF_testTMP3x.ino, D. Brooks, September 2021
  Shows how to compute accurate reference voltage
  From the internal 3.3 V power output pin. The
  result may be different from the default value
  of 5.0, depending on how the board is powered.
    Adds TMP35 sensor to look at values with
  Serial Plotter.
*/
const int Vref_pin=A0,TMPpin=A1;
int Vref_digitized,T_digitized;
float Vanalog=3.30,Vref,T;
void setup() {
  Serial.begin(9600);
  Vref_digitized=analogRead(Vref_pin);
  Vref=Vanalog*1023./Vref_digitized;
  //Serial.print("Vref                    =
");Serial.println(Vref,3);
}
void loop() {
  T_digitized=analogRead(TMPpin);
  T=100.*T_digitized*Vref/1023.;
  Serial.println(T);
  delay(500);
}
```

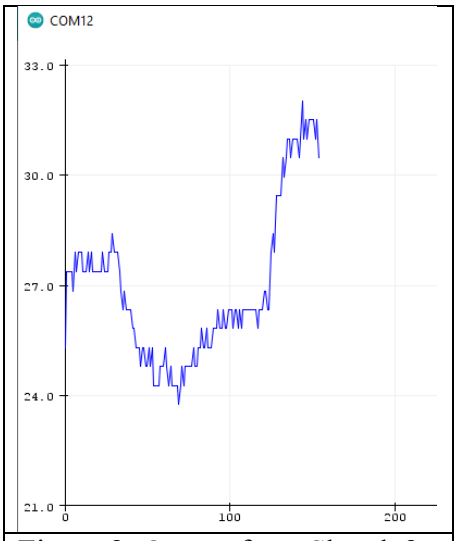


Figure 2. Output from Sketch 2.

Using an external reference voltage

Another solution for correctly interpreting analog voltage inputs is to use the 3.3 V power pin as an "external reference" for converting digitized voltages on any analog input pin back to their analog values. (You can also use any external known voltage source as an external reference, but that requires external hardware.) Instead of relying on the (approximate) default 5 V reference, even when properly defined as shown above, you can instead define a different reference voltage as long as it's less than the nominal 5 V and as long as the input voltage isn't higher than this new reference voltage. This improves the A-to-D resolution a little: 3.3/1023≈3 mV. Many Arduino-compatible analog sensors produce outputs less than 3.3 V, so this is a good solution.

It's easy to define an external reference voltage, but you have to be careful. In your code's `setup()` function, ***BEFORE*** any `analogRead()` statement, include this statement:

```
analogReference(EXTERNAL);
```

Connect the 3.3 V power pin to the AREF pin – see the top row of the UNO image in Figure 1, just to the left of the GND pin.

Now when you interpret analog inputs, assuming the 3.3 V power pin on your board produces exactly 3.3 V (you need to measure it only once), equation (3) becomes

(4) $V_{analog} = 3.3 \bullet V_{digitized}/1023$

An alternative to using Arduino's analog input pins

The above discussion of interpreting analog measurements provides simple approaches to improving the accuracy of A-to-D-to-A voltage conversions with Arduinos. These approaches are best applied to devices that produce output voltages spanning a significant portion of the total available range – 0 to 5 or 0 to 3.3 volts – and for which 10-bit resolution is acceptable.

There are many analog output devices for which neither of these two conditions will be met. In these cases, the solution is to bypass completely the Arduino's on-board analog input processing. An example of such a device is a pyranometer – an instrument that measures total incoming solar radiation on a horizontal plane at Earth's surface. An instrument available from the *Institute for Earth Science, Research, and Education* (www.instesre.org) produces about 0.25 V in response to the roughly 1000 W/m^2 of incoming solar radiation seen in full noontime summer sunlight under clear skies at temperate latitudes. This uses about 0.25/5 or only 5% of the available voltage range for a 5 V board, so an incoming solar radiation of 1000 W/m^2 is divided into only about 50 intervals, with a corresponding resolution of only about 20 W/m^2. Using a 3.3 V external reference voltage improves resolution a little, but still not to a level appropriate for this measurement.

There are several other commercially available pyranometers that produce output voltages of no more than a several tens or a few hundred millivolts, Of course, these voltages can be amplified, but these kinds of instruments are intended to be passive devices without on-board electronic amplification. A much better solution for Arduinos is to use a separate analog input device with much higher A-to-D resolution over a selectable range of input voltages.

The ADS1115 breakout board is a 16-bit, 4-channel, programmable A-to-D converter that bypasses Arduino's analog input pins for interpreting analog voltages. It has a code-selectable voltage range (not the same as an amplified "gain") based on selecting an on-board reference voltage. These devices are available from Adafruit.com and some online sources. They use an I2C communication interface, so they need a software library:

Sketch→Include Library→Manage Libraries.

Type ads1x15 in the search box and install the Adafruit ADS1X15 library. (In addition to the ADS1115 there is also a 12-bit version, ADS1015, which is a little less expensive but generally not worth the small cost differential; both are supported by this library.)

Hookup for the ADS1115 is straightforward. The four analog input channels are labeled A0 through A3; these designations have nothing to do with the analog input pin labels on the Arduino. Connect power, ground, and SDA/SCL for the I2C interface. Connect sensor inputs as needed. Just remember that because the code-selected voltage range applies to all four ADS1115 input channels, all the sensors should have roughly the same range of outputs.

Sketch 3 shows code for using a ADS1115 to read data from a TMP35 or TMP36 analog temperature sensor. A TMP35 produces an output of 0.25 V at 25°C and the TMP36, 0.75 V at that temperature. The datasheet says that the maximum output for either of these devices is 2.0 V, although nowhere near this value will be seen with indoor or outdoor environmental conditions; for reading normal indoor or outdoor temperatures, TMP3x sensors are a good example of an analog output device that covers only a small part of the total voltage range available on Arduino analog input pins. The TMP3x sensors aren't particularly accurate devices (it's reasonable just to read them on analog input pins despite their low voltage output), but they are an easy way to show how to use the ADS1115.

Sketch 3. Using an ADS1115 analog input board.

```
/* ADS1115_TMP36, D. Brooks, January 2019
  Shows how to read 1 channel from ADS1115 breakout.
*/
#include <Wire.h>
#include <Adafruit_ADS1015.h> // This library works for both 1015 and 1115
Adafruit_ADS1115 ads(0x48); // default address is 0x48
float V0,V1,V2,V3,DtoA,T;
```

```
int16_t adc0,adc1,adc2,adc3;  // ADC reading produces 16-bit integer
const int delayTime=500;
void setup(void)
{
  Serial.begin(9600);
  Wire.begin(); ads.begin();
  //ads.setGain(GAIN_ONE);     DtoA=0.125/1000;    // 4.096 V
  ads.setGain(GAIN_TWO);       DtoA=0.0625/1000;   // 2.048 V
  //ads.setGain(GAIN_TWOTHIRDS); DtoA=0.187506/1000; //6.144 V (default
range)
  //ads.setGain(GAIN_FOUR);    DtoA=0.03125/1000;  // 1.024 V
  //ads.setGain(GAIN_EIGHT);   DtoA=0.015625/1000; // 0.512 V
  //ads.setGain(GAIN_SIXTEEN); DtoA=0.007813/1000; // 0.256 V
}
void loop(void)
{
  adc0 = ads.readADC_SingleEnded(0);
  //adc1 = ads.readADC_SingleEnded(1);
  //adc2 = ads.readADC_SingleEnded(2);
  //adc3 = ads.readADC_SingleEnded(3);
  V0 = adc0 * DtoA;
  //V1 = adc1 * DtoA;
  //V2 = adc2 * DtoA;
  //V3 = adc3 * DtoA;
  //T=(V0-0.5)*100.; // for TMP36 sensor
  T=V0*100.; // for TMP35 sensor
  Serial.print("Temperature, *C: ");
  Serial.println(T,2);
  delay(delayTime);
}
```

The ADS1115 input range is selectable and applies to all four channels. In this code I selected the `GAIN_TWO` option to accommodate the maximum possible output from a TMP3x. Even with more accurate analog devices, the 16-bit resolution means that choosing a higher voltage range than will actually be produced by that sensor may not be a problem.

Note that, unlike Arduino analog pins that allow only positive voltage inputs, the input ranges for the ADS115 are "plus or minus." Hence, for converting a digitized voltage using `GAIN_TWO`, $V_{analog}=V_{digitized}\bullet(2\bullet2.048/2^{16}) = V_{digitized}\bullet0.0000625$.

Figure 3 shows some data from Sketch 3 graphed using Serial Plotter. Compared to the output shown with Sketch 2, the much higher 16-bit resolution is clearly evident. This doesn't mean that the temperatures are more accurate, but just that the analog sensor output is more precisely represented. TMP3x sensors have a stated *precision* of 0.1°C, but an *accuracy* of only ±2°C.

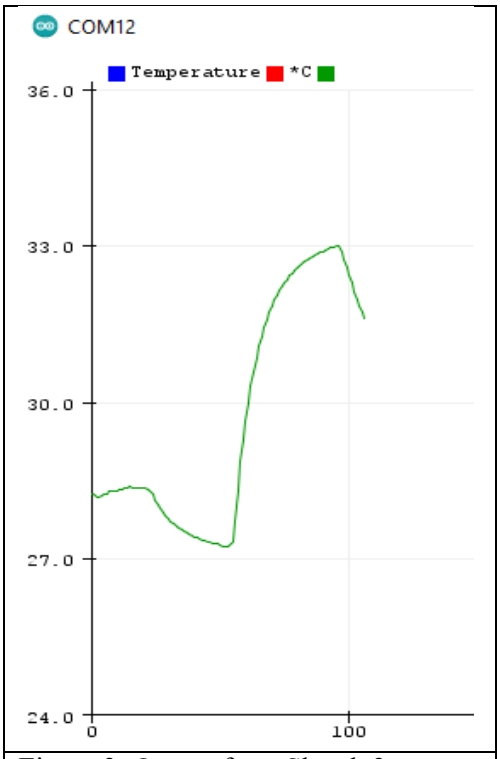
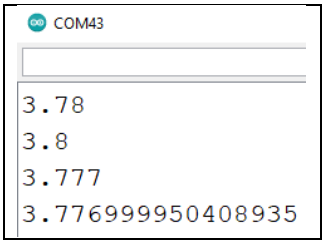


Figure 3. Output from Sketch 3.

The dramatic difference between the outputs displayed in Figures 2 and 3 serves to emphasize the importance of properly interpreting data from analog sensors. The high precision of the ADS1115's 16-bit A-to-D resolution compared to the Arduino's 10-bit resolution can make graphs "look better," but it may have nothing to do with the accuracy of the sensor. The temperatures reported in Figure 2 change in steps of about 0.5°C, but even that coarseness misrepresents the actual sensor accuracy; it would be fairer to report data from TMP3x sensors rounded to the nearest degree.

In general, when reporting data from analog sensors, it's always important to display those data with an implied accuracy that's consistent with the sensor and the A-to-D-to-A resolution. When using `Serial.print()` (and similar statements for some other display devices), significant digits can be controlled. Consider this code fragment:

```
x=3.777;
Serial.println(x);
Serial.println(x,1);
Serial.println(x,3);
Serial.println(x,15);
```



Included in a working sketch, these statements will display the values shown here. The default for real (floating point) numbers in Arduino programming is for `Serial.print()` to display two digits to the right of the decimal point. Other choices will properly round the displayed result. For real numbers in Arduino programming, 6 or 7 is the maximum *total* number of digits that have any significance (see https://www.arduino.cc/reference/en/language/variables/data-types/float/. Displaying more digits than that (not just digits to the right of the decimal point), whether for very large or very small numbers, is meaningless and produces just "junk" digits. Unlike some other programming languages, the Arduino language doesn't have a "double precision" option for floating point numbers.

Conclusions

The basic code for using analog input pins often found in beginning tutorials for Arduino programming – see equation (1) – is OK for (very) casual use, but proper treatment of analog-output sensors requires more care. For devices with voltage output values small compared to the entire input range, or in any situation where Arduino's 10-bit A-to-D resolution isn't adequate, additional hardware like the ADS1115 provides an inexpensive solution.