Episode 2.6 – Analog to Digital Conversion with Arduino

Welcome to the Geek Author series on Computer Organization and Design Fundamentals. I'm David Tarnoff, and in this series we are working our way through the topics of Computer Organization, Computer Architecture, Digital Design, and Embedded System Design. If you're interested in the inner workings of a computer, then you're in the right place. The only background you'll need for this series is an understanding of integer math, and if possible, a little experience with a programming language such as Java.

During our last episode, we introduced the steps through which an analog measurement is converted to binary. Simply put, we are mapping a sequence of binary integers to a range of analog values. To do this we needed three things: the bottom and the top of the analog range we want to represent and the number of bits we're going to use to digitally represent the analog value, also referred to as bit depth. Although we covered a lot in that discussion, there were two elements of our system that we overlooked. The first is a device called a sensor. A sensor converts a physical phenomenon such as temperature, sound waves, or force, into an electrical signal. Typically, this electrical signal is a voltage level. If you're not familiar with voltage, think of it as the pressure behind the electrons in a conductor. The stronger the pressure, the higher the voltage. No pressure, no voltage.

This voltage is sent to the second piece that was missing from our discussion, a device called an analog to digital converter, or ADC. The ADC is what maps the voltage level to an integer.
Is this a bit much to take in? Well maybe, but as programmers, our responsibility is to interpret the integer sent to us from the ADC. So let's take a look at that conversion process with a real-world application.

In Episode 1.4, we introduced the Arduino line of low-cost circuit boards used by anyone who wants to create programmable devices that interact with users and the environment.[1] In that episode, we showed how a simple function called analogWrite() could be used to generate an analog output by adjusting the duty cycle of a series of digital pulses. Now let's see how Arduino reads analog values. The typical Arduino board has a set of conductive pins along the edge of the board, some of which can be used as analog inputs. An on-board ADC will provide programmers with the digital value present at an analog input. So how do we access this ADC? As with the analog output, we need to use a couple of software functions.

First, let's take a look at the Arduino function analogReference().[2] This function is used to identify the voltage level that will be used to represent the upper limit of our analog range, in other words, the voltage that will be mapped to the binary value of all ones. By default, the voltage level used to power the Arduino board itself is used as the upper voltage level. Using the analogReference() function, however, we can change the level used as upper limit to an internal voltage of 1.1 volts or 2.56 volts. If you want to have more control over the circuit, you can connect an external voltage ranging from 0 to 5 volts and set that as the maximum voltage. Whichever limit you choose, remember that any voltage equal to or greater than the one you select will be converted by the ADC to a digital value of all ones.
So what is the minimum value, in other words, the voltage that will be represented by the binary value of all zeros? Well, the typical Arduino is hard-wired to use zero volts for the lower limit.
So that takes care of two of the three things we need. What about bit depth? How many bits will we use to represent our analog value? Most Arduino boards fix their bit depth to 10 bits. That means that the

values read from the ADC can take on one of 2^{10} or 1024 different values. Some versions of the Arduino, however, allow the programmer to modify the bit depth using the function analogReadResolution().[3] These boards have a 12-bit ADC which means they are capable of dividing the analog range into 2^{12} or 4096 different levels. The parameter sent to the function analogReadResolution() must be an integer between 1 and 32. This will tell the ADC how many bits to return for each analog measurement.

Lastly, the function analogRead() is used to capture an integer value from the ADC with the correct bit depth.[4] To show how analogRead() is used, let's take a look at a couple of examples.
Certain types of game controller joysticks have connections for two analog signals: one that identifies the position of the joystick in the x-direction and one that identifies its position in the y-direction. That means our code will be reading two different digital values from two different analog input pins. Let's assume that the joystick pin with the analog level for the x-direction is connected to pin A0 on the Arduino and the pin with the analog level for the y-direction is connected to pin A1. Assume also that the bit depth is set to 10.

If the command analogRead(A0) returns the value 511, then we know that the joystick is sitting at the half-way position in the x-direction since 511 is half-way up the range from 0 to 1023. If you were steering a car with this joystick, it would mean that we should be going straight.
Now assume that the command analogRead(A1) returns the value 921. Dividing 921 by 1023 gives us a value of about 0.9 or 90%. This means someone has pushed forward on the joystick so that it is 90% of the way to full throttle.

It turns out that there's a formula we can use to convert the digital value from an ADC to the analog value it is meant to represent. You got a flavor of it from our previous example. If you take the digital value from the ADC and divide it by the highest digital value the ADC can represent, in other word $2^n - 1$ where n is the bit depth, you get the percentage of the way that the digital value is from the minimum on its way to the maximum. For example, 511 is halfway from 0 to 1023 and 921 is 90% of the way from 0 to 1023. If you multiply this percentage by the size of the analog range, you can determine how far we are from the minimum analog value on the way towards the maximum. If the minimum analog value is not zero, then we will also need to add the minimum value to the result so as to bias our analog result to the correct range.

Okay, that description probably needs an example. Let's say that our Arduino is measuring the temperature of an oven with an ADC using a bit depth of twelve. We will use a minimum temperature of 200 degrees Fahrenheit and a maximum temperature of 500 degrees Fahrenheit. What temperature is our oven at if our ADC returns a value of 3071? Well, our analog range is 500 – 200 or 300 degrees, right? If we divide 3071 by the upper limit of our 12-bit digital value, $2^{12} - 1$ or 4095, we get about 0.75. That means our input is 75% of the way from 0 to 4095. Multiply this by our range of 300 degrees, and we get about 225 degrees. Does this mean our oven is running at 225 degrees? Nope. We forgot that our reading is actually a measurement from the bottom of our range at 200 degrees. Add 200 degrees to 225 degrees and we see that our oven is at 425 degrees Fahrenheit.

In the next episode, we will see what happens when we read a sequence of these analog measurements for applications such as audio recording. Until then remember that while the scope of what makes a computer is immense, it's all just ones and zeros.

References:

1 – Arduino. (n.d.). *Getting Started | Foundation > Introduction*. Retrieved April 09, 2019, from Arduino: https://www.arduino.cc/en/guide/introduction
2 – Arduino. (n.d.). *Reference > Language > Functions > Analog io > Analogreference*. Retrieved April 09, 2019, from Arduino: https://www.arduino.cc/reference/en/language/functions/analog-io/analogreference/
3 – Arduino. (n.d.). *Reference > Language > Functions > Zero due mkr family > Analogreadresolution*. Retrieved April 09, 2019, from Arduino: https://www.arduino.cc/reference/en/language/functions/zero-due-mkr-family/analogreadresolution/
4 – Arduino. (n.d.). *Reference > Language > Functions > Analog io > Analogread*. Retrieved April 09, 2019, from Arduino: https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/