Temperature Sensing with an Arduino Microcontroller

[redacted]

***Abstract*—An Arduino microcontroller was used to record analog temperature data from an LM61 temperature sensor. This was preformed using the microcontroller's built in 10-bit ADC at a fixed sample rate. Sample rates were improved by setting up an external timer with hardware interrupts.**

I. Introduction

THE Arduino microcontroller is a simple and effective tool for performing low level tasks in either analog or digital. Here an Arduino was used to record data from an LM61 temperature sensor. The LM61 is an analog device which produces a voltage that is linearly relational to its die temperature as seen from its transfer function

$$V_o = 10\,\text{mV}/°\text{C} \times \text{T}°\text{C} + 600\,\text{mV} \quad (1)$$

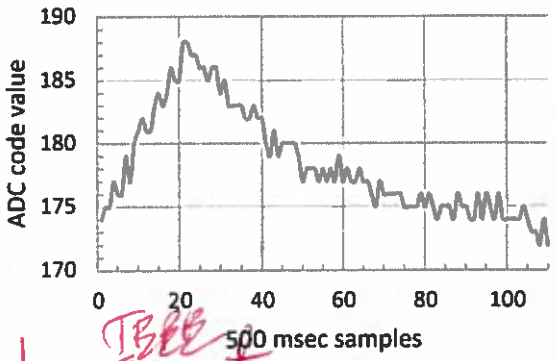
The Arduino's ADC is 10-bits and can produce code values from 0 – 1023. The voltage reference for the ADC was 0 – 5 volts. The voltage step for a code value was approximately 4.88 millivolts. Using the LM61's transfer function (1) we can calculate that the Arduino can sense 0.488°C per code value.

Data processing for this report was performed by printing out sample data through a serial connection to a computer. This data was loaded into Microsoft Excel for graphing.

The intermittent temperature source used for the procedures was a touch of a finger. The sensor was left at room temperature to stabilize before data collection began. Shortly after connecting the Arduino to power and a serial terminal the LM61 was touched to raise its temperature.

II. Results

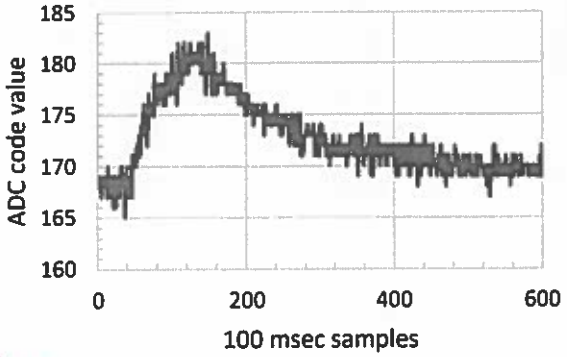
Several sets of temperature data were recorded. The first set of data, P1-1, was captured using a 500-millisecond sampling rate. This was done with a fixed delay in the sensor polling loop. For the second set of data, P1-2, the Arduino was configured to use an external timer. This timer interrupted the Arduino every 100 milliseconds to ensure a more accurate sample rate. The third set of data, P1-3, was run with the same sample rate as the second set, however only 256 samples were recorded. The fourth set of data, P1-4, was captured with extra code to calculate a running mean and variance. An additional set of data, P1-5, was captured without a touch to record background conditions.



P1-1. LM61 response and recovery from a touch. Data was recorded over approximately 60 seconds with a 500-millisecond fixed delay for sample timing. An ADC voltage reference of 5v was used.

III. Analysis

Mean and standard deviation are useful for statistical analyses. Calculations for these values rely on the summation of all samples and the summation of the squares of samples. The sample recording loop was modified to keep a running total of



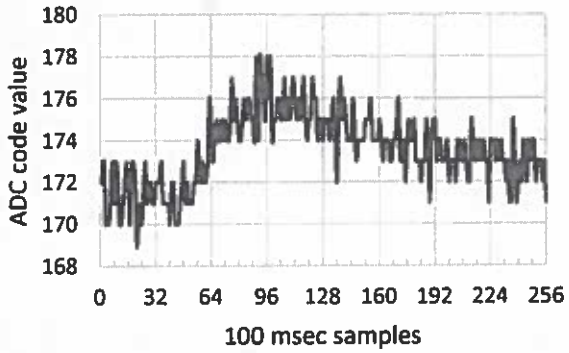
P1-2. LM61 response and recovery from a touch. Data was recorded over approximately 60 seconds with an external 100-millisecond interrupt for sample timing. An ADC voltage reference of 5v was used.

sample values and their squares. With these additional variables recalculating the mean and standard deviation are trivial and were performed and printed out for each sample. Fig. 1 shows the additional code required to carry out these calculations.

The running mean and standard deviation follow the rise in

```
... Globals ...
double total = 0, totalSquared = 0;

void loop() {
  ... loop code ...

  // Running mean and stdev
  total += sample;
  totalSquared += sample*sample;

  double mean = total/(double) nSmpl;
  double stdev = 0;
  if (nSmpl > 1)
    stdev = sqrt((totalSquare - ((total*total)/(double)
nSmpl))/(nSmpl-1.0));

  Serial.print(mean); Serial.print('\t');
  Serial.print(stdev); Serial.print("\r\n");

  ... rest of loop ...
}
```

Fig. 1. Additional Arduino code to print out running mean and standard deviation.



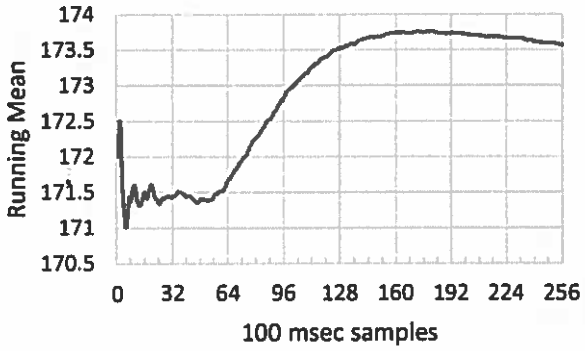
P1-4-1. LM61 response and recovery from a touch. Data collection was limited to 256 samples with an external 100-millisecond interrupt for sample timing. An ADC voltage reference of 5v was used.

temperature from the touch. The standard deviation has a sharper response and falls faster than the mean. This is due to how standard deviation is calculated. Standard deviation is the square root of the mean of the variances of the data. Variance is calculated as the square root of the difference of each sample from the mean of the samples. When the sensor is touched, value of the samples increases rapidly. The mean lags behind this spike. This difference between the mean and these new sample is relatively large and the square can be much larger. The result is that the standard deviation changes faster than the mean.

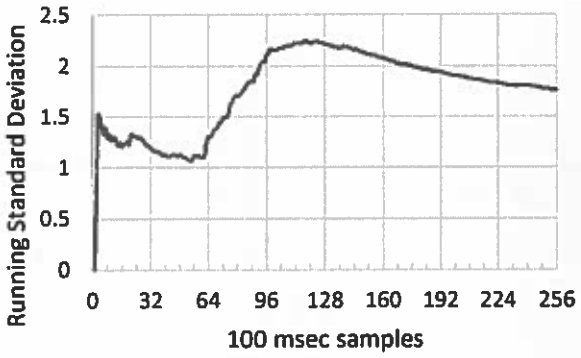
very good.

IV. Conclusion

The Arduino paired with the LM61 recorded a noticeable temperature change from an intermittent touch. This was



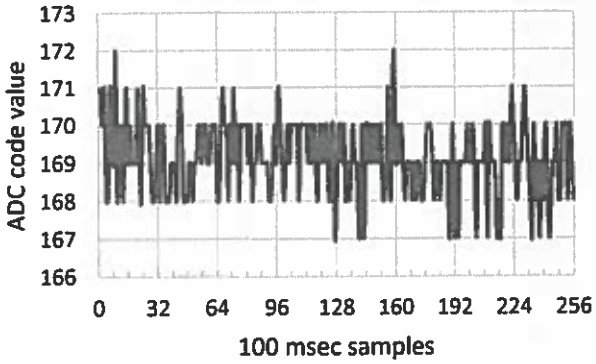
P1-4-2. Running mean of the ADC code values from P1-4-1.



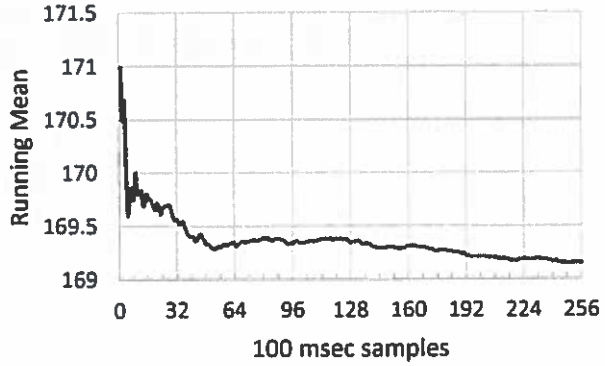
P1-4-3. Running standard deviation of the ADC code values from P1-4-1.

possible at the slower sample rate of 500-milliseconds, at the faster rate sample rate of 100-milliseconds, and when constrained to only 256 samples. The rise and fall in sensor temperature are clearly visible in all cases.

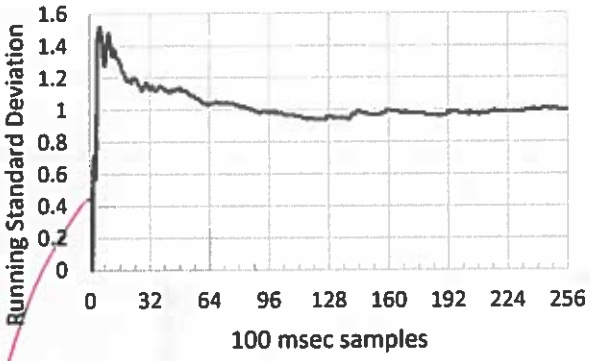
The code for external timing uses interrupts and flags. A flag, `sampleFlag`, is set true in the interrupt subroutine to signal an interrupt has happened. The main loop waits for the `sampleFlag` then sets it to false signifying that a sample has been taken and another interrupt is required to take another.



P1-5-1. LM61 at room temperature. Data collected was limited to 256 samples with an external 100-milisecond interrupt used for sample timing. An ADC reference voltage of 5v was used.



P1-5-2. Running mean of ADC code values from P1-5-1.



P1-5-3. Running mean of ADC code values from P1-5-1.

Comments?

Does this make sense?

Yes. The running mean reflects a slow downward drift in the data set

The running σ converges to ≈ 1. The code values are chattering ± 1 at fairly high density. Rearranging them, one could make a fairly decent square wave of $2\,CV_{pk\text{-}pk}$, which would have $\sigma = 1\,CV$.

The initial σ spike is due to the mean being initialized to zero. The first data point near 170 has a huge $(x_1 - \bar{x})^2$ contribution. Code could be improved by waiting a couple of ticks to get a mean estimate before enabling σ calculation