

Lab 3: Arduino Temperature Sensor

Argenis Aquino, Rachel DuBois, Diego Lopez, Jonathan Sumner Department of Engineering Technology,
Rochester Institute of Technology
1 Lomb Memorial Drive, Rochester NY, 14623, USA

Abstract—This document demonstrates how an Arduino Uno was used to record analog temperature data by connecting it to an LM61 temperature sensor. Data collection was optimized by leveraging the microcontroller’s 10-bit ADC along with the MsTimer2 library and hardware interrupts.

I. INTRODUCTION

ARDUINOS are open-source microcontrollers that provide a beginner-friendly introduction to circuit and firmware design. In this lab, the Arduino was used with an LM61 precision IC temperature sensor to collect temperature samples and plot them using MATLAB. The LM61 has a temperature range from -30°C to $+100^{\circ}\text{C}$. The datasheet provides the transfer function used to determine the temperature:

$$V_0 = (10 \text{ mV}/^{\circ}\text{C} \times T^{\circ}\text{C}) + 600 \text{ mV} \quad (1)$$

II. METHODOLOGY

For P1-1 a delay was hard-coded in order to collect samples at a rate of 500 millisecond.

```
1  int nSmpl = 1, sample; // global variables
2  void setup()
3  {
4      Serial.begin(9600); // Set baudrate to 9600
5      Serial.print("\nsmpl\tADC\n"); // column
6      headers
7  }
8  void loop()
9  {
10     sample = analogRead(A0); // Read ADC value on
11     pin A0
12     Serial.print(nSmpl);      // Print sample number
13     Serial.print('\t');       // Print tab
14     Serial.println(sample);    // Print sample value
15     ++nSmpl;                  // Increment sample
16     number
17     delay(500);                // Delay of 500
18     milliseconds
19 }
```

P1-2 data was collected with a few changes. Here the sampling rate was changed to be a rate 1 sample per 100 milliseconds. Next a timer interrupt was integrated with the MsTimer2 Arduino library.

```
1  #include <MsTimer2.h>
2
3  const int TSAMP_MSEC = 100;
4  volatile boolean sampleFlag = false;
5  int nSmpl = 1, sample;
6
7  void setup() {
8      Serial.begin(9600);
9      MsTimer2::set(TSAMP_MSEC, ISR_Sample); // Set
10     sample msec, ISR name
11     MsTimer2::start(); // start running the Timer
```

```
11 }
12
13 void loop() {
14     while (sampleFlag == false); // spin until ISR
15     trigger
16     sampleFlag = false; // disarm flag: enable
17     next dwell
18
19     sample = analogRead(A0);
20     // Display results to console
21     if (nSmpl == 1) Serial.print("\nsmpl\tADC\n");
22     Serial.println(String(nSmpl) + '\t' + String(
23     sample));
24     ++nSmpl;
25 } // loop()
26
27 void ISR_Sample() {
28     sampleFlag = true;
29 }
```

The P1-3 code setup() function was edited so that instead of the sample collecting beginning immediately after the microcontroller being flashed it waited for an input in the terminal. This allowed for the user to control the beginning of the data collection. Another change added was that a max amount of samples collected of 256 was set. By taking the ++nSmpl; and putting it in an if statement that compares its current value with the value of max samples it can be determined whether the max sample count has been reached or not. If the max count has been reached the ISR is not called and the program is infinitely stuck in the while (sampleFlag == false); loop.

```
1  #include <MsTimer2.h>
2
3  const int TSAMP_MSEC = 100;
4  volatile boolean sampleFlag = false;
5  int nSmpl = 1, sample;
6  const int NUM_SAMPLES = 256;
7
8  void setup()
9  {
10     Serial.begin(9600);
11     Serial.println("Enter 'g' to go .....");
12     while (Serial.read() != 'g'); // spin until 'g'
13     entry
14     MsTimer2::set(TSAMP_MSEC, ISR_Sample); // Set
15     sample msec, ISR name
16     MsTimer2::start(); // start running the Timer
17 }
18
19 void loop()
20 {
21     while (sampleFlag == false); // spin until ISR
22     trigger
23     sampleFlag = false; // disarm flag: enable
24     next dwell
25
26     sample = analogRead(A0);
27     // Display results to console
28     if (nSmpl == 1) Serial.print("\nsmpl\tADC\n");
```

```

25 Serial.println(String(nSmpl) + '\t' + String(
    sample));
26
27 if (++nSmpl > NUM_SAMPLES) MsTimer2::stop();
28 } // loop()
29
30 void ISR_Sample()
31 {
32     sampleFlag = true;
33 }

```

For P1-4 Matlab was used in unison with the Arduino IDE to collect and plot the IC data. The Arduino code sends a string “%Arduino Ready” on the terminal that Matlab will recognize and then send a ‘g’ in response. The Matlab scripts let us automate the process of recording the data, plotting it, and saving the data collected into a .mat file.

```

1 #include <MsTimer2.h>
2
3 const int TSAMP_MSEC = 100;
4 volatile boolean sampleFlag = false;
5 int nSmpl = 1, sample;
6 const int NUM_SAMPLES = 256;
7
8 void setup()
9 {
10     Serial.begin(9600);
11     Serial.println("%Arduino Ready"); // Send the
    ready string
12     Serial.println("Enter 'g' to go .....");
13     while (Serial.read() != 'g'); // spin until 'g'
    entry
14     MsTimer2::set(TSAMP_MSEC, ISR_Sample); // Set
    sample msec, ISR name
15     MsTimer2::start(); // start running the Timer
16 }
17
18 void loop()
19 {
20     while (sampleFlag == false); // spin until ISR
    trigger
21     sampleFlag = false; // disarm flag: enable
    next dwell
22
23     sample = analogRead(A0);
24     // Display results to console
25     if (nSmpl == 1) Serial.print("\nsmpl\tADC\n");
26     Serial.println(String(nSmpl) + '\t' + String(
    sample));
27
28     if (++nSmpl > NUM_SAMPLES) MsTimer2::stop();
29 } // loop()
30
31 void ISR_Sample()
32 {
33     sampleFlag = true;
34 }

```

P1-5 added the calculation of the mean and standard deviation of the samples collected. The Arduino code would calculate the mean and standard deviation. The standard deviation is calculated using this equation.

$$s^2 = \frac{1}{n(n-1)} \left(n \sum_{i=1}^n x_i^2 - \left(\sum_{i=1}^n x_i \right)^2 \right) \quad (2)$$

In the C code was written as

$$\text{runningMean} = \text{oldMean} + \frac{x_i - \text{oldMean}}{\text{numSamples}}$$

$$\text{runningVar} = \text{oldRunningSumVar} + (x_i - \text{oldMean}) \cdot (x_i - \text{runningMean})$$

$$\text{variance} = \frac{\text{runningVar}}{\text{tick} - 1}$$

```

1 // OPEN NEW ARDUINO SKETCH.
2 // CLICK IN THIS TEXT BOX. CTRL-A, CTRL-C.
3 // CLICK IN SKETCH. CTRL-A, CTRL-V.
4
5 // Lab 3 starter: Cook stats
6
7 const int TOTAL_SAMPLES = 600; // simulated
    samples
8 int numSamples = 1;
9
10 // Declare the variables that are computed in
    the calculateStats function
11 float sample, runningMean = 0.0, runningVar =
    0.0, stdev = 0.0, variance=0.0;
12
13 void setup()
14 {
15     Serial.begin(9600);
16
17     // This line tells MATLAB that the Arduino is
    ready to accept data
18     Serial.println("%Arduino Ready");
19
20     // Wait until MATLAB sends a 'g' to start
    sending data
21     while (Serial.read() != 'g'); // spin until 'g'
    entry
22
23 } // setup
24
25 void loop()
26 {
27     sample = simSample();
28
29     // Call the statistics calculation function
30     calculateStats(sample);
31
32     // Display the statistics
33     displayStatsData();
34
35     // Run TOTAL_SAMPLES iterations then halt
36     if (++numSamples > TOTAL_SAMPLES) while (true)
37     ;
38 } // loop()
39
40 float simSample(void)
41 {
42     // Simulate sensor for stats calculation
    development
43     float simSmpl, simAmp = 1.0, simT = 60;
44
45     simAmp = ((numSamples > 180) && (numSamples <
    300)) ? 0.125 : 1.0; // burst amplitude
46     // simT = ((numSamples > 180) && (numSamples <
    300)) ? 30.0 : 60.0; // burst frequency
47     simSmpl = 180.0 + simAmp*sin((numSamples/simT)
    *TWO_PI); // fixed amplitude, frequency
48
49     return simSmpl;
50 } // simSample
51
52 void calculateStats(float xi)
53 {
54     // Calculate running statistics per Cook
    pseudo code.
55     static int tick = 1;
56     static float oldMean, oldRunningSumVar;

```


Statistical analysis was performed to examine fluctuations in the recorded temperature values. Key metrics such as the mean and standard deviation were computed for different scenarios.

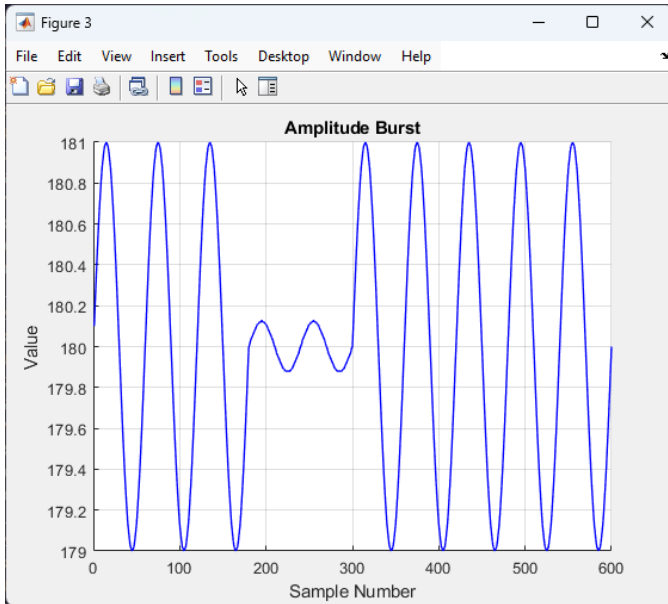


Fig. 5: Amplitude Burst Plot

This test introduced an amplitude burst in the data, simulating abrupt temperature changes. The results show clear deviations in the readings, as illustrated below.

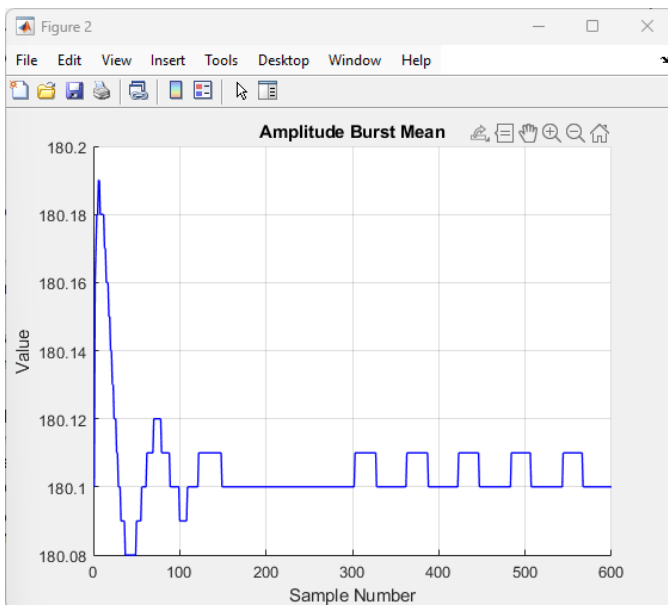


Fig. 6: Amplitude Burst Mean

The calculated mean of the amplitude burst dataset provides insight into the overall temperature shift

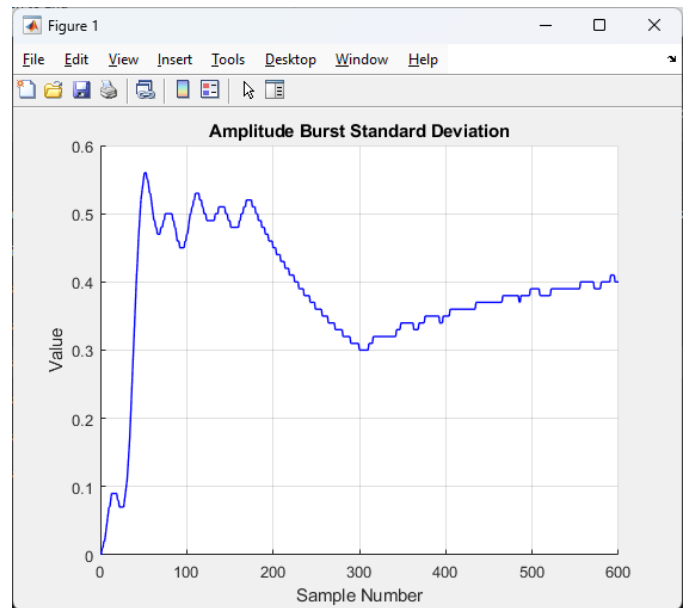


Fig. 7: Amplitude Burst Standard Deviation

The standard deviation plot highlights increased variability during the burst phase.

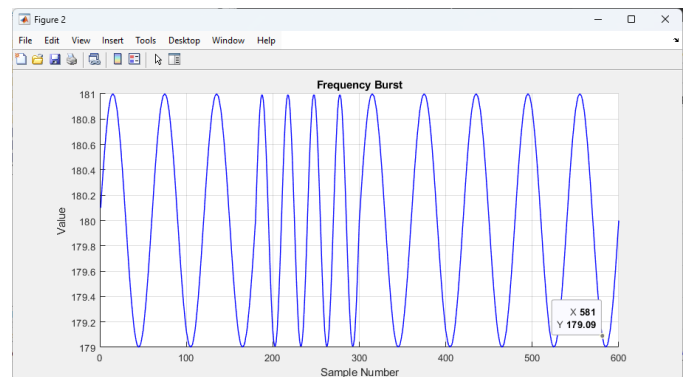


Fig. 8: Frequency Burst Plot

In this case, a frequency burst was introduced, altering the rate of temperature fluctuations. The resulting data captures rapid changes in sensor readings.

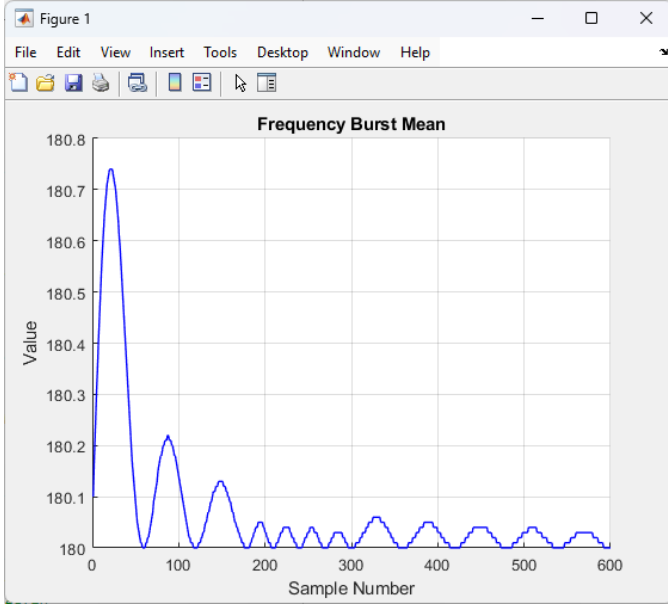


Fig. 9: Frequency Burst Mean

The mean temperature over the frequency burst interval is shown below.

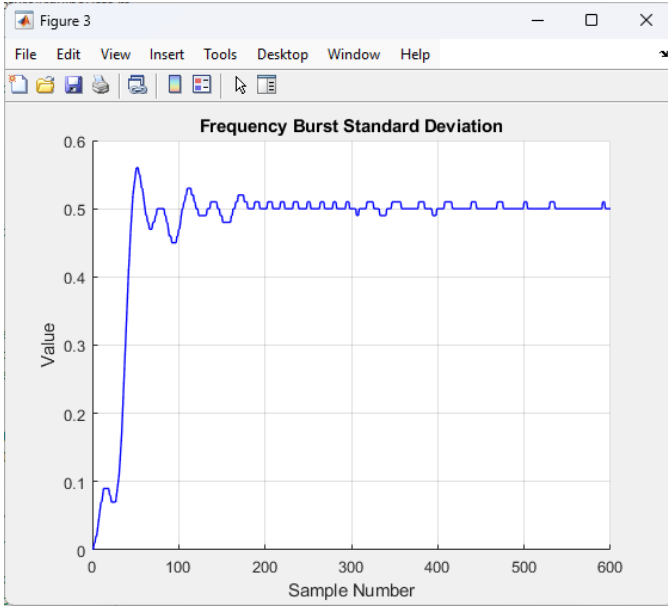


Fig. 10: Frequency Burst Standard Deviation

The standard deviation plot indicates increased variation due to the burst.

IV. ANALYSIS

The analysis of mean and standard deviation in statistics is crucial, as these variables are indicators of the procedure's success. For P1-5 the Arduino code was edited to calculate the mean and standard deviation of the samples.

The standard deviation is more responsive to temperature variations than the mean because it captures the extent of changes in the data, where the mean only provides a central value. The formula for standard deviation accounts for the

squared differences between each data point and the mean, making it more sensitive to larger deviations. In contrast, the mean smooths out variations by averaging all values, reducing the impact of extreme temperature changes. For instance, if temperature readings fluctuate significantly over time, the standard deviation will increase, reflecting the variation, while the mean may remain relatively stable. This makes standard deviation a better indicator of dynamic temperature changes.

V. CONCLUSION

The experimental setup successfully demonstrated an optimized method for temperature data collection using an Arduino and IC system. By progressively improving data recording techniques, the lab highlighted the advantages of interrupt-based sampling, user-controlled initiation, and MATLAB integration for enhanced data analysis. The statistical evaluation reinforced the reliability of the collected data and provided deeper insights into temperature changes.

Future work could explore additional enhancements such as adaptive sampling rates, machine learning-based anomaly detection, and integration with wireless communication for remote data monitoring. These improvements would further increase the system's versatility and applicability in real-world temperature monitoring scenarios.