Experiment #2

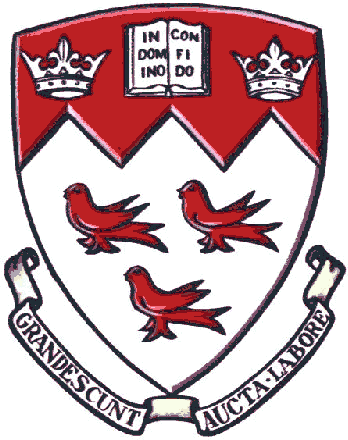
Sensor Data Acquisition, Digitizing, Filtering, and Digital I/O

Maxim Goukhshtein ID: 260426769

Olivier Laforest ID:260469066

Department of Electrical and Computer Engineering

McGill University, Montreal



February 21st, 2015

ECSE 426 Microprocessor Systems

# Abstract

The goal of the experiment presented in this report is to implement a temperature data acquisition system using the STM32F407 Discovery board and display the acquired data through the use of the on board Light Emitting Diodes (LEDs) in order to create a simple output display. This report will show how the built-in temperature sensor of the STM32F407 Discovery board, as well as the analogue to digital converter (ADC) and LEDs were used to implement the desired system. It will also be shown how filtering of the raw data was done and how pulse width modulation (PWM) was utilized in order to realize the desired display effects.

# Problem Statement

In this experiment, the internal processor temperature sensor of the STM32F407 Discovery board is used to get temperature readings of the microprocessor that will be converted into a visual LED display in order to let the user know if the temperature is increasing, decreasing or if the temperature has reach an upper threshold. The display is to be created using the LEDs that are positioned in a diamond shape on the board (i.e. LED 3 to LED6). While in normal operation (i.e. below the upper threshold), only one LED should be on at any one time. For each increase of 2 degrees Celsius, the display should cycle through the four LED lights in a clockwise fashion. In other word, if LED3 is currently lit, after an additional increase of 8 degrees Celsius, the display should have cycled through the four LEDs and LED3 should be lit again. For every decrease of 2 degrees Celsius, the display should cycle through the four LEDs in a counter clockwise manner. If the temperature of the microcontroller exceeds an upper temperature threshold, the display should enter an overheating alarm mode. The alarm mode consists in the four LEDs simultaneously flashing in a fade-in/fade-out manner. When the temperature falls back under the threshold, the alarm mode should be exited and normal mode should resume. Several challenges are associated with the LED display. While in normal operation, the transitions between LEDs should be as definitive as possible (i.e. the LED should ideally not flicker back and forth during a transition from one LED to the next). While in the alarm mode, all four LEDs must smoothly fade-in and fade-out from all the way off to fully on in a cyclic manner. In alarm mode, the LEDs should not be flickering on and off.

# Theory and Hypothesis

According to [1], the temperature sensor of the microcontroller on the STM32F407 Discovery board is an analogue sensor which outputs a voltage that varies linearly with temperature that ranges between 1.8V to 3.6V. The temperature sensor is internally connected to an Analogue to Digital Converter (ADC) which allows the analogue sensor readings to be converted to digital values. These digitized temperatures readings can then be used in an embedded C program to implement the LED output display. From the observation of the raw data of the digitized temperature readings outputted by the ADC shown figure 1 and from the fact that sensor data is prone to noise [2], it can be seen that the temperature sensor output is significantly affected by noise. It can be seen in figure 1 that raw data samples can vary by more than within only a few samples. Filtering of the raw data is therefore necessary.

Figure 1: Digitized temperature sensor readings outputted from the ADC

From the experiment 2 specifications [2], it is required to use a Kalman filter in order to filter the raw data. The Kalman filter is an optimal estimator for one-dimensional linear systems with Gaussian noise [3]. It is typically used to smooth noisy data and supply estimates of the filter's parameters [3]. From experiment 1 specifications [4], the parameters for the Kalman filter are the following:

|  |  |
| --- | --- |
|  | (1) |
|  | (2) |
|  | (3) |
|  | (4) |
|  | (5) |

These parameters need to be initialized to appropriate values before the filter can be used. To get a better insight into how to initialize some of these parameters it is useful to take a look at the equations governing each update of the Kalman filter which is performed every time a new measurement of the temperature is outputted from the ADC.

|  |  |
| --- | --- |
|  | (6) |
|  | (7) |
|  | (8) |
|  | (9) |

From a quick inspection of (6), (7), (8) and (9), it can be seen that the only 2 independent parameters (i.e. the parameters that are not changed by the updating process) are the process noise covariance (1) and the measurement noise covariance (2), therefore these parameters will have the most impact. From experimentation, the estimation error covariance (4) converges after only a few updates and its initial value is irrelevant. From equation (7), it can be seen that the kalman gain's initial value is not important since as soon as the first update is performed, the parameter will be set to its appropriate filtering value as it only depends on the estimation error covariance (4), which as mentioned above converges after a few updates and the measurement noise covariance (2) which is constant. The parameter (3) is the actual filtered measurement value and its initial value can be set to the first temperature measurement produced out of the ADC. Therefore, the two parameters which have an important impact on the filtered data are the measurement noise covariance (2) and the process noise covariance (1). From the documentation of the Discovery board [5], it can be found that the precision of the temperature sensor is and consequently the measurement noise covariance will be equal to,

|  |  |
| --- | --- |
|  | (10) |

The only parameter left to set is the process noise covariance (1) and since all the other degrees of freedom of the kalman filter have been fixed or are irrelevant and finding the process noise covariance of a sensor is not trivial, the appropriate value for (1) can be set by trial and error on a set of raw data. The value of (1) can be varied until the filtered data exhibit the desired characteristics (i.e. the filtered data follows the local average values of the unfiltered data, and the filtered data's variation from one sample to the next remains small enough for the proper operation of the LED display, otherwise the LEDs are going to flicker back and forth when the temperature approaches a transition value).

# Implementation

The process of implementation of the solution for the lab, can be divided into roughly 5 parts: setup and initialization of relevant components, sensor data acquisition, conversion of acquired raw data to suitable format, data filtering and implementation of routines for the 2 required modes of operation.

## Component setup and initialization

In the course of this lab, we used 3 components: the analog-to-digital converter (ADC), the GPIO used for the LEDs and the SysTick timer. The use of these component requires their initialization and/or setup.

**ADC**: The ADC is used to convert the sampled analog voltage data from the processor’s temperature sensor into a digital value (which is later to be converted into a temperature value). Initialization of the ADC involves the enabling of the high-speed bus (APB2) clock for the ADC1.

Afterwards, the common ADC structure, *ADC\_CommonInitTypeDef*, must be initialized. As we’re interested in sampling from a single channel, the ADC mode was set to independent mode (i.e. *ADC\_Mode\_Independent*) and the unneeded DMA access was disabled (i.e. *ADC\_DMAAccessMode\_Disabled*). Furthermore, the frequency of the clock the ADC was set to a division by 2 (i.e. *ADC\_Prescaler\_Div2*), which was deemed adequate for the purpose of this experiment. Finally, the delay between 2 subsequent sampling phases was set to 5 clock cycles (i.e. *ADC\_TwoSamplingDelay\_5Cycles*), which is the smallest possible delay, thus allowing for fast data acquisition (however, since 12-bits resolution was used, this parameter could have also been set to a higher delay).

Afterwards, the specific ADC structure, *ADC\_InitTypeDef*, was initialized. We set it to use the highest available bit resolution of 12 bits (i.e. ADC\_Resolution\_12b), to achieve the highest temperature resolution possible. As we were using the single mode, the scan and continuous conversion modes were disabled. As we didn’t utilize an external trigger for out samples conversion, the field was set to none (i.e. *ADC\_ExternalTrigConvEdge\_None*). We set the data to be aligned to the right (i.e. *ADC\_DataAlign\_Right*). Finally, the number of conversions was set to 1, as we’re only interested in a single conversion at time.

Once the common and specific ADC structures were initialized, the ADC1 module was enabled along with the temperature sensor, followed by the setting up of the temperature sensor channel 16 (also given by the macro *ADC\_Channel\_TempSensor*) and the ADC.

**GPIO:** In order to use the LEDs it is essential to configure the relevant GPIO. Based on the board manual GPIOD can be used to control the 4 LEDs. In order to configure the GPIO, we first enable the AHB1 peripheral clock. Then, the GPIO structure was initialized. In particular, the pins were set to the ORring of the GPIO pins 12-15, which correspond to the 4 LED pins. Naturally, mode of operation was set to the output mode (i.e. *GPIO\_Mode\_OUT*), and the speed was set to 50MHz (i.e. *GPIO\_Speed\_50MHz*), which was deemed sufficient for the task at hand. The output type was set to push-pull (i.e. *GPIO\_OType\_PP*) and as we don’t have an input the operation mode for the pulling was disabled (i.e. *GPIO\_PuPd\_NOPULL*).

**SysTick:** The SysTick clock which was used for generating the sampling frequency was setup to the required 50Hz, by passing the ratio of the 168MHz / 50Hz to the SysTick configuration function. Internally, this ratio is used to generate an interrupt (calling the SysTick handler) every ratio clock cycles of the system clock.

Data acquisition

The temperature sensor data was acquired using the ADC1. Every time, after the SysTick handler was called, we would call the *ADC\_SoftwareStartConv()* function on the ADC1 and subsequently wait until the end-of-conversion flag (i.e. *ADC\_FLAG\_EOC*) was set (this indicating that the sample conversion has been done). We then reset the end-of-conversion flag (to allow for the next conversion to take place later) and retrieved the converted value using the *ADC\_GetConversionValue()* function.

Conversion from voltage to Celsius

The acquired data from the sensor was converted from voltage to Celsius using the formula presented in the board manual as was described in the theory section. As the acquired data has a 12-bit resolution, it was necessary first to normalize it be dividing the obtained data by . This value was then mapped into the 0 – 3 V range of the sensor (as given in the manual) by multiplying it by the maximum possible voltage value of 3 V. This value is then considered to be the sensor voltage value which is used in the formula to extract the temperature value in Celsius.

Filtering

As the acquired data is subject to noise, the raw sensor data exhibited a lot of sharp fluctuations. In order to rid the data from this noise, the data was filtered using a 1D Kalman filter. The selection of the 4 filter parameters was done part experimentally and part theoretically. First we noticed that the Kalman gain, K, was immediately set in the calculation, therefore its initial value was irrelevant. The estimation error covariance parameter p, didn’t appear to make any significant difference in the result and quickly converged to the same value within a number of iterations. Therefore, the 2 parameters that were the most essential to configure were the measurement noise covariance r and process noise covariance q. Based on the manual, the temperature sensor gives readings within. We therefore, modeled the r as . At this stage, the only value to establish was q, which we did experimentally, by running a simulation of the Kalman filter with raw sensor data for various values of q. The results of these simulations can be found in the next section.

Temperature display and alarm routines

We implemented 2 routines for the 2 distinct modes of operation. Namely, the normal operation routine for when the sampled temperature is below the selected threshold and the alarm operation routine for the when the sampled temperature is above the threshold.

**Normal operation:** We defined an array of size 4, which holds the 4 GPIO pins which correspond to the LEDs (i.e. pins 12 – 15). In order to light the correct LED, we mapped the most recently sampled temperature into an index into this array using the following formula:

|  |  |
| --- | --- |
|  | (11) |

Where offset is defined as:

|  |  |
| --- | --- |
|  | (12) |

with the division being integer division. The offset was used to account for the (somewhat unlikely) situation when the processor is operating in sub-zero temperature (which it technically supposed to be able to do). For each decrease of 8 degrees another 4 is added to the offset in order to bring the term in the (11) into the range 0 – 3 range. The division by 2 of the temperature in (11) ensures that the index is incremented/decremented for each rise/fall. Finally, the modulus operation ensures the LED index is mapped into a valid index (i.e. 0 – 3).

**Alarm operation:** When the processor was found to be overheating, an alarm in the form of 4 LEDs fading in and out was implemented in a routine using a PWM-based technique to vary the intensities of the LEDs. One of the essential parameters of PWM is the chosen period. We designed our system to work with the SysTick timer. That is, a period X for our PWM, corresponds to X clock cycles of the SysTick. We found that the PWM period should be very short in order to reduce the “blinking” effect. We therefore chose, after testing multiple values, a period of 4. That is, our PWM period was 8ms (4 \* 2ms), corresponding to a frequency of 12.5 Hz (50 Hz / 4). Originally, we envisioned to increase/decrease the duty cycle after every PWM period. However, we found that repeating a duty cycle multiple times (in our case, 3 times) before modifying the duty cycle leads to a more “smooth” fade in/out patterns, as it makes the transition from one duty cycle to another (thus from one intensity to another) slower and thereby producing a more “fluid” transition effect between the LED internsities. Therefore, after every 24ms, we incremented/decremented the duty cycle (based on the value of a variable which indicates whether the duty cycle is to be incremented or decremented). When the duty cycle reached the maximum level (i.e. the period of the PWM), the variable was set to decrement. Similarly, when the duty cycle was the minimum (i.e. 0), the variable was set to increment.

To conclude, inside the main function a function to initialize the ADC, GPIO and SysTick components is called. Afterwards, we set a loop such that after every time the SysTick handler is called, a new temperature is sampled and converted into a digital value by the ADC. This value is then converted into a temperature reading (in Celsius) and filtered using a Kalman filter. Then, depending whether the temperature is above or below the pre-selected threshold, we proceed with the routine corresponding to the normal or alarm operation.

# Testing and Observations

The testing phase included the testing of the selection of the Kalman filter parameters and overall system behavior.

Kalman filter testing: we wrote a Matlab function (see Appendix) that performs Kalman filtering on a vector of data. We then collected 4096 raw sensor data samples, which we then filtered with our Matlab function and plotted and compared the results.

The following 4 plots showcase the effects of the varying the Kalman state parameter r (while keeping the other ones constant):

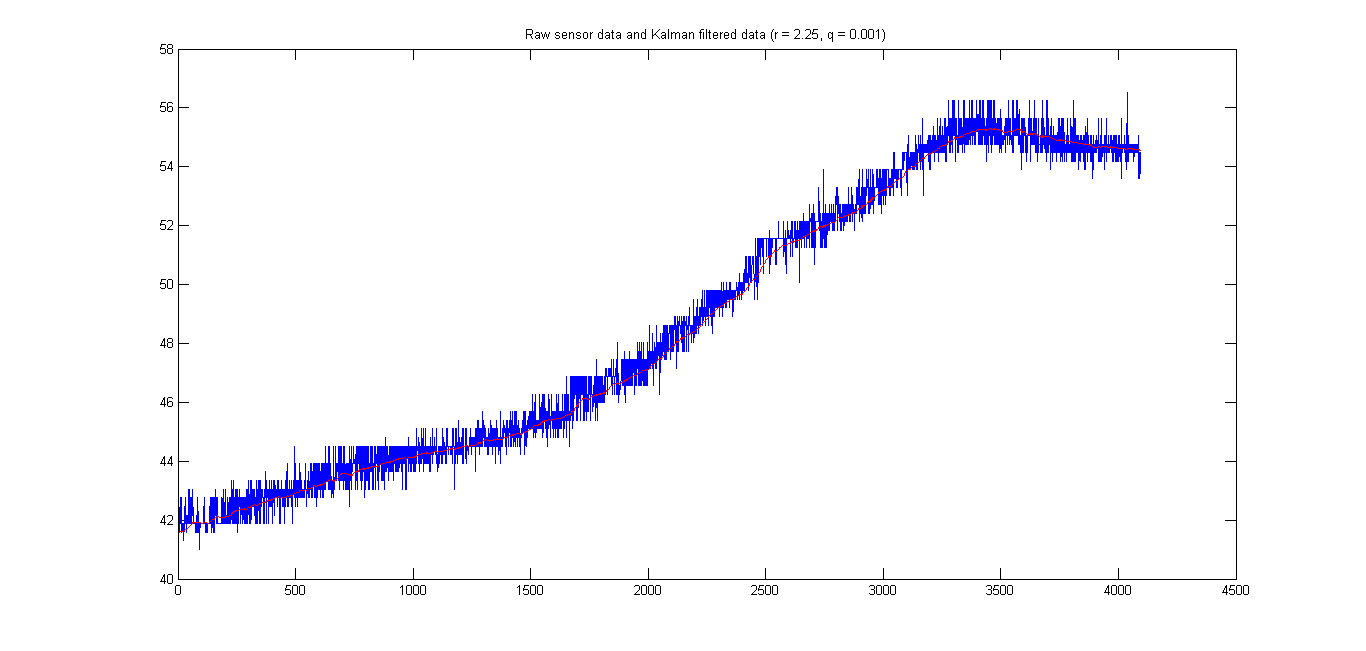


Figure 2: Kalman filtered data (red) and raw sensor data (blue) using a Kalman filter with initial parameters r = 2.25 and q = 0.001.

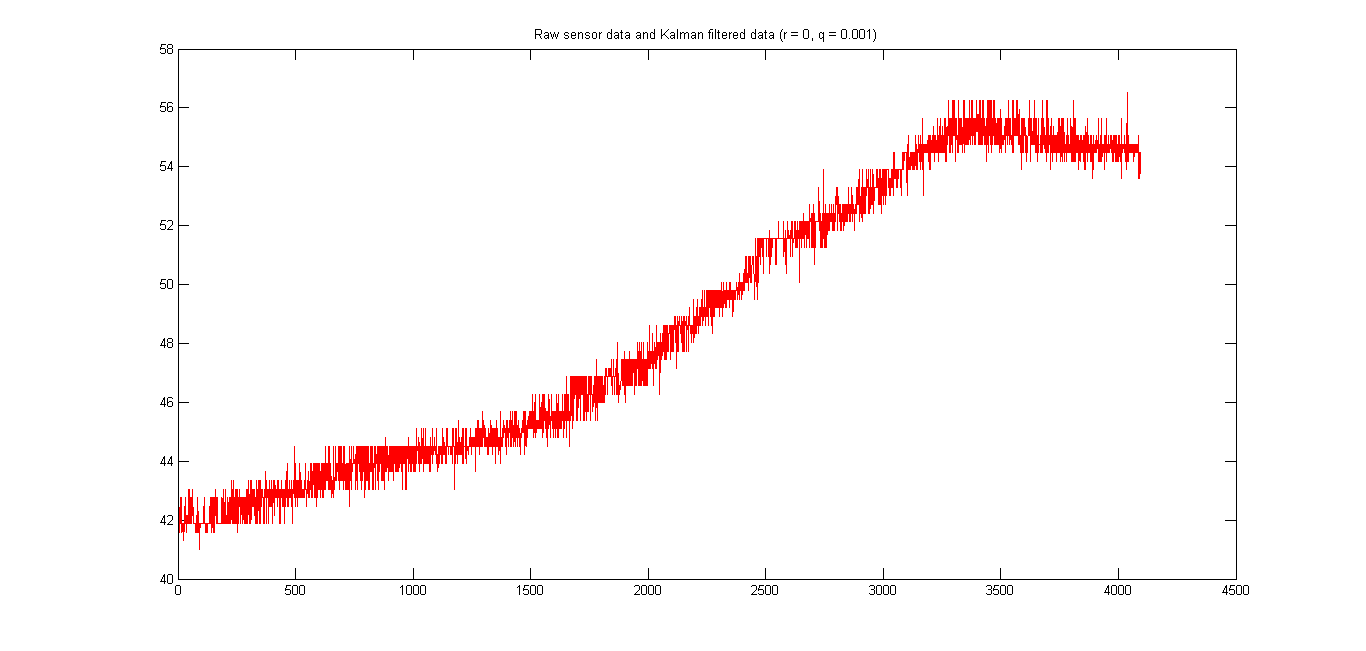
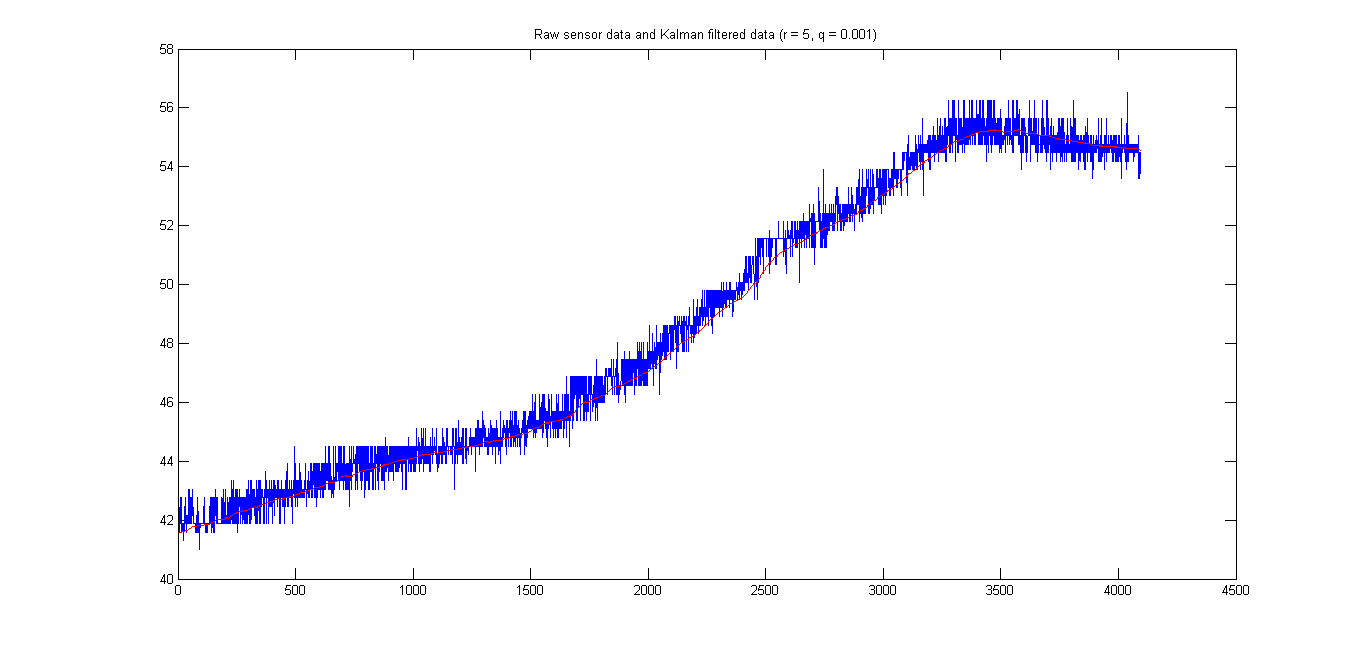


Figure 3: Kalman filtered data (red) and raw sensor data (blue) using a Kalman filter with initial parameters r = 0.0 and q = 0.001.

 Figure 4: Kalman filtered data (red) and raw sensor data (blue) using a Kalman filter with initial parameters r = 5.0 and q = 0.001.

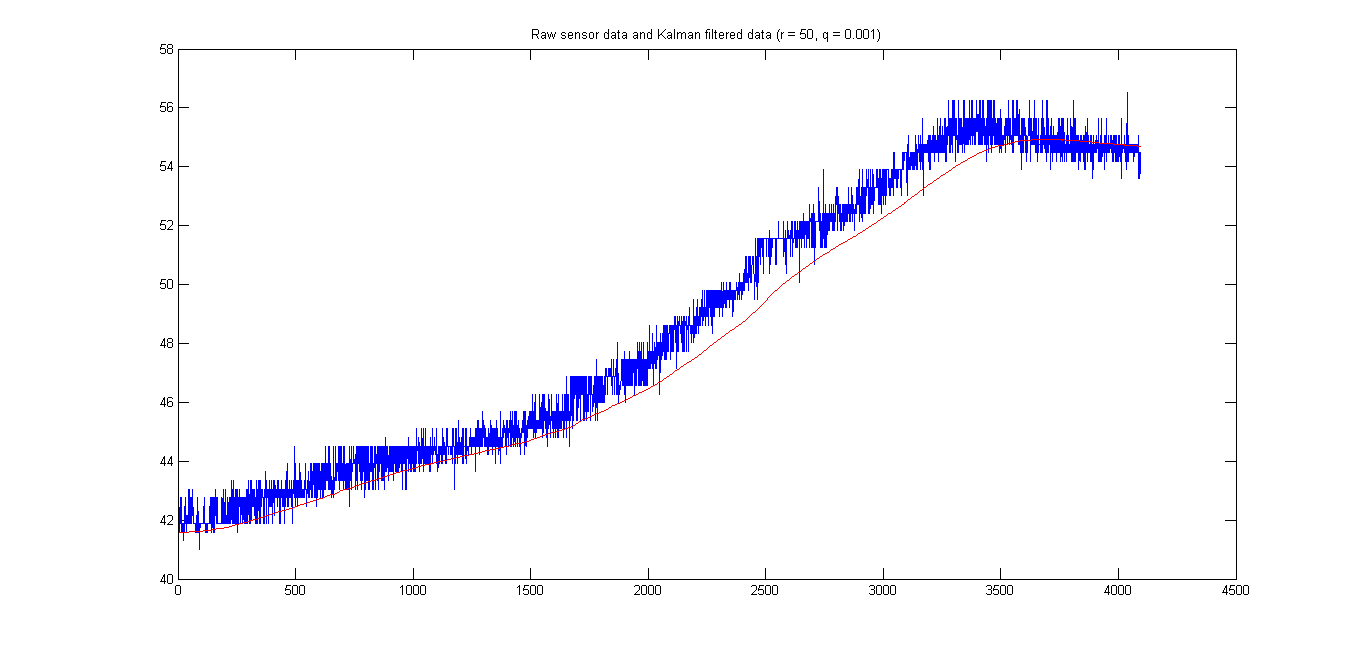


Figure 5: Kalman filtered data (red) and raw sensor data (blue) using a Kalman filter with initial parameters r = 2.25 and q = 0.001.

The following 3 plots showcase the results of the Kalman filter for various initial parameter q (while keeping the other ones constant and in particular, r = 2.25):

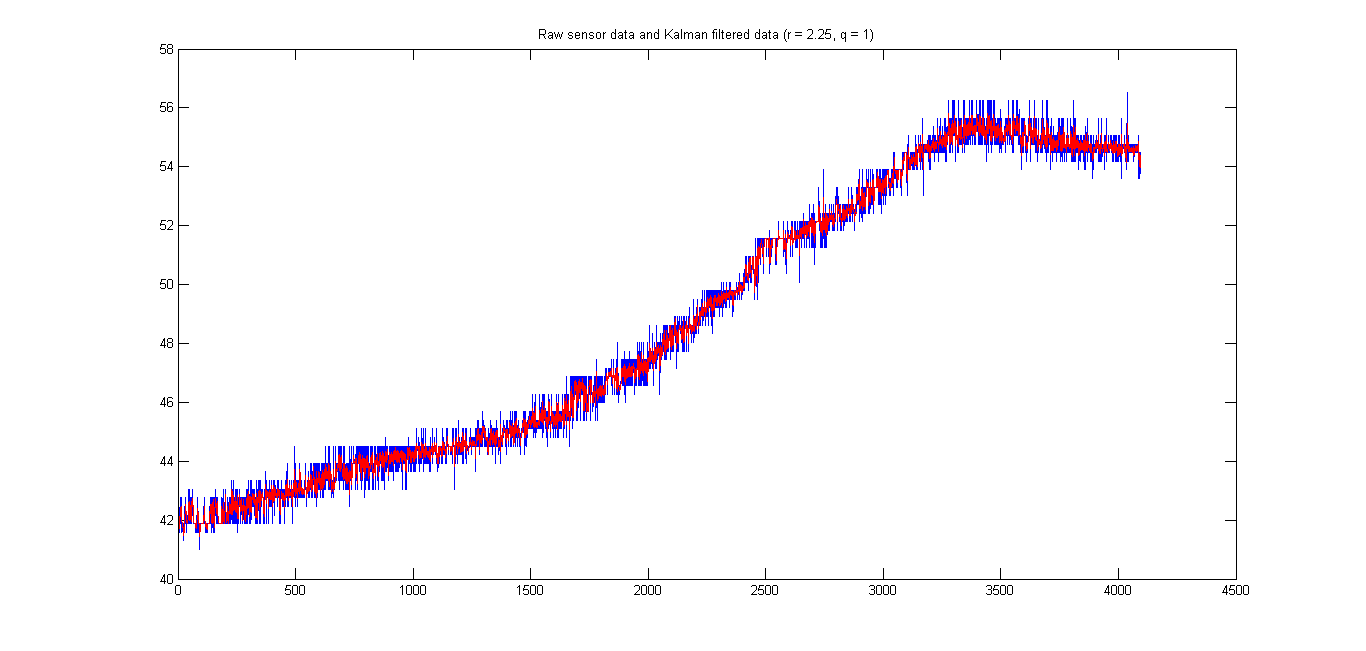


Figure 6: Kalman filtered data (red) and raw sensor data (blue) using a Kalman filter with initial parameters r = 2.25 and q = 1.0.

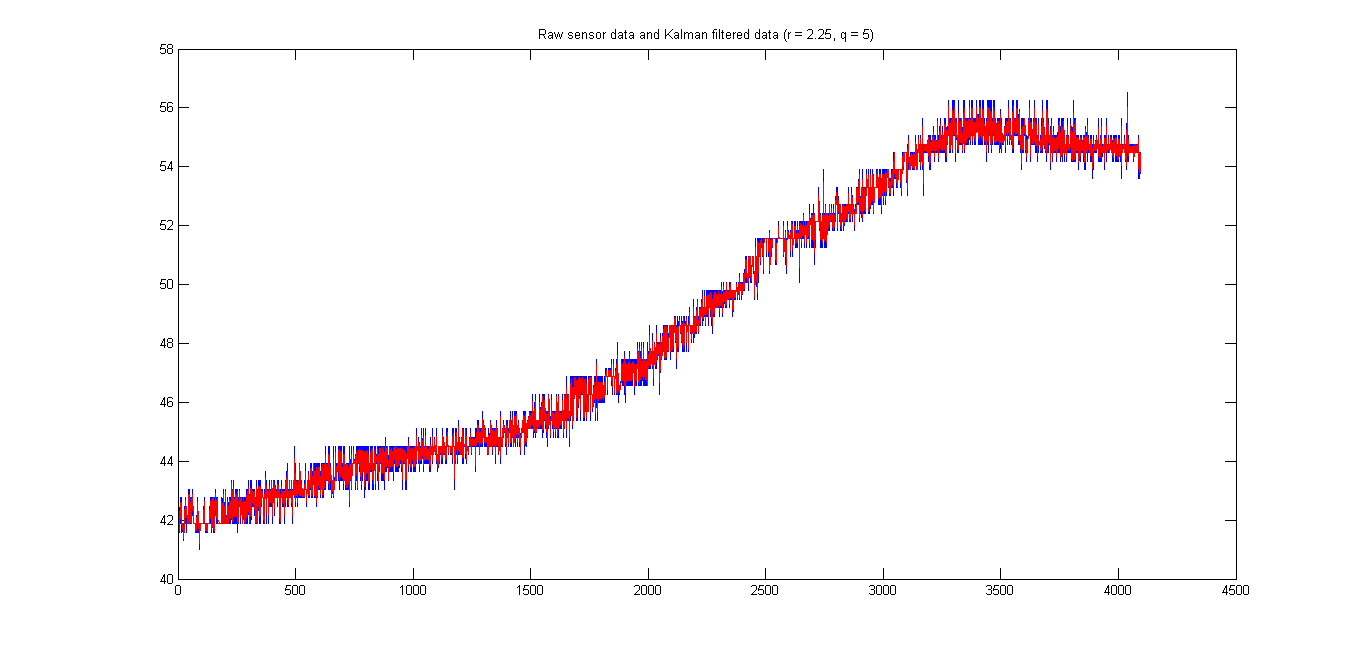


Figure 7: Kalman filtered data (red) and raw sensor data (blue) using a Kalman filter with initial parameters r = 2.25 and q = 5.0.

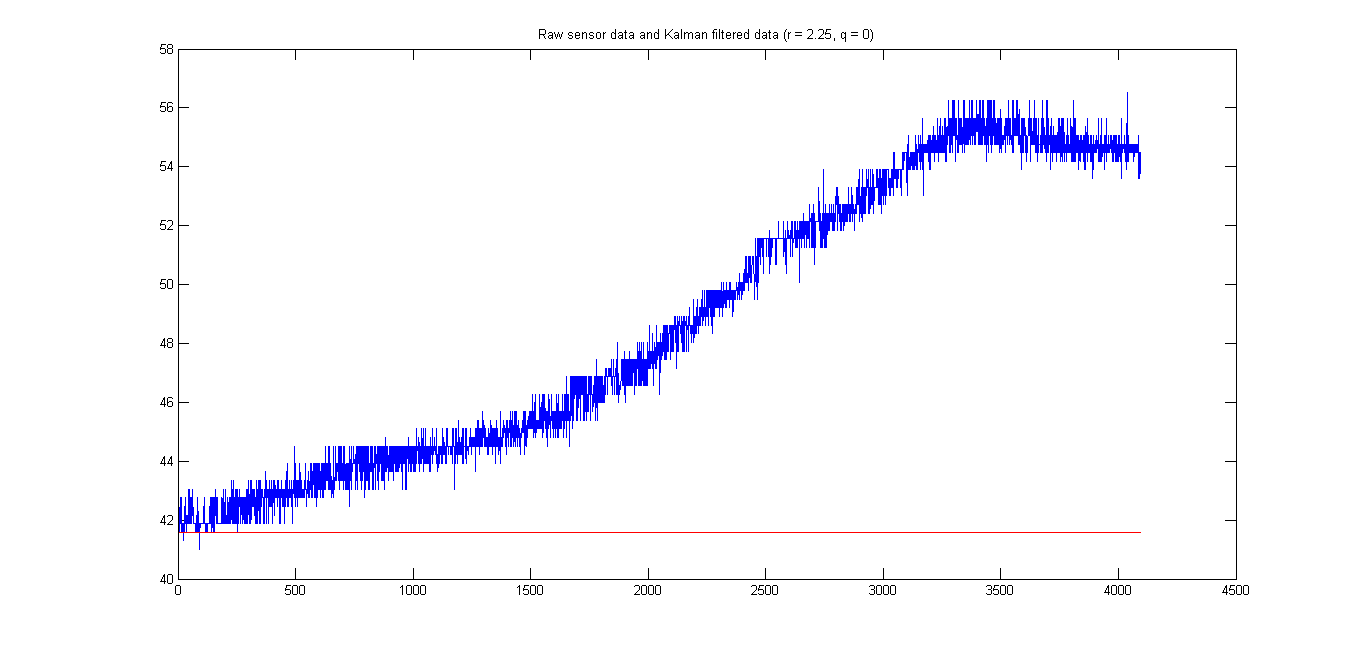


Figure 8: Kalman filtered data (red) and raw sensor data (blue) using a Kalman filter with initial parameters r = 2.25 and q = 0.0.

Using this script we also confirmed that varying the parameter p has a very minor effect on the overall performance of the filter over many samples. In particular, we compared the result of the filtering with various parameters p and found that the results were quite similar.

Based on these observations and the theoretical value for r, we concluded that using the parameters r = 2.25 and q = 0.001 (while arbitrarily setting the other ones), provided for an adequate filter setup. In particular, using these parameters, the filtered data appeared to very closely follow the general trend of the sampled data, while remaining smooth.

System testing

We setup a print function in our code (relaying on the provided code for redirecting the printf() function to the debug port), in order showcase the temperature in real time as the program runs. We then ran our program on the microcontroller in debug mode. We varied the temperature of the sensor using the hair drier and observed that the LEDs were turning on as expected, based on the printed temperatures. When we reached the threshold temperature, we observed that, as expected, the operation was switched into the alarm mode. Once the processor has cooled down to below the threshold temperature, the operation was switched back to the normal mode. Overall, after repeating the experiment multiple times, we observed that the operation of our system corresponded to the requirements.

# Conclusion

# References

|  |  |
| --- | --- |
| [1] | "www.st.com," STMicroelectronics, 2013. [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00037051.pdf. [Accessed 21 February 2015]. |
| [2] | A. Suyyagh, "ECSE 426 Microprocessor Systems Lab 2: Sensor Data Acquisition, Digitizing, Filtering, and Digital I/O," 2015. |
| [3] | R. Faragher, "Understanding the Basis of the Kalman Filter Via a Simple and Intuitive Derivation," *IEEE Signal Processing Magazine,* pp. 128-132, 20 August 2012. |
| [4] | A. Suyyagh, "ECSE 426 Microprocessor Systems Lab 1: One-Dimensional Kalman Filter," 2015. |
| [5] | "www.st.com/," STMicroelectronics, 2014. [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/reference\_manual/DM00031020.pdf. [Accessed 21 February 2015]. |
| [6] | "www.st.com," STMicroelectronic, 2014. [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user\_manual/DM00039084.pdf?s\_searchtype=keyword. [Accessed 21 February 2015]. |

# Appendix

Matlab code used for testing the Kalman filter parameters:

% 1D Kalman filtering of data using with initial Kalman state parameters

% p, r and q.

function [filtered] = KalmanFilter1D(data, p, ~, r, q)

filtered = zeros(length(data), 1);

% filtering

x = data(1);

for i = 1:length(filtered)

p = p + q;

k = p / (p + r);

x = x + k \* (data(i) - x);

p = (1 - k) \* p;

filtered(i) = x;

end

% plotting result

plot(1:length(filtered), data, 1:length(filtered), filtered, 'r');

title(['Raw sensor data and Kalman filtered data (r = ', num2str(r),...

', q = ', num2str(q), ')']);

end