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

Gueorgui Savadjiev (260572750)[1] and Sajad Darabi (260552196)[1]

[1]Department of Electrical Engineering McGill University

February 23, 2016

Abstract

Sensor data acquisition is a common task for Microcontrollers (MCUs). In this report, the procedures required to read the internal temperature of the sensor provided by the ARM STM32F4 processor are given. Furthermore, the drivers used to display the acquired data on a 7-segment display as well as on the LCD display are explained. Finally, an experimental run was performed to optimize the Kalman filter's initial state to smooth out the noisy data read from the sensor.

1 Problem Statement

The STM32F4 microcontroller (MCU) has an internal temperature sensor which measures the operating temperature of the processor. Firmware is to be implemented in order to use the analog-to-digital converter (ADC) modules provided internally to digitize the temperature readings. The reading is to be acquired at a sampling frequency of 100 Hz by using ARM's SysTick timer. The data is then to be processed and filtered using a 1-D Kalman filter. Lastly, the temperature readings are to be displayed on a 7-segment display and with bonus points associated with displaying the temperature on an LCD screen.

2 Theory and Hypothesis

2.1 Digitizing Temperature Data

Microcontrollers process digital signals, where as physical quantities are continuous signals (analog). To convert analog signals to discrete digital bits, an analog-to-digital converter (ADC) is used. ADCs have different $N-$bit resolutions, and detect different voltage levels as discrete multiples of a fixed resolution V_{STEP}, where V_{HIGH} is the maximal voltage and is mapped to all bits set to 1 ($ADC_{READING} = 2^N - 1$), and V_{LOW} is the minimal voltage and is mapped to all bits set to 0 ($ADC_{READING} = 0$). Hence an ADC with N bit resolution will have V_{STEP} calculated as follows:

$$V_{STEP} = \frac{V_{HIGH} - V_{LOW}}{2^N} \tag{1}$$

The V_{STEP} (V/increments) is used to convert ADC readings to real quantities as follows:

$$RealQuantity = ADC_{READING} * V_{STEP} + V_{LOW} \tag{2}$$

2.2 Displaying Data

2.2.1 7-segment Display

This display contains 7-segments as shown in the figure below. Individual segments can be driven through general purpose input output (GPIO) pins provided on the MCU.
To display different numbers it is necessary to drive different segment combinations as listed in figure 2.

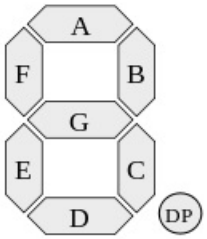

Figure 1: 7-segment display with segment labeling. Taken from [1]

In a 4-digit display it is only required to drive one of these 7-segments at a time, and cycle through each digit at a frequency which fools the eye to think all the segments are on at the same time.

segment outputs							display
a	b	c	d	e	f	g	
1	1	1	1	1	1	0	0
0	1	1	0	0	0	0	1
1	1	0	1	1	0	1	2
1	1	1	1	0	0	1	3
0	1	1	0	0	1	1	4
1	0	1	1	0	1	1	5
0	0	1	1	1	1	1	6
1	1	1	0	0	0	0	7
1	1	1	1	1	1	1	8
1	1	1	0	0	1	1	9

Figure 2: List of bit sequences required to turn on to display corresponding digit. Segment mapping corresponds to figure shown above.

2.2.2 LCD Display

Similar to the 7-segment display, the LCD display (HITACHI HD44780) contains 14 pins which can be driven to display different characters as depicted below. The communication with the LCD display consists of a set of commands and data writes. The LCD display must initially be configured to the proper settings using a series of commands, after which one can send characters encoded by their ASCII codes to display them on the screen.

2.3 Filtering Data

The Kalman filter is a set of mathematical equations that adaptively correct a series of physical measurements and converge to the desired "true" value. This filter can be used to compute an optimal state minimizing the error assuming the system is linear and its noise can be characterized as white Gaussian noise. The state of the filter is characterized by optimal measurement $\mathbf{x}$, the tuning factor $\mathbf{k}$, and the estimation error $\mathbf{p}$. The noise parameters are characterized by process noise covariance $\mathbf{q}$ and measurement noise covariance $\mathbf{r}$. Given these parameters and the state at t_1, the next state t_2 can be computed

as follows:

$$p(t_n) = p(t_{n-1}) + q \tag{3}$$

$$k(t_n) = \frac{p(t_n)}{p(t_n) + r} \tag{4}$$

$$x(t_n) = x(t_{n-1}) + K(t_n)(measurement - x(t_{n-1})) \tag{5}$$

2.4 Timing

In order to perform the ADC readings and data display at the proper frequency, a source of periodic timing is required. The MCU provides the SysTick timer, which generates an interrupt whenever the value of a counter register goes to zero. The SysTick counter decrements at the same rate as the system clock frequency, but the frequency at which the interrupt handler is called is set by the initial value stored in that register. The SysTick handler frequency is therefore the system clock frequency divided by the counter value.

3 Implementation

3.1 Digitizing Temperature Sensor Reading

The HAL ADC generic driver is used to configure the ADC. This driver is initialized by configuring the following structs: **ADC_ChannelConfTypeDef** and **ADC_HandleTypeDef**. In our implementation the following configurations were used:

Field	Value	Effect
ClockPrescaler	ADC_CLOCK_SYNC_PCLK_DIV8	Divide clock by 8 for prescaler
Resolution	ADC_RESOLUTION_12B	Bit resolution for temperature sensor is 12 bits
DataAlign	ADC_DATAALIGN_RIGHT	Places the padding zeros in the MS bits, most convenient as no shift is required to read
ScanConvMode	DISABLE	Use single channel mode
EOCSelection	ADC_EOC_SEQ_CONV	Not relevant as we don't check the EOC flag
ContinousConvMode	ENABLE	Continuous conversion desired as ADC will be read periodically
DMAContinousRequests	DISABLE	No DMA as we won't be using DMA to store temperature values
NbrOfConversion	1	Single conversion
DiscontinousConvMode	DISABLE	Chose continuous mode. Don't require discontinuous
ExternalTrigConv	ADC_SOFTWARE_START	No external trigger, hence disabled
ExternalTrigConvEdge	ADC_SOFTWARE_START	No external trigger

Table 1: ADC_InitTypeDef initialization field in the ADC_HandleTypeDef

Field	Value	Effect
uint32_t Channel	ADC_CHANNEL_16	choose channel 16
uint32_t Rank	1	N/A only one ADC is used in this lab
uint32_t SamplingTime	ADC_SAMPLETIME_480CYCLES	set ADC to 480 cycle sampling rate good enough as we need 100 Hz
uint32_t Offset	0	N/A

Table 2: ADC_ChannelConfTypeDef configuration used to initialize ADC.

As the internal temperature sensor of the processor is connected to ADC1_IN16, the **ADC_HandleTypeDef.Instance** field is set to *ADC1*. Furthermore, the ADC is configured to operate on channel 16. The values used for the clock prescaler (divide by 8) and the channel sampling time (480 cycles) were chosen because they were the largest available, in order to limit the ADC sampling frequency, which does not need to be too much larger than 100Hz to avoid wasting power. Since the ADCs have a maximum of 12 bits resolution, we chose this as our resolution, in order to get the most precise temperature readings. Data alignment was chosen such that the value can be conveniently read in little-endian order with the padding zeros on the left (data on the right) most significant bits. Since we are required to use polling, DMA was disabled and continuous conversion mode was enabled.

3.2 Reading Temperature at 100 Hz

The temperature sensor is read 100 times per second with the use of the SysTick timer. The SysTick timer is configured by calling : **HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/500)**. This stores the value HAL_RCC_GetHCLKFreq()/500 into the counter register which is decremented on each clock cycle. Since SysTick is using the main processor's clock running at 168 MHz, dividing by 500 will cause the SysTick Handler to be called at a frequency of 500 Hz. Note that it was possible to obtain 100 Hz by dividing the clock frequency by 100, but this was not done as it is necessary to refresh the 7-segment display more often to avoid flickering (to be discussed in next section).

A flag (SYSTICK_READ_TEMP_FLAG) is checked in the main loop prior to reading the temperature. This flag is set once every 5 calls to the SysTick handler (using a static counter) and hence the process will read the temperature sensor at 100Hz.

The temperature read from the sensor is first converted to voltage by using

$$tempinvolts = V_{SENSED} * \frac{V_{REF}}{4096} \tag{6}$$

where $V_{REF} = 3V$ and V_{SENSED} is the value read from the ADC connected to the temperature sensor. Prior to using V_{REF}, the tempinvolts was calculated using $tempinvolts = V_{SENSED} * \frac{V_{HIGH}-V_{REF}}{4096} + V_{REF}$[doc 6], where $V_{REF} = 1.8V$ & $V_{HIGH} = 3.6V$, though this did not return a correct temperature reading. Instead the aforementioned values were used. This value is then converted to the real physical quantity by using the temperature characteristics provided: $V_{25} = 0.76V$, $AVG\_SLOPE = 2.5$ [DOC 3]

$$temperature = \frac{tempinvolts - V_{25}}{AVG\_SLOPE} + TEMP\_REF + FUDGE\_FACTOR \tag{7}$$

We observed there was an offset in the temperature readings which caused the temperature to vary around 38 degrees Celsius. This may have been due to the internal processor's temperature, but since we wanted our measurements to fall in the range of 25 to 35 degrees to match room temperature, we added a constant offset called FUDGE_FACTOR to shift the readings down to the desired range.

3.3 Displaying Temperature

3.3.1 7-segment Display

The Discovery board consists of GPIO ports A-I each consisting of 16 pins, though not all pins can be readily assigned as they are reserved for different functions. The 7-segment display is driven using 12 GPIO pins, port E was used to serve this purpose. To enable PORT E, its clock must be enabled using __HAL_RCC_GPIOE_CLK_ENABLE();
The following figure depicts the pin numbering of the 7-segment display. The table below lists the connections and GPIO pin assignment.

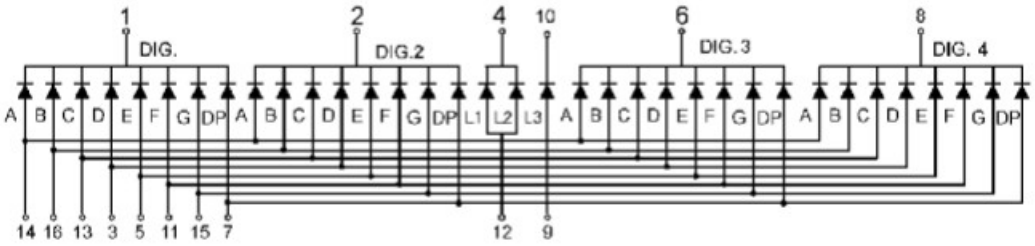

Figure 3: 7-segment circuit diagram with pin numbers.

GPIO_PIN_#	7-segment PIN	Purpose
GPIO_PIN_4-10	14, 16, 13, 3, 5, 11, 15	connection to segments
GPIO_PIN_11-14	1, 2, 6, 8	connection to individual 7-segments
GPIO_PIN_15	7	connection decimal point

Table 3: GPIO_InitTypeDef Port assignment. The ports used all correspond to port E.

All pins are configured using the following field parameters in the GPIO_InitTypeDef struct.

Field	Value	Effect
uint32_t Pin	0xFFF0	Enable pins 4-15
uint32_t Mode	GPIO_MODE_OUTPUT_PP	Pin can set to be as OUTPUT and can be driven LOW/HIGH
uint32_t Pull	GPIO_NOPULL	N/A since pin is set to OUTPUT pin does not need to be activated
uint32_t Speed	GPIO_SPEED_FREQ_LOW	Speed to drive GPIO set to minimum
uint32_t Alternate	N/A	no peripheral is connected to pin

Table 4: GPIO_InitTypeDef configuration used to initialize GPIO.

As mentioned in the table above, the configurations used are chosen as the pins will be driving the 7-segment display, i.e. the pins should be able to go HIGH (3.3 V) and LOW (0 V), which corresponds to the PUSH-PULL configuration used. Note that to limit the current through the LEDs, a series resistor connection is used prior to the connection to the 7-segment pins.
The pins can be set by overwriting the data register as follows:

```
GPIOE->ODR = GPIOE->ODR | 0xFFF0; // All pins 4-15 are set to high.
```

Here GPIOE is the GPIO port E, and ODR is the output data register. Each bit of the ODR corresponds to one of the 16 pins of the GPIO, bit 0 corresponding to GPIO_PIN_0 up to bit 15 corresponding to GPIO_PIN_15. A bit is set to 1 to turn the corresponding GPIO pin, and set to 0 to turn it off.

There were a few problems related to the display of the 7-segments. The first was flickering: if the digit switching frequency is not high enough, the eye will be able to notice that the digits turn on and off quickly, and will observe flickering. In order to solve this problem, we had to increase the 7-segment display frequency to 500Hz by setting the SysTick handler's own frequency to be 500Hz. The SysTick handler sets the flag SYSTICK_DISPLAY_SEGMENT_FLAG to 1 every 2ms. The main loop checks this flag independently of the SYSTICK_READ_TEMP_FLAG, and calls the **display_segment_val()** function which keeps track of the current digit and displays it on the 7-segment, switching to the next digit. Thus the digits are switched with a frequency of 500 Hz, and each single digit turns on and off with a frequency of 500 Hz / 3 = 167 Hz, which is larger than what the human eye can normally perceive (between 50 Hz and 90 Hz)[2].

An additional problem is that since temperature readings occur once every 10ms, displaying every temperature reading on the 7-segment display causes the digits to become blurred and dimmed. This is because the LEDs do not have enough time to completely turn on and off, and also because the eye is unable to perceive all the fast noisy changes in the least significant digits. In order to solve this problem, we display only one out of SEGMENT_DISPLAY_PERIOD temperature readings. The constant was set to 50, which means the same temperature is displayed for a duration of 50 temperature readings, or 500ms. This is long enough to fully turn on and off the 7-segments, and to distinguish clearly the least significant digits. The value which is currently being displayed is set as a global variable called displayed_segment_value, which is used by **display_segment_val()** when calculating the current digit to display.

3.3.2 LCD Screen

The LCD screen is driven by assigning pins from port B, and C. The choice of ports is arbitrary and could have been chosen to be other ports available on the board. Similar to the 7-segment display the pins were configured as in table 4. The pin assignments to individual pins are listed in the table below.

GPIO_PIN_#	LCD PIN	Purpose
GPIO_PIN_0-2 PORT C	4, 5, 6 respectively	RS, R/W, E, respectively
GPIO_PIN_1,2,4-9, 11 PORT B	7-14	connection to D0-D7 data bits

Table 5: GPIO_InitTypeDef Port assignment. The ports used all correspond to port B & C.

Prior to displaying characters on the LCD screen, it must be initialized as follows [3]:

- Clear the display
- Set function mode to line
- Display on

Characters can subsequently be sent to the LCD screen by resetting the cursor position, and writing individual characters to the data line (D0-D7). The R/W pin must be set and the enable line must toggle from 1 to 0 with a delay of $37\mu s$ for the operation to be completed. This delay is simply achieved by using a for loop with index variable bound calculated as follows, since the clock is 168 MHz it takes 168 clock cycles for $1\mu s$ hence 168 is multiplied by the desired μs delay (this listing is defined as a MACRO):

```
#define DELAY(US) {int i; for(i = 0; i < (int)168 * (US); i++){}}
```

Note that this delay is approximate, since nothing guarantees each iteration of the for loop to last exactly one clock cycle, but it is good enough for our purposes since the delay can be somewhat larger than that as well.

The LCD display temperature value is switched at the same frequency as the 7-segment display, and should normally display the same value as the 7-segments.

3.4 LED Temperature Alarm

When the temperature becomes higher than a certain threshold, the system should trigger an overheating alarm using the 4 LEDs LD3, LD4, LD5, LD6 on the board. The datasheet specifies that these LEDs are controlled by the GPIO pins PD13, PD12, PD14, PD15 respectively. Therefore we need to configure pins GPIO_PIN_12 to GPIO_PIN_15 of the GPIO port D. The configuration parameters are the same as those for the other pins as mentioned above in Table 4.

The temperature alarm is activated once the filtered temperature value becomes larger than THRESHOLD_TEMP. At this moment, the flag ALARM_TRIGGERED_FLAG is set to 1 in main. In order to avoid spurious noise, the alarm is not turned off when the temperature goes back below THRESHOLD_TEMP. When we initially did that, we observed that when the temperature is close to the threshold, the LEDs would turn on and off in a somewhat random fashion due to the temperature going above and below the threshold too fast. To fix this problem, we added a LOWER_THRESHOLD_TEMP constant which forces the alarm to wait until the temperature goes below this lower threshold before turning off. This provides a small noise margin and allows the LEDs to complete at least one full rotation. The value of LOWER_THRESHOLD_TEMP was set 1 degree lower than THRESHOLD_TEMP to provide a small margin.

In order to create a rotating pattern, the LEDs are each turned on and off in succession starting from LD3 and moving clockwise. This is achieved by a function called **toggle_LEDs()**. The period with which this function is called is set by the constant ALARM_LED_TOGGLE_PERIOD, and this causes the LEDs to toggle once every ALARM_LED_TOGGLE_PERIOD readings of the temperature sensor, if the ALARM_TRIGGERED_FLAG is currently set to 1. The current value set for this constant is 60, which means the frequency of toggling is 100Hz / 60 or around 1.7 Hz. This is good enough to obtain a clear rotating pattern.

3.5 Filtering Data

The temperature value read from the ADC is passed to a Kalman filter written in assembly. The state vector **struct kstate(q, r, x, p, k)** is initialized and passed to the function. The assembly code implements the filter by converting the following into their assembly equivalents:

```
p = p + q;
k = p / (p + r)
x = x + k * (measurement - x)
```

```
p = k * r // p = k * r = (1 - p/(p+r)) * p = (1 - k) * p
          // Above arguments follow from algebraic manipulation
```

Data is filtered by calling the respective function **Kalmanfilter_asm(&temperature, &filtered_temp, 1, &kstate)**, where &temperature is the pointer to the current temperature reading, &filtered_temp is pointer to variable to start filtered data, argument 1 is the length of the array (1 float) and kstate is the initialized struct.

3.6 Super-loop implementation

The super-loop, which runs in main, reads temperature values, displays temperature on 7-segment/LCD and handles the alarm is shown in the flow graph below. On individual decisions the red path denotes "FALSE," whereas "GREEN" denotes "TRUE". Initially the temperature flag is checked, then if set the ADC is polled for conversion. If the ADC converts successfully, the value is read. This value is converted to the physical temperature value using equation (7), and is passed to the Kalman filter. The final filtered value is compared to a threshold, if it surpasses this threshold the LEDs will turn on and start rotating in clockwise fashion. Further, if a new temperature value is read the LCD screen is refreshed, whereas each segment is refreshed at the end of the loop.

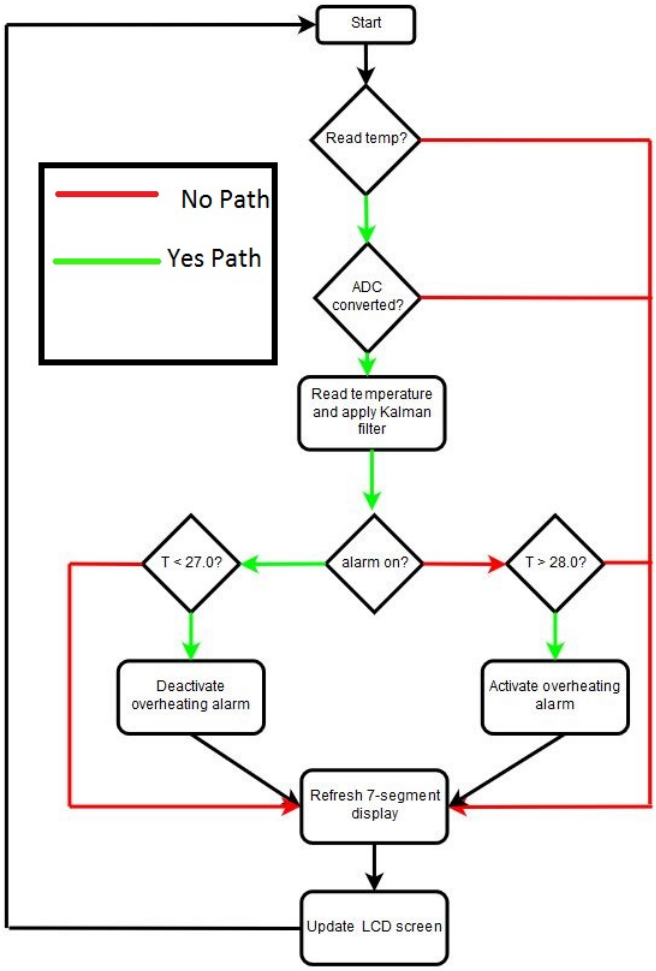

Figure 4: Flow chart of super-loop.

4 Testing & Observations

The overall behavior of the circuit was tested as follows: first the fluctuation of the unfiltered data was observed by plotting data in MATLAB as depicted in the figure below. If we are to zoom in to this data and observe the fluctuations, one can see there is an approximate 0.3 fluctuation around the mean. A value close to the 0.3^2 this is set to the noise process parameter.

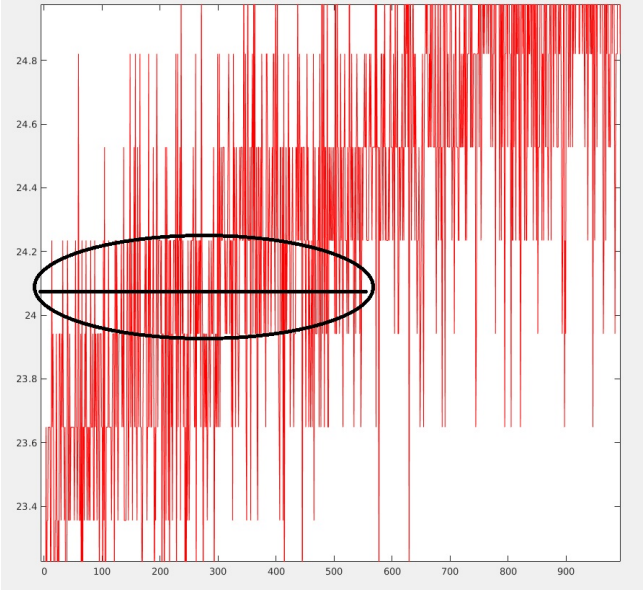

Figure 5: As circled, the noise fluctuates approximately with a pk-pk swing of 0.3. By observing such deviations the Kalman filter noise process parameter is chosen.

Using the above observations and assigning a smaller weight to the process covariance, the filtered data will follow the unfiltered data, but will smoothen extreme fluctuations as depicted in the figure below. The kstate is initialized as: **struct kstate(q, r, x, p, k)** $= (0.01, 0.3, 0.0, 0.1, 0.0)$

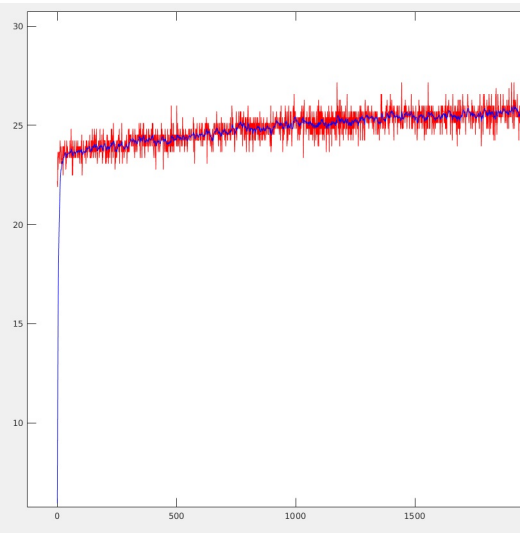

Figure 6: Red graph is the noisy raw temperature values, and blue depicts the filtered temperature value. As shown, the filter does not over smoothen the temperature, and the filtered data nicely follows the unfiltered data.

This result was satisfactory, and no other tests were done to optimize the filter parameters.

To test the displays, the values of the filtered data were printed to the console and compared to the values displayed on the 7-segment display as well as the LCD screen. The frequency of the refresh rates for the 7-segment display were adjusted such that flickering is not present. As was mentioned individual segments are refreshed at 167 Hz. As for the LCD, flickering is not an issue since it retains its display and hence it is only necessary to refresh it when the new temperature is to be displayed at 167 Hz.

5 Conclusion

The goal of this project was to use a microcontroller to read temperature values from its internal sensor, filter them, and to display the values on a 7-segment display and LCD display. Additionally, the system had to include an overheating alarm consisting of rotating LEDs when the temperature reaches a certain threshold. Throughout the implementation, we encountered problems related to timing: LED flickering and blurriness of digits. These problems were solved by making use of the SysTick timer's functionality to control the frequency of display refreshing. Our results show that proper timing is critical in the resolution of such issues. Additionally, the Kalman filter was shown to be accurate, and allows the measurement noise from the temperature sensor to be greatly reduced.

6 References

[1] Wikipedia,. "Seven-Segment Display". N.p., 2016. Web. 23 Feb. 2016.

[2] J. Davis, Y.-H. Hsieh, and H.-C. Lee, "Humans perceive flicker artifacts at 500 Hz," Scientific Reports, vol. 5, p. 7861, 02/03/online 2015.
[3] https://www.protostack.com/blog/2010/03/character-lcd-displays-part-1/ N.p., 2016. Web. 22 Feb. 2016. [3]