MCGILL UNIVERSITY

ECSE-426

Lab 2: Sensor Data Acquisition, Digitizing, Filtering, and Digital IO (Group-11)

Tanuj Sharma (260367436), Benjamin Swearingen (260398370)

10/19/2013

Contents

Abstract

This lab report describes Sensor Data Acquisition, Digitizing, Filtering, and Digital I/O of an embedded microprocessor system. The main requirement of this lab was to measure the operating temperature of the processor and update the four LEDs based on the change in temperature. Finally, a simple software pulse width modulation (PWM) was implemented to cycle the LEDs from their brightest to their dimmest. A button was used to toggle between these two modes of operation. At the end of this lab, the system was able to communicate temperature changes, and the direction of the change, through the LEDs.

Problem Statement

Lab 2 entailed the operation of embedded microprocessor system in sensor data acquisition and signal conditioning. Several design questions had to be addressed for efficient use:

- The first task involved initializing the internal temperature sensor on the STM23F4 microcontroller so that the operating temperature of the processor could be viewed and recorded. The lab required the use of an Analogue to Digital Converter (ADC). This was used to digitize the temperature values obtained from the sensor. Readings were converted from voltage format to temperature in Celsius.

- The ARM's SysTick timer was used to provide basic interval timing for a consistent time base. This set up an interrupt when the timer expires. Since interrupts run on high priority their runtime should be reduced, otherwise the system can be delayed for other tasks and events. The temperature sensor was required to operate at 20Hz.

- The sensor data, which was passed on as a signal, was prone to noise (from electromagnetic interference, thermal noise, ADC conversions etc). Therefore, a filter was designed to eliminate such noise. For the purpose of our experiment a moving average filter was implemented, which acted as a low pass filter by performing average of several past values. This smoothed the signal by eliminating unnecessary noise.

- The four diagonal LED displays, which showed the trend of change of temperature, were updated with only one LED on at a time. If the internal processor temperature was rising, the LEDs rotated their active state in a clockwise fashion (for each 2°Celsius change). However, if the temperature was falling, the LEDs rotated in an anticlockwise fashion.

- There were two modes of operation to be handled in the laboratory. This was managed by the user button, which toggled between the changes in temperature readings and the movement of the LEDs associated with it.

- Finally, a simple software pulse width modulation (PWM) was implemented to show that LEDs will light from off state to their brightest state and then back again to the off state.

Theory and Hypothesis

The temperature sensor was used to measure the ambient temperature (TA) of the STM23F4 microcontroller, which was measured in voltage. This result is converted to °Celsius by using the following equation obtained from the **datasheet**:

$$\textbf{Temperature (in °C)} = \frac{V_{Sense} - V25}{Avg_{Slope}} + 25$$

Where:

V_SENSE = value obtained from ADC

V25 = VSENSE value for 25° C = 0.76 V (760mV)

Avg_Slope = average slope of the temperature vs. VSENSE curve = 2.5 mV/°C

The VSENSE is multiplied by a value of 3000 as the default value of voltage needed is 3 Volts. It is furthermore divided by 2^resolution (2^12 = 4096) to be in the required range of between 0 and 1. This equation should give the correct conversion of temperature from reference voltage format to degrees Celsius during the experiment.

A moving average filter was implemented to remove noise from the signal. The moving average filter finds the average of several past values. The filter maintains a buffer called filter depth of size D, which determines how smooth the signal is. Since the size of D is fixed, new measurements replace old ones and hence it filters higher frequency components. Therefore, the moving average filter acts as a low pass filter by eliminating noise through smoothing. The plot line connecting all the fixed averages is the moving average. Mathematically, the moving average is the convolution of the datum points with a fixed weighting function. Therefore, the moving average is a simple linear system. Figure 1 shows an example of a moving average filter smoothening data in blue.

Fig 1: Example of a moving average filter

For our experiment, it was expected that for a large size of buffer D the output signal will be smoother as the higher frequency components get filtered out. This can be seen in the section titled “Observations and Testing”.

A software pulse width modulation (PWM) was used so that the LED’s light from off state, gradually increase and then go back again to off state. This was done by changing the SysTick period and using software delay subroutine. The pulse width within the period is changed. This is referred as a duty cycle. This allows us to configure the duration for which the LEDs are turned off and on. The LED was on for the duration of pulse and off otherwise. Figure 2 shows a PWM of a square wave signal with different duty cycles.

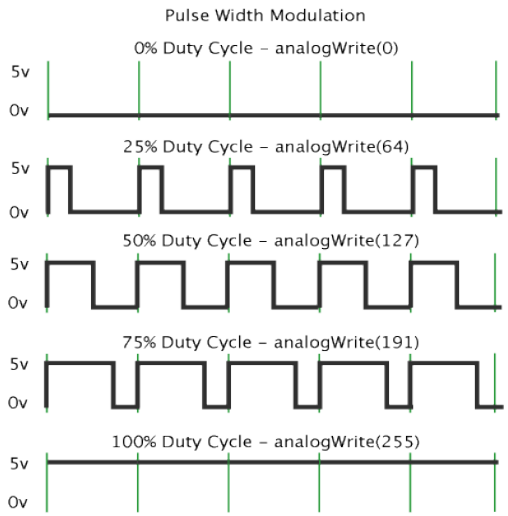

Fig 2: Example of PWM where green lines represent regular time period.

During the experiment, it was expected that the longer the pulse width the longer the LED is turned on and higher the intensity. If the pulse width is equal to the period then the LED is always on. This can be seen under the section titled “Observations and Testing”.

Implementation

The first problem that arose was the need to set the SysTick clock to a frequency of 20Hz, and then create an interrupt handler to deal with the resulting interrupt. Initializing and configuring the System Timer occurs in the same method, SysTick_Config(uint32_t ticks). This method initializes the timer and sets the number of ticks between two interrupts. SystemCoreClock is the number of ticks that occur in one second. Therefore SysTick_Config(SystemCoreClock/FREQ) sets Systick to occur with a frequency of FREQ. The interrupt handler for SysTick sets a flag, uint_fast16_t ticks, to 1.

LED initialization and the initialization of the button occur inside the same method GPIO_Configuration. Before either is configured, the AHB1 peripheral clocks to GPIOA and GPIOD are enabled. Then LEDs are configured. GPIO_Pin is set to pins 12 through 15, as these are the pins that the LEDs are connected to. Mode is set to output mode, as LEDs are outputs. OType is set to Push Pull, as current must be pushed to the LEDs to light them. The LEDs don’t need either a pull up or pull down resistor, so they are set to NOPULL. The speed is set to 100MHz as the LEDs need current pushed to them quickly. The LEDs are then initialized.

The button is set to Pin_0, as the button is on Pin_0 of GPIOA. Mode is set to IN, as the button is an input. Pull up/down is set to down so that the button only has current flowing through it

when it is pressed. This should be more power efficient than having a pull up resistor. The speed is left as the default value, as it does not require a fast checking. It will only be pressed occasionally, and any press will last multiple cycles. The button is then initialized.

Next two methods to rotate the LEDs were needed, a clockwise method and a counterclockwise method. To successfully rotate them, the state of the LEDs must be maintained. Rather than reference state by a number corresponding to which LED was on, an enum representing this number was created. The positioning of the LEDs on the board is reminiscent of a compass, so the state is represented by North, South, East, or West as seen in Figure 3. With this in place, rotating is simply a matter of passing the state to a switch, and then toggling the proper two LEDs for the given state. This is how both clockwise and counter-clockwise rotation is handled.

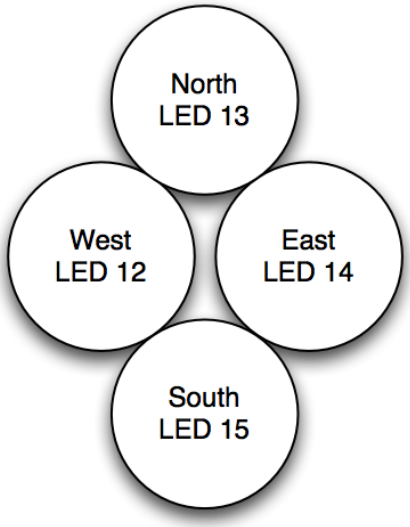

Fig 3: LED State and corresponding LED number

With the LED control in place, the temperature sensor and ADC must be configured and initialized so that the change in temperature results in a change in the LEDs. First the clock to the High Speed APB2 must be enabled, as the ADC is on that bus. Then using ADC_DeInit(), all ADCs are set to a known state. After the ADC is in a known state, it must be configured to work with the onboard temperature sensor. As it will be the only ADC operating, ADC_Mode is set to Independent. ADC_Prescaler is set to ADC_Prescaler_Div6, which sets the frequency of the clock to all ADCs to 131 kHz, which is fast enough that the conversion process won't stall the program. As the ADC is not in multiple ADC mode, ADC_DMAAccessMode can be disabled. ADC_Resolution is set to 12 bits, as this is the highest resolution that the ADC offers and the duration of the conversion is not going to impact the program. A higher resolution leads to a longer conversion time. The conversion will only be occurring in one channel, so

ADC_ScanConvMode is disabled. There is no external trigger, so ADC_ExternalTrigConvEdge is set to None. Data is aligned right so that any padding 0's occur on the left. One conversion occurs in each attempt. The ADC is then initialized with the given settings.

Reading the temperature occurs in the method Read_Temp(). This method configures the channel on which the conversion is going to occur, and then runs the conversion. The channel is set to occur in ADC1, which is to read from ADC_Channel_TempSensor for 480 cycles. As only one conversion will occur, the rank in the sequencer is set to 1. With the channel configured, the conversion is started. It runs until the End of Conversion(EOC) flag is thrown, then clears the EOC flag, and reads the value. Then the value must be converted from a reference voltage to an actual temperature. This is done through the equation detailed in the Theory and Hypothesis section:

$$Temp = \frac{\frac{V_{sense} * 3000}{4096} - 760}{2.5} + 25$$

After reading from the sensor, the new value must be filtered. This is done with a Low Pass Filter, implemented in a structure called Lowpass_Filter. It contains an array of floats of size six that represents the sample window(buff), a float that represents the average(average), and an integer representing where in the window the new value goes(new_placement). A configuration method initializes all values to 0. The filter_Value method takes in a value, places it in buff[new_placement], and then computes the average of the window. Then new_placement is incremented, with a modulus taken so that it can only have values from 0 to 5. If the window is not completely full, it returns a value of -100 so that the LEDs are not updated.

With the temperature complete, the button is used to switch between modes. Testing showed that a button press resulted in a string of 1s, rather than the expected toggling between 1s and 0s before settling to a steady string of 1s. As such, our debouncing took the form of a Finite State Machine, with states for On, Off, and On/Off in Run as seen in Figure 4.

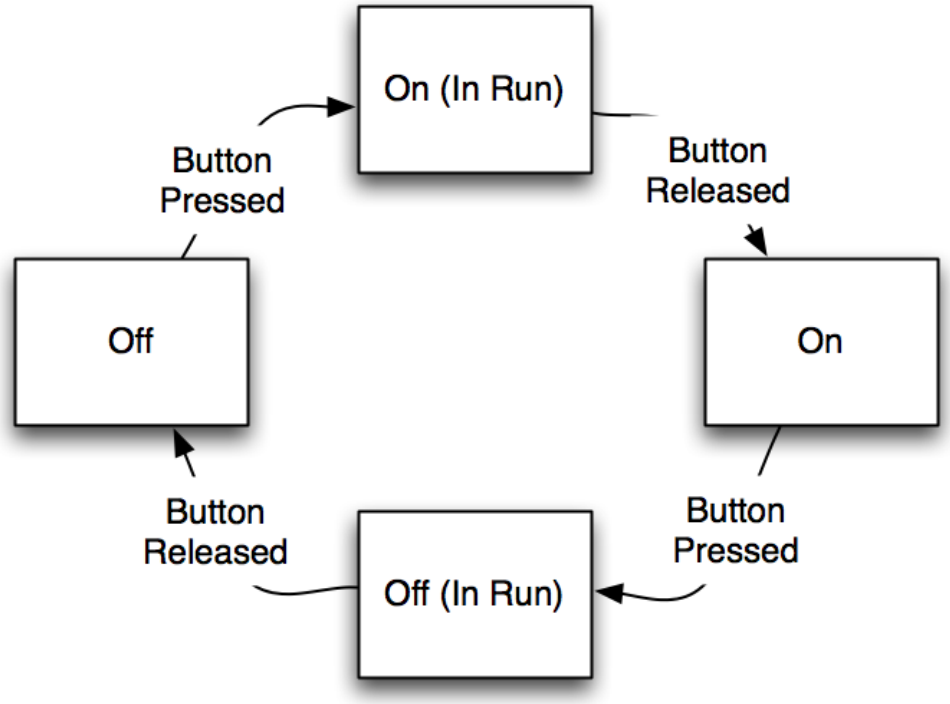

Fig 4: Button FSM

The state logic is handled by the Button_State method, as seen in Figure 5, which implements the FSM as a switch. When switching from Temperature mode to PWM mode, the SysTick is configured so that the frequency is 100Hz and all LEDs are turned off. This is because at 20Hz the LEDs would flicker. Upon switching back SysTick is reset and the LEDs are set to the proper state.

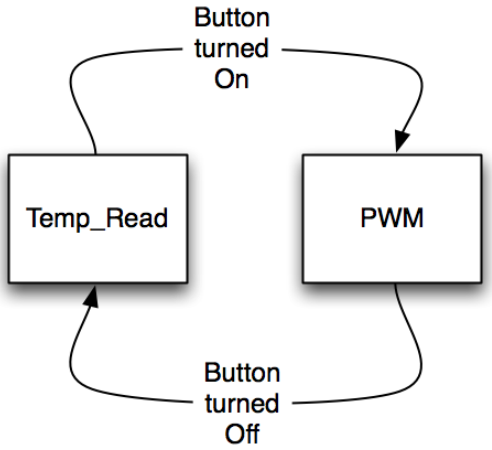

Fig 5: State FSM

The State FSM works off of the button state. When the button is in an “Off” state, PWM is off, and the temperature is being read. When the button is in an “On” state, PWM is on and the temperature is not being read.

The final problem was implementing PWM. To do this, the frequency of SysTick interrupts had to be increased so that flickering was minimized. Each clock cycle consists of a set number of ticks, SystemCoreClock/100, and it consists of two parts, the “On”, or duty cycle, section and the “Off” section. First, the LEDs are toggled on, and the percent of the total cycle that duty cycle represents is passed to delay.

$$percent = \frac{Duty\ Cycle}{Total\ Cycle}$$

Delay calculates the number of ticks that percent represents, and then uses a spinlock to wait until those numbers of ticks have occurred. Then the LEDs are toggled off and the inverse of percent is passed to delay, which again spinlocks until the correct number of ticks have occurred. The duty cycle is then increased or decreased so that the LED brightness increases or decreases.

Observations and Testing

Initial testing of the temperature sensor occurred before the ADC output to temperature conversion was implemented. This resulted in values of around 1000 being read. Upon completing the conversion formula, these changed to values from 29^{o}C to 34^{o}C being read with no external heat source.

Initial testing of the LED control occurred separately from that of the temperature sensor. In this testing situation, the program ran an infinite loop that called the rotational methods until it was stopped. The clockwise rotation behaved as expected initially, but the counterclockwise test initially behaved erratically, not doing any sort of rotation. It revealed that the LEDs were set to toggle incorrectly. After fixing this issue and testing again, the counterclockwise test passed.

The final testing of the temperature sensor and the LED control occurred at the same time. While the program was running, a heat source (air from a hair dryer) was applied to the chip. The temperature of the chip quickly rose from around 30^{o}C to about 40^{o}C. This resulted in a complete revolution in the LEDs, plus an extra rotation. After removing the heat source, the temperature slowly fell to around 34^{o}C, which did not complete a revolution, but did test one of the edge cases. As the edge cases are handled in identical ways, and both rotational methods

passed their initial testing, it is safe to say that the temperature successfully controls the rotation in the proper manner.

Before implementing state switching, the button was tested. The value the button's pin was output to the screen in an infinite loop. This test showed that rather than a noisy change from being unpressed to being pressed, the button smoothly transitioned into a string of 1s. This meant that debouncing could take the form of an easy to implement, but overly complex, FSM.

PWM testing occurred at the very end of the project's lifecycle. Essentially at this point the entire program was implemented, as the temperature reading and LED control were occurring as well. Even though they were occurring, the focus was on the toggling between modes and how PWD was preforming. Initially PWM was set to increase linearly, but this resulted in long periods of time at the brightest setting. Changing from a linear increase and decrease to an exponential increase/decrease made a much smoother cycle.

The testing of the filter was carried out by generating a test vector in MATLAB and comparing the filter output to MATLAB's. The input sample was subjected to filters of different size and the results of each were recorded. When then buffer size D was small, the original noisy data got filtered at fewer depths as compared to the ones with a larger buffer size. When the buffer size was increased to a larger value the signal curve smoothened and noise from the signal was eliminated from the signal. This was expected from as seen under the section "Theory and Hypothesis".

While testing, we subjected the input signal to three different size filters as follows:

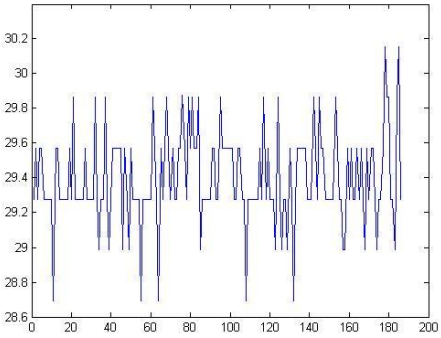

Fig 6: Input signal of the waveform

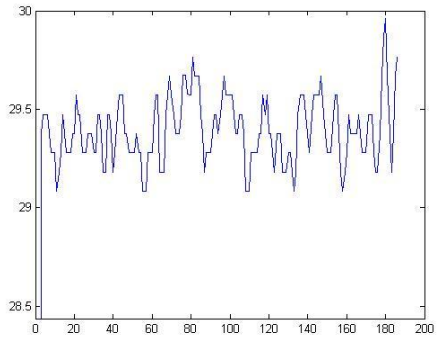

Fig 7: A 3 window plot showing smoothing of edges of the input signal

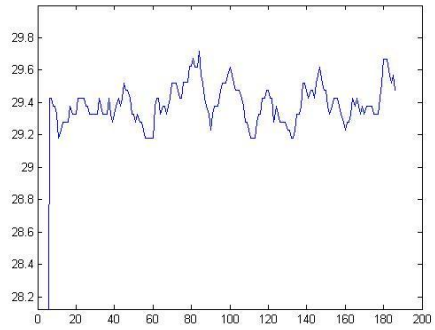

Fig 8: A 6 window plot showing smoothening of input signal

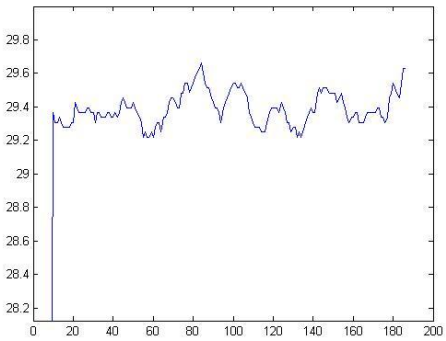

Fig: A 10 window plot showing smoothening of input signal

As seen from the above plots, the larger the size of the buffer the more noise is removed from the input signal. The results of the test correlated with theory and hypothesis.

Conclusion

This experiment involved in measuring the operating temperature of the processor from a temperature sensor. These temperatures values were digitized using ADC and passed through a weighted moving average filter to remove noise. The filter was implemented to maintain a buffer size of the filter depth D. As seen from the results of the experiment, the larger the size of buffer of the filter the smoother the input signal was as higher frequency components were removed. Finally, a pulse width modulation was implemented in the use LED's, to cycle from brightest to dimmest. The LED is on for the time period when the duty cycle is in its active state. Otherwise, the LED is off. These observations and results are supported by various figures that are contained in the section titled "Observations and Testing".

The end result of this experiment was a successfully working device that communicated temperature changes to the user and had a second mode that showed the capabilities of the LEDs. As the end product would not be connected to a computer, there would be no way of knowing the current temperature, or any specific numbers for temperature data beyond 2^{o}C.

Appendix

- Pulse width modulation (Theory and Hypothesis-fig.2)
  http://arduino.cc/en/uploads/Tutorial/pwm.gif
- Moving weighted average (Theory and Hypothesis-fig.1)
  http://en.wikipedia.org/wiki/Moving_average