ECSE426 Microprocessor Systems

Fall 2013

Lab 2: Sensor Data Acquisition, Digitizing, Filtering, and Digital IO

Introduction

A common operation in embedded microprocessor systems is sensor data acquisition and signal conditioning. If the signal is known to be noisy, it is beneficial to oversample and then apply a low-pass filter to remove some of that noise. In this experiment you will construct such a system, performing data acquisition and conditioning while providing a simple graphical output using the STM32F4-Discovery's LEDs.

To begin this experiment, you will first have to go to the desk on Trottier's 4th floor to claim your team's development kit. You need to go in groups, present your IDs and sign forms to acquire your lab kit. Development will be much like on the simulator but with a bit of hardware initialization code and the ST-LINKV2 SWD debugger. **Tutorial II** has already covered in extreme details how to use ST peripheral drivers, debugging, reading the board schematics and basics of interfacing. Moreover, to learn more on embedded C programming, the slides as well as an embedded C tutorial are posted for your reference.

Functional Requirements

In this lab you are required to do the following:

1. The STM32F4 microcontroller has a crude built-in temperature sensor. The sensor measures the operating temperature of the processor. You are required to write the code to initialize the internal processor temperature sensor and set up the Analogue to Digital Converter (ADC) to digitize the temperature sensor values (where the sensor output (as most sensors are) is in voltage). The sensor data should be acquired at a frequency of 20Hz. You are referred to the ADC section in the STM32Ffxx datasheet, consult the tutorial and the standard peripheral library in file documentation to get started with the setup.

2. Setup ARM's SysTick timer to generate the required frequency. No software delays are allowed **for this part**. Do not use any of TIMx peripherals yet. Only SysTick is allowed for acquisition frequency. The SysTick body should be lightweight as described in the tutorial.

3. Sensor data as any other signal is prone to noise, there are many sources of noise: electromagnetic interference, thermal noise and quantization noise due to ADC conversion to name a few. As such, one needs to filter these noise sources. There are quite a number of filters used in all sorts of engineering problems. To make things simple, we will use a very basic yet efficient filter called the moving average filter which is a simple linear filter that performs the function of a low-pass filter, i.e., smoothing. The moving average filter can in general performs a weighted averaging,

often done in a way that puts more weight to the more recent samples. The filter simply maintains a buffer of size *D*; called filter depth. The value of *D* determines how smooth a signal is. The filter takes as an input one new measurement at a time and returns the average of the buffer, that is the average of the most recent measurement *i* and the *d-1* values in the buffer. New measurements replace the oldest ones in this fixed size *D* buffer. The buffer is initialized to values of zero. You might wish to have a C struct having the internal state of the filter; that is the buffer, the index which keeps track of the new measurement placement, the sum or average. You can then pass a pointer to this filter state to the filter along with the new measurement.

Make sure you have a correct functioning filter. To test the filter, you can generate a test vector of your own in MATLAB and compare your filter output to MATLAB's.

Finally, you are required to investigate what filter depth size *D* suits your application. You should carry an actual analysis of the temperature sensor output (i.e. the printf statement could be beneficial in extracting the data). Then, use any mathematical package of your choice to analyze and compare the original noisy data with different versions of the filtered data at various depth sizes. You have to present a solid convincing case to justify your choice.

4. Convert the acquired readings from voltage format to temperature format in Celsius. The equations are to be found under the ADC section in the STM32F4xx datasheet

5. You are also asked to update the four diagonal LED display to show the trend of change of temperature values. If the internal processor temperature is rising, the LEDs will rotate their active state in a clockwise fashion. That is, for each 2°C rise, the next LED will turn on while the current one will turn off and so on. However, if the temperature change trend is falling, the LED display will rotate the active LED in an anti-clock wise fashion. Only one LED is on at a time. Your implementation should be resilient to small variations in temperature. The LEDs should also loop around as the temperature overflows the 8 degree range available using four LEDs in such a configuration

6. Finally, you are to make use of the user button to switch between two modes of operation. By default, your code functions as described above, once the button is pressed, you will light on the LEDs from their off state to their max gradually and then back again to their off state again and so on. This is achieved by simple software pulse width modulation (PWM). Pressing the push button again toggles back to the default operating mode and so on. For this part, you might need to reconfigure SysTick period and use a software delay subroutine. The concept is quite simple. You need to choose a period and change the pulse width within that period (duty cycle). The LEDs are on for the duration of the pulse and off for the remaining time of the period. The longer the pulse width, the more time the LED is on and the higher the intensity. When the pulse width is equal to the period it means it is always on. Modulating the duty cycle have the effect of average the total voltage/current going to the LED. Thus, giving you the visual change of LED intensity. Though PWM is supported in hardware timer modules in the STM32F4, You are required to do (simulate) it in software.

Testing

Naturally, most temperature readings should be below 30 C° but these sensors have lots of error and values above 30 are reasonable to see. You should extrapolate accordingly to whatever temperature range you have and decide on the ranges. To force the processor to heat up, you might try a heat source: exhaust from laptops or the machines in the lab should be hot enough to change the temperature sufficiently -- though give it some time since the heat will be transferred more slowly through air. Just be careful not to short any pins on the board while handling it near metal cases. Do not, however; subject the board to heat sources which could damage it. Don't even use ice to cool it down ☺

Debugging

For the hardware part, you will no longer be running your program in simulation mode. We will be using real time debugging through the ST-LINK debugger (conveniently built-in on the discovery F4 board). To switch from simulation mode to hardware debugging, right-click on Target 1 in the left most project pane, choose Debug →Use ST-LINK debugger from the drop down menu. Click settings next to it and choose SWD protocol. Refer to Figure 1 below. Similar settings are used for programming: in the same window, click on Utilities, also choose ST-LINK Debugger from the drop down menu. Finally click settings→ Add → STM32F4xx Flash. Save and rebuild; now your project should be working. Figure 2 illustrates setting up the programmer. To make life easier for you, we have prepared a base project with default setting set up for you.

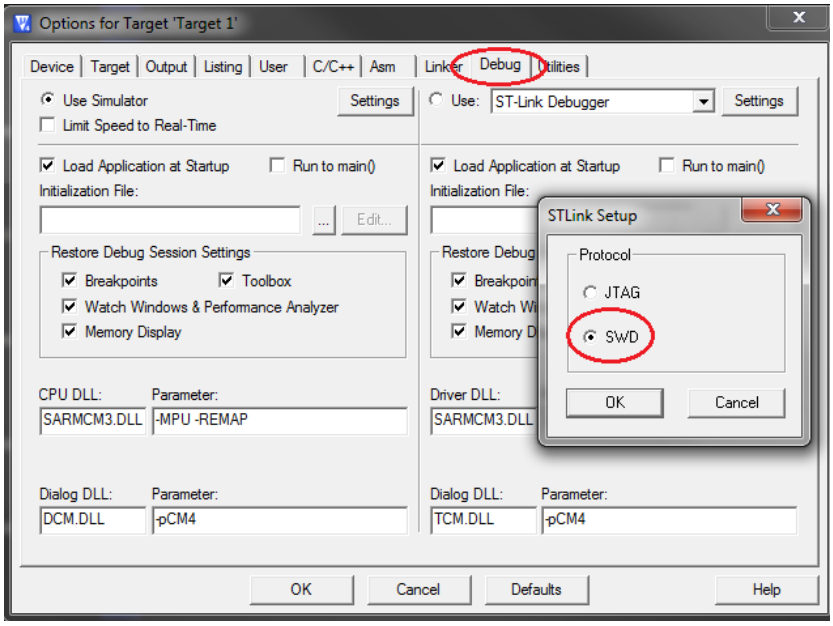

Figure 1 - Setting up the Debugger

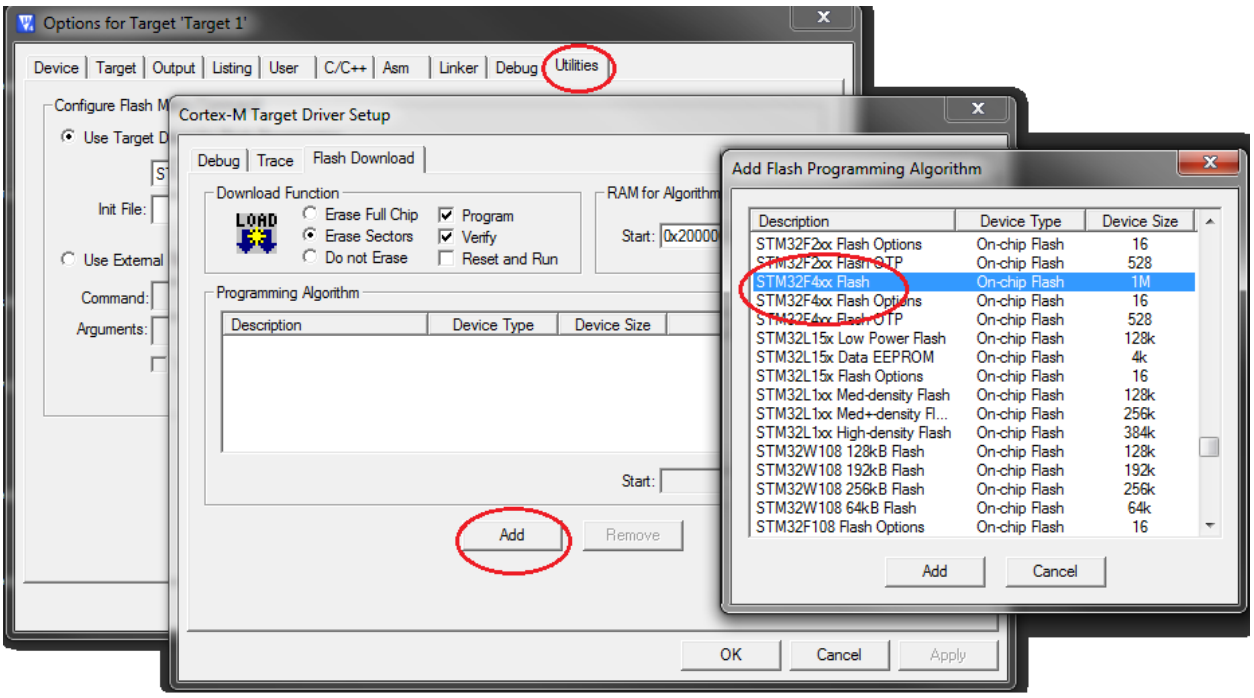

Figure 2 - Setting up the programmer

Hints

LEDs and Push buttons

The use of general-purpose IOs (GPIOs), ADC and the temperature sensors is fully described in the STM32F4 Reference Manual. To see how the user button and the LEDs are connected to the processor, please consult the F4-Discovery User Manual.

SysTick handlers and Interrupts

DO NOT perform the sampling in your interrupt handler routine. Interrupts happen in a special processor mode which should be limited in time. You can start the sampling process in the handler but do not wait for the conversion to complete. This can be achieved by setting a flag inside the ISR which can be observed from your regular application code once you return from your interrupt routine.

Don't worry about interrupts outside of the SysTick timer. "void SysTick_Handler(void){}" which will execute each period of the SysTick timer. You are not required to use any **other** sources interrupt beside SysTick's of for this experiment.

Examples Codes to get you started:

Many examples exist for various elements of this exercise. Browse the STM32F4 peripheral library examples for ADC usage, SysTick, and GPIO interactions. With an understanding of how to use the ADC, you can read in the STM32F4 Family Reference Manual about how to use the temperature sensor with the ADC.

Reading and Reference Material

- Tutorial II
- C Tutorial
- F4 Discovery User Manual
- STM32F4 Reference Manual. Chapter 10: ADCs and Temperature Sensor.
- Example codes. Available at http://www.st.com/internet/com/SOFTWARE_RESOURCES/SW_COMPONENT/FIRMWARE/stm32f4discovery_fw.zip
- Of course you are free to refer to any other posted documents which you see useful.

Demonstration

Demos will take place Friday, October the 18th. The exact time will be decided later. We expect that your code be ready by 3:00 P.M that day and that you will be ready to demo. No extensions allowed without penalties. You will lose 25% per day for the demo. Weekends are considered one day.

Final Report Submission

Reports are due by Monday October 21st at 11:59 P.M. Late submission will be penalized by 10% a day. For reports, weekends are considered two days not one as in the demos case. Submit the PDF report and the compressed project file separately. **Don't include your report in the archive.**