ECSE 426 - Microprocessor Systems
Lab Report 1: Analog Data Acquisition, Filtering, and Digital I/O

Harley Wiltzer (260690006)
Matthew Lesko

February 19, 2018

Contents

List of Figures

List of Tables

0.1 Implementation

The design of the voltmeter was fairly complex and was composed of several modules. These modules included the user input module for processing user input via the push button, the data acquisition module for digitizing analog data on the board, the data processing module for filtering the data and associating meaning to its digital values, and the output module for displaying the data to the user. This section will be divided into several subsections, each corresponding to a module, in order to organize the design decisions that were made.

Since all of these modules needed to work together, they had to be synchronized appropriately, and this was achieved with the `SysTick` timer. The `SysTick` timer invokes interrupts at a chosen frequency, and the interrupt handler was used to coordinate all of the modules. Since the configuration parameters of the `SysTick` timer were heavily influenced by the modules described above, they will be explained independently in the sections following where the design decisions were made.

0.1.1 The User Input Module

One of the requirements of the voltmeter was to provide three display modes to the user: a display of the RMS voltage, and a display for each of minimum and maximum voltage updated within the past ten seconds. As such, there was a need for user input to switch between these display modes. This was achieved with the user button on the STM32F407 board, which allowed the user to cycle through each of the display modes by pressing the button.

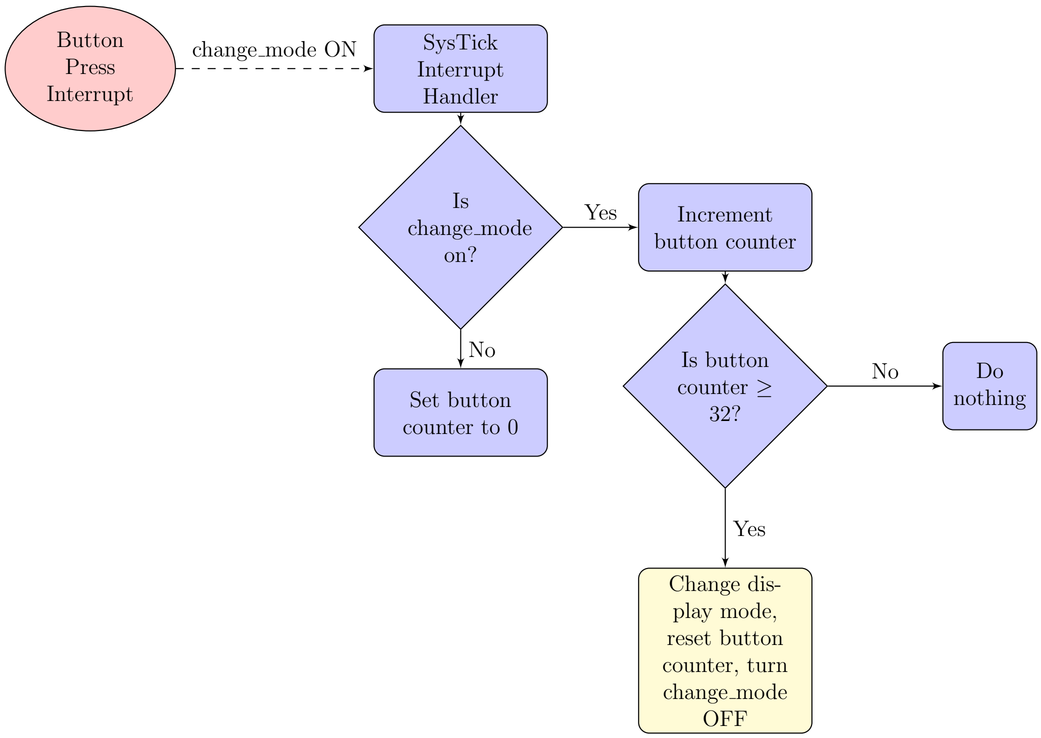
The first challenge dealt with how to process the button presses. There were two main options: polling for button presses and handling interrupts. Since polling the button at every iteration in a loop seemed inefficient, the interrupt method was chosen. Therefore, an NVIC interrupt was configured at priority 0 for `EXTI0`. However, even with the interrupts set up, there was still a major challenge to correctly process the button presses, as one button press often caused several interrupts. This could have been do to button bouncing, or possibly the fact that a what seems like a short *click* to a human actually goes over several clock cycles of the processor. To prevent this from occuring, it was decided to enforce a time delay between consecutive button press handling routines, and this was achieved using the `SysTick` timer. The process is shown in Figure 1.

In brevarium, the `EXTI0_IRQHandler()` function (invoked by the button press interrupt) asserts a `change_mode` signal, and when the `SysTick_Handler()` function (invoked by `SysTick` interrupt) sees that, it waits for 32 consecutive `SysTick` interrupts before taking action. The number 32 was achieved via trial and error, as it was unknown exactly how long an average human button press lasts. That being said, this number was chosen at a `SysTick` frequency of 200Hz, so a delay of 160ms was imposed.

0.1.2 The Data Acquisition Module

The data acquisition module was responsible for gathering analog data and digitizing it so it could be processed. Firstly, however, it was helpful to set up a digital to analog converter (DAC) in order to test the performance of the analog to digital converter (ADC). Setting up the DAC was fairly straightforward, and most of the work was carried out by the HAL Cube software. The DAC

Figure 1: Debouncing button presses



was configured on channel 1, and it wrote to pin PA4 on the board. Furthermore, its resolution had to be chosen. Since the performance of the voltmeter ultimately depended on the resolution of the ADC, the resolution of the DAC was chosen to be the same as that of the ADC, which was 8 bits, right aligned. This decision will be explained when discussing the ADC parameters below. Finally, the DAC needed to output some analog voltage. A value was chosen arbitrarily and passed to the DAC via the `HAL_DAC_SetValue()` driver function, and the conversion was instantiated via `HAL_DAC_Start()`. This starts a conversion in polling mode, which was deemed appropriate for the purposes of this experiment as the conversion would only occur once.

Setting up the ADC was considerably more complicated. Again, the basic initialization was done by the HAL Cube software, and the ADC1 unit was set up on channel 1. Single conversion mode was chosen, as it was required for one conversion to occur at a given frequency. Once again, the resolution had to be determined. Since it was known that the displayed voltages would be shown with two decimal places of precision, the user could only see voltages in increments of 0.01V. The

Table 1: Accuracy of ADC by resolution

Resolution (bits)	Voltage difference between consecutive digital values (V)
6	0.047
8	0.012
10	0.003
12	4.89e-4

accuracy of the ADC by its resolution is shown in Table 1. The accuracy is defined here as the change in voltage when increasing the digital reading by 1. Since the voltage range of the ADC is 3V, the accuracies were calculated according to

$$A = \frac{3}{2^R} \tag{1}$$

were A is the accuracy (rightmost column) and R is the resolution (leftmost column). Clearly, a resolution of 6 bits is not a great choice, as it cannot resolve voltages within 0.047V from each other. Since the display of the voltmeter allowed two decimal places, this accuracy is insufficient. With a resolution of 10 bits, however, the accuracy is relatively high. At an accuracy of 0.003V, the display would only change after a change of 4 in the digital reading of the ADC. The 8 bit resolution could resolve voltages that are 0.012V apart which is very close to the accuracy on the display. Ultimately, the 8 bit and 10 bit resolutions were the main contenders, because the 8 bit resolution is slightly worse than that of the display, and the 10 bit resolution is much stronger than that of the display. In the end, the 8 bit resolution was chosen as it was deemed strong enough for the purposes of this experiment, it would cause lower power consumption, and it matched one of the possible resolutions of the DAC which made the code simpler.

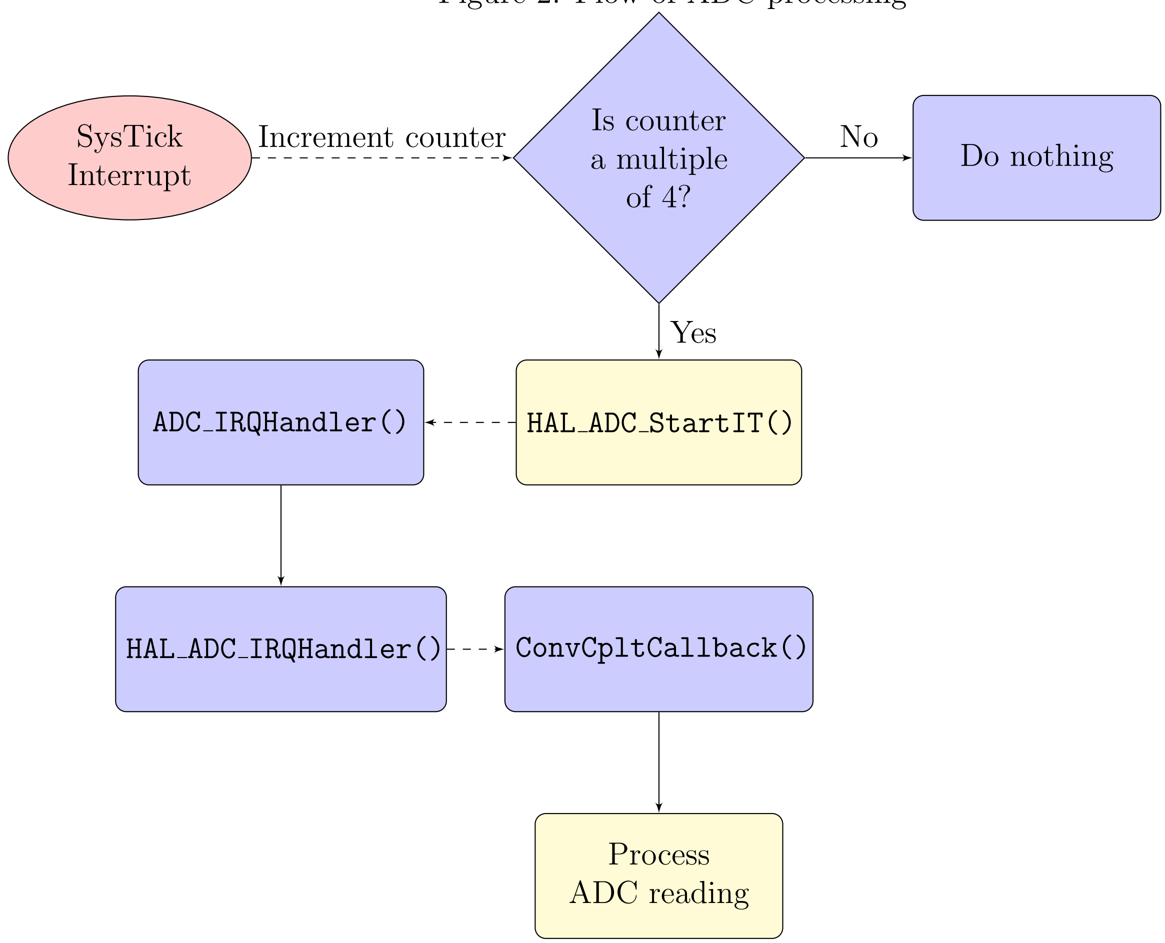
Next, the conversion mode of the ADC had to be chosen. Polling mode was not considered a viable option, since ADC conversions would happen frequently and thus polling would waste a considerable portion of the CPU's cycles. Although DMA was a very good alternative, the developers did not have time to do the requisite research. Therefore, interrupt mode was selected. Despite the conversion mode, however, the frequency of ADC conversions remained to be implemented. A sample rate of 50Hz was required, so the `SysTick` interrupts were used to time the ADC conversions. Since the `SysTick` interrupts were occuring at 200Hz (see the Output subsection below), it was required to implement a prescaler in the `SysTick_Handler()` function in order to sample at 50Hz. The process of handling ADC conversions is shown in Figure 2. The `SysTick_Handler()` maintains a `counter` and increments it at every `SysTick` interrupt, that is to say, at 200Hz. Since ADC conversions were to be taken at 50Hz, they had to be triggered at a rate four times less frequent than `SysTick`. Thus, the `HAL_ADC_StartIT()` function was called on every fourth `SysTick` interrupt to start ADC conversions at 50Hz. This was accomplished by starting the ADC conversion when the `counter` variable was a multiple of 4.

0.1.3 The Data Processing Module

Given the digital readings from the ADC, the next step was to filter them to reduce noise and to then translate the readings into meaningful values. These steps were carried out in the `ConvCpltCallback()` function (see Figure 2). In order to reduce noise, the digital readings were passed through a 4^{th} order FIR filter, which effectively output the average over the past 5 inputs. A 9^{th} order filter was experimented (which effetively output the average over the past 10 inputs), but it had a negligible effect on the performance of the voltmeter. Since the 9^{th} order filter required considerably more memory, the 4^{th} order filter was chosen. By passing each new ADC reading through the filter, potential noise will be "smoothed out", as it will be converted into the average of that reading and the previous 4.

Upon filtering the ADC readings, the filtered output was then translated into its analog rep-

Figure 2: Flow of ADC processing



resentation according to

$$x_a = V_{DD}\left(\frac{\mathcal{F}(x_d)}{2^R - 1}\right) \tag{2}$$

where x_a is the analog representation of the new ADC reading, $\mathcal{F}(x_d)$ is the filtered digital ADC reading, R is the resolution (in bits) of the ADC, and V_{DD} is the highest voltage that the ADC is rated for. Finally, the analog value is passed to the `plot_point()` function, which is responsible for curating the inputs into the form in which they should be displayed.

The goal of the `plot_point()` function is to maintain the values of the RMS voltage over the past 10 seconds, as well as the minimum and maximum voltages over the past 10 seconds, while making efficient use of memory. It was decided to design this in a "moving window" fashion, meaning the values to be displayed will always taken into account data from within the past 10 seconds, and not in discrete 10 second blocks. With the requirement of keeping memory usage down to at most 20 samples, this was a difficult task. Firstly, it was decided that the minimum and maximum voltage do not need to be updated very frequently. Of the 20 samples, 5 samples were allocated to each of minimum and maximum voltages, and the remaining 10 samples were reserved for RMS voltage. With only 5 samples for each of minimum and maximum voltages, each sample would have to represent the minimum or maximum sample in a 2 second interval. Likewise, for RMS, each stored sample represents the running RMS over a 1 second interval. The minimum samples, maximum samples, and RMS samples were each stored in their own circular list (see Appendix

A) for efficient insertion and removal. For each 100 samples (2 seconds) passed to `plot_point()`, the minimum and maximum were calculated and added to their respective circular lists. For the RMS calculations, a new value was added to the RMS circular list for every 50 samples (1 second), however the value to be added could not be the RMS exactly. RMS is calculated as follows

$$RMS = \sqrt{\frac{1}{|N|}\sum_{n \in N} n^2} \tag{3}$$

where N is the set of samples to calculate the RMS over. Since the square root operation is non-linear, the RMS itself cannot be passed to the RMS circular list every second. Rather, the *sum of squares* $\sum_{n \in N'} n^2$ is passed, and the RMS is calculated over the past 10 sum of squares values.

With the circular lists mentioned above, the `plot_point()` function was able to return the RMS, minimum, and maximum voltages over the past 10 seconds fairly easily. For the minimum and maximum voltages, this consisted of simply finding the minimum value in the minimum voltages circular list and the maximum value in the maximum values circular list. The RMS over the past 10 seconds was computed as follows:

$$RMS_{10} = \sqrt{\frac{1}{50|S|}\sum_{s \in S} s} \tag{4}$$

where S is the set of the 10 most recent second-long sum of squares calculations described above. Since each second consists of 50 readings, the sum of sums of squares had to be divided by $50|S|$, because it summed over 50 points for each $s \in S$.

Note that although the RMS is only being updated once per second and the minumum and maximum are only being updated once every 2 seconds, the circular lists still store enough data to report the exact RMS, minimum, and maximum over a moving 10 second window. Due to the constraint on memory usage, the update frequency of the RMS, minimum, and maximum could not be improved with a moving window design.

Appendix A

Circular Lists

In order to calculate RMS, minimum, and maximum on a moving window, a "circular" list data structure was used. Circular lists contain a capacity C describing how many items the list can hold, an index L describing where the next sample should go, and a vector δ which stores the data of the list. Each time a new element ϵ is to be added to the list, it is inserted by $\delta[(L{+}{+})\%C] = \epsilon$, where $\%$ represents the modulo operation, and $L++$ increments L. Note that when the circular list is full, that is to say $L = C$, the least recent data point in the list is overwritten. For example, if one was to add the values $1, 2, 3, 4, 5, 6$ in a circular list of capacity 3, the progression of the circular list would look as follows:

$$
\begin{array}{ll}
\delta = [] & \text{Add } 1 \\
\delta = [1,] & \text{Add } 2 \\
\delta = [1, 2,] & \text{Add } 3 \\
\delta = [1, 2, 3] & \text{Add } 4 \\
\delta = [4, 2, 3] & \text{Add } 5 \\
\delta = [4, 5, 3] & \text{Add } 6 \\
\delta = [4, 5, 6] &
\end{array}
$$

Clearly, the δ in the previous example is always storing the latest 3 values that it receives. Therefore, the circular list is storing a running window of its latest C inputs.

This data structure is particularly convenient due to the running time of the operations it provides for storing running windows. To store a running window, one must add an item to a list and remove the oldest item from a list for each new sample in the worst case. With this data structure, adding a new item involves calculating an index by $(L++)\%C$, which is an $O(1)$ operation. However, adding the new item removes the oldest item implicitly! Therefore, the process of updating a running window with this data structure has $O(1)$ time complexity. In terms of storage, the circular list stores the C most recent samples as well as the value C itself and the current index into the list, L. Therefore, the space complexity of the circular list is $O(n)$, which cannot be improved. The addition of the two extra parameters L and C is a fairly low cost.