ECSE 426 - Microprocessor Systems Analog Data Acquisition, Filtering, and Digital I/O

(Laboratory Assignment 2)

Isaac Chan*, Tiffany Wang†
*School of Electrical and Computer Engineering
McGill University, Montreal, Quebec
Email: isaac.chan@mail.mcgill.ca
Student ID: 260624096
†School of Electrical and Computer Engineering
McGill University, Montreal, Quebec
Email: tiffany.wang@mail.mcgill.ca
Student ID: 260684152

***Abstract*—In this report, we demonstrate the STM32F407's capability to output analog interpretations of its digital data, and we show that through the capabilities of a well-tuned filter, an appropriate sampling rate, and the inherent analog-to-digital conversion (ADC) drivers in the STM32 board, that we can still retain the most important aspects of the data despite the inherent loss in an ADC. For the purposes of this experiment, we output an arbitrarily defined voltage through the digital-to-analog converter (DAC) and re-read that same information through the ADC and subsequently, output 3 fundamental values (min, max, and RMS) through a 7-segment LED display in order to demonstrate that fundamental modules of information are still retained despite the information loss.**

I. Problem Statement

The primary objective of this experiment is to display important modules of output voltage once it has been passed through a DAC and ADC system (Walden 1999). We sample the analog signal at a rate of 50 Hz, and pass it through an Finite Impulse Response (FIR) filter (Cetin et. al 1997) as the read signal has an inherent noise component. The filter's coefficients were defined via the window design method, and this technique was chosen in order to scale the magnitude of the filtered data to a more comprehensible size. We then extract the minimum and maximum values from the past 10 seconds, and the past-10 Root-Mean-Squared (RMS) (Cartwright et. al 2007) values are displayed onto the 7-segment LED display. The system level block diagram can be found in Appendix A

The primary challenge faced in this experiment was the implementation of interrupts. The inherent system-level interrupts from ADC essentially dominates the bus and prevents the processor from receiving and executing code from other interrupts in the system. This issue was particularly prevalent with regards to the interrupt that was generated by the blue button input, and the blue button interrupt would not be registered by the system as the ADC would be perpetually occupying the bus. The solution to this problem was that to deal with all interrupts equitably (essentially, the implementation of equal-level priorities to all interrupts), each interrupt generates a flag instead of executing the relevant code, and a subsequent method then checks for what flags - corresponding to the interrupts - were raised, and executes commands related to those interrupts.

II. Theory and Hypothesis

This system hinges on multiple signal processing, microprocessor, interrupt, and electrical theories on its function. We detail their functional dynamics below:

A. Digital to Analog Conversion

A DAC (Dempsey et. al 1999) converts a precision floating point value to a physical continuous analog signal (in this case voltage). An ideal DAC converts the abstract numbers into a conceptual sequence of impulses that are then processed by a reconstruction filter using some form of interpolation to fill in data between the impulses. However, in practicality, a DAC creates a piecewise-constant continuous function that are made up of a series of rectangular functions. The difference between the theoretical ideal DAC and the practical DAC is shown in figures 1 and 2.

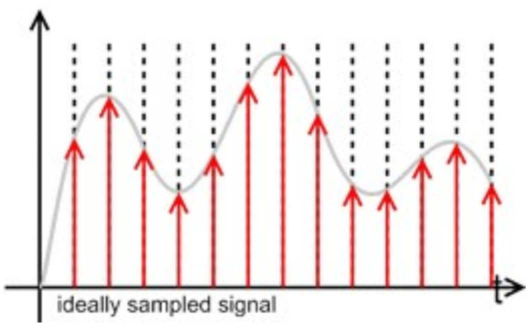

Fig. 1. Ideal DAC signal

There are multiple metrics that determine the quality of a DAC, all of them pertaining to the informational loss that is

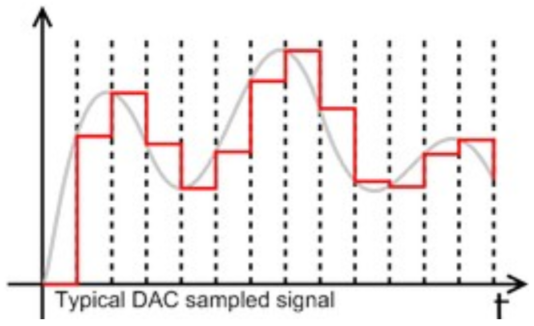

Fig. 2. Practical DAC signal

possible via such a conversion. The most important of these metrics is the resolution of the inherent digital signal. For our application, since we were concerned with voltages between 0 and 3V with up to 2 decimal places, we decided that a 1024-bit resolution would be sufficient to create a signal that had insignificant loss.

B. Analog-to-Digital Conversion

An analog-to-digital converter converts an analog signal (Walden et. al 1999) (in this case - voltage), to a floating point precision digital signal. As this process requires the quantization of a continuous input, it by definition has to introduce loss of information. Additionally, as the ADC does the conversion periodically by sampling the input, it limits the bandwidth allowable bandwidth of the input signal.

The performance of an ADC is quantified by its bandwidth, resolution, and signal-to-noise ratio. In our instance, as the analog signal was converted from the board's own digital signal, the bandwidth is trivially fully comprehensive of the information. We measure the dynamic range as a function of the resolution and the effective number of bits (ENOB) as a measure of the signal-to-noise ratio. We compute the resolution Q with the following formula

$$Q = \frac{E_{FSR}}{2^M} \tag{1}$$

where E_{FSR} is is the full scale voltage range, which is given by

$$E_{FSR} = V_{RefHi} - V_{RefLow} \tag{2}$$

where V_{RefHi} and V_{RefLow} are 3 and 0V respectively, and where M is 10 bits.

We also alleviate the issue of the signal-to-noise ratio by passing the ADC information through a FIR filter, which is described below.

C. FIR Filter

A Finite Impulse Filter, or an FIR filter, is filter whose response to an impulse is of finite duration as it eventually converges to 0. This effectively parameterizes the digital interpretation of an analog signal such that it reduces the signal to noise ratio.

In our causal discrete-time system, the FIR filter of order N that we apply can be summarized with the following:

$$\begin{aligned} y[n] &= b_0 x[n] + b_1 x[n-1] + ... + b_N x[n-N] \\ &= \sum_{i=0}^{N} b_i * x[n-i] \end{aligned} \tag{3}$$

whereupon we can see that this operation is a discrete convolution function. This filter brings about multiple advantages.

1) The filter is inherently BIBO stable
2) Requires no feedback, and as a result, the relative error of each of the outputs is the same as rounding errors would not be compounded through summation

The coefficients b_i were chosen using the Window design method, where we scaled the coefficients based on the size of the window such that the output of the filter is on the same level of magnitude as its input.

D. Interrupts

There are effectively 2 methods for peripherals in a system to interact with the CPU. Polling or interrupts (Corbet et. al 2014). Polling, while much simpler to implement, has one fundamental flaw: When the CPU requires the peripheral to make a computation, the CPU will continuously poll the peripheral until the computation is completed, and will therefore not make any other computations in the interim. However, an interrupt method will allow the CPU to move onto other tasks until the peripheral is completed with its computation, by which time the peripheral will send an interrupt to the CPu and pass along the information that the CPU has requested. The interrupt system allows for no wasted clock cycles whereas the polling system will experience a lot of latency due to its fundamental inefficiency.

III. Implementation Testing

In this section, we will discuss the following 8 components of our system, and we will present the methodologies used to implement the system, issues that we encountered during its implementations, and the solution that was used to remedy the problem.
1) ADC
2) DAC
3) GPIO different pins
4) Blue Button
5) NVIC
6) Display - interleaving
7) Systick
8) FIR filter and C_MATH

We have included details of the implementation of the code base in the form of a class and state diagram which can be found in Appendices B and C respectively.

A. DAC

We used a resolution of 10 bits such that the voltage that is given between 0 and 3V has a linear interpolation of 1024 intervals. To validate that these results were indeed correct, we computed the theoretical values that the output of the DAC would give, and compared the theoretical values to a measured value from an oscilloscope. As there is inherently a certain amount of offset, we implemented a finite offset to the code in order to justify that difference. Currently, the voltage is a constant value that is declared as a global variable in the beginning of the code, however, it can certainly be modified that it gives a discrete sine wave or any other function depending on the specific functional use case.

B. ADC

The value we read from the ADC would be between 1 to 1024 bits, and is evenly spaced between 0 and 3V, and therefore, a conversion is necessary. The equation that we use to show this conversion is shown below

$$DV = \frac{AV * 3}{1024} \tag{4}$$

where DV is the Discrete Voltage, and AV is the Analog Voltage, we multiply by 3 as that is the VDD, and divide it by 1024 as that is our resolution. The ADC was implemented in 2 different ways.

1) Polling mode: Our initial hypothesis was that the polling mode would be functional and we selected it as it would be easier to implement.In this mode, we would want to sample at each systick, however, as the system would be polling continuously until a manual interrupt was enacted, the system would effectively be stuck at the sampling due to the high clock speed.

2) Interrupt mode: We remedied the polling problem by placing the system in an interrupt mode instead. We did this by using the start_IT function instead, and selected the NVIC mode. What this effectively meant is that an interrupt would happen every time the function were called, but it wouldn't stop the CPU from doing other computations. In practicality, this functioned much better than the Polling mode, and as a result, was what we eventually selected to complete the rest of the experiment.

C. GPIO

The pin setups were completed by the CubeMX software, and all the LED pins were therefore initialized through that as well. We selected to implement the LED pins and the 7-segment outputs of the system in push-pull mode because we don't want to have any high impedance floating point values. This is done by having pull up and pull down transistors at all times, which allows for the implementation that we wanted.

D. Blue Button

Our first step in this experiment was to setup the blue button. Initially, it functioned correctly when there were no other interrupts in the system, however, once we introduced other interrupts, the interrupts began conflicting and the blue button output (LED lights) no longer functioned. To remedy this issue, we set up the blue button as an interrupt that had higher priority compared to the other interrupts. However, since the function that enacted the methods were placed within the interrupt handler, there was a race-condition issue that prevented the blue button from functioning. As a result, we used a flag and used a separate method to check for the flag before running the interrupt-relevant code.

One final issue that we encountered was that the sampling frequency of the blue button interrupt was much greater than human reaction. As a result of this, we added a debouncer so that the button function would only trigger one action, and not multiple actions despite the frequent sampling rates.

All these functions were validated with the use of LED lights, different LED lights corresponded to different outputs. This was the easiest way for us to validate that the display was displaying the results from the correct mode.

E. NVIC

The NVIC (Arm 2007) vectored interrupt system was only usable because we did not have too many peripherals. The vectored interrupts operate much quicker than regular interrupts, and as a result, were ideal for the performance of this low-peripheral system.

F. Display

Our display showed our voltage up to 3 digits (2 decimal places). As all 3 LEDs shared the same 7-segment decoder, we had to interleave the display of the digits so that it appeared that all 3 digits were displayed at the same time to the user. We did so by displaying each of the digits for a very short period of time. This implementation was done by adding a for loop that iterated up to 5000 to make it visible.

G. SysTick

The system counts from 1 to 2^{24} before sampling, where 2^{24} is the equivalent of 168MHz (Which is the rate of the board). As a result, if we wanted to sample 50 times in a second, we would simply divide that number (168MHz) by 50, so that it would count from 1 to $\frac{2^{24}}{50}$ before sampling, allowing it to sample 50 times per second.

To test this, we applied breakpoints to the sampling points and verified that the system did indeed sample every 0.02 seconds (which would be equivalent to sampling 50 times in a second.

H. FIR Filter and C_MATH

The implementation of the FIR filter (Cetin et. al 1997) is mostly detailed in the previous section. However, one important thing to note is that we counted 500 sampling cycles before we updated the minimum and maximum values so that we could meet the functional requirements of this experiment by updating those values every 10 seconds.

IV. Conclusion

In this experiment, we developed a system that would output a analog voltage through a DAC converter, then re-read that analog voltage using an ADC converter. We then pass the read the data through an FIR filter as there is some noise that would be interpreted through the ADC converter. We then extract the mean, max, and RMS values in this data which are displayed through a 7-segment decoder, where the 3 modes are controlled through a blue button. The ADC and the blue button interrupts had to share the same bus, and were therefore controlled through the use of flags and debouncing in order to ensure that neither interrupt perpetually occupied the bus.

References

[1] A. E. Cetin, O.N. Gerek, Y. Yardimci, "Equiripple FIR filter design by the FFT algorithm," IEEE Signal Processing Magazine, pp. 60-64, March 1997.

[2] Cartwright, Kenneth V. "Determining the Effective or RMS Voltage of Various Waveforms without Calculus". Technology Interface. 8 (1): 20 pages, Fall 2007

[3] Walden, R. H. . "Analog-to-digital converter survey and analysis". IEEE Journal on Selected Areas in Communications. 17 (4): 539550 1999.

[4] ARM, Application Note 179 - Cortex M3 Embedded Software Development, http://infocenter.arm.com/help/topic/com.arm.doc.dai0179b/AppsNote179.pdf, March 2007

[5] Jonathan Corbet; Alessandro Rubini; Greg Kroah-Hartman (2005). "Linux Device Drivers, Third Edition, Chapter 10. Interrupt Handling", Dec 2014

[6] Dennis Dempsey and Christopher Gorman, "Digital-to-Analog Converter," U.S. Patent 5,969,657, filed July 27, 1997, issued October 19, 1999.

Appendix A
System Level Block Diagram

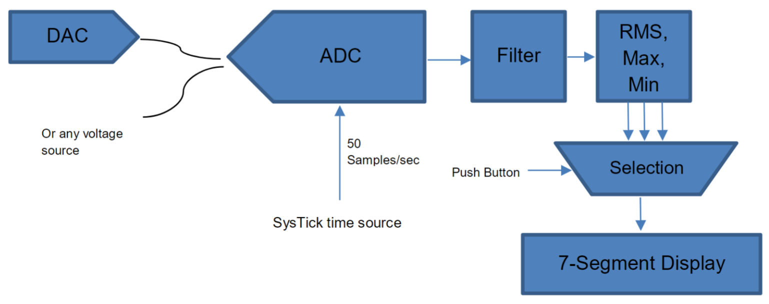

Fig. 3. System Level Block Diagram of System

Appendix B
Class Diagram

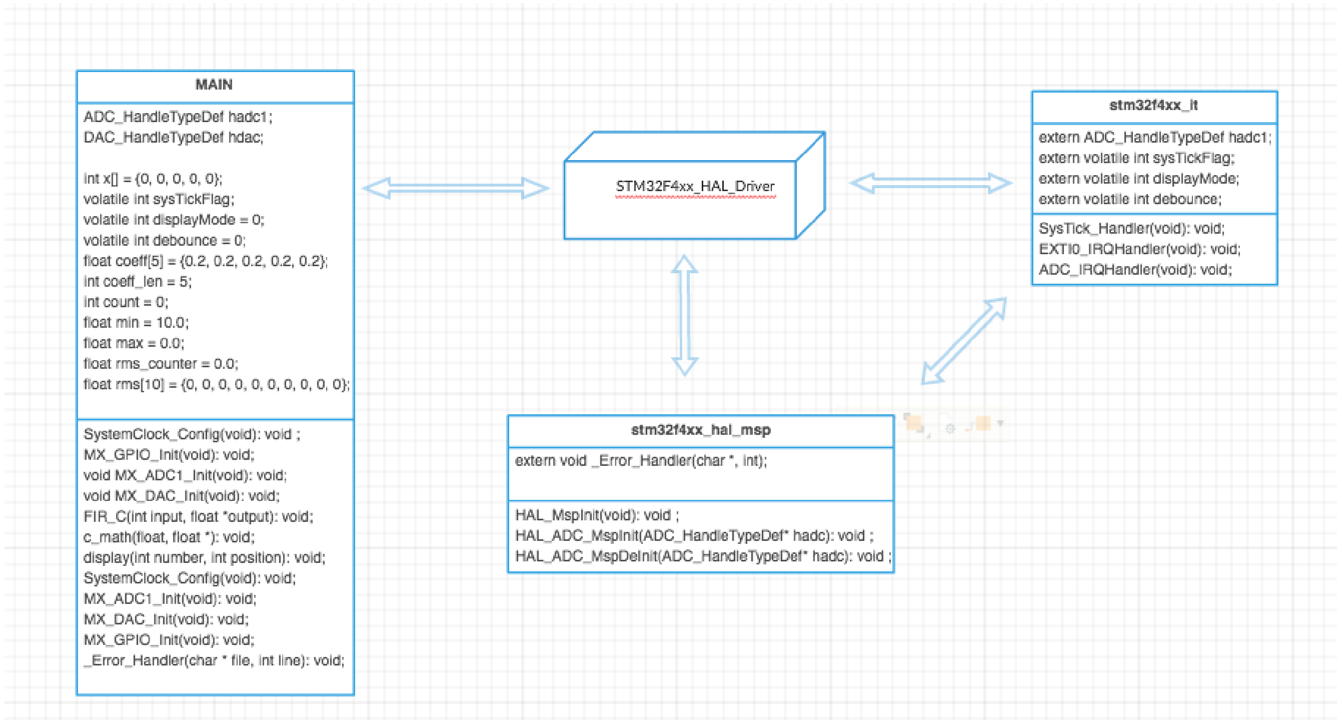
MAIN
ADC_HandleTypeDef hadc1;
DAC_HandleTypeDef hdac;
int x[] = {0, 0, 0, 0, 0};
volatile int sysTickFlag;
volatile int displayMode = 0;
volatile int debounce = 0;
float coeff[5] = {0.2, 0.2, 0.2, 0.2, 0.2};
int coeff_len = 5;
int count = 0;
float min = 10.0;
float max = 0.0;
float rms_counter = 0.0;
float rms[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
SystemClock_Config(void): void ;
MX_GPIO_Init(void): void;
void MX_ADC1_Init(void): void;
void MX_DAC_Init(void): void;
FIR_C(int input, float *output): void;
c_math(float, float *): void;
display(int number, int position): void;
SystemClock_Config(void): void;
MX_ADC1_Init(void): void;
MX_DAC_Init(void): void;
MX_GPIO_Init(void): void;
_Error_Handler(char * file, int line): void;
STM32F4xx_HAL_Driver
stm32f4xx_it
extern ADC_HandleTypeDef hadc1;
extern volatile int sysTickFlag;
extern volatile int displayMode;
extern volatile int debounce;
SysTick_Handler(void): void;
EXTI0_IRQHandler(void): void;
ADC_IRQHandler(void): void;
stm32f4xx_hal_msp
extern void _Error_Handler(char *, int);
HAL_MspInit(void): void ;
HAL_ADC_MspInit(ADC_HandleTypeDef* hadc): void ;
HAL_ADC_MspDeInit(ADC_HandleTypeDef* hadc): void ;

Fig. 4. Class Diagram

Appendix C
Class Diagram

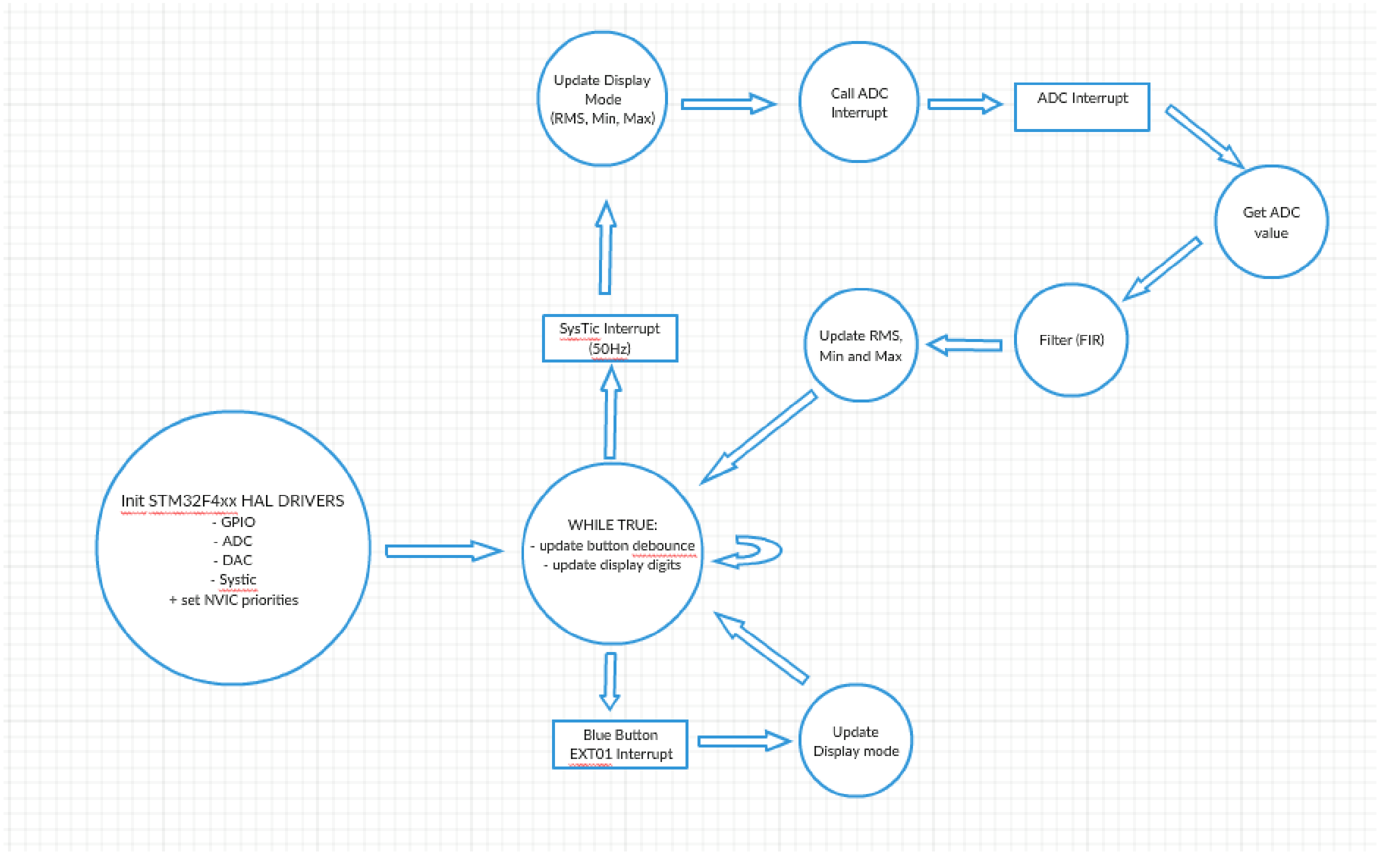

Fig. 5. State Diagram