# Microprocessor Systems Lab 4 Report

Weiheng Zhuang  Ziyu Zhu

## Table of Contents

# Task 1: Simple Voltmeter

## Introduction

A circuit that converts an analog signal into a digital signal is called an analog-to-digital converter (referred to as A/D converter or ADC, Analog to Digital Converter). The function of A/D conversion is to convert analog signals with continuous time and amplitude. It is converted into a digital signal with discrete time and discrete amplitude. Therefore, A/D conversion generally goes through four processes of sampling, holding, quantization and encoding. In practical circuits, some of these processes are combined, for example, sampling and holding, quantization and encoding are often implemented simultaneously in the conversion process. For the ADC function we want to implement on STM32F769NI, we mainly need another part, one of which is the sampling part, and the second is the conversion part.

The STM32F769NI has 3 built-in 12-bit successive approximation ADCs with up to 19 channels. Sampling of multiple channels in succession is done through channel grouping. There are two kinds of channels groups: regular and injected. A regular group is a set of channels that are set and used whenever a program calls for a conversion, while an injected group is similar to an interrupt: an external trigger causes the injected group to interrupt the conversion of a regular group.

The first task we have to do is to use the ADC(Port A3) to read the voltage at the interface and convert it into the digital circuit we need and present it on the terminal. Since the button is used to determine the timing of sampling, we actually want to use the inject mode. When the conversion formula is given, we can start to complete the lab. Basically, we first set the ADC, GPIO, and some other settings, and then we can start to call the internal functions of the ADC to achieve our task requirements.

## Low Level Description

First we need to initialize the ADC. The first step is to turn on the clock to the ADC. We use ADC3 here, so Instance is naturally ADC3. According to the requirements of the experimental document, Resolution is set to 12bit. Set ScanConvMode to disable, we don't need it here, then set ContinuousConvMode to enable. NbrOfConversion is set to 1, and we need the End Of Conversion flag to be set to ADC_EOC_SEQ_CONV. ExternalTrigConv is set to ADC_SOFTWARE_START. Finally we need to implement our initialization with HAL_ADC_Init(). Then there is the initialization setting for the channel of the ADC. The PF10 interface on our board provides channel 8 of the ADC, and we use this channel. Because we only use this ADC, set Rank to 1. SamplingTime is set to 56CYCLES after our repeated testing. After setting the ADC channel, use the function HAL_ADC_ConfigChannel binding to set our enabled ADC3.

```
ADC_ChannelConfTypeDef chanconfig;
HAL_ADC_ConfigChannel(&ADC_h3, &chanconfig);
```

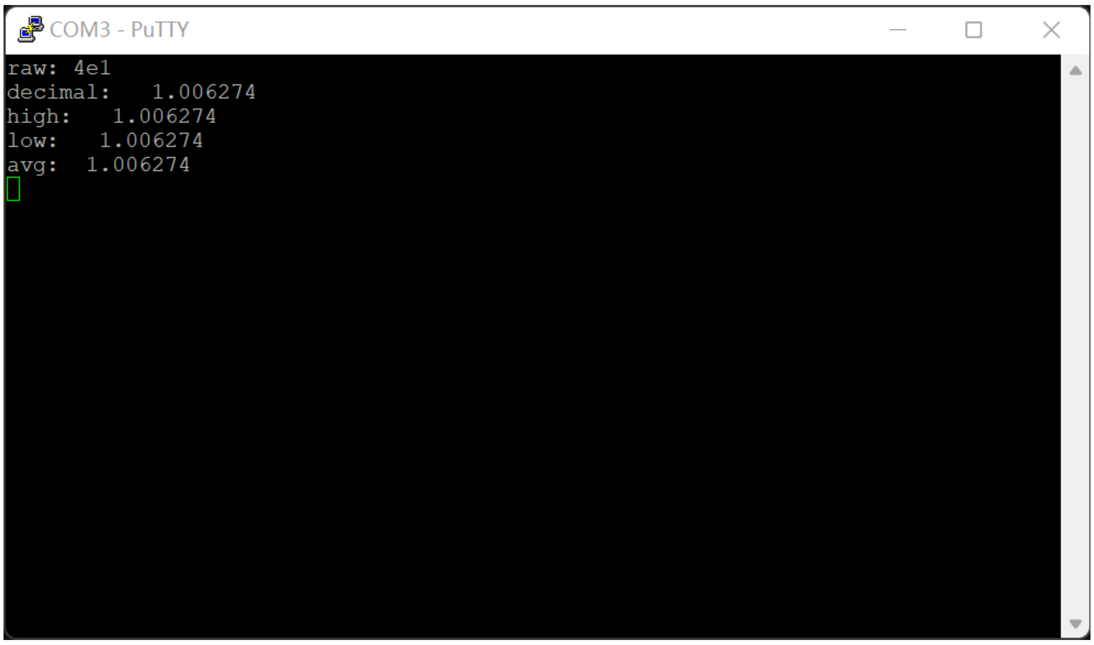
The next step is to set the GPIO interface and initialize the GPIO interrupt. Because we want to use the button, we also have to open the interface of the button at the same time. The interface of the ADC is PF10, and the interface of the button is A0. Then we need to use the EXTI0 used by lab2 to set the interrupt of the button. In the interrupt function of HAL_GPIO_EXTI_Callback, we need to read the value of the digital circuit returned by the ADC after conversion. That's what's going to be written next.

For the ADC, the first step would definitely be to turn on the ADC and get him to work. After checking the manual, we will call the HAL_ADC_Start() function to start the ADC. Then we need to wait for the conversion of ADC to complete, we need to call HAL_ADC_PollForConversion() to wait for the conversion to end. The main thing is that a timeout value is needed here, and we need to continue to look at the manual. When set at 12bit resolution, it takes a minimum of 0.5μs and a maximum of 16.4μs. Based on this information, we can continue to use the 1000 value used in the previous lab. Finally, after waiting for the conversion to finish, we need to call the function HAL_ADC_GetValue() to get the final value we want.

After getting the value converted by the ADC, we also need to convert it into a decimal number that we can understand. The formula in the lab file is used here. Because we also need the max, min and mean. So we also need 3 variables to store the maximum value, the minimum value, and the total amount. In the end, we just need to print out these values to complete the task.

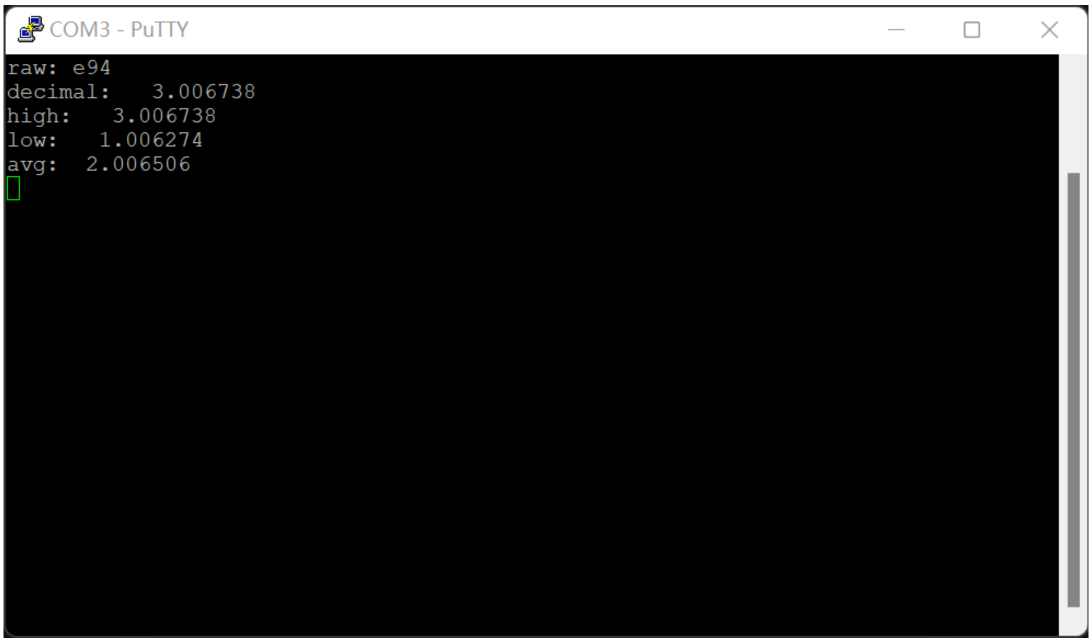
## Results and Analysis

The first and second screenshots show that the normal running sequence from terminals.



```
raw: 4e1
decimal:    1.006274
high:    1.006274
low:    1.006274
avg:   1.006274
```

Figure 1: input 1V voltage



```
raw: e94
decimal:    3.006738
high:    3.006738
low:    1.006274
avg:   2.006506
```

Figure 2: input 3V voltage

# Task 2: DAC Output

## Introduction

Digital-to-analog conversion (D/A converter) is the bridge between the computer acquisition control system and the analog control object. The role of the D/A converter is to convert discrete digital signals into continuously changing analog signals. The basic principle of this converter is to sample the input analog signal at a specified time interval and compare it with a series of standard digital signals. The digital signals converge successively until the two signals are equal. Then display the binary number representing this signal. There are many kinds of analog-to-digital converters, such as direct, indirect, high-speed and high-precision, ultra-high-speed and so on. Each has many forms. The opposite function of the analog-to-digital converter is called "digital-analog converter", also known as "decoder", which is a device that converts digital quantities into continuously changing analog quantities, and there are many kinds and many forms. Simply put, DAC is the inverse of ADC.

The STM32F769NI has 2 built-in 12-bit DACs with 1 output channel each. Operation and configuration of the DACs is much less flexible than the ADCs.DACs and ADCs are very important in some applications that process digital signals. The intelligibility or fidelity of the analog signal can be improved by converting the analog input signal to digital form using an ADC, then the digital signal is "cleaned" and the final digital pulse is reconverted to analog using a DAC.

## Low Level Description

The first is to set the DAC. The DAC has more flexible settings than the ADC. The DAC settings basically correspond to the ADC we use. The first step in setup is always to turn on the clock. The default DAC resolution of stm32 is 12bit, so we don't need to set it. After setting Instance as DAC, the HAL_DAC_Init equation can be called directly. Then there is the setting of the DAC channel. Need to set DAC_Trigger to DAC_TRIGGER_NONE and DAC_OutputBuffer to DAC_OUTPUTBUFFER_ENABLE. After completing the above steps, use the HAL_DAC_ConfigChannel equation to complete the initial setup of the channel. The final initialization setting is GPIO. After consulting the manual, we know that the DAC interface is A4.

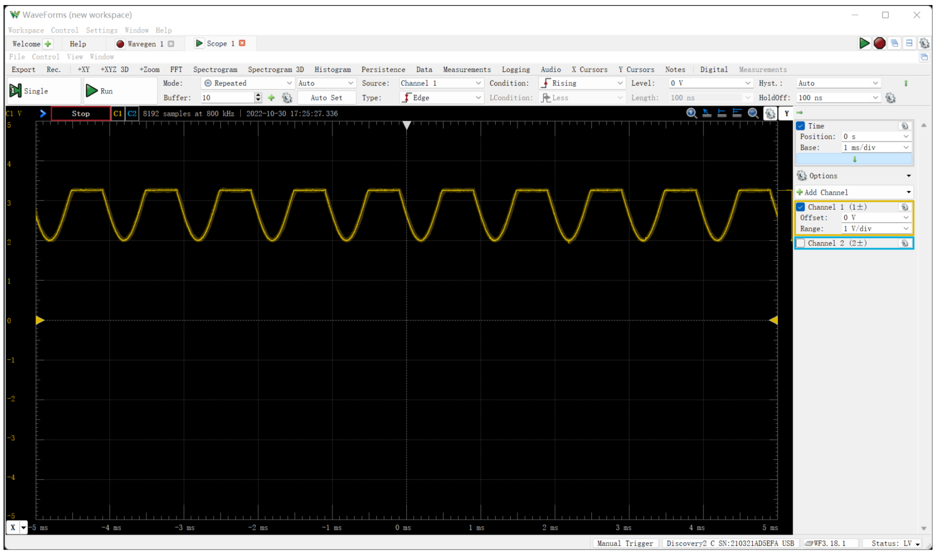
```
    HAL_DAC_ConfigChannel(&dac1, &dacctd, DAC_CHANNEL_1);
```

Then we came to the writing of the main function. First we call the ADC to sample and convert the voltage at the interface and convert it into a digital signal. Then use the DAC function HAL_DAC_SetValue() to convert the digital signal to an analog signal. It is worth noting that because the conversion of the ADC is not continuous, the final conversion result will also look a little stiff.

```
read_value =  checkvoltage();
HAL_DAC_SetValue(&dac1, DAC_CHANNEL_1, DAC_CHANNEL_1, read_value);
```

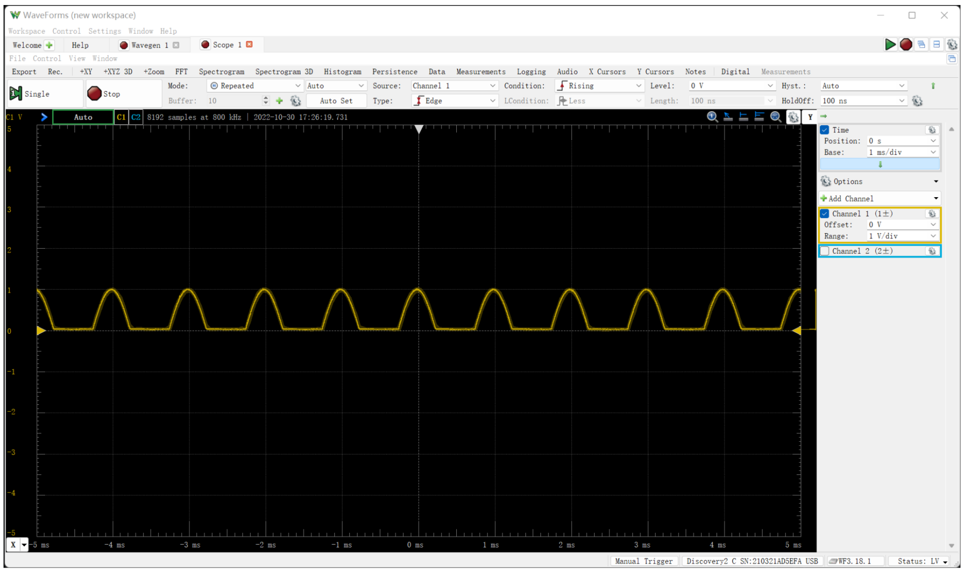
## Results and Analysis

The first and second screenshots show that the cut-off of DAC. The third shows the normal running sequence.



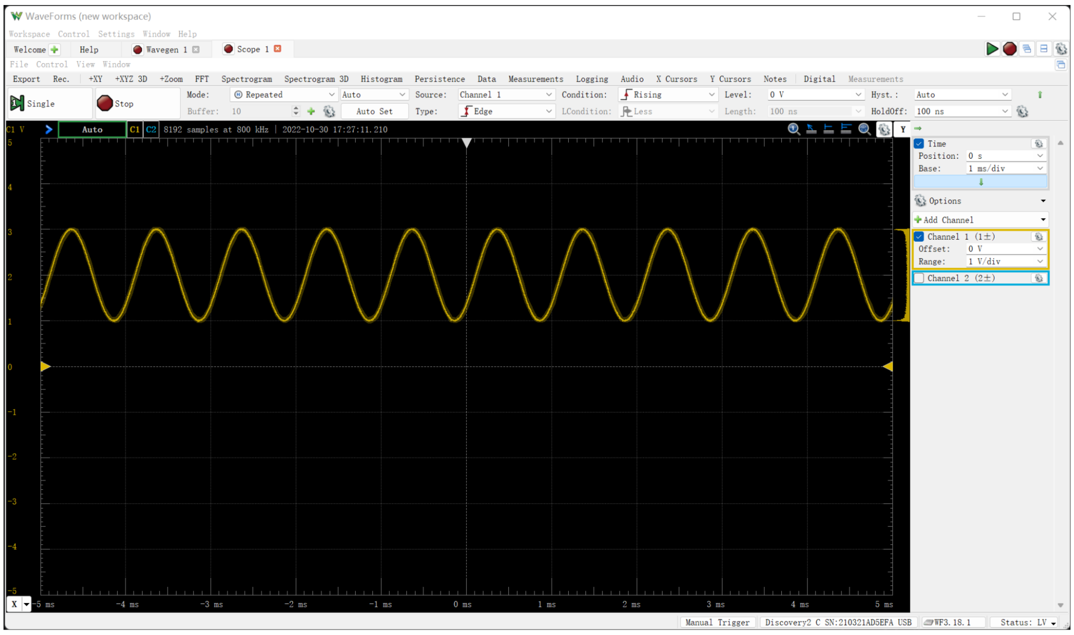
```
WaveForms (new workspace)
Mode: Repeated  Auto    Source: Channel 1   Condition: Rising   Level: 0 V        Hyst.: Auto
Buffer: 10      Auto Set  Type: Edge        LCondition: Less    Length: 100 ns    HoldOff: 100 ns
Stop  C1 C2  8192 samples at 800 kHz | 2022-10-30 17:25:27.336
Time  Position: 0 s  Base: 1 ms/div
Channel 1 (1±)  Offset: 0 V  Range: 1 V/div
Channel 2 (2±)
Manual Trigger  Discovery2 C SN:210321AD5EFA USB  WF3.18.1  Status: LV
```

Figure 3: Upper filter of 3.26V



```
WaveForms (new workspace)
Mode: Repeated  Auto    Source: Channel 1   Condition: Rising   Level: 0 V        Hyst.: Auto
Buffer: 10      Auto Set  Type: Edge        LCondition: Less    Length: 100 ns    HoldOff: 100 ns
Auto  C1 C2  8192 samples at 800 kHz | 2022-10-30 17:26:19.731
Time  Position: 0 s  Base: 1 ms/div
Channel 1 (1±)  Offset: 0 V  Range: 1 V/div
Channel 2 (2±)
Manual Trigger  Discovery2 C SN:210321AD5EFA USB  WF3.18.1  Status: LV
```

Figure 4: Lower filter of 0V


WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Wavegen 1 Scope 1
File Control View Window
Export Rec. +XY +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Data Measurements Logging Audio X Cursors Y Cursors Notes Digital Measurements
Single
Stop
Auto Set
Options
Add Channel
C1 V
C1
C2
5
4
3
2
1
0
-1
-2
-3
-4
-5
X
-5 ms
-4 ms
-3 ms
-2 ms
-1 ms
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
Y

Mode: Repeated Auto
Buffer: 10
Source: Channel 1
Type: Edge
Condition: Rising
LCondition: Less
Level: 0 V
Length: 100 ns
Hyst.: Auto
HoldOff: 100 ns

Auto | 8192 samples at 800 kHz | 2022-10-30 17:27:11.210

Time
Position: 0 s
Base: 1 ms/div

Channel 1 (1±)
Offset: 0 V
Range: 1 V/div
Channel 2 (2±)

Manual Trigger | Discovery2 C SN:210321AD5EFA USB | WF3.18.1 | Status: LV

Figure 5: Normal running sequence

# Task 3: Simple Assembly Math

## Introduction and High-Level Description

Assembly Language is any low-level language used in electronic computers, microprocessors, microcontrollers or other programmable devices, also known as symbolic languages. In assembly language, the opcodes of machine instructions are replaced by mnemonics, and the addresses of instructions or operands are replaced by address symbols or labels. In different devices, assembly language corresponds to different machine language instruction sets, and is converted into machine instructions through the assembly process. There is a one-to-one correspondence between a specific assembly language and a specific machine language instruction set.

Assembly language is the fastest and most efficient language provided by the computer to the user, and it is also the only language that can utilize all the hardware features of the computer and can directly control the hardware. But because writing and debugging assembly language programs is more complicated than high-level languages, its applications are not as extensive as high-level languages.

Assembly language is more readable than machine language, but still less readable than high-level languages. However, the programs written with it have the characteristics of less storage space and faster execution, which cannot be replaced by high-level languages. In practical applications, whether to use assembly language depends on the trade-offs of specific application requirements, development time, and quality.

Task 3 of this lab requires us to use simple mathematical operations of assembly speech. This is basically an introductory task for assembly language. Reading assembly speech is also a necessary skill as a hardware engineer. In the following task 4, the practical application scenarios of assembly language mathematical operations will also be guided, but at present we only need to use it for simple operations in task 3.

## Low Level Description

### Load and add

We need to first give each of the two registers a number, then add them up, and print the final result. Because our printf function cannot print the value of the assembly register, at the end of the operation, we need to assign the value of the register to the variable of our c code. The specific operation is to use the LDR instruction to give two register numbers. Then use the instruction ADD to add them, and then call the STR instruction to assign the value in the register to the variable of our c code. As shown below:

```
    asm("LDR r1, = 34");
    asm("ADD r2, r0, r1");
    asm("STR r2,%0" : "=m" (var));
```

### Multiply 2 int32_t variables

The basic flow is the same as "Load and add" above, but the MUL instruction will be used instead of the ADD instruction. A method similar to formatting is used here, that is, 0% represents the first filled variable to be assigned, 1% represents the second filled variable, and so on.

```
    asm("MUL %0,%1,%2" : "=r" (var): "r" (variable_0), "r" (variable_1));
```

### Multiply 2 single precision floats

There are two main differences between the floating-point version of multiplication and the integer version. The difference is that the instructions are different, the instructions for floating point are VMUL instead of integer MUL, because we need single precision floating point, so I use F32, and double precision requires F64.

```
asm("VMUL.F32 %0,%1,%2" :"=t" (f_var): "t" (variable_0), "t" (variable_1));
```

### Evaluate the equation through 32-bit integer math

This is actually a summary task for addition and multiplication, and the basic processes and techniques have also appeared above. First store the number 2 into register r2. The result of multiplying the value of r2 by the value in x is placed in register r0. Then store the number 3 into register r3. Then divide the value in r0 by the value in r3 and place the result in register r1. Finally, use the ADD instruction to add 5 to r1 and assign the value to the variable of the c code, and finally print it out

```
    asm("ADD %0,r1, #5":"=r" (var));
```

## Evaluate the equation through floating point addition

The floating-point part is almost exactly the same as the integer. Basically, the ADD instruction is replaced by VADD, the LDR instruction is replaced by VMOV, and VMIL and VDIV.

```
    asm("VMOV s5, #5.0");
    asm("VADD.F32 %0, s1, s5": "=t" (f_var));
```

## Evaluate the previous equation using integer MAC commands

The MAC instruction is an MLA instruction in assembly. This instruction simply integrates addition and multiplication. The second register of the MLA instruction is the multiplicand, the third register is the multiplier, and the fourth register is the number to be added. After understanding how MLA works, let's take another look at our equation. Since we are multiplying by 2, dividing by 3 and adding 5 at the end, it is obvious that integer MLA cannot be applied directly, so we can multiply the whole formula by 3 and divide by 3 at the end. The equation becomes the sum of 2 times x add 15 divide it by 3. Now we can use the MLA instruction first, and then the DIV instruction to get the answer.

```
    asm("MLA r0,%0,r2,r1" :: "r" (x));
    asm("LDR r3, =3");
    asm("UDIV %0, r0, r3": "=r" (var));
```

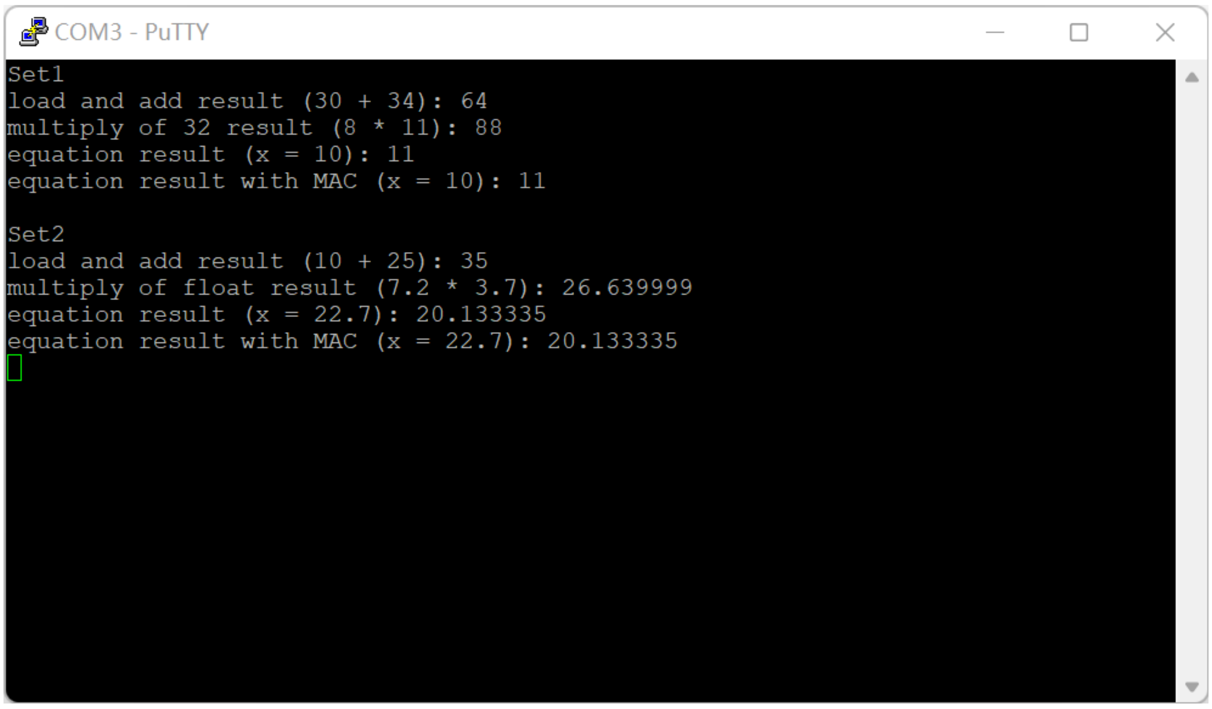
## Evaluate the previous equation using floating point MAC commands

Since it is a floating-point operation, we can directly multiply x by two-thirds, and the other processes are no different from the previous ones.

```
    asm("VMLA.F32 s3,%0,s2" :: "w" (x));
```

## Results and Analysis

The program would echo a message as expected.



```
Set1
load and add result (30 + 34): 64
multiply of 32 result (8 * 11): 88
equation result (x = 10): 11
equation result with MAC (x = 10): 11

Set2
load and add result (10 + 25): 35
multiply of float result (7.2 * 3.7): 26.639999
equation result (x = 22.7): 20.133335
equation result with MAC (x = 22.7): 20.133335
```

Figure 6: Output of Task3

# Task 4: IIR Filter Implementation

## Introduction and High-Level Description

Task 4 combines Task 2 and Task 3 to produce a IIR(infinite impulse filter), acting upon a received signal from the ADC and outputting the filtered signal using the DAC. The filter to be implemented by the following transfer function.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\frac{10}{32}(z + \frac{5}{13} + j\frac{12}{13})\,(z + \frac{5}{13} - j\frac{12}{13})}{z\,(z - \frac{19}{20} \cdot \frac{10}{32})}$$

which leads to the difference equation.

$$y(k) = \frac{10}{32}\,(x(k) + \frac{10}{13}x(k-1) + x(k-2) + \frac{19}{20}y(k-1))$$

where x(k) is the input from the ADC at sample time k, x(k − 1) is the previous sample.

The infinite impulse response is a type of digital filter that is used in Digital Signal Processing applications. A filter's job is to allow certain types of signals to pass and block the rest. The infinite impulse response filter is unique because it uses a feedback mechanism. It requires current as well as past output data. An impulse response $h(t)$ which does not become exactly zero past a certain point, but continues indefinitely. Though they are harder to design, IIR filters are computationally efficient and generally cheaper.

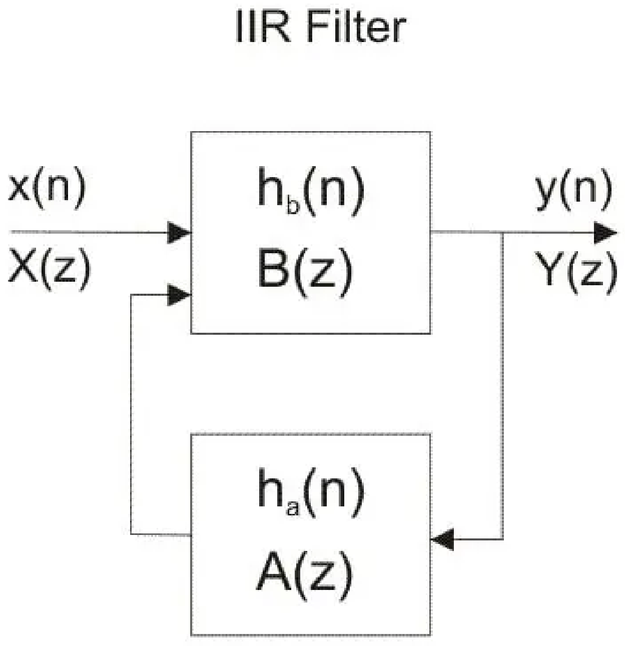


Figure 7. IIR Filter

In Task 4, the program should initialize both ADC and DAC, then sample the input sine wave. After sampling, put the current and previous sample result into the difference equation to calculate the current responce.

The wiring of this task is the same as Task 2. The only difference is that the Analog Discovery should read the input signal on the oscilloscope, so the input connects to both ADC and channel 1 on Analog Discovery.

## Low Level Description

The calculation for every sample in normal calculation is this

```
    sum = 0.3125 * x + 0.240385 * xminus1 + 0.3125 * xminus2 + 0.296875
     * yminus1;
```

In assembly, most of the calculation is the same, but it should be noted that the type of the variable may cause problems. So, while multiplying, the uint16_t typed variable should be changed into float to avoid errors.

```
    asm("VMUL.F32 s0, %0, s1" ::"t"((float)x));
```

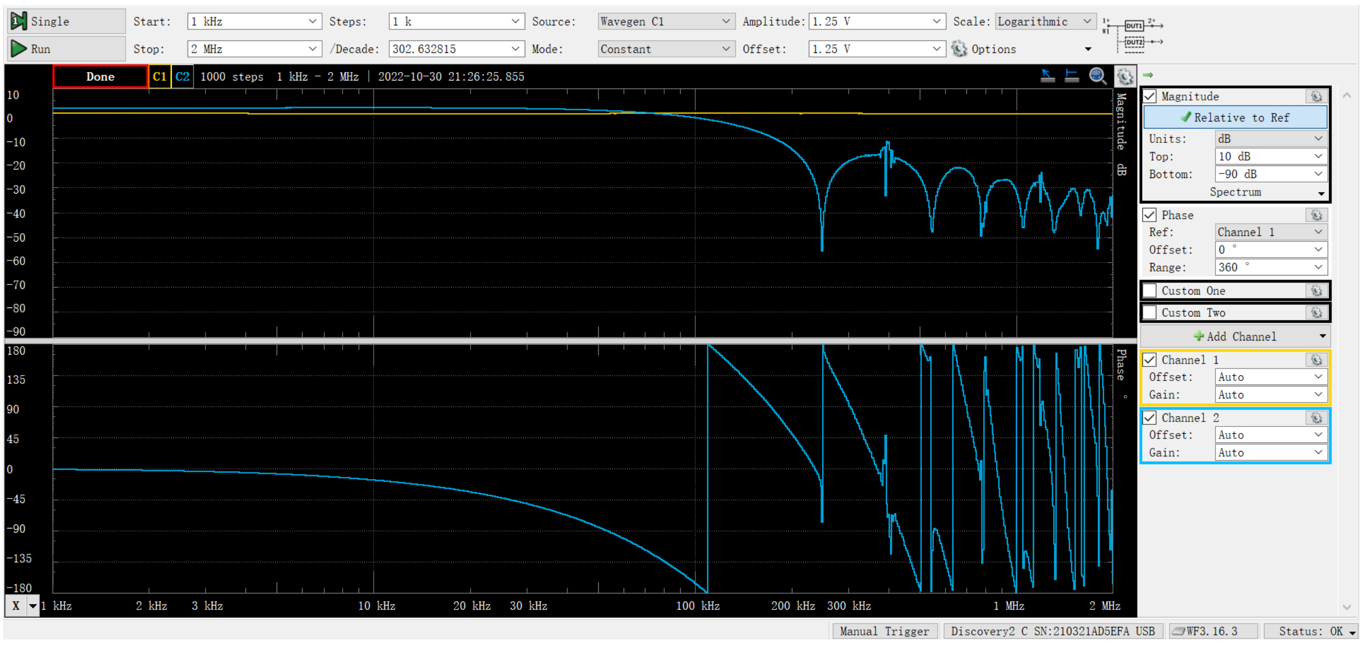
To calculate the difference equation, the previous result should be stored. In this Task, after 3 samples are taken, the first result can be calculated. So, before the infinite loop, there should be 3 sampling and voltage recorded.

```
    xminus2 = checkvoltage();
    xminus1 = checkvoltage();
    x = checkvoltage();
```

After storing the read voltage, put all the values into the filter, then output the value into the DAC function, as mentioned in Task 2. The type output the output value should also be uint16_t, but not float in the calculation.

```
    HAL_DAC_SetValue(&dac1, DAC_CHANNEL_1, DAC_CHANNEL_1, (uint16_t)yminus1);
```

## Results and Analysis



Start: 1 kHz
Stop: 2 MHz
Steps: 1 k
/Decade: 302.632815
Source: Wavegen C1
Mode: Constant
Amplitude: 1.25 V
Offset: 1.25 V
Scale: Logarithmic
Options

Done C1 C2 1000 steps 1 kHz - 2 MHz | 2022-10-30 21:26:25.855

Magnitude
Relative to Ref
Units: dB
Top: 10 dB
Bottom: -90 dB
Spectrum

Phase
Ref: Channel 1
Offset: 0 °
Range: 360 °

Custom One
Custom Two
Add Channel

Channel 1
Offset: Auto
Gain: Auto

Channel 2
Offset: Auto
Gain: Auto

Manual Trigger | Discovery2 C SN:210321AD5EFA USB | WF3.16.3 | Status: OK

Figure 8: IIR filter waveform

The filtered waveform of the sine wave is as expected. The filter notch is shown at 300 kHz.

# Task 5: [Depth] Frequency Mixer

## Introduction and High-Level Description

Task 5 is a more complex filter than Task 4. Instead of using one wave, Task 5 uses 2 sine wave signals with different frequencies. The triggering for the ADCs must be directly provided by a timer and not through code. The rate should be relatively fast (e.g., >100 kHz). Once the ADC samples are acquired, they should be “Mixed,” which is just multiplying them together. The filter to be implemented by the following transfer function.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0.001-0.002z^{-2}+0.001z^{-4}}{1-3.116z^{-1}+4.417z;^{-2}-3.028z^{-3}+0.915z^{-4}}$$

which leads to the difference equation.

$$y(k) = 0.001x(k) - 0.002x(k-2) + 0.001x(k-4) + 3.166y(k-1) - 4.418y(k-2) + 3.028y(k-3) - 0.915y(k-4)$$

where x(k) is the input from the ADC at sample time k, x(k – 1) is the previous sample.

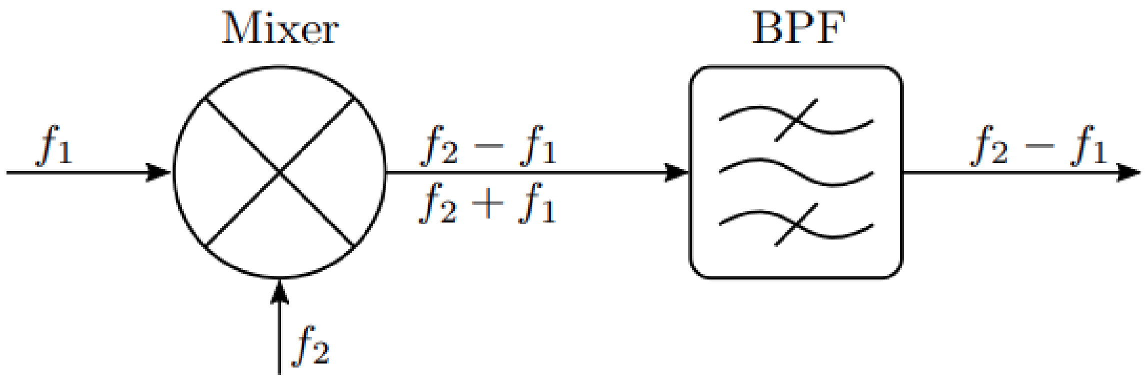


Figure 9. Mixer block diagram

In Task 5, the program should initialize 2 ADCs and DAC, then sample the input sine wave. Flags may be set/cleared in the ADC interrupts to track ADC completion. For example, if ADC2 completes first, a flag is set; then, ADC1’s interrupt is triggered, this flag is read and the filter is executed. After sampling, put the current and previous sample result into the difference equation to calculate the current responce.

The wiring of this task is simple. The 2 sine waves should be connected to 2 ADCs, while the DAC is connected to channel 1 of the Analog Discovery.

## Low Level Description

The filter calculations and cautions are the same as Task 4. The main difference is the interrupt of ADC.

In order to check the interrupt is triggered for each ADC, a variable is set in the program to check the status. It should start by sampling ADC1 and update itself to 1. Then ADC_IRQHandler should take charge and get the converted voltage of ADC1. Then in the callback function, the program should start sampling ADC3 and update itself to 2. Then ADC_IRQHandler of ADC3 should be triggered and get the value of ADC3. With both values of ADC recorded, the filter can be calculated and get the output.

```
    if (hadc->Instance == ADC1)
    {
        f1 = HAL_ADC_GetValue(&ADC_h1);
        adc_state = 2;
        HAL_ADC_Start_IT(&ADC_h3);
    }
    if (hadc->Instance == ADC3)
    {
        f2 = HAL_ADC_GetValue(&ADC_h3);
        adc_state = 3;
        filter();
    }
```

For the timer, the triggering for the ADCs must be directly provided by a timer and not through code. So DAC timer6 is used for this program. The interrupt of the timer is discussed in Lab2, with Prescaler set to 9 and Period set to 99. So the interrupt is 108kHz.

The DC biases of the input signals should be removed prior to multiplication in order to prevent excessively large values and additional frequency components. Add the DC biases back in for DAC transmit. In this program, the Dc bias is considered 1V.

## Results and Analysis

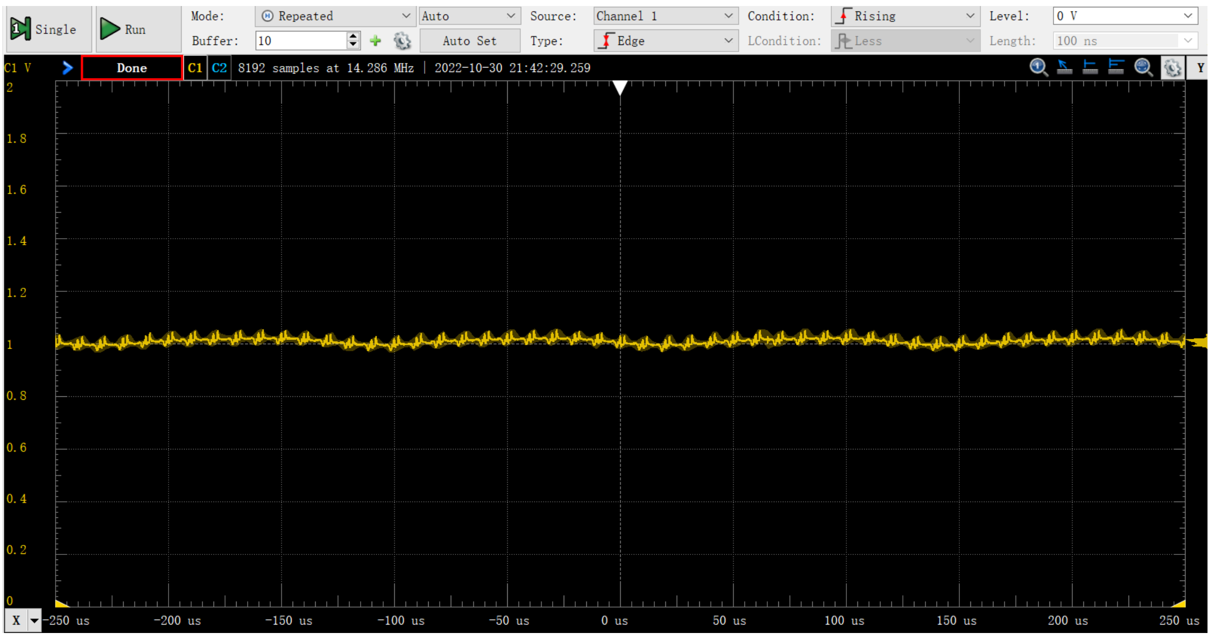


Figure 10. 100kHz and 100kHz mix

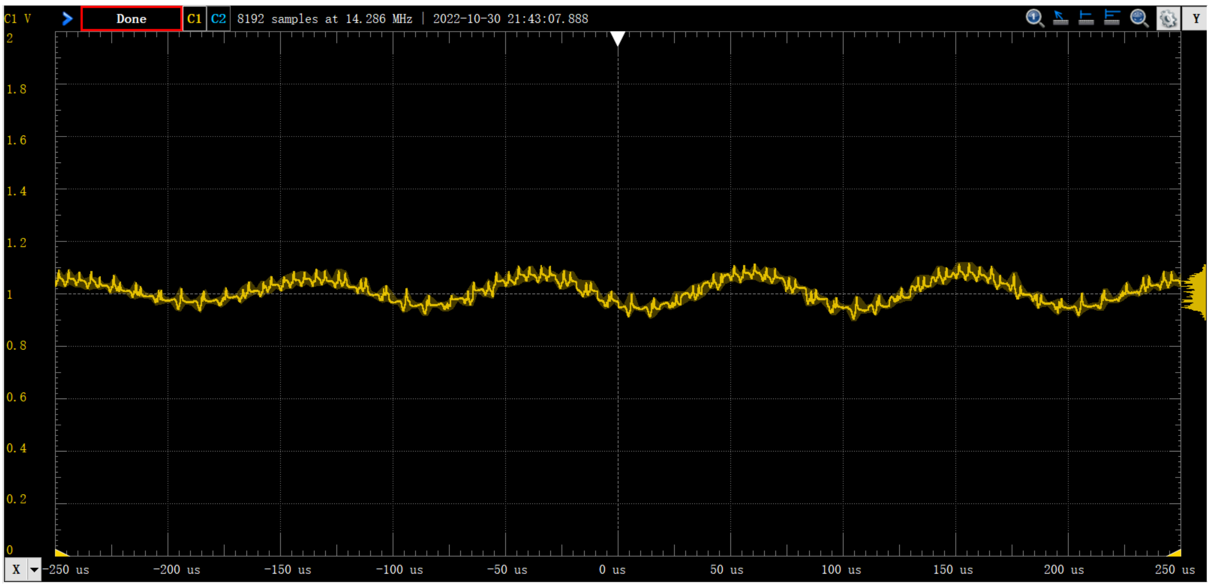


Figure 10. 100kHz and 110kHz mix

The result is as expected. When 100kHz and 100kHz sine wave are modulated, the filtered waveform can be considered flat. While 100kHz and 110kHz sine wave are modulated, the waveform frequency is changed.

# Conclusions

Through this lab, we focused on learning the example operations of ADC and DAC on STM32F769NI, as well as the basic writing and application of assembly speech.

## ADC & DAC

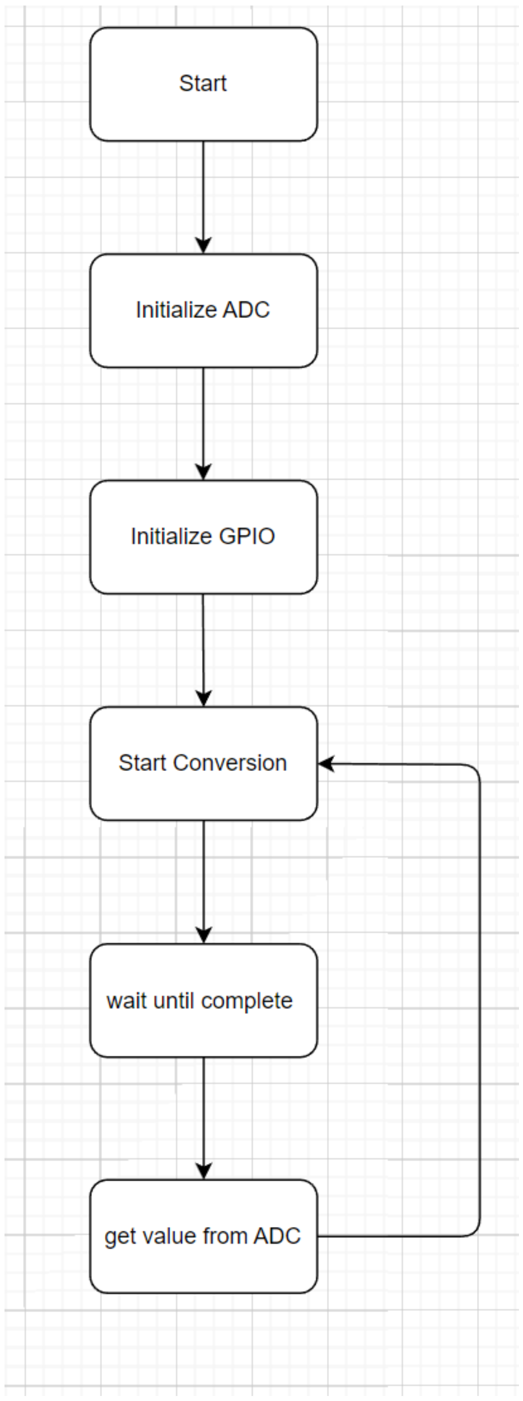
ADC is a kind of signal chain chip in analog chips, and from the perspective of sub-products, ADC is one of the most important signal chain products with a large market scale, and the market size of ADC/DAC accounts for the proportion of analog circuit market share. up to 15%. The computer cannot directly process the signal of sound. First, convert it into a digital signal that can be recognized by the computer, and then use the DAC in the sound card. It converts the sound signal into a digital signal. It needs to be sampled and converted in two steps. A digital-to-analog converter, a device that converts a digital signal into an analog signal.

## Assembly Language

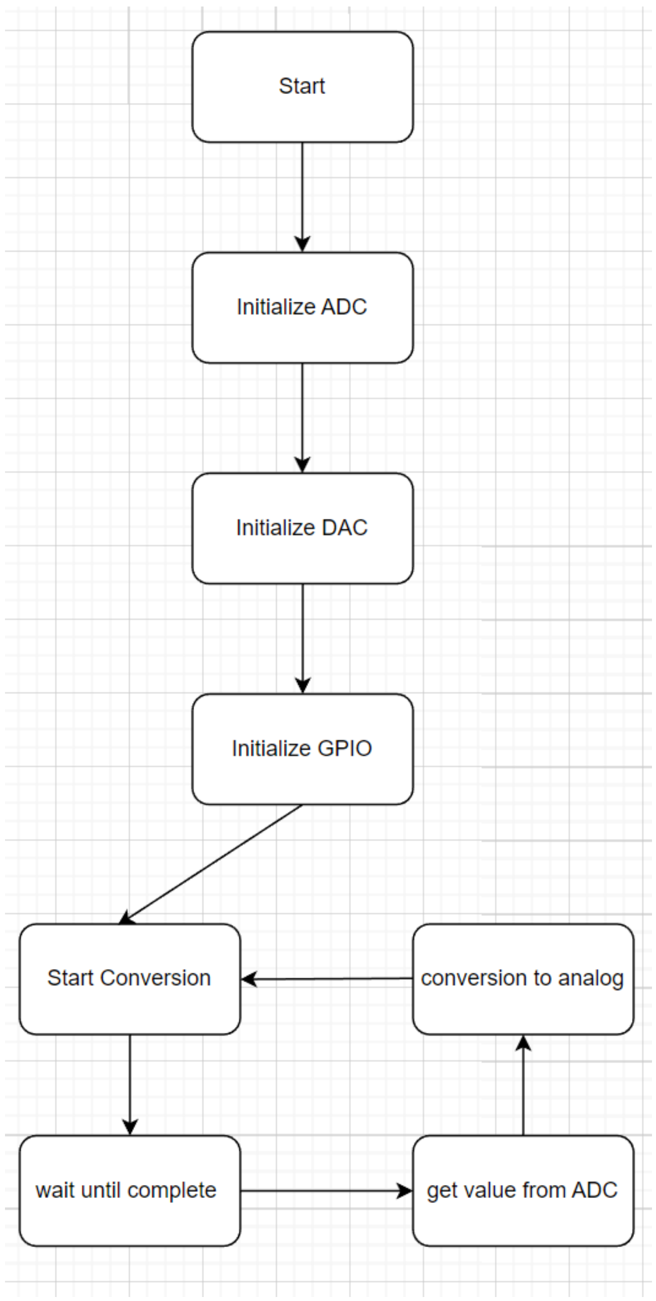
With the increasing complexity of modern software systems, a large number of packaged high-level languages such as C/C++, Pascal/Object Pascal also emerged. These new languages make the development process simpler and more efficient for programmers, enabling software developers to cope with the demands of rapid software development. However, assembly language has gradually reduced its scope of application due to its complexity. But that doesn't mean compilation is useless. Because assembly is closer to machine language and can operate directly on hardware, the generated program has higher running speed and takes up less memory than other languages. Therefore, in some programs that require high timeliness, many large-scale programs The core module of the program and a large number of applications in industrial control.

## Appendix

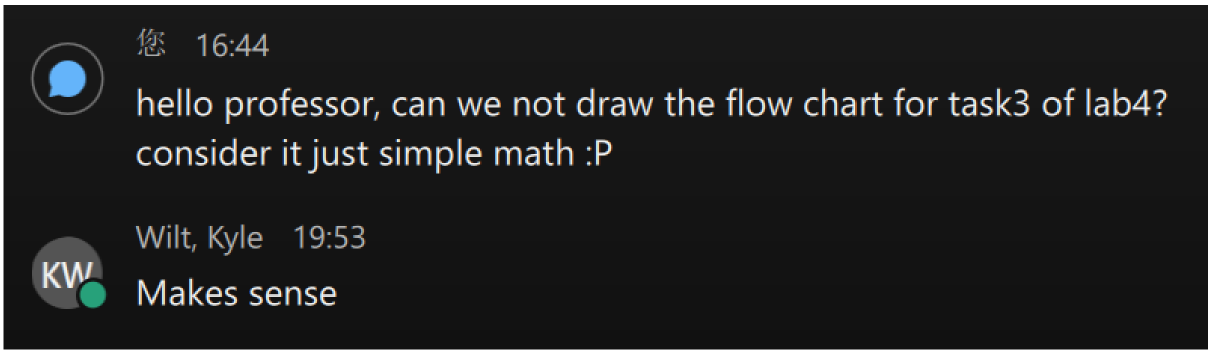
[1] Task 1 flowchart.



[2] Task 2 flowchart.



[3] Task 3 flowchart.



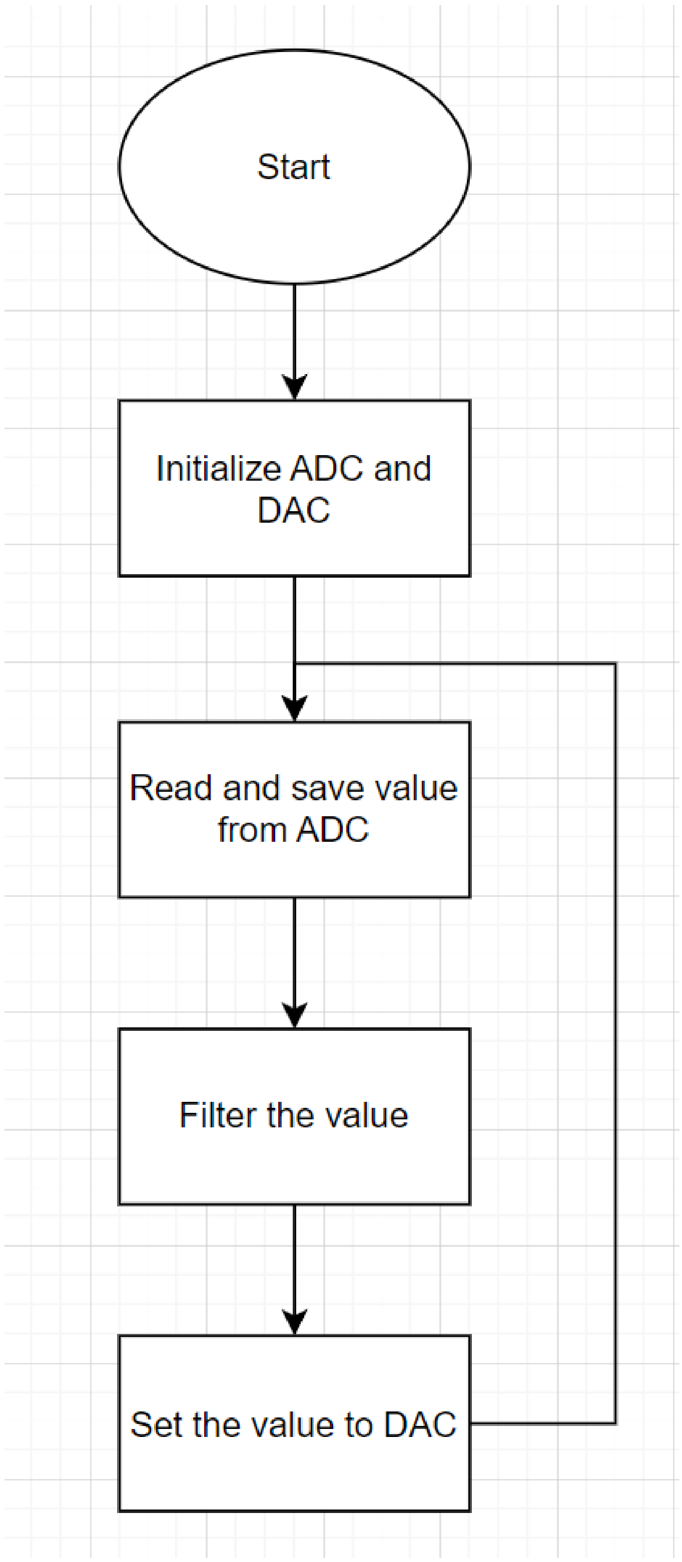
您 16:44

hello professor, can we not draw the flow chart for task3 of lab4? consider it just simple math :P

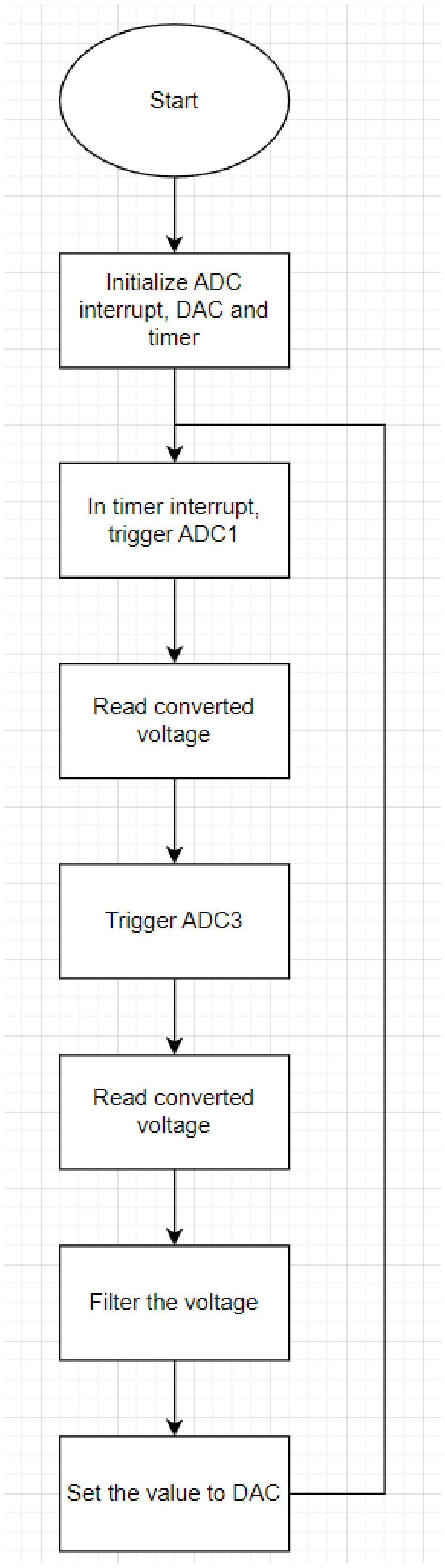
Wilt, Kyle 19:53

Makes sense

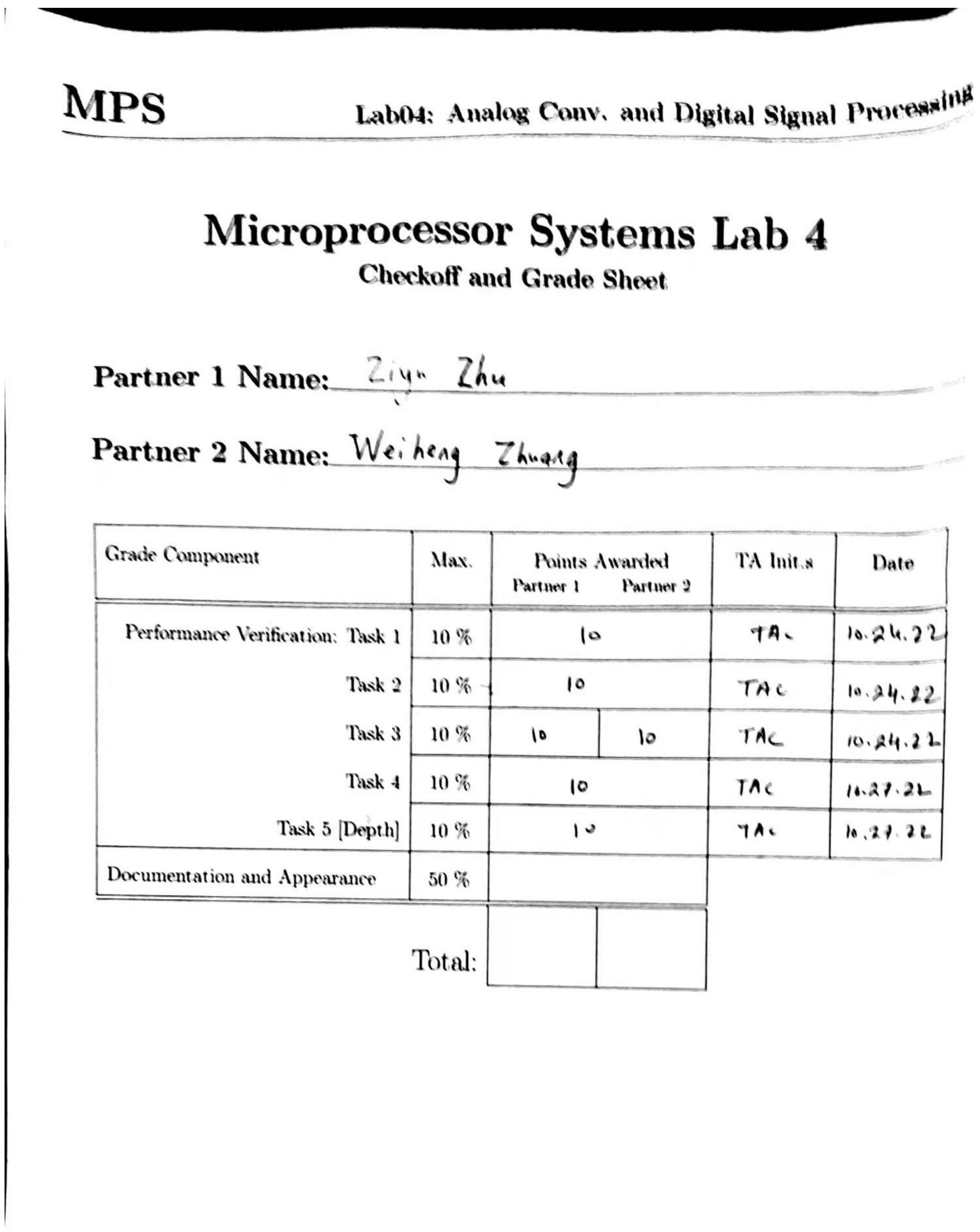
[4] Task 4 flowchart.



[5] Task 5 flowchart.



[6] Checkoff and Grade Sheet



MPS Lab04: Analog Conv. and Digital Signal Processing

# Microprocessor Systems Lab 4

**Checkoff and Grade Sheet**

**Partner 1 Name:** Ziyu Zhu

**Partner 2 Name:** Weiheng Zhuang

| Grade Component | Max. | Points Awarded Partner 1 | Points Awarded Partner 2 | TA Init.s | Date |
|---|---|---|---|---|---|
| Performance Verification: Task 1 | 10 % | 10 | | TAC | 10.24.22 |
| Task 2 | 10 % | 10 | | TAC | 10.24.22 |
| Task 3 | 10 % | 10 | 10 | TAC | 10.24.22 |
| Task 4 | 10 % | 10 | | TAC | 10.27.22 |
| Task 5 [Depth] | 10 % | 10 | | TAC | 10.27.22 |
| Documentation and Appearance | 50 % | | | | |
| Total: | | | | | |

1