Microprocessor Systems Lab 5 Report

Weiheng Zhuang Ziyu Zhu

Table of Contents

Task 1: DMA Performance Example

Introduction

Direct Memory Access, or DMA, is an extremely useful efficiency tool. In a nutshell, DMA modules allow for peripheral devices to transfer data to and from memory directly without intervention from the CPU, greatly speeding up memory operations and freeing up CPU cycles to be spent on other tasks. DMA transfers copy data from one address space to another. When the CPU initiates the transfer, the transfer itself is performed and completed by the DMA controller. A typical example is moving a block of external memory to a faster memory area inside the chip. Operations like this don't stall processor work, but can be rescheduled to handle other work. DMA transfers are important for high performance embedded system algorithms and networks. In circular mode, the DMA will transfer the requested amount of data then restart. When the circular mode restarts, both the source and destination addresses are reset to original and the same amount of requested data is retrieved again.

When implementing DMA transfer, the DMA controller directly manages the bus, so there is a problem of bus control transfer. That is, before the DMA transfer, the CPU should give the bus control to the DMA controller, and after the DMA transfer, the DMA controller should immediately return the bus control to the CPU. A complete DMA transfer process must go through four steps: DMA request, DMA response, DMA transfer, and DMA end.

The first task is done simply to show the benefits of using the DMA in moving data. In this task, the DMA should be configured to move data from one C buffer to another C buffer. This can be done by configuring DMA in normal mode and in the memory-to-memory direction.

Low Level Description

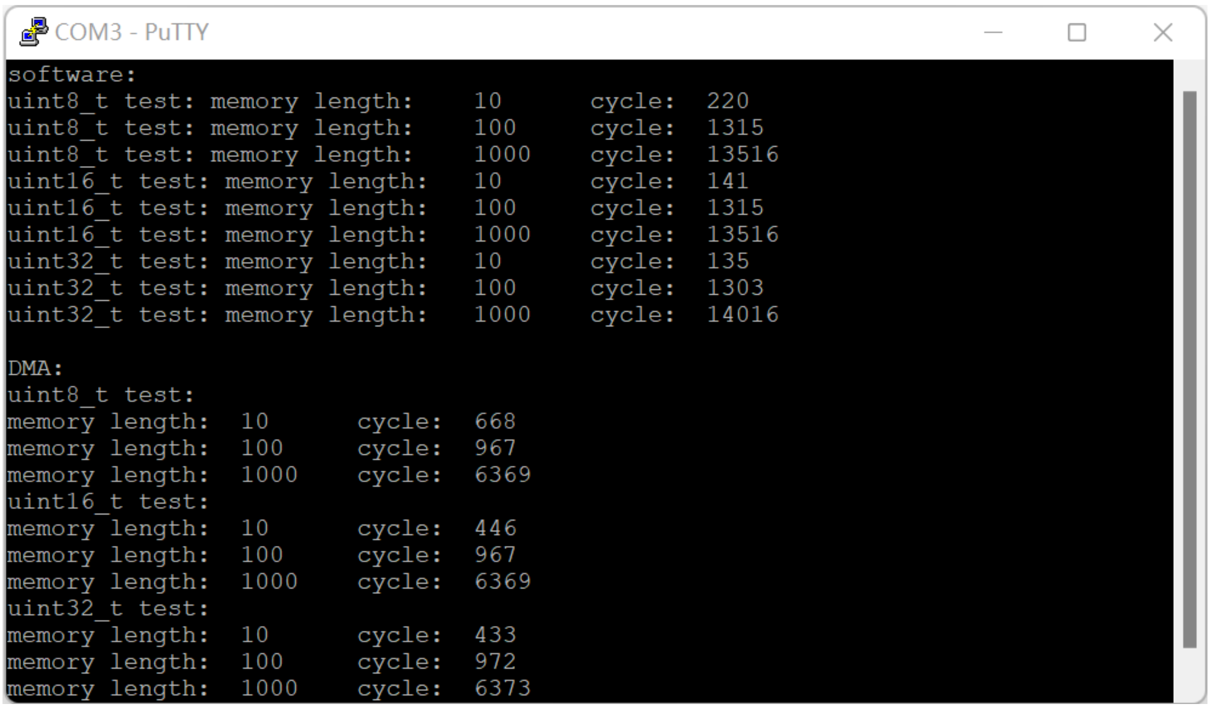
To complete this task, we first need to make some initial settings for the DMA. First create a global DMA_HandleTypeDef variable for the initial setup of DMA. The first step in the initial setup is always to turn on the clock. Then we need to check the instance and channel of the DMA to be used. Since we do not need these tools for buffer data transfer, but because we are using memory-to-memory transfer, we use channel0 of DMA2. Then set the direction to DMA_MEMORY_TO_MEMORY. Open MemInc for memory transfer data. Finally, use HAL_DMA_DeInit and HAL_DMA_Init to complete the initialization settings.

After initialization, we can start writing our main function. The first algorithm was written in software. We directly create two arrays of size 1000. Then fill up one of the arrays, and finally inject the values in it into the other array through the write loop. When performing the above operations, we need to use the Cycle Counter of the Data Watchpoint Trigger module to complete the cycle counting function. Then there is the part that supplements the DMA algorithm. For this task, we will use interrupt mode to complete. Since we are not using any other tools or equipment for this task, we will not use any callback equations either. So we will directly replace him with the IRQhandler function. We can know by consulting the documentation that the interrupt is triggered at the end of the data transmission, so we can start the timing in the main function before the data is transmitted, and then end the timing in the IRQHandler function after the data transmission and print the result.

The last requirement of lab is to use different lengths from 10,100 to 1000 and different data types, uint8_t, uin16_t, and uint32_t. The completion methods of these tests are all completed as described above, and the results can be viewed on the results page that follows.

Results and Analysis

The first and second screenshots show that the normal running sequence from terminals. It is within our expectations.



```
software:
uint8_t test: memory length:    10      cycle:  220
uint8_t test: memory length:    100     cycle:  1315
uint8_t test: memory length:    1000    cycle:  13516
uint16_t test: memory length:   10      cycle:  141
uint16_t test: memory length:   100     cycle:  1315
uint16_t test: memory length:   1000    cycle:  13516
uint32_t test: memory length:   10      cycle:  135
uint32_t test: memory length:   100     cycle:  1303
uint32_t test: memory length:   1000    cycle:  14016

DMA:
uint8_t test:
memory length:  10      cycle:  668
memory length:  100     cycle:  967
memory length:  1000    cycle:  6369
uint16_t test:
memory length:  10      cycle:  446
memory length:  100     cycle:  967
memory length:  1000    cycle:  6369
uint32_t test:
memory length:  10      cycle:  433
memory length:  100     cycle:  972
memory length:  1000    cycle:  6373
```

Figure 1: Task 1 output

Task 2: SPI DMA

Introduction

Many modern and major implementations of protocols introduced in the previous labs, such as SPI, may be used in conjunction with DMA in devices such as SD card readers and flash memory interfaces. In the project development, the SPI interface needs to be used to transmit a large amount of data. In order to save the working time of the CPU, it is wise to use DMA for data transmission. While the DMA is capable of transferring a single data point at a time, its usefulness arises when many data points or samples need to be moved in and out of memory without CPU intervention. In order to allow for this, the DMA controllers have functionality to increment both the source and destination addresses after each transfer.

The STM32F769NI has two DMA controllers: DMA1 and DMA2. While the functionality of both are similar, they are not identical as there are different limitations on which modules the controllers may be connected to. The DMA controllers may be configured to operate in three directions: peripheral-to-memory, memory-to-peripheral, and memory-to-memory.

For this task, we need the code for task 3 of lab3. We need to use DMA to manage the transmission and reception of the SPI loopback data. The DMA streams again can be configured in normal mode. But unlike the previous task, this task needs to read a whole line of characters and print them out. Once enter is pressed, the buffer should be sent across the SPI lookback channel via DMA and then printed on the terminal.

Low Level Description

For this task, the first thing is to set up DMA and SPI. The initialization of SPI is exactly the same as the code written before and does not need to be modified. Let's go straight to setting up the DMA. The first step is to turn on the clock. Since we are going to use the SPI tool this time, we need to consult the manual to know that the TX and RX of SPI2 use two different DMAs, so we need to set them separately. Here RX uses DMA1_Stream3 and DMA_CHANNEL_0. This DMA transfer is due to RX, so the direction is DMA_PERIPH_TO_MEMORY. We open MemInc. Next, the difference from task 1 is that we need to use the __HAL_LINKDMA function to connect the

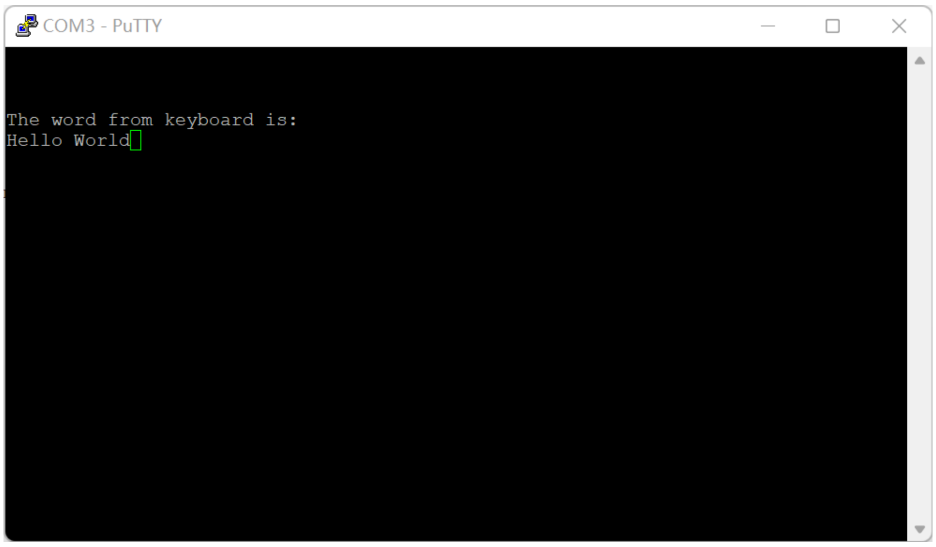
```
__HAL_LINKDMA(&SPI_2,hdmatx,TX_DMA);
__HAL_LINKDMA(&SPI_2,hdmarx,RX_DMA);
```

DMA and SPI RX together. Then there is the TX part, but basically the RX is almost the same. But on the direction, TX uses DMA_MEMORY_TO_PERIPH. Finally, the initialization settings are completed through HAL_DMA_DeInit and HAL_DMA_Init.

Back to the main function, we need a loop, the condition of this loop is that the received character is not 13. Because 13 is the Ascii code of the newline key. Then we put each received character into the buffer in turn. When a newline character is detected, the loop terminates, and we call HAL_SPI_TransmitReceive_DMA to transfer our input character to the output buffer. Then we use HAL_Delay to delay it a little to prevent the program from ending prematurely. Finally, we also print the result in the IRQHandler of RX.

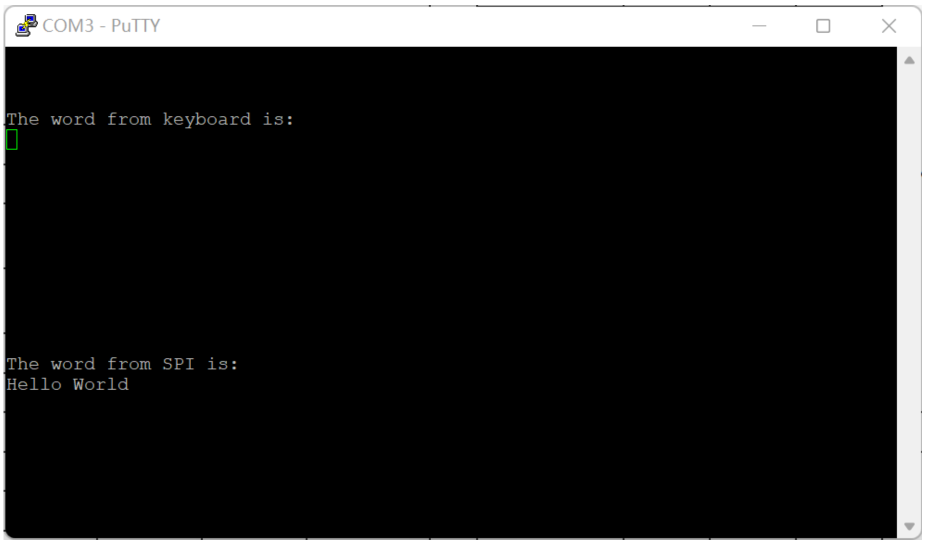
Results and Analysis

The first screenshots show that the input interface of this task. The second shows the output results.



```
The word from keyboard is:
Hello World
```

Figure 3: input interface



```
The word from keyboard is:





The word from SPI is:
Hello World
```

Figure 4: output results

Task 3: IIR Filter DMA

Introduction and High-Level Description

Task 3 is the same of the Lab 4 Task 4. Except it is used by DMA. The introduction of DMA is discussed in previous 2 Tasks, with no new feature to use in this Task. The filter to be implemented by the following transfer function.

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

Implement a circular DMA stream for the ADC only, transferring one sample at a time. We were waiting for the ADC result in a loop, which isn't an effective way of using processor resources. It is better to trigger a conversion and wait for the conversion to complete the interrupt. This way, a processor can do other tasks rather than wait for ADC conversion to complete. The DMA would transfer the data from the ADC into a buffer which its size is 1, then wait for the next interrupt, while CPU is processing the data into a IIR filter and output it into the DAC to or a frequency output.

Use of the DMA with the DAC is only useful in this type of application if there is an external trigger controlling conversion timing.

The wiring of this task is the same as Lab 4 Task 4. 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 main ADC and DAC initialization is same as Lab 4 Task 4. However, there is a DMA set tup in ADC parameter that is shown below.

```
    // HAL and Low-layer drivers P. 88
    ADC_h3.Init.DMAContinuousRequests = ENABLE;
```

For the Stream and Channel in DMA, it is DMA2 with Stream0 and Channel 2 for ADC3.

```
    // Reference_Manual P. 249
    ADC_DMA.Instance = DMA2_Stream0;
    ADC_DMA.Init.Channel = DMA_CHANNEL_2; //ADC3
```

To connect DMA to the ADC buffer, following weak function should be implemented to overwrite.

```
    __HAL_LINKDMA(&ADC_h3, DMA_Handle, ADC_DMA);
```

The following is DMA initialization. The mode of this lab is circular. A circular DMA stream for the ADC only, transferring one sample at a time. In addition, the data of the ADC input may be greater than 4000, so the data buffer in DMA should be WORD, with 2^{16} numbers in it.

```
    ADC_DMA.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
    ADC_DMA.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;
    ADC_DMA.Init.Mode = DMA_CIRCULAR; // circular
```

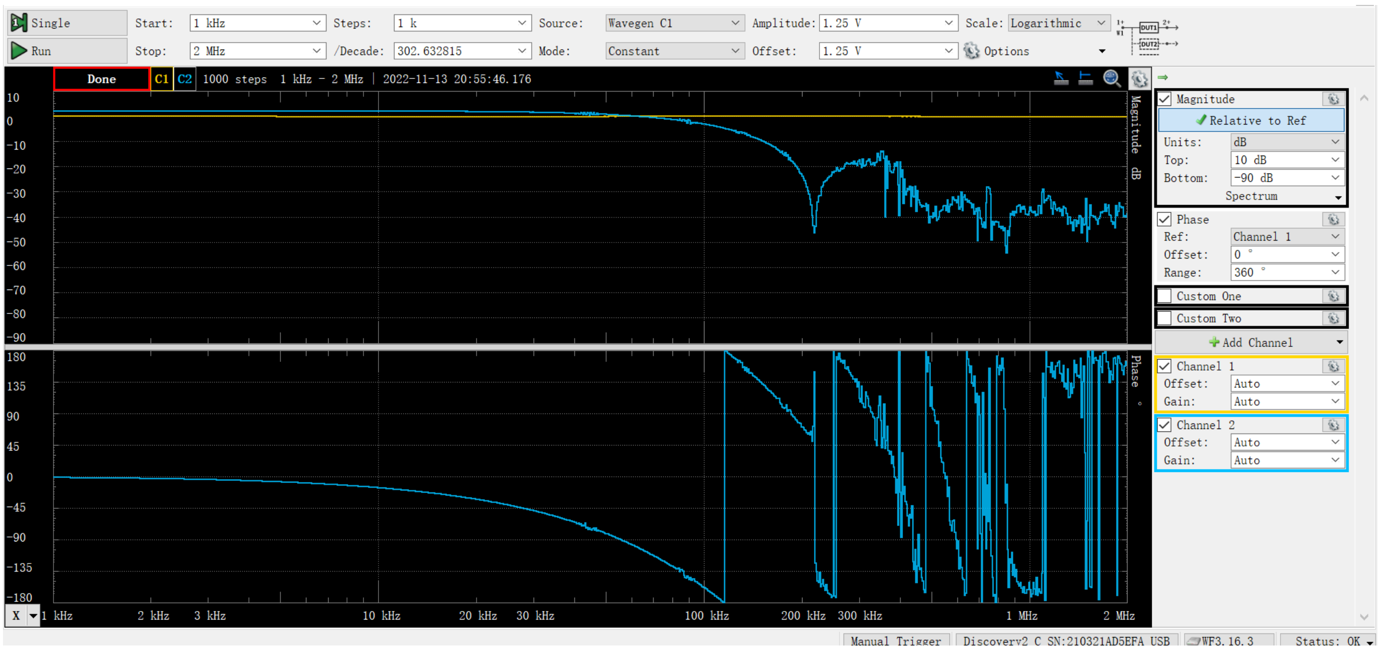
Instead of start ADC itself, use the following function to start ADC with DMA. The last argument is the size of the destination data buffer. In this task, it is 1.

```
    HAL_ADC_Start_DMA(&ADC_h3, (uint32_t *)ADC_buffer, 1);
```

After storing the read voltage, put all the values into the filter, then output the value into the DAC function inside the ADC callback function. So the CPU would automatically handle the IIR filter and set the DAC value. 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

Done C1 C2 1000 steps 1 kHz - 2 MHz | 2022-11-13 20:55:46.176

Magnitude
Relative to Ref
Units: dB
Top: 10 dB
Bottom: -90 dB
Phase
Ref: Channel 1
Offset: 0 °
Range: 360 °
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. This implementation is less efficient than the original implementation without DMA. Because the buffer size is only 1, so the DMA is not efficient. The CPU has to give the memory control to DMA for one cycle and then close it. It uses lots of resource and time, but only calculates one sample. So it is less efficient.

Conclusions

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

DMA

The DMA method is to directly rely on the hardware to transfer data between the main memory and the device, and does not require CPU intervention during this period. It is characterized by randomness, higher parallelism, fast transmission speed and simple operation. Applications:

(1) Data block transfer for high-speed external memory such as disks;

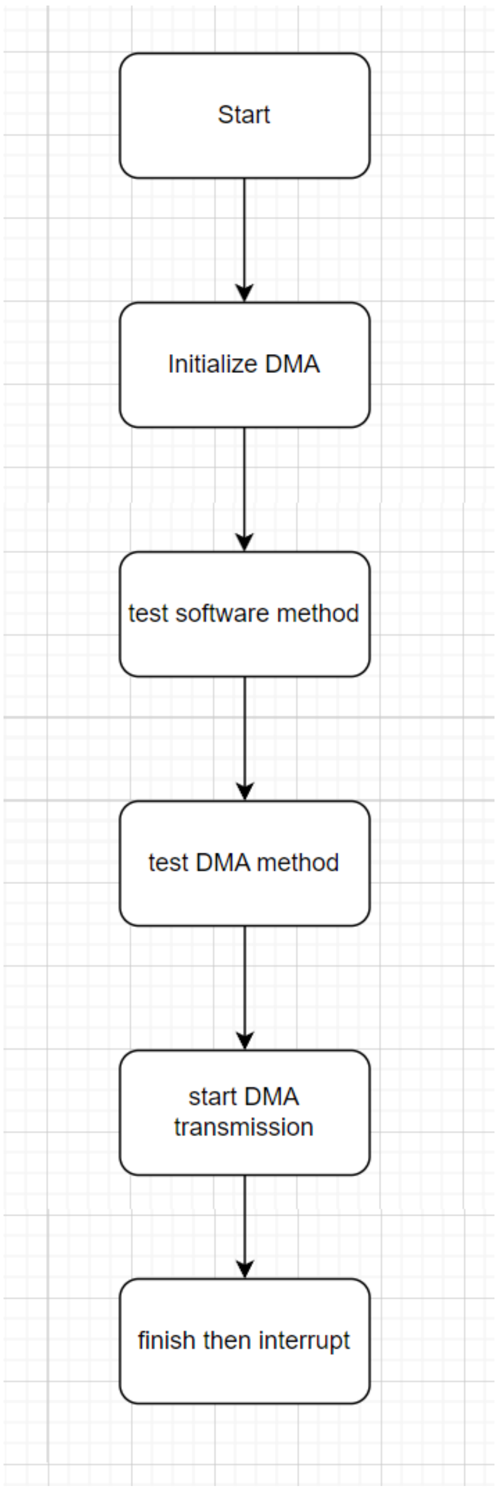
(2) Data frame transfer for high-speed communication equipment;

(3) High-speed data acquisition;

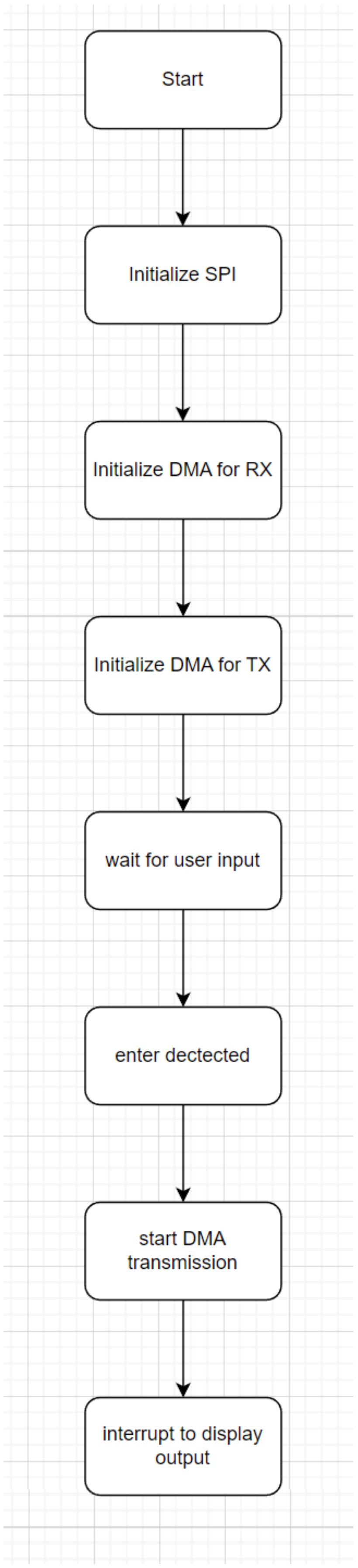
(4) Dynamic memory refresh.

Appendix

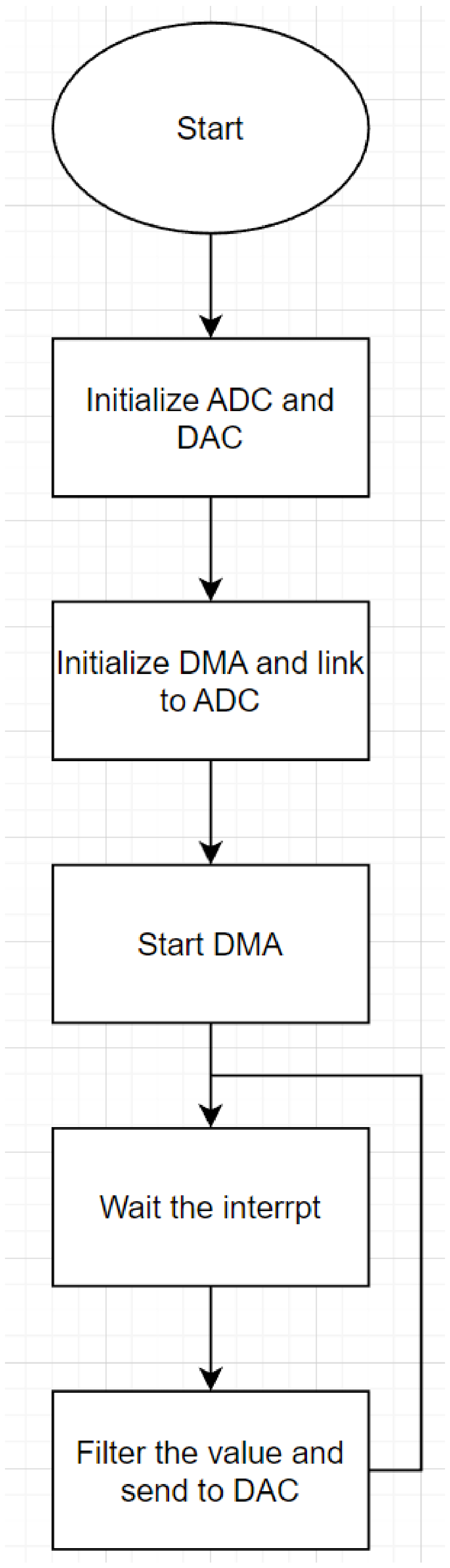
[1] Task 1 flowchart.



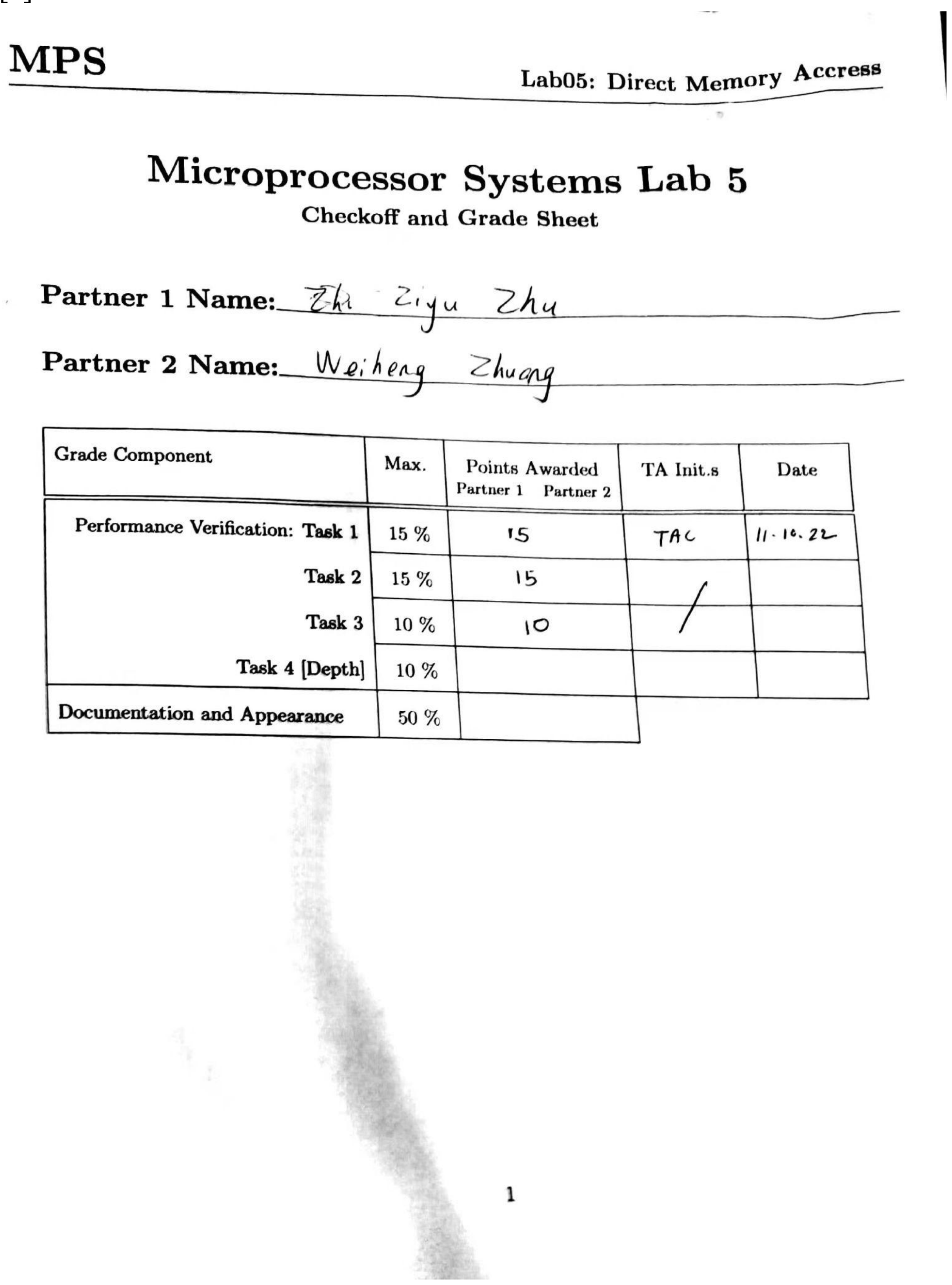
[2] Task 2 flowchart.



[3] Task 3 flowchart.


Start
Initialize ADC and DAC
Initialize DMA and link to ADC
Start DMA
Wait the interrpt
Filter the value and send to DAC

[6] Checkoff and Grade Sheet



MPS Lab05: Direct Memory Accress

Microprocessor Systems Lab 5

Checkoff and Grade Sheet

Partner 1 Name: ~~Zh~~ Ziyu Zhu

Partner 2 Name: Weiheng Zhuang

Grade Component	Max.	Points Awarded		TA Init.s	Date
		Partner 1	Partner 2		
Performance Verification: Task 1	15 %	15		TAC	11.10.22
Task 2	15 %	15		/	
Task 3	10 %	10			
Task 4 [Depth]	10 %				
Documentation and Appearance	50 %				

1