

Lab 5 Report

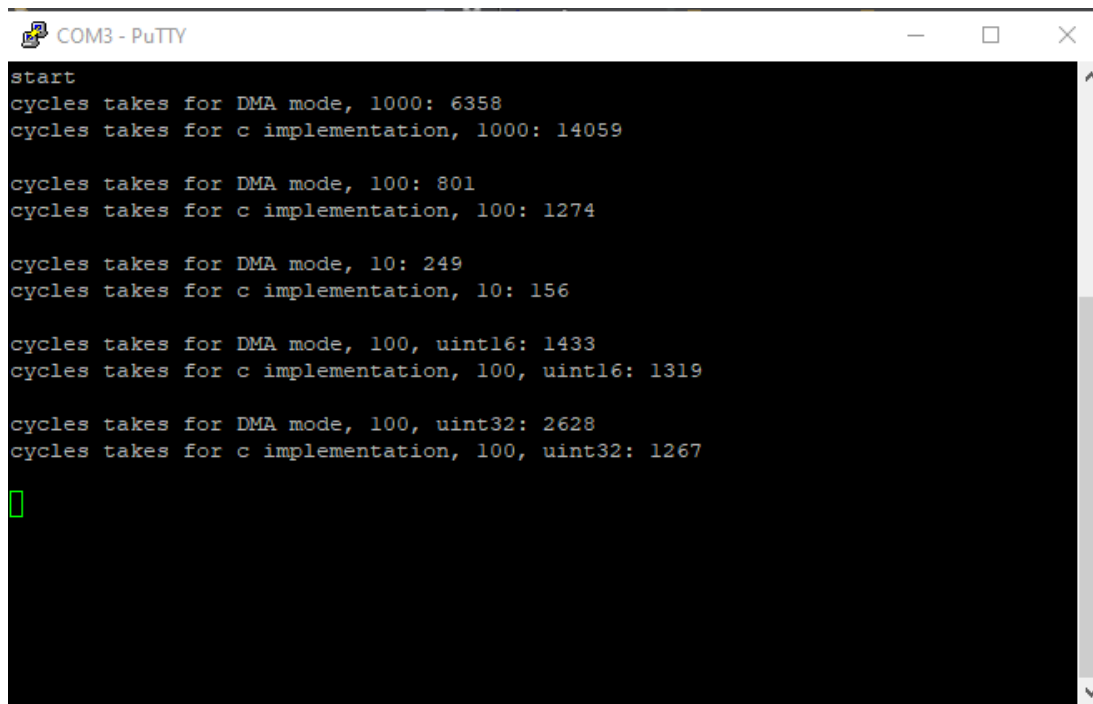
Task 1: DMA Performance Example

- High Level Description
 - Transfer data from one C buffer to another C buffer using DMA in normal mode and in the memory-to-memory direction.
 - Should use 2 separate algorithms for copying the buffers. One is implemented in software and one using the DMA.
 - Should time both of these methods using the Data Watchpoint Trigger module's Cycle Counter.
 - Should try to test and copy different types of variables and different lengths of buffers.
 - Flow chart in appendix
- Low Level Description
 - First, use `hdma_memtomem_dma2_stream0.Instance = DMA2_Stream1;` and `hdma_memtomem_dma2_stream0.Init.Channel = DMA_CHANNEL_0;` to set DMA2 channel0 stream1.
 - Then, use `hdma_memtomem_dma2_stream0.Init.Direction = DMA_MEMORY_TO_MEMORY;` to modify the memory-to-memory direction.
 - Enable both increment with `hdma_memtomem_dma2_stream0.Init.PeriphInc = DMA_PINC_ENABLE;` and `hdma_memtomem_dma2_stream0.Init.MemInc = DMA_MINC_ENABLE;`
 - Use `hdma_memtomem_dma2_stream0.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;`
`hdma_memtomem_dma2_stream0.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;` `hdma_memtomem_dma2_stream0.Init.Mode = DMA_NORMAL;` and `hdma_memtomem_dma2_stream0.Init.Priority = DMA_PRIORITY_LOW;` to set the data alignment to byte and to set the priority to low.[1]
 - Finally, disable the FIFO and enable the DMA clock with `hdma_memtomem_dma2_stream0.Init.FIFOMode = DMA_FIFOMODE_DISABLE;`
`hdma_memtomem_dma2_stream0.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;` `HAL_RCC_DMA2_CLK_ENABLE();` and `HAL_DMA_Init(&hdma_memtomem_dma2_stream0);`
 - To calculate the time, we first clear the cycle counter, `DWT->CYCCNT = 0;`, and then store the counter, `cycles = DWT->CYCCNT;`, to find out how many cycles it took to transfer the data.
 - We set `uint8_t flashData[1000];` `uint8_t sramData[1000];` `uint16_t flashData2[1000];` `uint16_t sramData2[1000];` `uint32_t flashData3[1000];` and `uint32_t sramData3[1000];` as global variables to

find out the difference between transferring different type of variables. And then we use `for (int j = 0; j<1000; j++){sramData[j] = flashData[j];}` `for (int j = 0; j<100; j++){sramData[j] = flashData[j];}` and `for (int j = 0; j<10; j++){sramData[j] = flashData[j];}` to test the difference between transferring different lengths of variables.

- Results and Analysis

- The program worked just as described in the Lab instruction. When we start the program, it will automatically transfer different types and lengths of variables, and store how many cycles it took to finish each transfer.
- What's new in this lab is how to properly set the DMA figuration and start the DMA transfer.
- Results:



```
start
cycles takes for DMA mode, 1000: 6358
cycles takes for c implementation, 1000: 14059

cycles takes for DMA mode, 100: 801
cycles takes for c implementation, 100: 1274

cycles takes for DMA mode, 10: 249
cycles takes for c implementation, 10: 156

cycles takes for DMA mode, 100, uint16: 1433
cycles takes for c implementation, 100, uint16: 1319

cycles takes for DMA mode, 100, uint32: 2628
cycles takes for c implementation, 100, uint32: 1267
```

Figure 1: Task1 result

Task 2: SPI DMA

- High Level Description
 - This program should use the DMA method to achieve the same results as the program in Lab 3 Task 3.
 - Instead of transferring only one letter at a time, this program should be able to transfer a full line of characters from the terminal into a buffer. Stop the line and start to transfer only when the user presses enter. Then the line should be sent across the SPI lookahead channel and printed on the terminal.
 - Flowchart in appendix

Table 1: Task 3 schematic

	Connected to	Connected to
MISO to MOSI	D11 DISCO board	D12 DISCO board

-
- Low Level Description
 - First, we have to configure the DMA settings. But this time, instead of using DMA2 channel2 and stream1, we have to use DMA1 channel 0, stream 3 and 4. These are the channels for rx and tx in SPI2.
 - Compared to Task 1, we have to set direction to peripheral to memory this time, with `SPI_DMA_rxHandle.Init.Direction = DMA_PERIPH_TO_MEMORY;`. We only need an increment for the peripheral to receive memory. So we are going to use `SPI_DMA_rxHandle.Init.PeriphInc = DMA_PINC_DISABLE;`
`SPI_DMA_rxHandle.Init.MemInc = DMA_MINC_ENABLE;`
`SPI_DMA_rxHandle.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;` and `SPI_DMA_rxHandle.Init.PeriphDataAlignment = DMA_MDATAALIGN_BYTE;` for the rxHandle. The setting for the txHandle is almost the same as the reHandle except txHandle will use Stream4 instead of Stream3.
 - And this time, we also need to set the mode to circular with `SPI_DMA_txHandle.Init.Mode = DMA_CIRCULAR;` [2]
 - The priority is set to low and the FIFO is disabled as in task 1.
 - The settings for SPI are the same as the settings in Lab3 task3 but without First Bit and CRCCaluation.
 - Since we are using interrupt, the IRQ Handler for Stream4 and Stream3 are configured by `void`
`DMA1_Stream4_IRQHandler(){HAL_DMA_IRQHandler(&SPI_DMA_txHandle);}`
and `void`
`DMA1_Stream3_IRQHandler(){HAL_DMA_IRQHandler(&SPI_DMA_rxHandle);}`.
 - The SPI2 Handler is configured by `void`
`SPI2_IRQHandler(){HAL_SPI_IRQHandler(&SPIHandle);}`.

- Then start the DMA with `HAL_SPI_TransmitReceive_DMA(&SPIHandle,&inputs,outputs,sizeof(inputs));` and we store the inputs to the variable "outputs".
- `length = uart_getline(&USB_UART,inputs,19);while(flag == 0){`
`printf("\033[12;1H\033[K");`
`fflush(stdout);`
`for (int j = 0; j< length; j++){`
`printf("%c",outputs[j]);`
`printf("\r\n");` are used to print out the data the user entered. And to print out the data received, we have to use

```
printf("\033[12;1H\033[K");
fflush(stdout);
for (int j = 0; j< length; j++){
    printf("%c",outputs[j]);
}
printf("\r\n");
```

- Results and Analysis

- The program worked just as described in the Lab instruction. The program will make the input from the user into a line. And after the user hits enter, it will be transmitted across the SPI lookback channel and then printed out on the terminal.
- This task combines Lab5 task1 and Lab3 task3 together. We used `HAL_SPI_TransmitReceive_DMA()` this time instead of the one we used in Lab3.
- Here are the results.



```
COM3 - PuTTY
start
sdlfsjdfsd
sdlfklwe
```

Figure 2; task 2 result 1



Figure 3: task 2 result2

Task 3: IIR Filter DMA

- (a) *Introduction and High-Level Description*
 - The purpose of this to build a IIR filter using the ADC and DAC function in the DISCO board. The board will read a input signal and process it. The output is a processed signal.
 - This task is similar to task 4 in LAB 4, the difference is using the DMA instead of polling. Unlike polling, DMA will do the reading without using CPU cycle. An interrupt will be triggered when the reading is complete. With the interrupt we can know when the reading is complete.
 - The basic idea of this IIR filter is the output is a sum of current input, last two input, and the last output signal voltage. Each with different coefficient.
 - Flow chart is attached in the appendix
 - Wire connection:

Table 2: Task 4 schematic

	Connected to	Connected to
ADC	A0 DISCO board	Discovery board wave generator W1

DAC	A1 DISCO board	Discovery board oscilloscope 2
Ground	GND DISCO board	Discovery board ground
Oscilloscope 2	GND DISCO board	Oscilloscope 2-
Oscilloscope 1	Discovery board wave generator W1	GND

- *(b) Low Level Description*
 - The DAC setting are the same as LAB4
 - The DMA setting here is different from the previous task. This DMA is peripheral to memory and all the increment need to disable. The ADC keeps reading the data so the DMA.Init.Mode should be circular. And the FIFO mode needs to be turned off.
 - Since the ADC is using the DMA mode, in ADC initialization, DMAcontinuousReuquests = ENABLE is used here to open the DMA function.
 - The Other idea are the same as in LAB4, Interrupt is used here for notifying the DMA transmission is complete. Once it complete, a flag will be set in the

interrupt function. Then the data can be accessed by HAL_ADC_GetValue. Then the flag is cleared.

- (c) *Results and Analysis:*
 - The program worked as expected
 - The filter did not perform well on frequency higher than 10kHz, the reason of this issue is the sampling rate chosen here is 480 cycles. Students changed the sampling rate to 56 cycles, then the filter works as expected.
 - The result are as followed. Zeros showed in the figure.

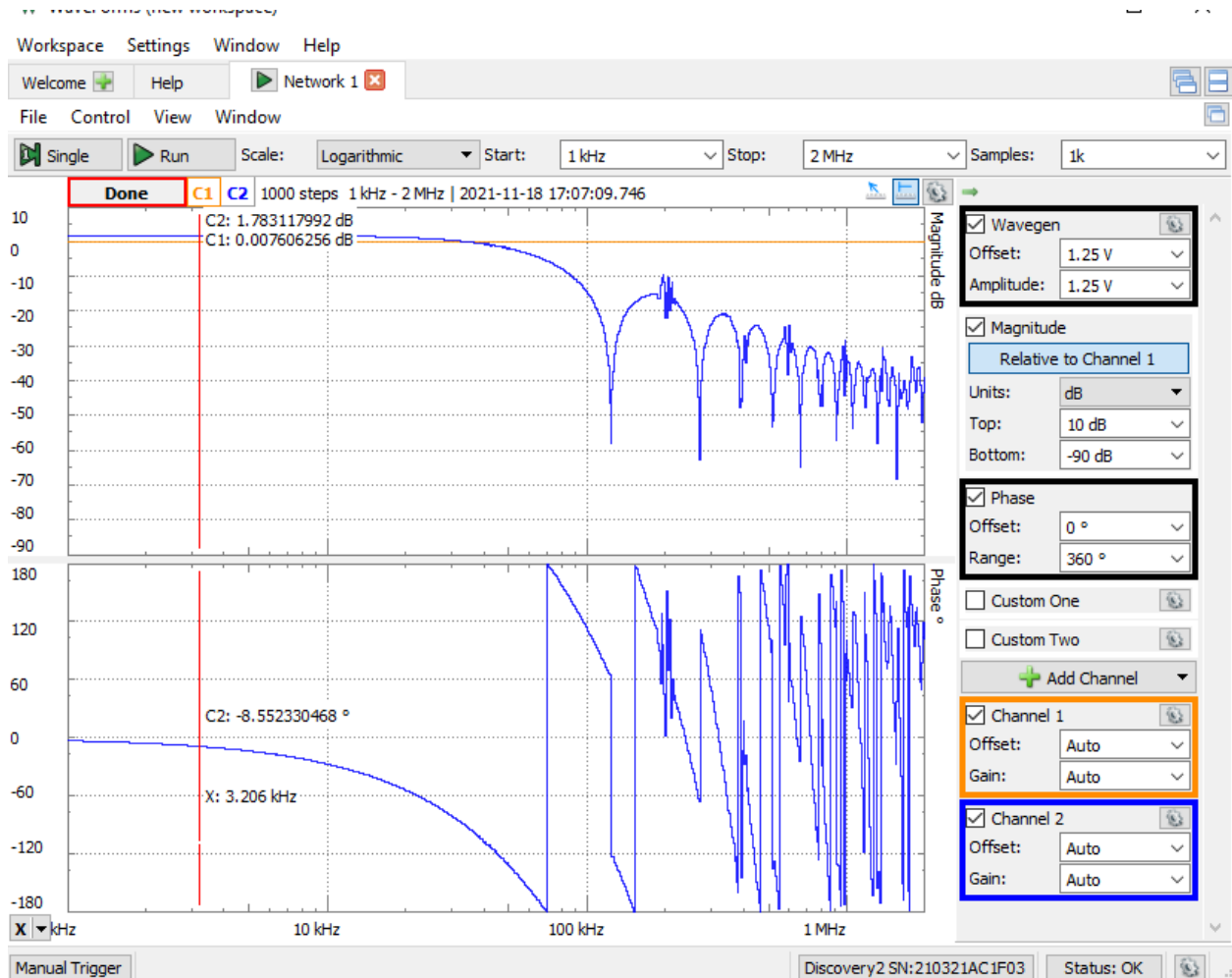


Figure 4: task 3 result

Conclusion

In this lab, student learned how to use DMA on the STM32 DISCO board. DMA is a strong tool for transmitting data between memories or memory and peripheral or peripherals. The property of DMA is it build a channel to exchange data without using CPU cycles.

In the first lab, the DMA has great advantage in transfer large number of data from one memory to another. The procedure is fast and does not need to use CPU cycles. And it can also be used to perform SPI. The advantage of this is also it does not need to use DMA. In the last part, the DMA ADC conversion has a better performance compared to LAB4. Without using CPU cycles, the microprocessor can do something else when processing DMA.

The application of the DMA can be in a case where the microprocessor needs to read multiple ADC result and processing them at the same time. Like the depth task in LAB4, this cannot be implemented by polling. It can also let the microprocessor processing other task when the DMA is working.

Reference

[1] *Mastering STM32*, Carmine Noviello, USA, 2018, pp. 284.

[2] *RM1905 User manual*, STMicroelectronics, NV, USA, 2017, pp.279-285.

Appendix

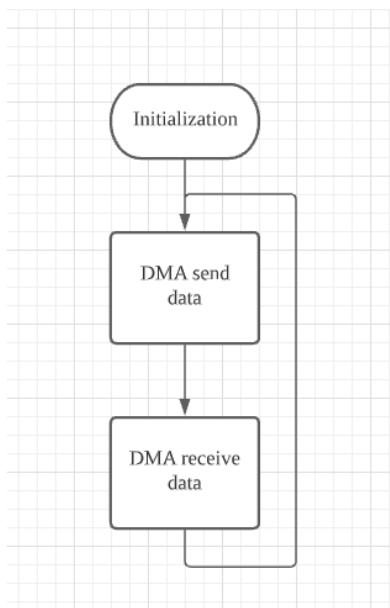


Figure 5: task 1 flowchart

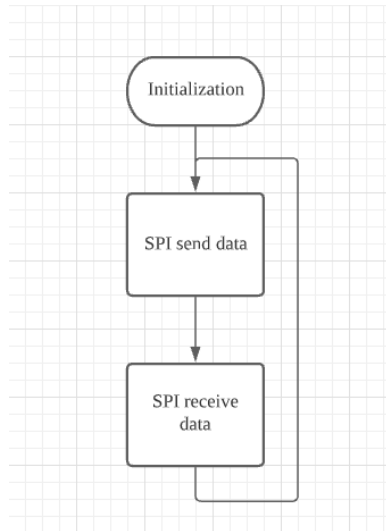


Figure 6: task 2 flowchart

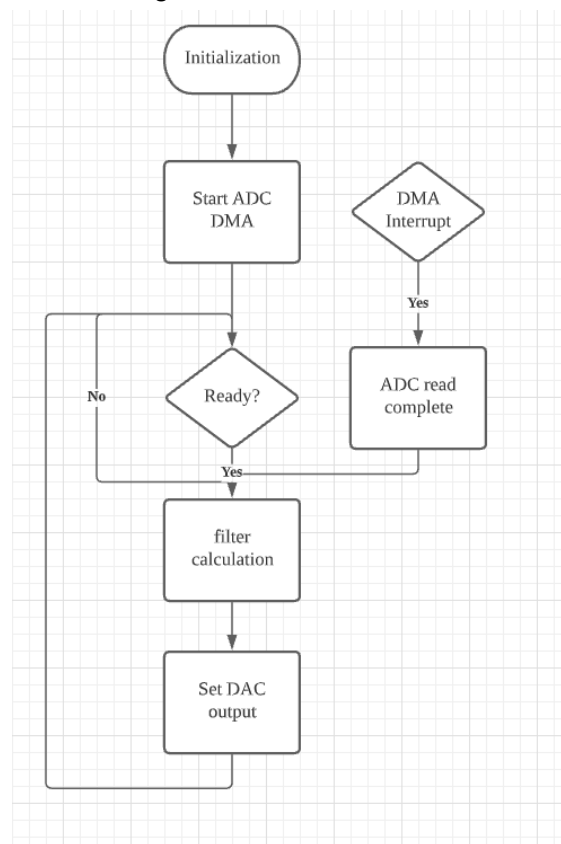


Figure 7: task 3 flowchart