Microprocessor Systems Lab 5

Checkoff and Grade Sheet

Partner 1 Name: _____

Partner 2 Name: _____

Grade Component	Max.	Points Awarded Partner 1	Points Awarded Partner 2	TA Init.s	Date
Performance Verification: Task 1	10 %				
Task 2	10 %				
Task 3	10 %				
TA Questioning	20 %				
Documentation and Appearance	50 %				
Aide Deduction	-				
Total:					

Grader's signature: _____

Date: _____

→ Laboratory Goals

By completing this laboratory assignment, you will learn about and to use:

1. the Direct Memory Access (DMA) module.

→ Reading and References

R1. Mastering STM32: Chapters 9

R2. UM1905-stm32f7_HAL_and_LL_Drivers.pdf: Chapter 20

R3. RM0410-stm32f7_Reference_Manual.pdf: Chapter 8 (

R4. AN4031-stm32_DMA_Controller.pdf: Overview of DMA use and functionality.

R5. Lab05_DMA_Template.zip: Project Template for Lab 5.

Direct Memory Access (DMA)

→ Introduction to DMA

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 many CPU cycles to be spent on other tasks. This is accomplished by having the peripheral device, like an ADC, interface with a *DMA controller*, which sends the same signals to memory that the CPU would if it were mediating the transfer. In other systems, the devices essentially take on the role of DMA controller in what is called *bus mastering*. The STM32F769NI has a programmable DMA controller, but examples of devices that use bus mastering (which is faster, since there is no "middleman" at all) includes PCI devices and IDE hard disk and optical drives.

The next evolution after bus mastering is what PCI-Express uses. This mechanism is similar to combining DMA with Ethernet, which allows very high-speed, full-duplex (i.e., simultaneous) read/write operations. PCI-Express also features low-latency switching, enabling multiple devices to share and utilize the same bus virtually simultaneously.

Back to DMA controllers: Many modern and major implementations of protocols introduced in the previous labs, such as SPI, are used in conjunction with DMA in devices such as SD card readers and flash memory interfaces. Since the DMA is not useful by itself (generally), this lab will entail the use of the code produced in said previous labs in order to incorporate DMA. Significant performance increases are not expected generally for this lab, however the techniques introduced my be extended to produce much more efficient program implementation than would otherwise be possible without the DMA.

→ DMAs on the STM32F769NI

The STM32F769NI has two DMA controllers: DMA1 and DMA2, while the functionality of both are similar, they are not identical as these are restricted in the modules that may be connected to them. Tables 27 and 28 of R3 (p. 249) describe what modules and signals may be connected to which. Note that each DMA module has 8 *streams*, which allow each module to manage up to 8 signals, instead of simply one signal per one DMA. Care must be taken, however, as multiple streams on one DMA cannot be active concurrently. To account for this the priority of each stream is configurable, allowing extremely time/speed sensitive streams to take precedence over others.

The DMA controllers may be configured to operate in three directions: peripheral-to-memory, memory-to-peripheral, and memory-to-memory (DMA2 only). This implies that to have bi-directional access to a module, for example, USB for both reading and writing, then at least two DMA streams would be required: one for providing data to the peripheral and one for receiving data from the peripheral. The data sizes for each of these may also be configured as well, allowing for byte (8 bits), half-word (16 bits) or word (32 bits) data widths.

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 as memory is written. For example, when taking consecutive samples from the ADC, the DMA may place the samples in a contiguous chunk of memory to prevent overwriting of the previous samples. In this case, the destination address (memory) would be configured as incrementing whereas the source address `ADC_DR` would remain constant.

The DMA streams can also be configured to operate as *normal* or continuous, or *circular*, transfer types. In normal mode, the DMA will transfer only the amount of data requested from it then stop. 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. Both of these cases will trigger a DMA interrupt (`DMA`x`_Stream`y`_IRQ`n) which of course could then be used to set a global variable indicating completion, ready the next set of data to be sent, or consume the received set of data, etc.

Although using the DMA controllers effectively replaces polling or interrupt management of peripherals, indication of completed transfers by the DMA is done through polling or interrupts! However, in the case of DMA interrupts, ideally these interrupts occur much less frequently then otherwise. Care must also be taken here as the normal and circular modes do not operate the interrupts in the same way. For example: in normal mode for when managing a UART peripheral, the HAL DMA callback function will trigger in turn trigger the associated USART interrupt; whereas in circular mode the HAL DMA callback function will bypass the USART interrupt and directly call the associated UART callback function (e.g., `HAL_UART_TxCpltCallback()`. Of course, the peripheral interrupts may be left disabled if there is no need for them.

Finally, the DMA controllers have FIFO buffers available for use which will further enable efficient memory access as it is possible to accumulate a chuck of data to be placed into memory prior to writing. This prevents multiple successive accesses to the SRAM from the DMA controller; allowing for other modules to access in that time. In this lab, the FIFO buffer does not need to be used.

→ Configuring DMA

When using the HAL, configuring the DMA is relatively straight forward if the configuration options available are understood. The following steps may be taken to enable DMA for various peripherals:

1. Disable the DMA stream to be configured if already active,
2. Create a (global) `DMA_HandleTypeDef` handle variable to hold the DMA stream's configuration information.
3. Associate the DMA stream and the channel to the handle,
4. Specify the direction, data size, address incrementation, mode, and FIFO buffers,
5. Enable the DMA through `HAL_DMA_Init()`,
6. Associate the DMA stream to the peripheral using the HAL macro `__HAL_LINKDMA()`,
7. Enable the stream IRQ handler, if using interrupt mode (vs. polling mode),
8. Start a DMA transfer through the peripheral's ***xxx*`_DMA()` functions.

◇ Task 1: UART DMA

Using the previously developed Lab01 task 2 code, implement DMA into the `uart.h/uart.c` functions `_write()` and `_read()`[1]. The DMA streams should be configured in normal mode[2]. All modifications for this task should be done in `uart.h/uart.c` only! The code from Lab01 is only necessary for testing functionality.

NOTES:

1. You should copy your Lab1 task 2 code into this project. Do not modify the original project.
2. The file `stm32f7xx_hal_uart.c` has some helpful information on setting up the DMA for use with UARTs.
3. Since work on this task will likely break `printf()`, the debugger may be extremely useful here (or alternative uart functions from `uart.h/uart.c`.
4. Interrupt priorities may be problematic for this task, but more so in the next task...
5. The USART interrupts are not enabled in `uart.h/uart.c`. If using them, don't forget to enable them.
6. The DMA has a clock too! (enable it!)

[1] These are the functions used to implement `printf()`, `scanf()`, `putchar()`, and `getchar()`.

[2] Here we are only periodically asking for or sending data. If it was the case that data was continuously being sent/received; then circular mode would be more appropriate

◇ Task 2: SPI DMA

Copy the Lab 3 task 3 code into this project. Modify this code to have DMA manage the transmission and reception of the loopback data. The DMA streams again can be configured in normal mode. Ensure that `HAL_SPI_Transmit()`, `HAL_SPI_Receive()`, and `HAL_SPI_TransmitReceive_DMA()` operate properly. Continue to use the modified `uart.h/uart.c` files from Task 1.

NOTES:

1. Interrupt priorities will likely be problematic for this task. Make sure the more important DMA and peripheral interrupts have priority over the others.
2. It may be confusing how the `HAL_SPI_TransmitReceive_DMA()` operates in terms of the interrupts triggered. Hint: only one of the DMA streams will have an enabled interrupt when this function is called.

◇ Task 3: IIR Filter DMA

Copy the lab 4 task 4 code into this project. Implement a circular DMA stream for the ADC *only*. The CPU be handling the signal processing math and passing the output value to the DAC. Is this implementation more or less efficient than your original implementation? Why or why not?

NOTES:

1. Use of the DMA with the DAC is only useful if there is an external trigger controlling conversion timing (e.g., a timer).