Microprocessor Systems Lab 5

Checkoff and Grade Sheet

Partner 1 Name: ____________________

Partner 2 Name: ____________________

Grade Component	Max.	Points Awarded		TA Init.s	Date
		Partner 1	Partner 2		
Performance Verification: Task 1	15 %				
Task 2	10 %				
Task 3	15 %				
Task 4 [Depth]	10 %				

→ 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-stm32f76xxx_Reference_Manual.pdf: Chapter 8

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

R5. Lab05_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 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 includes PCI devices and IDE hard disk and optical drives. Bus mastering ends up being faster than using a DMA controller as there is no "middleman," which is the DMA controller itself.

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, may be used in conjunction with DMA in devices such as SD card readers and flash memory interfaces. Since the DMA is not useful by itself[1], this lab will entail the use of code produced in said previous labs in order to incorporate DMA. Significant performance increases are not generally expected 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.

[1]DMAs can usually move data in memory from one location to another (i.e., the *memory-to-memory* mode); however, this mode is not the primarily use of the module

→ 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 there are different limitations on which modules the controllers may be connected to. Tables 27 and 28 of R3 (p. 249) describe what modules and signals may be connected to each. 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 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 extracting 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*. These data widths control the size of the data that is transferred for each DMA action.

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. For example, when taking consecutive samples from the ADC, the DMA may place the samples in a contiguous chunk of memory (e.g., a target array) 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 be configured to operate as *normal* or *circular* (continuous) transfer types (see Figure 1). 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.

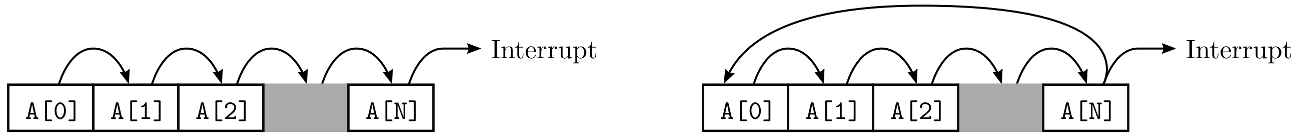

Figure 1: Flow of DMA accessing buffer `A` with length `N` in (left) normal and (right) circular modes.

An extension to the circular mode of the DMA is the *double-buffered* mode, shown in Figure 2. In this configuration, the DMA alternates movement of data between two memory buffers; for example: buffer `A` is filled by the DMA, the DMA interrupt (if used) is triggered, and then the DMA begins filling buffer `B`. In this way, the software may consume `A` within or after the interrupt without concern of the DMA overwriting the buffer prior to use; assuming the software algorithm for consuming the buffer will complete before the next DMA interrupt. After the next interrupt, the DMA moves back to `A` while the software consumes `B`.

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 these DMA

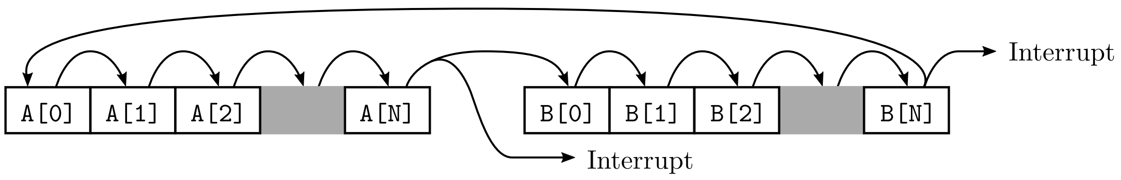

Figure 2: Flow of double-buffered DMA accessing buffers `A` and `B`, both with length `N`.

interrupts ideally 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: when managing a UART peripheral in normal mode, the HAL DMA callback function will 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()`)[2]. 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 within the DMA 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.

[2]There does not seem to be concrete documentation on this implementation or the reasoning behind it.

◇ Task 1: DMA Performance Example

This 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. To accomplish this, the DMA stream should be configured in normal mode and in the memory-to-memory direction. Reviewing the requirements for this "direction" is strongly suggested (R3 p.250). For implementation of the DMA, using the interrupt functionality is preferred, though polling is acceptable. It should be noted that the HAL documentation R2 states that the polling operation is included for legacy compatibility only.

Two separate "algorithms" for copying the buffers should exist in this task: one purely implemented in software and one using the DMA. Both of the methods should be timed using the *Data Watchpoint Trigger* (`DWT`) module's *Cycle Counter*, `CYCCNT`. Functionality for using the `DWT_CYCCNT` has been implemented within the template project as documentation for implementation is sparse[3].

Several evaluations of this task's final code should be done with varying length buffers (e.g., 10, 100, 1000) and different variable types and associated DMA widths (e.g., `uint8_t`, `uin16_t`, and `uint32_t`), with this data being presented within the Laboratory Report.

NOTES:

1. Don't forget to wait for the DMA transfer to complete!
2. Any arrays/buffers modified by the DMA should either be globally defined or initialized with `malloc()` such that their positions in memory do not move. **Do not use local variable arrays!**
3. The data in the arrays does not need to be anything specific (could just be zeros); however, filling them with identifiable data will aid in determining if the DMA transfer is successful or not.
4. If using the interrupt mode for the DMA, be aware that since a peripheral is not being used, there will not be a callback function registered with the DMA handle by default, as opposed to when the DMA is incorporated with the SPI, for example. A callback function may be manually added through the DMA handle field `XferCpltCallback` or the callback may be omitted.
5. It is expected that most groups will implement the software version of buffer copying using a `for` loop. It is also valid that `memcpy()` be used as well. It should be noted that `memcpy()` will inherently be either much faster or equivalent to using a `for` loop.
6. Although the DMA may be configured using 8-, 16-, or 32-bit data widths, it is very likely to be found that DMA efficiency will be poor compared to the software implementation for 8- and 16-bit transfers. This is due to the DMA being inherently a 32-bit bus, where extra cycles are required to isolate 8-bit and 16-bit values: "...interface supporting only 32-bit accesses." [R3].
7. The template project has compiler optimizations turned off to accentuate the differences between the C implementation and the DMA implementation. If optimizations are turned on or if the C code is written very efficiently, it's possible that the DMA will never perform as well as the C implementation due to DMA performance overhead; however, the gain in free CPU cycles during the DMA transfer is still generally a greater trade-off. You may turn optimizations back on through: *Project Properties* →[*On left: C/C++ Build* →*Settings*] →*Tool Settings* →*MCU GCC Compiler* →*Optimization*. The standard configuration for this class's projects is "Optimize for Debug."

[3]Yay for StackOverflow!

◇ 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 SPI loopback data. The DMA streams again can be configured in normal mode.

In addition to the changes above, the program should read a full line of characters from the terminal into a buffer, stopping when the user presses enter (`\n`), instead of just one character. Once enter is pressed, the buffer should be sent across the SPI lookback channel via DMA and then printed on the terminal.

NOTES:

1. Unlike the previous task, once the DMA is initialized, there should be no calls directly to the `HAL_DMA_`*`xxx`*`()` functions as these are generally managed by the peripheral functions (e.g., `HAL_SPI_TransmitReceive_DMA()`). This guidance excludes `HAL_DMA_IRQHandler()` which of course must be called within the DMA interrupt handler.

◇ Task 3: IIR Filter DMA

Copy the lab 4 task 4 code into this project. Modify the code such that a circular DMA stream takes the ADC output and places it in a buffer. Similarly, the DAC should be loaded from a circular DMA stream. The trigger for both of the DMA streams should be provided from a single timer, running at a relatively fast rate (e.g., 100 kHz). Set the length of the DMA buffers to be much larger than 1 (e.g., 100-1000). Once *half the ADC buffer is filled* the filled portion of the ADC DMA buffer should be filtered and saved into the corresponding locations in the DAC DMA buffer. This configuration allows for the program to process a large chunck of information at a time while simutaneously allowing data to be captured and/or transmitted at the same time. This is a more efficient approach; however, the aggregation of samples prior to processing will cause the filter to have a high delay or *latency* as the DAC data is delayed by the time it takes to transmit one-half of the output buffer. This delay will manifest itself in the phase response of the filter; causing it to be significantly distorted as compared to Lab 4.

NOTES:

1. Only one implementation of the IIR filter is required here: c-code implementation or assembly. Verification and operation of both is not needed.
2. While DMA buffer length(s) of 1 would work here, it makes much more sense to have large buffers; for example, 100-1000. A value of 1 may be used for testing, but larger values must be used for checkoff.
3. Triggering of the ADC and DAC conversions via a timer is a fairly straight-forward process. Review the `ADC_InitTypeDef:ExternalTrigConv` and `DAC_ChannelConfTypeDef:DAC_Trigger` configuration fields. Do not start the timer until all configuration is complete!
4. Care must be taken to ensure that previous samples used to calculate the filter are appropriately saved between interrupts; especially if these values are being read straight from the ADC/DAC buffers (e.g., `ADCbuffer[-1]` would likely be $x(k-1)$ in the IIR filter equation).
5. The latency of the filter must be calculated and verified for the report. One way to do this is to apply a low-frequency square-wave input and measure the filter delay with respect to an input edge.

◇ Task 4: [Depth] Serial File Transfer with DMA

Implement a method for two Disco boards to be able to send and receive large "files" across a UART interface, in this case just simply buffers. The buffers in this case should be a variation of `uint8_t`, `uint16_t`, and `uint32_t` all ranging in length from 100 to 10,000 in length (or greater). As the buffer size and type may change, the serial protocol must indicate both the length and the type of the data being sent.

As UART is only capable of sending single bytes each transfer, the DMA FIFO buffers must be used in order to send `uint16_t` and `uint32_t` data types.

Each transfer must start with some "header" information, identifying what type of data is being sent and its length, along with any other information deemed necessary. There is no requirement to use the DMA for receiving the header information. It is suggested to add a "checksum" to the header as well such the the receiver may verify the received buffer is correct.

NOTES:

1. The instructor may provide specific header files which contain the data buffers to be sent/received.
2. It is suggested that the buffers are sent when specific keyboard keys are pressed. For example: press `a` to send buffer 1, etc.
3. It is not allowed *for this task* to typecast the "files" to be sent. You *must* use the DMA FIFO buffers to do the applicable type conversions.