**MPS Direct Memory Access Lab Exercise**

**Direct Memory Access**

Student's name & ID (1): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Partner's name & ID (2): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Your Section number & TA's name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Notes:**

You must work on this assignment with your partner. Hand in a printed copy of your software listings for the team. Hand in a neat copy of your circuit schematics for the team.

These will be returned to you so that they may be used for reference.

------------------------------- do not write below this line -----------------------------

|  |  |  |  |
| --- | --- | --- | --- |
|  | POINTS (1) (2) | | TA init. |
| Grade for performance verification (50% max.) |
| DMA Part 1 (20% max.) |  | |  |
| DMA Part 2 (10% max.) |  | |  |
| DMA Part 3 (20% max.) |  | |  |
| Grade for answers to TA's questions (20% max.) |  |  |  |
| Enhancement (5% max.) |  | |  |
| Grade for documentation and appearance (25% max.) |  | |  |
|  |  |  | TOTAL |

Grader's signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**DMA**

**GOAL**

By doing this lab assignment, you will learn:

1. Direct Memory Access (DMA)

**PREPARATION**

• References: *769 Reference Manual (Register Map).pdf*

Ch. 8 (DMA)

*STM32 DMA Overview.pdf*

All

*Mastering STM32*

Ch. 9 (DMA; skip 9.2.1 & 9.3)

*769 Description of HAL Drivers.pdf*

Ch. 20 (DMA), 64 (TIM)

(Reference only; alternatively, “stm32f7xx\_hal\_dma.h/.c”

contain more or less the same info in their comments)

**DIRECT MEMORY ACCESS (DMA)**

**1. Introduction to DMA**

Direct Memory Access, or DMA, is one of the most significant technological concepts to arise this side of the invention of the transistor. In a nutshell, it is all about enabling 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.[[1]](#footnote-1)

This is accomplished by having the peripheral device, like an ADC, send the same signals to memory that the CPU would if it were mediating the transfer. In systems that employ circuitry known as the *DMA controller*, the controller handles the signaling between the peripheral and memory to lighten the load off the CPU. In other systems, the devices essentially take on the role of the 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’s no middle man at all) include PCI devices and IDE hard disk and optical drives.

The next evolution after bus mastering is actually what PCI-Express uses, which is similar to combining DMA with Ethernet, and it allows super 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 – you could think of it as DMA+++.

Many modern, major implementations of protocols introduced in previous labs, such as SPI, are used in conjunction with DMA in devices such as SD card readers and flash memory interfaces. As such, this lab will involve revisiting some of your code from previous labs in order to incorporate DMA. If you take the time to optimize your code, you will see that performance improvements achieved with DMA versus regular CPU polling/interrupt methods are huge (in this lab, this is easiest to see with the FIR filter part).

**2. Configuring DMA**

Configuring a bus to use the STM32F7’s DMA controller is actually not that complex, as it is similar to how we have been setting up our previous peripheral configurations:

1. Disable the stream to be configured in DMA\_S*x*CR (where *x* is from 0 to 7) and then check to make absolutely sure the EN bit is cleared
2. Ensure that all previous stream-specific data has been cleared (check DMA\_LISR and DMA\_HISR)
3. Set the peripheral port address in DMA\_S*x*PAR for the desired peripheral source
4. Set the destination memory address in DMA\_S*x*MA0R (and in DMA\_S*x*MA1R if using double buffer mode)
   1. Note that the memory can be the source for a memory-to-peripheral transfer, though this does not change the fact that S*x*PAR is always for peripherals and S*x*MA0R is for memory regardless of which is the source/destination
5. Set the total number of items to be transferred in the DMA\_S*x*NDTR register, which auto-decrements after each item has been transferred
6. Configure FIFO usage in DMA\_S*x*FCR
7. Set the DMA channel to be used, the stream priority, and PFCTRL to 0 in DMA\_S*x*CR
   1. Since the only peripherals that can act as a flow controller are SDMMC1 (the SD card reader) and the hardware JPEG decoder, PFCTRL will always be 0 (DMA controller is the flow controller) for the purposes of this lab
8. Set data transfer direction, burst mode, data widths, circular mode, increment offset sizes, and/or double buffer mode, and enable any interrupts and error flags desired in DMA\_S*x*CR
9. Enable the DMA controller’s clock
10. Activate the stream by setting the EN bit
    1. It will be ready for use immediately upon activation
11. Start the desired peripheral once the EN bit is successfully set to 1

**WARNING:** **If for any reason a peripheral needs to be disabled as part of the software program, the DNA stream that the peripheral is connected to MUST be disabled first (and then make sure EN = 0). Failure to do so risks data corruption or undefined behavior.**

Using HAL initialization structures to do this as per Ch. 18 in 7*69 Description of HAL Drivers.pdf* and Ch. 9 in *Mastering STM32* greatly simplifies the configuration and operation of DMA streams.

**3. Using DMA**

Once a DMA stream has been set up, it will automatically shuttle data as it has been programmed to. In your code, you will need to use the DMA versions of peripherals’ start and init functions (if applicable) to have the peripherals send data using DMA streams (e.g. HAL\_SPI\_TransmitReceive\_DMA()).

*Important note:*

While putting together the DMA init structures, you may notice that peripherals appear to have DMA handles in their HandleTypeDef (e.g. UART\_HandleTypeDef has a [Structure].hdmatx). Ignore this and use the \_\_HAL\_LINKDMA() macro instead, as it states to do in *Mastering STM32*. Using \_\_HAL\_LINKDMA() was found to be significantly more reliable (and caused significantly fewer issues) than directly using the HandleTypeDef’s inbuilt DMA link.

**DMA PROGRAMMING TASKS**

The following programming exercises will get you familiar with the performance benefits of DMA. The document *STM32 DMA Overview.pdf* will be very handy here. You will only be using peripheral-to-memory and memory-to-peripheral DMA (without FIFOs) in these exercises. You should also be implementing DMA with DMA interrupts, as polling defeats the purpose of using DMA transfers.

**PART I – UART**

Using your Lab01 Part 2 code, implement circular (i.e. continuous) DMA into uart.h/uart.c (the USB UART connection). If you’d like, you can use this DMA version of uart.c from here on out. You will need to use two streams for this. You will also need to slightly modify init.c/init.h for this. There is no need to modify your Lab01 code (except maybe for the #includes if you change any file names).

The file “stm32f7xx\_hal\_uart.c” has some helpful information on setting up DMA for use with UARTs.

*Tip:* Only enable UART DMA when you need to transfer data, and stop it when you don’t need it. You will probably want to consult the UART chapters again to find the DMA-related UART functions.

If you have trouble implementing HAL\_UART\_Receive\_DMA() in the “uart\_getchar()” function, it appears that HAL\_UART\_Receive\_DMA(), after being called the first time, will only write to the first pointer it was given. This can be worked around by creating a small global buffer (i.e. a uint8\_t) just before the uart\_getchar() function and having the function return that buffer containing the read data.

See this post for more information about this behavior:

<https://community.st.com/thread/9360#comment-163320>

Separately, it appears as though HAL\_UART\_Transmit\_DMA() has difficulty sending single bytes at a time after a UART DMA receive has occurred, but it’s fine with buffers of 2 or more. In functions like “uart\_putchar(),” which is intended to be a uart-redirectable version of “putchar(),” you can use HAL\_UART\_Transmit\_IT() for single-byte transmits. The same applies for the “echo” part of uart\_getchar(). (Using putchar() itself does not have this problem, but we can’t redirect it on-the-fly.) DMA should be used for all other sizes of transmissions.

**PART 2 – SPI**

Go back to your Lab03 Part 3 (SPI) code and implement DMA into your program to shuttle the SPI data to/from the STM32F7’s SPI peripheral and your send and receive buffers. You will need to use two SPI DMA streams and SPI interrupts for this. You may have to mix circular and normal mode DMA depending on the stream.

*Tip:* Though you may have been able to get away without doing so in part 1, you need to pay attention to the NVIC priorities you assign your DMA streams. This is particularly important if you use UART DMA and SPI DMA together.

**PART 3 – ADC/DAC**

Go back to your Lab04 Part 5 (FIR Filter) code and implement circular mode DMA for the ADC and DAC so that the CPU only handles signal processing math. You will need to use three DMA streams for this. You only need to add DMA to those assembly-using functions you made (floats + your choice).

1. Like responding to interrupts, running 30 web browser tabs of cat videos, and running circuit simulations – all at the same time, of course. [↑](#footnote-ref-1)