Microprocessor Systems Lab 6-JPEG

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	25 %				
Task 2	25 %				
Task 3	0 %				
Task 4 [Depth]	0 %				

→ Laboratory Goals

By completing this laboratory assignment, you will learn to:

1. Read from an SD card using the FatFs library
2. Decode a JPEG image using a hardware peripheral
3. Configure DMA to move data to and from the JPEG peripheral
4. Configure DMA2D to move an image into a framebuffer

→ Reading and References

R1. UM1905-stm32f7_HAL_and_LL_Drivers.pdf: Chapters 19 (DMA), 20 (DMA2D), 37 (JPEG), 57 (SD)

R2. RM0410-stm32f76xxx_Reference_Manual.pdf: Chapters 8 (DMA), 9 (DMA2D), 21 (JPEG), 39 (SD)

R3. Mastering STM32: Chapters 25 (FatFs)

R4. UM1721-stm32_FatFs_Library.pdf: Overview of FatFs Type Filesystem Driver included in STM32Cube.

R5. http://elm-chan.org/fsw/ff/00index_e.html: Documentation for Generic FatFs Driver

R6. JPEG Wikipeida Reference

R7. Lab06-SD-JPEG_Template.zip: Project Template for Lab 6 JPEG/FATFS/DMA2D

BSP - SD

For every development board, STM ships Board Support Packages (BSPs) which are libraries to help use functionality on the specific development board. These packages are available in the template project in *Libraries/BSP/STM32F769I-Discovery/* and provide functions for using the audio, eeprom, SD card slot, and other peripherals. The required source files to use the SD card, and the LCD (if available) have been enabled for this project.

The source file *stm32f769i_discovery_sd.h* contains several functions to access and use the SD card. Since this lab makes use of the FatFs library (described below), the SD card access will be performed through the file system. As a result, the library does not need to be directly called by the user.

WARNING: Do not leave the SD card in the Disco board if you put it back into the plastic case. The SD card likes to get stuck when you take it out and can break. You've been warned.

FatFs

→ FatFs Overview

FAT and exFAT are two filesystem types that are used to organize files and directories on physical media. FatFs is a generic library to use FAT and exFAT filesystems on embedded systems. FatFs is “generic” in that it does not handle any of the Input/Output (IO) layer. Instead, it requires that the user provide several functions which FatFs then uses to perform the disk IO. These functions are `disk_status`, `disk_initialize`, `disk_read`, `disk_write`, and `disk_ioctl`. Fortunately, these functions are again provided by STM to support interfacing with these filesystems on an SD card.

→ Initializing FatFs

To setup FatFs, the user must provide the functions for FatFs to perform diskio. These are in the struct `Diskio_drvTypeDef`. An instance of this struct, with all the variables initialized, is passed to the function `uint8_t FATFS_LinkDriver(const Diskio_drvTypeDef *drv, char *path)`. The “path” argument should be “0:/” as the first drive connected. The FatFs SD driver provided by STM has already created the `Diskio_drvTypeDef`, so it juts needs to be passed to `FATFS_LinkDriver` to setup FatFs.

```
typedef struct
{
  DSTATUS (*disk_initialize) (BYTE); /* !< Initialize Disk Drive */
  DSTATUS (*disk_status) (BYTE); /* !< Get Disk Status */
  DRESULT (*disk_read) (BYTE, BYTE*, DWORD, UINT); /* !< Read Sector(s) */
#if _USE_WRITE == 1
              /*!< Write Sector(s) when _USE_WRITE = 0 */
  DRESULT (*disk_write) (BYTE, const BYTE*, DWORD, UINT);
#endif /* _USE_WRITE == 1 */
#if _USE_IOCTL == 1
              /*!< I/O control operation when _USE_IOCTL = 1 */
  DRESULT (*disk_ioctl) (BYTE, BYTE, void*);
#endif /* _USE_IOCTL == 1 */
}Diskio_drvTypeDef;
```

→ FatFs Interface

The FatFs application interface is available in the driver, or on their website at R5. The functions provided, such as `f_open`, `f_read`, and `f_gets` are very similar to their Unix counterparts. The reference provides example usages for most of the functions.

◇ Task 1: Read files from SD Card

Read a FAT/FAT32 SD card (most SD cards are FAT/FAT32 by default) and display its contents on the serial terminal. A library for FatFs interaction is provided in the project template. Create a file selector to allow the user to select a file. Print out the contents of that file to the serial terminal.

Make sure to:

- Link to the provided FatFs driver with `FATFS_LinkDriver()` function
- Mount the drive (with `f_mount`) before attempting to access the files

JPEG Peripheral

→ Introduction to JPEG + JPEG Peripheral

JPEG is a commonly used lossy image compression format. It compresses images by performing a form of quantization, where the number of individual frequencies present in the image is reduced to reduce the total amount of data(R6). Several steps are required to convert an image to JPEG:

1. Image converted to YCbCr, which has lower redudancy than RGB
2. The resolution of chroma is reduced by several bits since the human eye is less sensitive to chroma than luma
3. The image is split into blocks, and each block undergoes a discrete cosine transform
4. Higher frequency components are stored with lower resolution than lower frequency components
5. All the blocks are then compressed further using a Huffman encoding (lossless encoding method with lookup table)

The STM32F769NI contains a JPEG peripheral which can convert YCbCr data to a JPEG image with a proper header, or extract the raw YCbCr data from a JPEG formatted data stream. Other chips, such as the H7 series, can also convert between YCbCr and RGB, but on the STM32F769NI a software library is provided to do this.

This hardware peripheral allows the microcontroller to offload the work of decoding/encoding an image, and when combined with DMA, can reduce the time the microcontroller spends moving data around. This is important when the microcontroller needs to handle timing critical tasks, or if it is handling user input and shound't appear sluggish.

→ JPEG Peripheral Callbacks

The JPEG peripheral works on data streams. Regardless of if it is decoding or encoding an image, it always has an input data buffer and output data buffer. When either the input or output data buffer fills up, the JPEG peripheral, and the HAL, initiate a callback. When the input data buffer fills up, the user must provide a new buffer for the JPEG peripheral to read from (or refill the same buffer and tell the peripheral to look at the same buffer). When the output buffer fills up, the user must provide a new location for the output to be placed.

The following shows the flow of using the peripheral to decode a jpeg image:

1. Make sure to call the `HAL_JPEG_IRQHandler` in the `JPEG_IRQHandler`. Otherwise the HAL won't be able to process any callbacks.
2. Initialize the JPEG with `HAL_JPEG_Init`. Note that the JPEG handle's `Instance` must be specified. No other handle field need to be initialized.
3. Read some data from the jpeg file into a buffer
4. Start the decoding with `HAL_JPEG_Decode` (or `HAL_JPEG_Decode_DMA`)
5. Wait until the decode is complete `HAL_JPEG_DecodeCpltCallback` will be called

- If `HAL_JPEG_DataReadyCallback` is called, a new output buffer is needed. Configure this output location with `HAL_JPEG_ConfigOutputBuffer`
- If `HAL_JPEG_GetDataCallback` is called, more input data is needed. Read more from the file. Configure the JPEG to look at the new input buffer with `HAL_JPEG_ConfigInputBuffer` (even if the new buffer is at the same place as the old one)

6. The data in the configured output buffer location(s) is now in YCbCr format!

◇ Task 2: JPEG Decode Image

Decode an image from the SD card using the JPEG peripheral. Some helper code has been provided for assistance.

- The easiest way to read in data is to create a buffer, use FatFs functions to fill it, then pass it to the JPEG. When new input data is required (`HAL_JPEG_GetDataCallback`), simply re-fill this buffer with more data, and reconfigure the JPEG with this same buffer.
- Put the output data starting at `(uint8_t*)JPEG_OUTPUT_DATA_BUFFER`. This puts it into memory in a region generally used by layers for the LCD. This was done based on the examples provided by STM and makes it easy to configure later output locations: just append them to the previous output data buffer each iteration. `HAL_JPEG_DataReadyCallback` reports the number of bytes decoded and the previous location of the buffer; where both can be used to calculate the next output buffer location.
- To determine if the image was decoded correctly, information about the image, such as size, can be retrieved from the `JPEG_ConfTypeDef` structure, from the `HAL_JPEG_GetInfo` function.
- Since the data is extracted into YCbCr format, code has been provided to convert the data to RGB. Call `uint8_t *colorConversion(uint8_t *addr, int num_bytes)` to convert the colors. The return value is the pointer to the location of the RGBA data.
- Print the data to PuTTY (terminal) using the `printPutty` function. Make sure to size the terminal such that it is large enough for the image. The terminal must also have 24-bit color mode enabled. It is highly recommended to test with small images (e.g., 60x60 px^2) so that they are easily viewable.

  ```
  void printPutty(uint8_t *raw_output, JPEG_ConfTypeDef info)
  ```

- The Memory tab in STM32CubeIDE can be used to inspect locations in memory to see if data is being moved to that location.
- Several different image sources were tested when developing the lab. JPEG images downloaded from the internet, and created in microsoft paint, seem to work as expected. JPEG images created in GIMP seem to cause the peripheral to lock up.

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 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," 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 (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 gererally 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.

→ 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 R2 (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 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 (DMAx_Streamy_IRQn) 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.

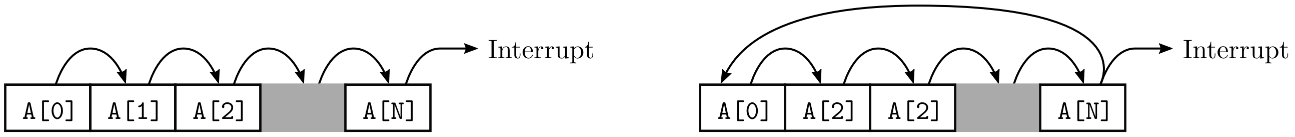

Figure 1: Flow of DMA accessing buffer A with length N in (left) normal and (right) circular modes.

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 but 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 is capable of consuming the buffer before the next DMA interrupt. After the next interrupt, the DMA moves back to A while the software consumes B.

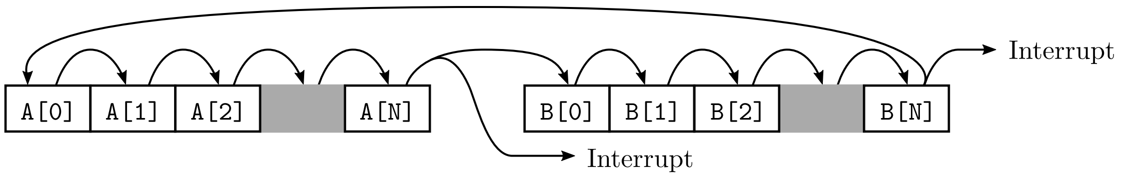

Figure 2: Flow of double-buffered DMA accessing buffers A and B, both with length N.

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 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 mde, 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())[1]. 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.

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

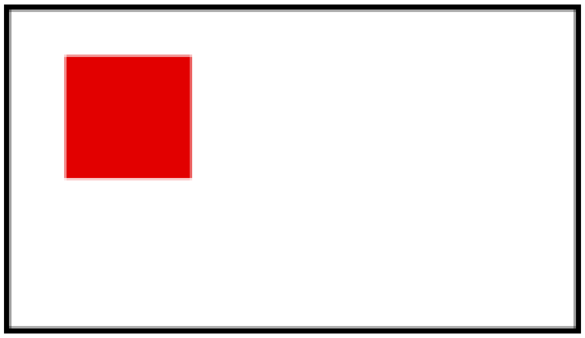

Figure 3:

→ 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 3: JPEG Decode Image - Transfers with DMA

Repeat Task 2, but instead use `HAL_JPEG_Decode_DMA()`. Configure the DMA for the JPEG in the `HAL_JPEG_MspInit()` function. Note, two DMAs will be needed, one for data input and one for data output.

→ Purpose of DMA2D

While the DMA peripheral is useful for moving data from one location to another, it is not ideal for working with images. Generally, images will be stored in a 1D array, as done in tasks 2 and 3. However, when images are moved to a buffer to be displayed on a screen they get split up. Consider the red image displayed on the screen in Figure 3. Since the screen is one large array, the red box gets spread across the screen framebuffer, it is not feasible to use standard DMA. Standard DMA would only be able to copy one row at a time, before having to be set to a new row by the CPU.

DMA2D also has several other useful features for working with images. It contains a foreground and background layer, and can use the alpha component of the pixel to perform blending with the foreground and background. It can also convert between pixel representations, such as ARGB8888, RGB888, RGB565, and therefore is very versatile for different tyes of images and displays. It can also just fill rectangles with color, useful for drawing backgrounds/simple icons without CPU intervention.

→ Display Memory Layout

Different displays use different pixel configurations for the display. The display on the Discovery board is memory mapped, meaning data written to a certain location in memory is drawn on the screen. This display uses 4 byte pixels, with 8 bits each for red, green, blue, and alpha. See the ARGB8888 color mode in the DMA2D section of the reference manual(R2). As a result, the byte offset for a certain pixel can be calculated with the algorithm in Equation 1, where x and y are the pixel coordinates, w is the width of the screen in pixels (there is a function to find that in the lcd BSP). You will need to use this even if you don't have the LCD to put the image at the correct place in memory.

$$\text{byte offset} = (y * w + x) * 4 \tag{1}$$

→ DMA2D Initialization Settings and Layers

Unsurprisingly, DMA2D has many settings that need to be configured. The `DMA2D_HandleTypeDef` has two sets of settings that need to be configured: `Init` settings and `LayerCfg` settings. `LayerCfg` is an array, you only need to set up `LayerCfg[1]` (foreground)

Notes about init settings:

- ColorMode: should be ARGB8888 to match the LCD pixel format
- OutputOffset: Used by the DMA to figure out where to place the next row. It adds this value to the address of the end of one row to start the next row. See `DMA2D_FGOR` register in reference manual(R2).

Notes about LayerCfg settings:

- InputOffset: This value is provided. It is used for some extra buffer to make the data size in memory a multiple of 8 or 16 (depending on image color data). See template for the code used, adapted from STM example.

→ Using DMA2D

1. Setup `DMA2D_HandleTypeDef`: `Instance`, `Init`, `LayerCfg[1]`
2. Intialize DMA2D with `HAL_DMA2D_Init`
3. Choose foreground with `HAL_DMA2D_ConfigLayer` and a `LayerIdx` of 1
4. Start the transfer with `HAL_DMA2D_Start`
5. Wait for the transfer to complete with `HAL_DMA2D_PollForTransfer`

◇ Task 4: [Depth] DMA2D - Moving images in memory

Use DMA2D to move the image into the framebuffer. You should be able to choose an x and y location where the top left corner of the image will be placed. The top left corner is (0,0), and x increases to the right, and y increases going down. If using a DISCO board with LCD, it will appear on the screen. Otherwise, use the provided function

```
void printPuTTY2D(uint8_t *framebuffer,
                  uint16_t xPos,
                  uint16_t yPos,
                  JPEG_ConfTypeDef *info)
```