iUS Documentation

User Guide: App

Local Beamforming

Enter our mobile app and click on the “Local Beamforming” button. Click on the select file button. For multiple files, click select on the top right corner before selecting each file. Once the files have been selected, click done and an activity monitor should appear showing that the images are being processed. The images will display on the screen, swipe to the left to see each consecutive image. For gif mode, click on “toggle gif” for a video display of the images.

User Guide: Ultrasound Simulation in MATLAB

Clone the CIS_iUS repository and start MATLAB and navigate to the “Image_Processing” directory. Run the ‘simulation.m’ script and a file should have been created, ‘rf_data_[timestamp].mat’, where [timestamp] is the current date. Then run the “..\CIS II\CIS_iUS\Image_Processing\createCSV.m” to create the CSV file that will be used for local beamforming and testing the C code.

User Guide: Image Processing in MATLAB

Although image processing is meant to be performed on the mobile app, it is also available in MATLAB for testing or other purposes. To process the data and display the image, run the “..\CIS II\CIS_iUS\Image_Processing\make_image.m” script. It currently only supports processing 1 ultrasound image at a time.

Clinical Requirements

Mobile app

The clinical goal of the implantable ultrasound device is to monitor neurosurgical patients after surgery. The current standard is to perform a MRI scan every 3 months, amounting to 4 scans a year. With this ultrasound device, the rate of imaging can increase significantly, allowing neurosurgeons to better monitor and enable them to catch the regrowth of tumors or possibly even incorporate machine learning to predict tumor regrowth. Ultimately the goal of the device is to take images either every day or once every week (frequency not yet determined, could depend on battery capacity or method of wireless charging) and for the neurosurgeon to review the images on the app during appointments.

Hardware Device

The clinical goal for the device is for it to fit in a 14 mm burr hole that is created during every neurosurgery. Three of these holes are created so it would also be possible to take advantage of all three, however that would mean each device would need to somehow wirelessly communicate between each other. The preference is for the device to fit into one 14 mm circular hole, with a height of 4-5mm.

Hardware-Mobile App Interaction for Bluetooth Monitoring

In order to monitor the patient's brain through bluetooth, the surgeon needs to be able to control several properties of the ultrasound device from the designated iPad or iPhone device. At each appointment, it would be useful for power conservation purposes to toggle between a low fidelity, or low resolution, mode and high fidelity, or high resolution, mode. A surgeon should also be able to change the imaging frequency (once a day or once a week, etc.) depending on the images he sees at each appointment. On the patient side, it is important to see the battery life, and charging status of the implanted device to help the patient keep the device charged and ready to use, just like a patient would charge his/her phone when the battery indicator is low.

Safety

Above all, the most important feature of this device is safety. There is a team of masters students working on the security of the device, to prevent hacking via bluetooth since the data will be transmitted wirelessly. In terms of materials, this device would be fully encased in PMMA, which is a clear plastic acrylic material that is biocompatible. This would be securely fixated on the skull using screws and plates and has been used in other FDA-approved neural implants.

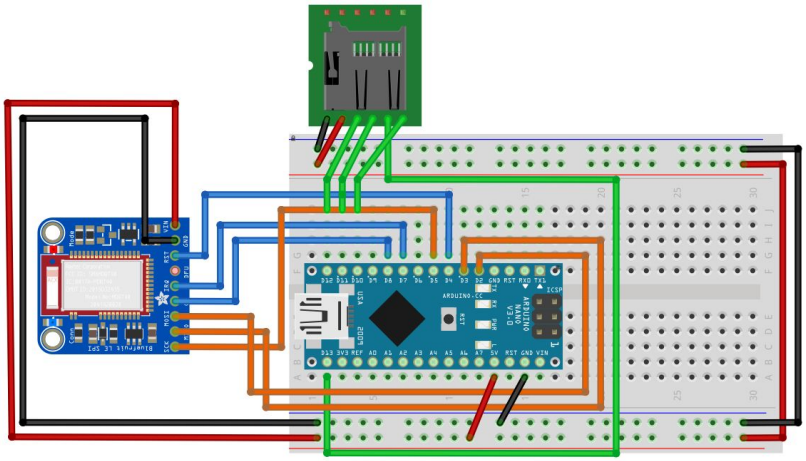
Design Decisions

1. Focusing on CSV file bluetooth communication instead of streaming data in real time.
    a. The rationale behind this decision is our major clinical requirement, being able to take images at a certain frequency and for the clinician to review them after several images have been taken.

    b. Since real time imaging is not required, we will focus on data acquisition and beamforming locally saved data in CSV files for image display.
2. Developing using iOS instead of Android.
    a. iOS development using XCode was the simplest solution given most of the lab members have iPhones, which will enable rapid testing and demos.
3. Using Arduino UNO, Adafruit Bluefruit LED SPI Friend, and microSD module for bluetooth data transmission to simulate ultrasound device
    a. Although this does not satisfy the clinical need of a miniature device, this Arduino setup was chosen for its prototyping ease and ability to simulate the ultrasound device with minimal code on the Arduino end.
    b. With time permitting, (or in the future), we aim to minituarize this hardware setup using Microchip chips.
4. Simulating ultrasound data using the Field II library in MATLAB instead of purchasing the raw data package from Clarius
    a. We did not want to rely on data output from Clarius, as this may change when a CMUT array is used. The highly costly package was not necessary for our project because the ultimate goal of the lab is to also design the ultrasound device, in which we would not require this data package anymore. Simulating the ultrasound data should be sufficient for testing our mobile application.
5. Develop the image processing code in C based on MATLAB functions
    a. With code in MATLAB outlining the steps to processing the rf_data, we chose to base the code in C on these steps because it provided an accurate way to test each function we wrote in C

Hardware Interface Documentation

Arduino Setup Schematics



[connect the Arduino nano to laptop via USB for power]

- The green chip represents the microSD Card Module: the pins are as follows:

Adafruit Chip Bluetooth Communication

The bluetooth chip, Adafruit Bluefruit LE SPI Friend, uses SPI protocol, or serial peripheral interface, for two devices to send and receive data. Typically hardware SPI is used (where the SCK, MISO, MOSI pins are connected to digital pins 11, 12, and 13. However, since the microSD card module requires these digital pins as well, software SPI is used instead. The difference between software and hardware SPI is speed. Software SPI is typically slower than hardware, however, in this case, it is the only option. For software SPI, the pins SCK, MISO, and MOSI can be wired digital pin, in this case, pins 5, 3, 2. These pins are specified in the “BluefruitConfig.h” file.

SenMod Micro SD Card Communication

The microSD card communicates via a standard SPI interface (very similar to the Adafruit bluetooth chip). In this case, it utilizes hardware SPI, by connecting SCK, MISO, and MOSI to pins 11, 12, 13.

Software Documentation

MATLAB

All MATLAB code is located in the "../CIS_iUS/Image_Processing" folder. The simulation code is dependent on the Field II MATLAB library, which is located in the "../CIS_iUS/Field_II" folder. The following code was developed based on the Field II example that is located in the "../CIS_iUS/Image_Processing/Field_II_Example" folder for reference. It is later used as a reference for developing the image processing code in C.

createCSV.m

- Description: creates CSV text files from selected workspaces saved with simulation.m.
- Code walkthrough:
    - Ask user to select the ".mat" files
    - For each workspace:
        - Load the variables
        - Open a text file with the same filename as the .mat file
        - Print the number of lines used in the simulation and size of data matrix on the first line
        - Print the data matrix to file by appending with a comma delimiter
        - Close the file

cyst_pht.m

- Description: creates a model of a cyst phantom that contains 5 point targets, 5 different sized water-filled cysts, and 5 different-sized scattering regions.
- Dependent libraries: Field II
    - Directly obtained (with no additional modifications) from the Field II Examples → refer to Field II documentation for further details.

make_image.m

- Description: loads saved workspace from simulation script and produces an image that shows 60dB of dynamic range.
- Code walkthrough:
    - Ask user to select a saved workspace created by simulation.m
    - Initialize a series of constants
    - For each line, find the hilbert transform of the input data
        - Note: in the MATLAB documentation, the definition of the hilbert transform is a function that returns a complex array of numbers, in which the real part is the original input and the imaginary part is the hilbert transform. Instead of using just the hilbert transform, the imaginary component of that output was combined with the real input array. This is primarily done for consistency between the output of the MATLAB code and C code for image processing.

- Perform a logarithmic compression
    - The sampling frequency is initialized to be 10. For a low fidelity (low resolution) image, increasing the frequency to 70 will greatly improve time performance.
- Create the interpolated image
    - There are 2 ways of doing this, one that was used to outline the C code, and one that is more efficient in MATLAB due to pre-defined functions.
    - The current code takes advantage of the interp function in MATLAB.
    - Using MATLAB documentation of the interp function, a very similar sequence of functions is written in the comments. These functions are easier to implement in C and is exactly how the interp function was written. The only difference is that the sequence of functions introduces **a delay in the data** that was fixed by shifting the output by half the size of the FIR filter.
- Crop and display the image
    - The uncommented lines (98-105) were taken from the Field II examples. They take advantage of built-in MATLAB functions for cropping and display the image scaled in SI units.
    - In the comments (69-91), there is code written to manually crop the image. However, it does not currently scale the image in SI units.

rf_data_[timestamp].mat
- Description: example workspace created by the simulation.m script. Filename is "rf_data_" + a date. The dates were manually changed for testing purposes. Workspace contains 1 "data" variable that holds the rf_data and t_start variables computed from simulation.m

rf_data_[timestamp].txt
- Description: example CSV text file created by the createCSV.m script from the rf_data_[timestamp].mat file.

simulation.m
- Description: simulates a linear array B-mode system scanning of an image. Created by combining scripts from the Field II Example for easier user control.
- Dependent libraries: Field II
- Code walkthrough:
    - Initialize the Field II Simulation.
        - Replace the second input to path() with the name of the directory that holds the Field II m-files and executable.
    - Make the scatters for the simulation using cyst_pht()
    - Begin the simulation by initializing transducer constants
    - Set the sampling frequency
        - Note: increasing this should create a high fidelity image, decreasing would create a low fidelity image → should directly impact battery consumption of an ultrasound device if sampling frequency can be toggled before acquiring and transmitting raw data

- Perform simulation → for further details, refer to the Field II library
- Format the variables for saving in 1 large data matrix
- Get the timestamp and format to display just the calendar date, not time
    - Decided to only mark data with the date not time because in the clinical setting, the ultrasound device would take images of at most a frequency of 1 image per day.
- Save data variable in the file “rf_data_[timestamp].mat”

C

All image processing for XCode is written in C. There is a copy of the relevant code in the XCode project, "../CIS_iUS/CISII/CISII" with minor modifications for the XCode compiler, but the code is primarily developed in "../CIS_iUS/CISII/image_proc_c_driver/", based on the image processing in MATLAB, where there is a Microsoft Visual Studio solution setup in the folder "../CIS_iUS/CISII/image_proc_c_driver/image_processing". There is an included library, FFTW, which includes fast C routines for computing the discrete Fourier transform (DFT). This is located in the folder "../CIS_iUS/CISII/image_proc_c_driver/image_processing/fftw-3.3.5-dll64". Further documentation on this library can be found here: http://www.fftw.org/fftw3_doc/Introduction.html#Introduction

The solution is set up to include the library above.

Data is stored and written to the folder "../CIS_iUS/CISII/image_proc_c_driver/data".

- rf_data.pgm: initial example image file for the simulated cyst phantom rf data.
    - .pgm is used for simplicity and due to the simulated image being grayscale by default.
    - Is not compatible with Swift's image display
- rf_data.bmp: example image file for simulated cyst phantom
    - Used to allow Swift in XCode to read and display the image natively
- rf_data.txt: example CSV text file used for the image processing to create the .pgm/.bmp image.
- test_data.txt: test CSV used for debugging functions

Image processing C files are located in the "../CIS_iUS/CISII/image_proc_c_driver/image_processing/" folder, which also contains the solution setup for Microsoft Visual Studio. The current solution is set to output an application for direct running in the terminal to test the image processing using the main function within csv_parser.c. There are also commented out main functions within other files that were used for testing individual functions.

csv_parser.c

- Description: defines a function that parses the CSV created by the MATLAB createCSV.m script to extract the rf_data needed to create the ultrasound image.
- Code walkthrough
    - printData()
        - Used to print debugging information for the custom struct DataMatrix. With input 0, the size information of the struct will print. With input 1, the size and data stored will print.

    - readCSV()
        - Open the CSV file for reading, exit if unable to open
        - Read the first line to get the total number of lines and maximum number of elements
        - Read the data line by line
            - The first element is the t_start value

            - Using the maximum number of elements, read in the data and fill in with zeros for shorter vectors (when the buffer return null)
        - Create the struct to hold the rf_data
        - Close the file to save
        - Free variables that were allocated in the heap with malloc() or calloc()

- Main function
    - Hard-coded file paths were used for testing
    - CSV file is read and data is stored in a DataMatrix struct
    - With read success, make the image and save the .bmp file
    - Free the data variable
    - Return success boolean

csv_parser.h
- Description: header file with function declarations for csv_parser.c

datamatrix.c
- Description: creates the custom struct DataMatrix defined in the header file.

datamatrix.h
- Description: includes the function declaration and defines the custom DataMatrix struct with the total size of the data array, number of rows, number of columns (primarily used for 2d array indexing) and the data array itself.

filter.c
- Description: defines functions used to filter data
- Code walkthrough:
    - upsample(): upsamples a row of data by inserting n-1 zeros between each input value
        - Check that input DataMatrix is a 1D row array, return original DataMatrix if not
        - Initialize an output array
        - For each input value, insert the value into the output array and n - 1 zeros
        - Create the output DataMatrix and return

    - firFilter(): designs a simple low-pass FIR filter by returning the filter coefficients
        - Note: this code was adapted from https://liquidsdr.org/blog/basic-fir-filter-design/ Refer to this link for further details on FIR filters and design
        - Initialize the array for the output filter coefficients using the input filter length
        - Calculate each filter coefficient by generating a time vector centered at 0, a sinc function using the time vector, a Hamming window, and multiplying the sinc function with the hamming window for the coefficient
        - Create a DataMatrix using the filter coefficient array and return

    - filter(): apply an FIR filter using convolution

        - Note: this code was adapted from https://lloydrochester.com/post/c/convolution/ Refer to this link for further details about the mathematical concept of convolution
        - Initialize convolution output size using the filter length and input length
        - In the inner for loop, an if/else block was added to handle infinite values, which occur in the log compression of the image processing which occurs before the filtering is called. If an infinite value is reached, store the infinite value back into the output and break the inner for loop.
        - Compute the delay caused by the filtering operation
        - Resize the output and shift by the delay to match the size of the input for image processing

    - Commented out main function
        - Used to test the filtering functions defined above by hard-coding inputs (obtained through MATLAB) and comparing outputs to outputs of built-in MATLAB functions

filter.h

- Description: declares functions in filter.c, value for pi constant and a min/max definition (obtained from https://lloydrochester.com/post/c/convolution/)

hilbert.c

- Description: creates a discrete-time analytic signal using the Hilbert transform based on the algorithm outlined in the MATLAB documentation: https://www.mathworks.com/help/signal/ref/hilbert.html
    - The complex analytic signal is made up of the real part, which is the original data, and an imaginary part, which contains the Hilbert transform
- Dependent libraries: FFTW (Fastest Fourier Transform in the West). Link to extensive documentation: http://www.fftw.org/fftw3_doc/Introduction.html#Introduction
- Code walkthrough:
    - convertDoubleToComplex(): convert a double real array to a complex array by inserting 0 for the imaginary part and the input for the real part

    - hilbert()
        - Initialize the input and output complex arrays for the fourier transform
        - Create the plan for a forward complex one-dimensional discrete fourier transform
        - Initialize the input array after creating the plan
        - Execute the plan to compute the fourier transform of the input, and destroy the plan
        - Initialize an array of constants to 0
        - For even input length, the constant is 1 for the first and middle element, 2 for the first half of the input (not including the first and middle element) and 0 otherwise

        - Create the plan for a backward (inverse) complex one-dimensional discrete fourier transform
        - Initialize the input to the inverse fourier transform as the element-wise product of the array of constants, h, and the complex array output of the forward fourier transform
        - Execute the inverse fourier transform and destroy the plan
        - Normalize the output using the length of the original input
        - Free variables that were allocated using malloc() or calloc()

    - Commented out main function
        - Used to test the hilbert function and compare with the MATLAB output

hilbert.h
- Description: declares functions in hilbert.c.

make_image.c
- Description: perform the image processing based on the MATLAB make_image script.
- Code walkthrough:
    - max_array(): find the maximum finite value of an array

    - min_array(): find the minimum finite value of an array

    - save_image(): save a DataMatrix with data in a range of [0 255] as an image (.pgm) file
        - Open the file for writing in binary mode, if unable then exit
        - Print the magic number to file
            - P2 is used for a portable graymap (pgm) ASCII type. For further details on the .pgm format refer to: https://www.geeksforgeeks.org/c-program-to-write-an-image-in-pgm-format/
        - Print the height and width of the image to file
        - Print each value in the array
        - Close the file and return

    - createBitmapFileHeader(): helper function that creates the bitmap file header using the height and width of the image and padding size
        - File header includes a signature (‘BM’), image file size in bytes, unused reserved bytes (default to 0), and a pixel offset for the start of the data

    - createBitmapInfoHeader(): helper function that creates the bitmap info header using the height and width of the image

    - save_image_bmp(): save a DataMatrix with data in a range of [0 255] as an image (.bmp) file

        - Convert DataMatrix to an unsigned char matrix
        - Get the Bitmap file and info header using the helper functions
        - Open the image file for saving
        - Write the file header and info header to the file
        - For each row in the matrix, get each line and write the line to file and pad with 0's
        - Close the image to save
        - Free the image variable that was allocated using malloc()
- linearInterpolation(): perform a linear interpolation given 2 points and the value at those 2 points

- resize_image(): resize the image using bilinear interpolation
    - Note: adapted from https://www.ece.iastate.edu/~alexs/classes/2012_Fall_185/sample_exams/midterm2/2010/Lab_v1/3_BilinearInterpolation.c
    - Calculate the scaling factor, from pixels to SI units (mm)
    - Calculate the dimensions of the new image
    - Calculate the scaling factor to convert between indices of the new image and indices of the original image
    - Initialize the variable to hold the data of the new image
    - For each pixel in the new image:
        - Calculate the corresponding indices in the original image
        - Get the indices of 4 corners surrounding the current pixel
        - Perform a linear interpolation between the top 2 corners
        - Perform a linear interpolation between the bottom 2 corners
        - Perform a linear interpolation in the y direction to get the bilinearly interpolated value for the new pixel
    - Create a DataMatrix to hold the data of the new image
    - Free the variable allocated using malloc()

- gamma_correction(): perform a gamma correction (a nonlinear operation used to change luminance) on an image
    - For each pixel, calculate the new value using the equation: 255 * (val/255) ^ gamma

- make_image()
    - Initialize constants used for image processing
    - Find the envelope of the input signal (rf_data for each line in this case) by computing the absolute value of the Hilbert transform
        - For details on envelope detection, refer to: https://www.mathworks.com/help/dsp/examples/envelope-detection.html
    - Sample the envelope and compute the logarithmic compression

        - Changing this sampling frequency affects the resolution of the resulting image (higher frequency creates lower resolution)
    - Design a FIR filter
    - Interpolate the image by upsampling each row of the logarithmic envelope and filtering it using the FIR filter
    - Crop the image to be [-20 20] mm wide and [35 90] mm tall
    - Convert the range of values to [0 255] (for images)
    - Resize the image to be proportional to the SI unit dimensions
    - Perform a gamma correction to the image to darken the image, making it resemble the Matlab produced image
    - Save the image
    - Free variables allocated using malloc() or calloc()

make_image.h

- Description: declares functions for make_image.c

XCode

The iOS app was developed in XCode and is dependent on the FFTW library as mentioned in the C code. However, one important distinction is that since the C code in Microsoft Visual Studio was built on a Windows machine, and the FFTW library in the "image_proc_c_driver" folder does not translate to the one used for app development.

A local copy of the FFTW library for Mac (not iOS) is located in "../CIS_iUS/CISII/CISII/fftw-3.3.8". This was downloaded from http://www.fftw.org/download.html, in which the .tar.gz file was extracted and installed. To install the library, in the Mac terminal, run "./configure", "make", "make install" as 3 separate consecutive commands. However, this library is built for macOS, which is different from being built for iOS. In the same local library folder, the script "universalFFTW.command" will build the library for iOS. This script was modified from an answer on: https://stackoverflow.com/questions/3588904/how-do-i-link-third-party-libraries-like-fftw3-and-sndfile-to-an-iphone-project

To make the "universalFFTW.command" script executable, open the terminal in the fftw library folder and run "chmod u+x (full file path)" where (full file path) could be "~/Desktop/.../universalFFTW.command". Then run "./universalFFTW.command" to create the fftw universal library which will be located within the library folder at "../fftw-3.3.8/ios_libs". This library should resolve any build error with "Building for iOS but the library is built for macOS…" and is included in the repository.

In the XCode project, to include the iOS library, make sure that under the Build Phases of the project, under "Link Binary with Libraries" the file "libfftw3_ios.a" is included. Then, under Build Settings, there should be a linker flag for "-lfftw3_ios", header search paths and library search paths with the full path to the "ios_libs" folder and "api" folder (also located within the "fftw-3.3.8" folder. This should all be configured in the current XCode project but for debugging purposes, start here.

Within the XCode project, in the CISII folder ("../CIS_iUS/CISII/CISII") there is also a copy of the C files for image processing that were modified to compile using the C99 utility. To use the C code in Swift, there is a bridging header ("CISII-Bridging-Header.h") that imports the header files so that the project can access the C code and call functions directly. Please refer to the C documentation for this code for further details.

The Application is split up into the following swift files that each serve a unique purpose in the app.

The following files are the ones that were added to the application, not including the default files of AppDelegate.swift and SceneDelegate.swift.

SwiftUI utilizes three essential stacks in order to create views: a Horizontal Stack (HStack), a Vertical Stack (VStack) and a Stack on top of another view (ZStack). These simply align views by which stack is used in the UI code.

ContentView.swift:

The content view is the home page created for the application. Ultimately, this view utilizes three buttons that are nested in a Navigation View so that they can be linked to each of the other views created. Each of the buttons are customized in order to have the colors that are included in the given logo mockup.

ClariusView.swift:

The Clarius View is the Navigation destination from the home screen that has text outlining the proposed use of the view with further development.

BluetoothView.swift:

The Bluetooth View is the Navigation destination from the home screen that has text outlining the proposed use of the view with further development.

LocalView.swift:

This view uses several boolean operators to distinguish which nested view should be displayed. The initial screen contains a button in order to Toggle GIF view (only available after the images are displayed in the scroll view), the default image (a placeholder for the scroll view with the one to multiple images), a button to select files (shows the native Files application in order to select files), and simple text outlining the basic function.

The initial workflow begins by selecting the file from the native Files application. The DocumentPicker subview is then called upon and handles the selection of the files in the application. The paths of the files are used in the subview in order to run the imageProcessing code on the selected files and the outputs (images) are then stored in the same directory as the selected files. The activity monitor begins once the select files button is pressed and ends once the imageProcessing code is finished running. Once the paths are established, our view is constantly looking for a path to be given, so once provided after the imageProcessing code is finished running, the images are displayed in a scroll view to the user with the associated file name as a label. Once the images are then in the application, we can then toggle our GIF view button in order to show the subview ImageCarouselView which essentially scrolls through the images in a given time, giving the look of a GIF.

ImageCarouselView.swift:

This View is used as a subview in the LocalView once the GIF Toggle button is pressed. Essentially this view establishes a series of adjacent images based on the number of images imported into the view. The adjacent images are then cycled through by a set timer to scroll one image in a set amount of time and the length of the timer creates the illusion that it is a GIF.

Arduino

Arduino code is organized so that each folder within the 'Arduino' folder is a collection of code (.ino or .h) that will upload to the Arduino altogether (even if the code is just 1 file, it will be within a folder due to Arduino's default file organization).

Debugging: if code is unable to be uploaded to the Arduino, check that the proper port was selected or if the proper driver is installed for the hardware used. Some Arduino chips use the FTDI chip, others use the CH340.

Instructions for downloading the CH340 drivers can be found here:
https://learn.sparkfun.com/tutorials/how-to-install-ch340-drivers/all
Instructions for downloading the FTDI drivers can be found here:
https://learn.sparkfun.com/tutorials/how-to-install-ftdi-drivers/windows---quick-and-easy

Syntax below will be: **Folder_name:** file_name
Folder names can be found within the Arduino folder of the repository.
Due to incremental code development, the following documentation is organized from simple testing code to the final pieces of code that will be used in our project. Those will be noted as such; therefore assume all other pieces of code are only used for testing the hardware.

sdcard_comm_read: sdcard_comm_read.ino

- Description: This function reads in a saved text file (named "DATA.txt") on the SD card of the Arduino setup (that is connected to the computer) and prints each line onto the Serial Monitor.
- Dependent libraries: SPI.h, SD.h
- Code walkthrough
    - Begin the serial communication using a baud rate of 9600 → adjust if serial monitor outputs wonky symbols
    - Wait for the serial port to connect
    - Begin initializing the SD card (print statements will show if successful or not)
    - SD card will open the file "DATA.txt"
    - If the file was successfully open, read each line, write to the Serial Monitor and repeat until there are no lines left and close the file

sdcard_comm_txtfile: sdcard_comm_txtfile.ino

- Description: This function writes pre-programmed strings to a text file on the SD card.
- Dependent libraries: SPI.h, SD.h
- Code Walkthrough
    - Begin the serial communication using a baud rate of 9600 → adjust if serial monitor outputs wonky symbols
    - Wait for the serial port to connect
    - Begin initializing the SD card (print statements will show if successful or not)
    - SD card will open the file "test.txt" for appending.

- Both strings and numbers will be appended to the file in different lines.
- Close the file and print “done” to signify the completion of data saving.

sdcard_comm_serial: sdcard_comm_serial.ino

- Description: This function sends keyboard input line by line to the Arduino via the Serial Monitor. Each line is appended to a “DATA.txt” file. At the end of data entry, user must send a period (“.” without quotations).
- Dependent libraries: SPI.h, SD.h
- Code walkthrough
    - Begin the serial communication using a baud rate of 9600 → adjust if serial monitor outputs wonky symbols
    - Wait for the serial port to connect
    - Begin initializing the SD card (print statements will show if successful or not)
    - SD card will open the file “DATA.txt” for writing
    - Input in serial monitor will be read in as a string
    - If the input is not empty (nothing entered) or isn’t a period, append the string to the text file and print the same string to the Serial Monitor.
    - If the input is a period, close the file (saves to SD card).

sdcard_comm_readwrite: sdcard_comm_readwrite.ino

- Description: This function uses keyboard input in the Serial Monitor to read or write data to the Arduino. Prompts in the serial monitor will guide users for the type of input expected at each step. This function combines the 2 functions above and broadens data entry by allowing file name input.
- Dependent libraries: SPI.h, SD.h
- Code Walkthrough
    - Begin the serial communication using a baud rate of 9600 → adjust if serial monitor outputs wonky symbols
    - Wait for the serial port to connect
    - Begin initializing the SD card (print statements will show if successful or not)
    - Serial Monitor will pause 5 seconds for user input → read/write, and print the user input.
    - Serial Monitor will pause 5 seconds for user input → file name (include .txt extension), get rid of the new line character, and print the user input
    - If the first user input was “read”, the file with the file name provided by the second user input will open for reading data
    - If the first user input was “write”, the file with the file name provided by the second user input will open for writing (appending) data
    - Else, an error statement will print in the Serial Monitor.
    - If the file was successfully opened for reading, each line will be read in, and printed in the Serial Monitor until there are no lines left.
    - If the file was successfully opened for writing, and the input in the Serial Monitor is not empty or a period, it will be appended to the text file and the read in string will print to the Serial Monitor. If the input is a period, data entry will end.

sdcard_bluetooth_readwrite: BluefruitConfig.h

- Description: This header file is used to configure the pins and type of communication to use. In the comments there are other options available, however, currently, the file is configured for software SPI where the SCK pin goes to the digital 5 pin on Arduino, the MISO pin goes to the digital 3 pin on Arduino, and the MOSI pin goes to the digital 2 pin on Arduino.
- The exact location of the pins (5, 3, 2) can be changed by modifying lines 54-56.

sdcard_bluetooth_readwrite: sdcard_bluetooth_readwrite.ino

- Description: This function allows the user to read and write data to the SD Card using the Serial Bluetooth Terminal App on a mobile phone. Once connected to the Adafruit chip, the terminal will display prompts asking for specific user input for file reading or writing. All user input must terminate with a '#' to signify the end of that line and a period '.' is used to signify the end of user input for writing data.
- Dependent libraries: Arduino.h, SPI.h, Adafruit_BLE.h, Adafruit_BluefruitLE_SPI.h, Adafruit_BluefruitLE_UART.h, SD.h
    - Include the header file: BluefruitConfig.h
- Code Walkthrough
    - Set the pins defined in the BluefruitConfig.h file by creating the bluefruit object using software serial
    - Begin the serial communication using a baud rate of 9600 → adjust if serial monitor outputs wonky symbols
    - Begin initializing the SD card (print statements will show if successful or not)
    - Initialize the bluetooth module
    - Once bluetooth module is found, perform a factory reset if enabled
    - Disable the command echo from the module
    - Print the module information
    - Set the module to DATA mode
    - Mobile terminal will ask for user input and wait 10 seconds: read or write
        - User input is read in by checking for available input and reading in each character and building the character array. After reading in the line, it is converted to a string, trailing symbols are removed.
        - All user input must end with a number symbol ('#') → this is used to remove extra trailing symbols at the end of the input.
    - Terminal will read the input and print it to both the serial monitor on the laptop and the terminal itself
        - Currently, the printing to terminal function takes in a string, converts it to a character array and prints using a function of the bluefruit object which prints the ASCII number in the terminal.
    - Terminal will ask for user input for the filename and wait 10 seconds
    - If the first input was read, the file with the filename from the second input will open for reading

- If the first input was write, the file with the filename from the second input will open for writing, otherwise the terminal will print an error message
- For reading, if the file was opened, it will print each available line to the terminal, close the file and print a statement
- For writing, the input will be read in, if it’s not empty or a period, that input will be appended to the file and printed in the terminal for confirmation. If it is a period, data entry will terminate, the file will be closed and saved, and a statement printed in the terminal