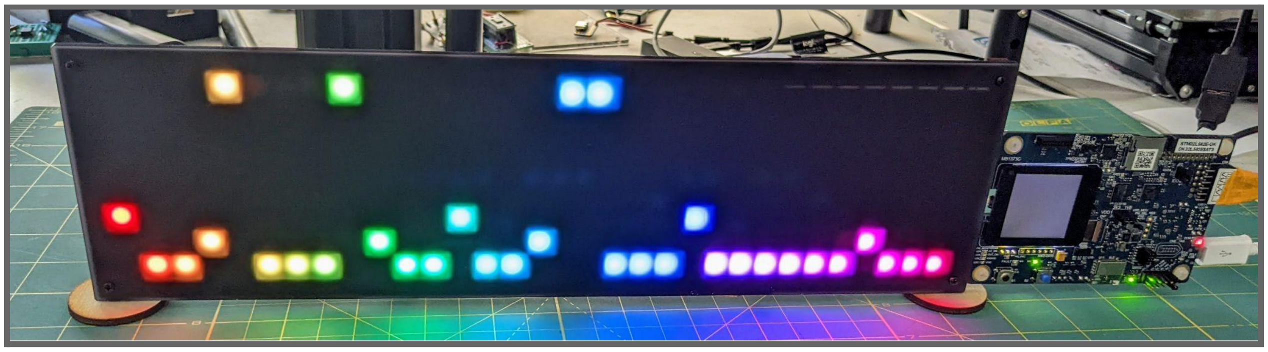
Spectral Speech Rate Analyzer

aka

"DMA All the Way"



Debra Ansell
Embedded Systems Final Project
February 19, 2021

Table of Contents

Application: Spectral Speech Rate Analyzer

Motivation

I'm frequently told that I speak quickly, making my words difficult to understand. I can adjust my speaking speed when I notice the problem but find it hard to remain aware of my speaking rate for extended time periods. I would find it helpful to have a device capable of notifying me when my speech rate (measured in words per minute) exceeds a set threshold.

Proposed Project Solution

Ideal Form

In its ideal form, the Spectral Speech Rate Analyzer (SSRA) is a small, battery powered, unobtrusive wearable device. It can remain powered for a period of hours, remaining idle until it detects the user's speaking voice. It then begins recording regular sound samples, and, in real time, analyzes the samples to compute an approximate word-per-minute (WPM) count. If the WPM count exceeds a user-specified threshold, the device will provide subtle feedback (e.g. haptic motor, BLE app notification, indicator light).

Short-Term Development Challenges

Knowledge/Time

Optimizing both the software and hardware subsystems of the SSRA will require significant research and experimentation. Additionally, developing an efficient algorithm to provide a real-time WPM count will require research in topics outside the scope of this class, including Digital Signal Processing (DSP) and possibly Machine Learning.

Design

Software design challenges arise from the need to distinguish a single speaker's voice amidst background noise which might include multiple speakers. In the hardware, the small size and portable power requirements of wearable electronic devices adds a layer of complexity to the physical assembly.

Experience/Skill

My previous C language coding experience has been primarily in Arduino. Being new to embedded development, I've struggled to learn some of its tools (the STM32Cube IDE in particular.) This class and the final project provide terrific motivation to learn new systems which bring a lot of additional flexibility and power to my current toolset. I've also gained a newfound appreciation for the "it just works" simplicity of my previous development environment! While these challenges are inherent when entering any new technical domain, they also make for slower than usual progress in developing new projects.

Minimally Viable Product

Given the development challenges, the SSRA prototype presented here (V1.0) is a minimally viable version of its Platonic ideal, altered for practicality and simplicity (and a bit for fun). Design choices have been influenced by the following rationales:

- **Plugged-in (large-ish) prototype**: If social interactions occur via webcam, the SSRA need not be wearable, portable or subtle. It can be plugged into a wall jack for power and hidden just off-camera.
- **Limited signal processing**: If the user is wearing headphones for online interaction, background noise discrimination is unnecessary, which is good because there hasn't been enough time to explore various speech rate detection algorithms yet.
- **Keep hardware simple/easy**: Choices of peripherals were based on quick acquisition/development and most project sensors I used were already built into the STM32 Development Kit. The addressable LED matrix was an item I had on hand that I have some experience with, making development easier.

SSRA V1.0 Current Functionality

Even in its preliminary form, the SSRA V1.0 contains significant functionality, described below. Greater detail about the implementation of these functions are discussed in the Software section of this report.

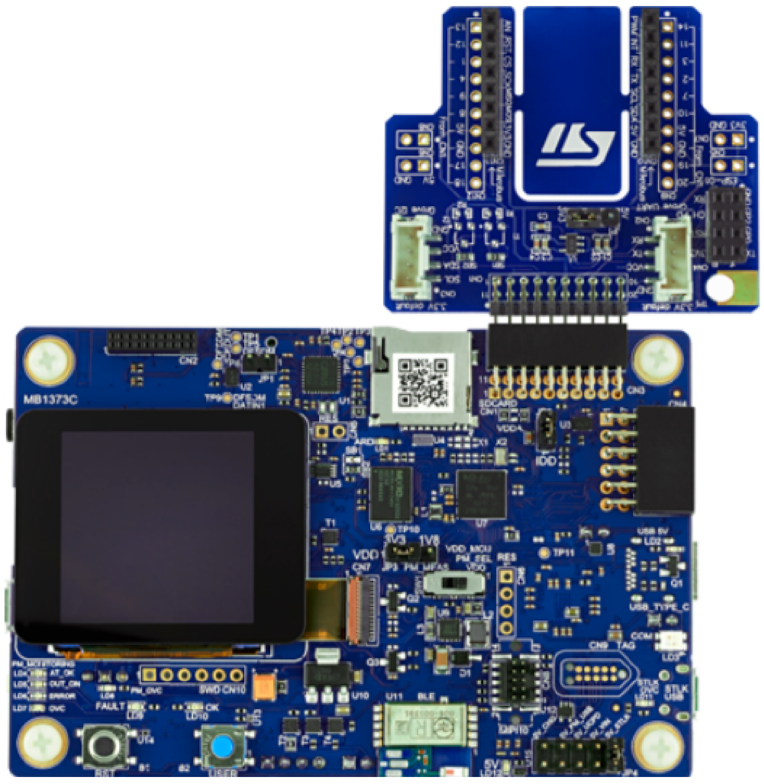
- Records regularly spaced audio samples over a specified time interval
- Performs a Fast Fourier Transform (FFT) of the audio data to generate a frequency spectrum
- Displays a real-time graph of the frequency spectrum (refactored into 32 frequency bins) on an 8x32 WS2812B LED matrix grid.
- Outputs real-time computation/data analytics (e.g. frequency bin size, # time samples, FFT processing time) via UART to a computer terminal emulator.
- Responds to a push button by entering into a responsive mode in which a user can enter single-character commands into the terminal emulator which modify the data processing that happens in the program and also the way the data appears on the LED matrix.

Hardware:

Components

STM32L562E-DK Discovery Kit

The SSRA uses the processor and some of the sensor/input peripherals in the STM32L256E-DK Discovery Kit shown below. (source: Discovery kit with STM32L562QE MCU - User manual)
Board peripherals utilized in the SSRA are listed below:



- STM32L562QEI6Q microcontroller (Cortex-M33 based)
    - 110 MHz Frequency
    - Floating Point Processing Unit
    - 512 Kb Flash
    - 256 Kb SRAM
    - 2 DMA Controllers with 14 DMA Channels
    - Multiple timers
    - Multiple GPIO
- User push button (Pin PC13)
- MEMS Microphone
- ST-Link 3 Debugger
- Two Indicator LEDs

Additional Electronics Hardware

These hardware elements are external to the Discovery Kit.

- USB to TTL cable to provide UART serial connection to my PC
- 8x32 WS2812B Flexible LED Matrix

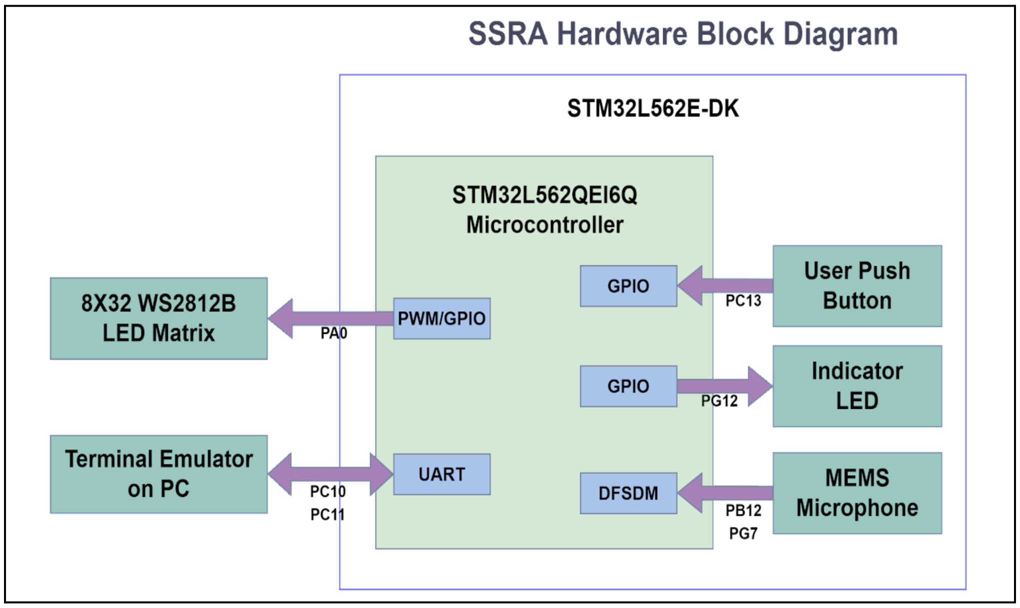
Display Case Parts

A custom stand/diffuser holds the LED matrix to the STM32L562E-DK and allows both to stand freely, while covering the LEDs with acrylic that tempers LED brightness.

- Custom Laser-cut wood baffle and supports for the LED matrix. Design files here.
- Custom Laser cut diffusion panel of LED Acrylic + screws/nuts to hold it to the LED matrix and Discovery Board

Peripheral Connections

Physical electronics assembly was very easy as the push button, indicator LEDs, and microphone are already electronically connected to the microcontroller on the discovery board. Electronic connections between the board and peripherals used in this project are shown below*.



*note there are actually TWO indicator LEDs in use (GREEN on PG12 and RED on PD3)

Software

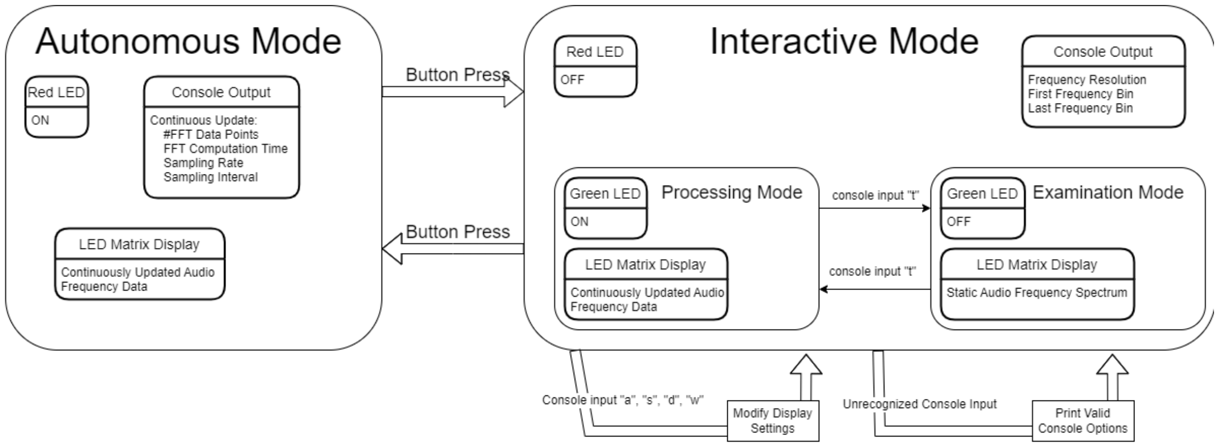
Overview

Functionality

The SSRA has two primary functions (modes):

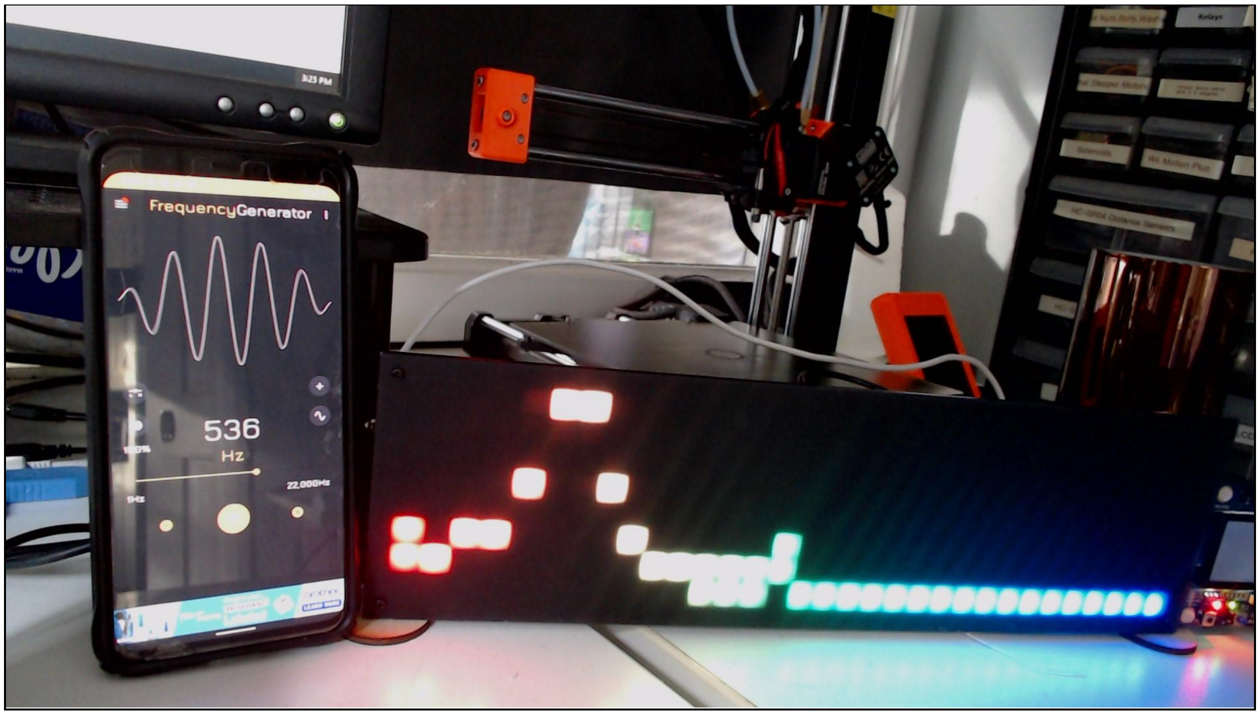
1. *Autonomous* real-time acquisition and display of the frequency spectrum of audio data
2. *Interactive* setting of display and data analysis parameters to allow better inspection and understanding of the data

The diagram below shows the relationship between autonomous and interactive mode and the status of the display peripherals in those modes.

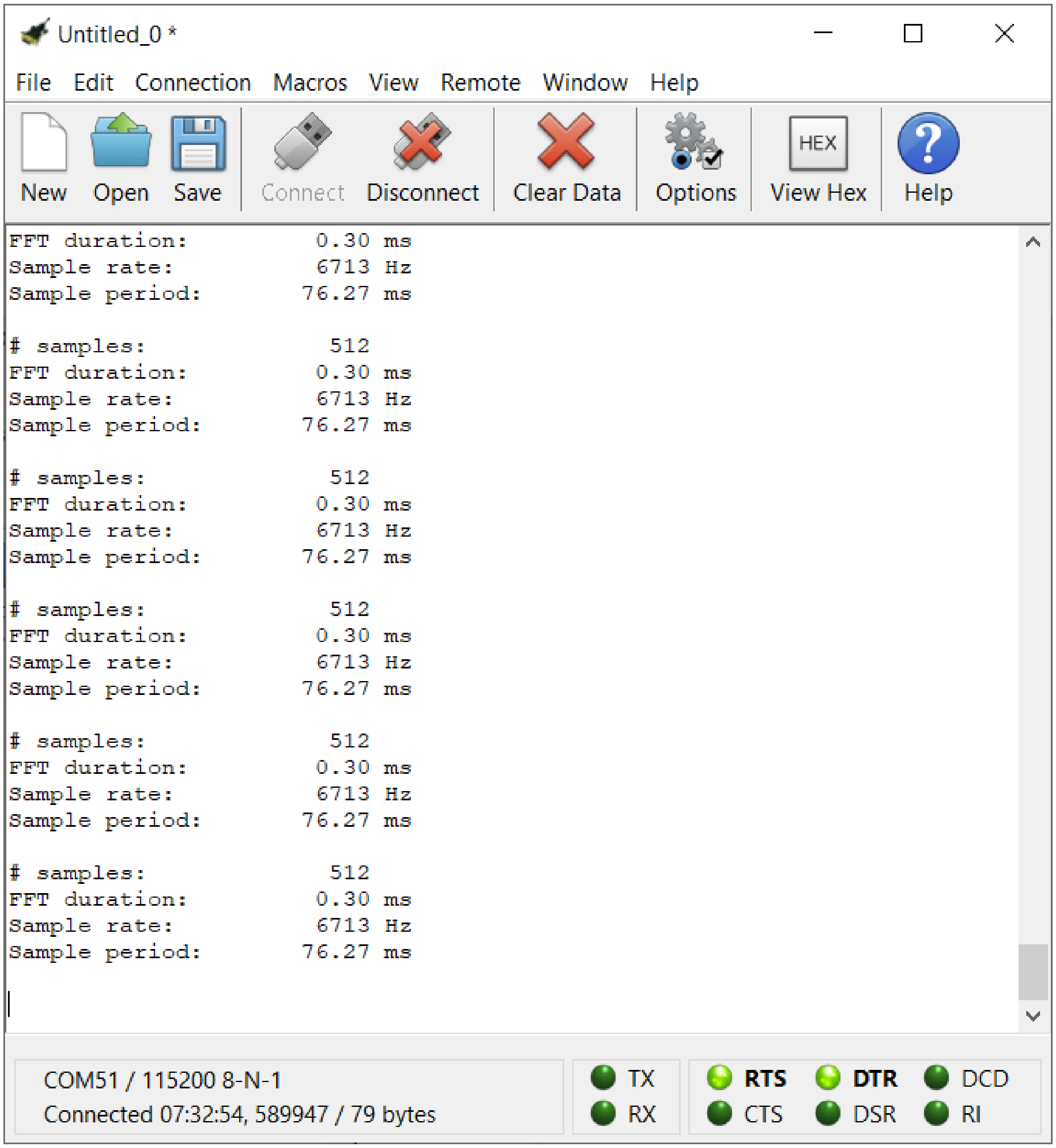


Autonomous Data Display

The program starts in autonomous mode with a real time spectral display of the detected frequency spectrum over the default range on the LED matrix. The picture below shows a pure audio tone of 536 Hz displayed as a single peak in the frequency display.



The SSRA also repeatedly sends simple statistics resulting from the ongoing data analysis to the serial port via UART, as shown in the image below:



```
FFT duration:        0.30 ms
Sample rate:         6713 Hz
Sample period:      76.27 ms

# samples:           512
FFT duration:        0.30 ms
Sample rate:         6713 Hz
Sample period:      76.27 ms

# samples:           512
FFT duration:        0.30 ms
Sample rate:         6713 Hz
Sample period:      76.27 ms

# samples:           512
FFT duration:        0.30 ms
Sample rate:         6713 Hz
Sample period:      76.27 ms

# samples:           512
FFT duration:        0.30 ms
Sample rate:         6713 Hz
Sample period:      76.27 ms

# samples:           512
FFT duration:        0.30 ms
Sample rate:         6713 Hz
Sample period:      76.27 ms
```

COM51 / 115200 8-N-1
Connected 07:32:54, 589947 / 79 bytes

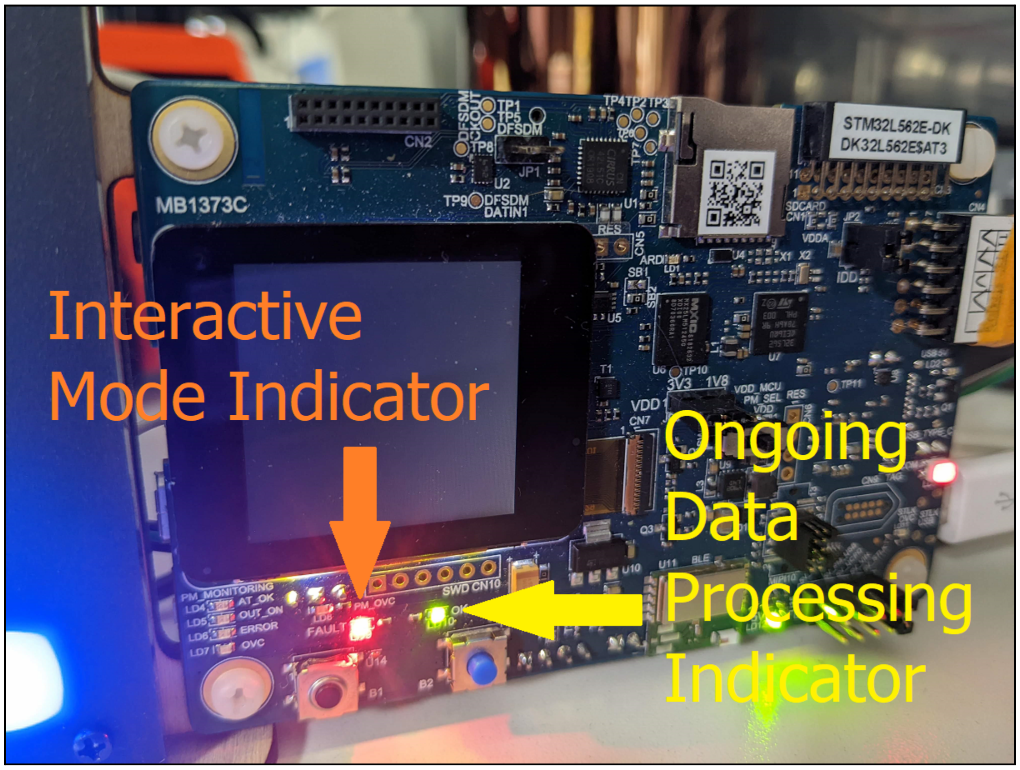
Currently, the display only shows:

1. the number of audio samples used as FFT input
2. the time taken by the processor to perform the FFT
3. the sampling rate
4. the length of time over which the samples were taken

As the DFSDM peripheral doing the recording can perform integration over a specified (IOSR parameter) number of samples (see description of DFSDM settings), the sample period can change without altering the other parameters. Eventually, I plan to add more interesting statistics (high frequency bin, DB level readings) to this display.

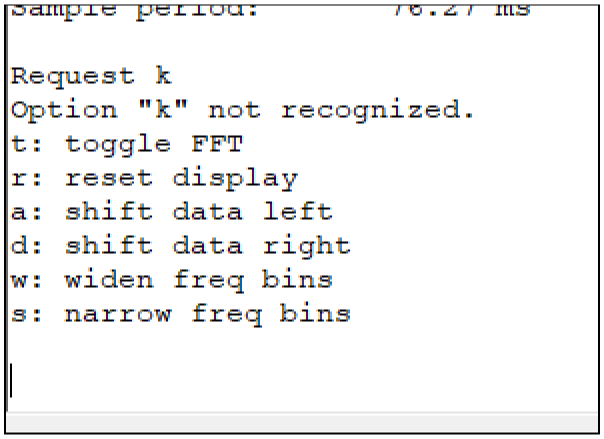
Interactive Parameter Setting

When the user presses the push button on the discovery board, an interrupt is triggered which puts the display into interactive mode. The discovery board's red indicator LED turns on to indicate that the board is in interactive mode.



In this mode, the LED display continues to respond with real-time visualization of frequency data, but the periodic output analytic data to the serial console stops, allowing the user to enter commands and view the responses in the serial terminal. This console command feature was implemented as simply (and quickly) as possible, utilizing single character commands and will likely be upgraded in future versions.

If an unrecognized command is entered in interactive mode, the user is prompted with the following menu:

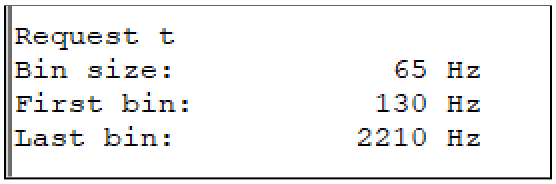


```
Sample period:        76.27 ms

Request k
Option "k" not recognized.
t: toggle FFT
r: reset display
a: shift data left
d: shift data right
w: widen freq bins
s: narrow freq bins
```

The purpose of this mode is to allow the user to change both display and data acquisition/analysis settings, though SSRA V1.0 has only a few options. These will definitely be expanded on in future revisions.

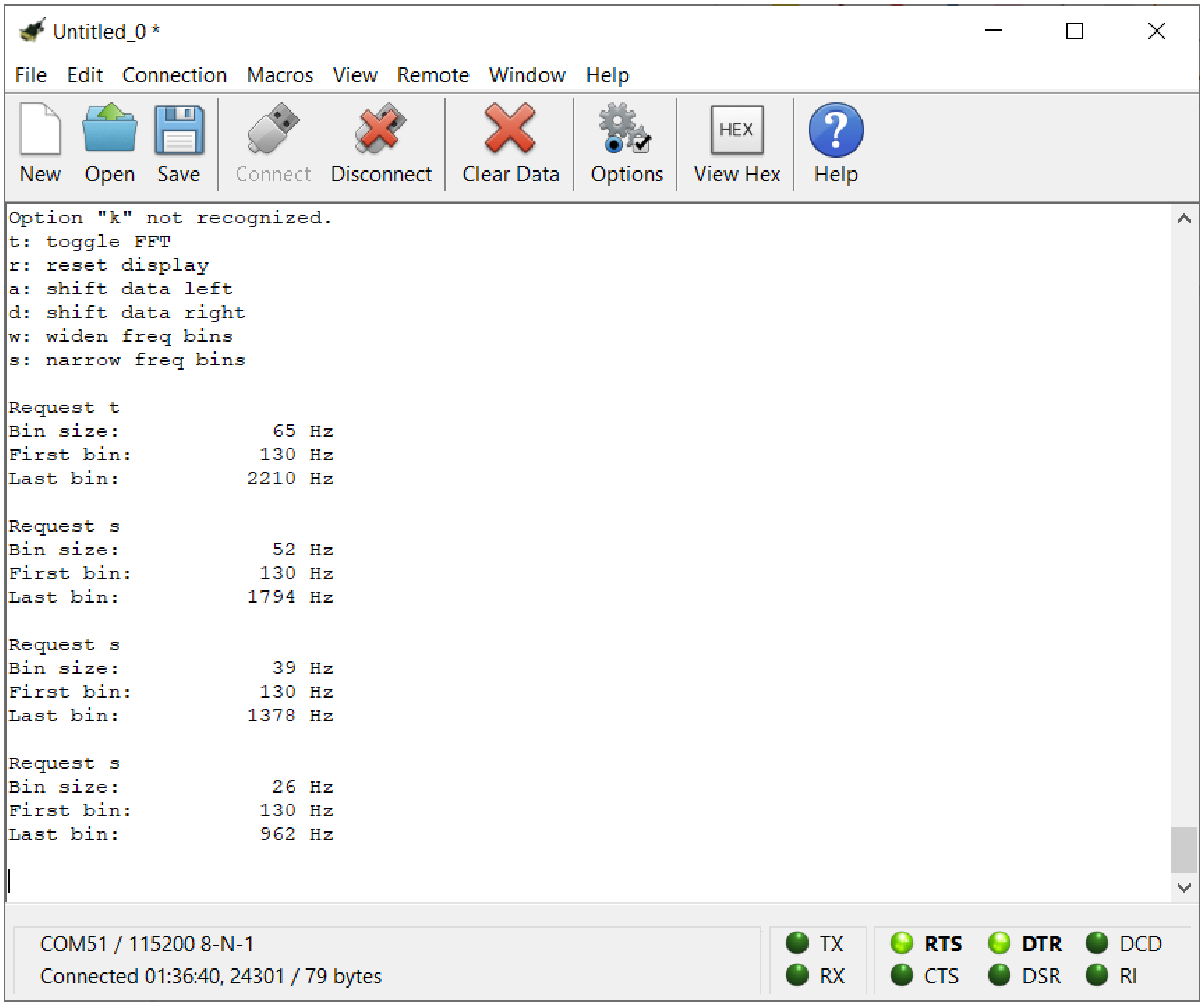
When a console command is entered, the resulting data and display parameters are printed to the console, as shown below. The currently accepted command set only changes the frequency range and resolution of the LEDs, shown below.



```
Request t
Bin size:            65 Hz
First bin:          130 Hz
Last bin:          2210 Hz
```

Option “t” stops the processing of new audio data. In this mode the LED matrix will continue to display only the frequency spectrum that was acquired at the moment the “t” option was entered. Entering this mode turns off the board’s green indicator light to show that audio data is not currently being processed.

Options “a”, “d”, “w” and “s” allow the user to change the resolution and displayed portion of the frequency spectrum on the LED matrix. SInce the FFT contains many more data points than the 32 columns of the LED display, there is enough data resolution to zoom in on portions of the data. Pressing “s” several times will provide a closer look at the spectrum detail. The image below shows the results of pressing the “s” button several times while viewing the same 536 Hz peak shown previously. The data on the LED display runs from 130 Hz to 962 Hz, and the 536 Hz peak shows up in the middle of the display, as expected.


Untitled_0 *
File Edit Connection Macros View Remote Window Help
New Open Save Connect Disconnect Clear Data Options View Hex Help
COM51 / 115200 8-N-1
Connected 01:36:40, 24301 / 79 bytes
TX RX RTS CTS DTR DSR DCD RI

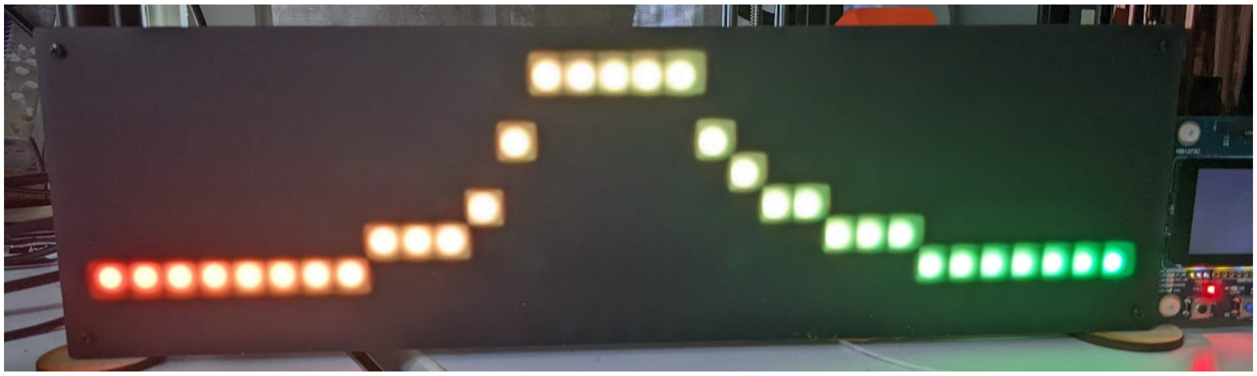
```
Option "k" not recognized.
t: toggle FFT
r: reset display
a: shift data left
d: shift data right
w: widen freq bins
s: narrow freq bins

Request t
Bin size:            65 Hz
First bin:          130 Hz
Last bin:          2210 Hz

Request s
Bin size:            52 Hz
First bin:          130 Hz
Last bin:          1794 Hz

Request s
Bin size:            39 Hz
First bin:          130 Hz
Last bin:          1378 Hz

Request s
Bin size:            26 Hz
First bin:          130 Hz
Last bin:           962 Hz
```



The zoomed-in spectrum can be moved right/left with the “a”/”d” keyboard commands, and the “s” command returns all display values to their default settings. Note that the colors of the LEDs in the display are associated with a specific frequency, providing a visual cue to the zoom level and positioning of the frequency spectrum shown on the LED display. If we are zoomed in, the pixels on the display will show only those hues associated with the graphed frequency range.

Any changes made to display/data acquisition parameters in Interactive mode will persist when Autonomous mode is resumed by pushing the user push button. Parameters may only be changed in Interactive mode.

Development Environment/Code Repository

All original code was written and compiled in the STM32Cube IDE (aka “The Devil’s IDE”), and the entire code repository is on GitHub at:
https://github.com/geekmomprojects/SpectralSpeechRateAnalyzer

Structure

Optimal Design Pattern: Model View Controller

This application would benefit greatly from following the model-view-controller design pattern as the data, view and console command interfaces operate independently and don’t share much hardware. In particular, it would be nice to be able to easily swap in and out different displays so the frequency spectrum can be viewed at greater than 8x32 pixel resolution.

In practice, the code was developed with twin philosophies of “just make it work” and “STM32Cube is a special kind of hell”, and I’ve only recently started refactoring the code to make it fit the modular MVC model. It’s kind of halfway there. I’ve created three separate data structures to hold parameters for the display, the data and the console, but their functionalities are still fairly intertwined in the code. Refactoring the code well is a high priority item on the improvement list.

Program Files

All of my written code (i.e. that which was not auto-generated by CUBE MX) is in “main.c”, including overridden callback functions for several of the peripherals. I’m afraid it isn’t particularly sensibly organized yet. “Borrowed” code libraries (CMSIS DSP and the SK6812 LED libraries) are in separate files, as explained in the next section.

“Borrowed” Code

SK6812 Library

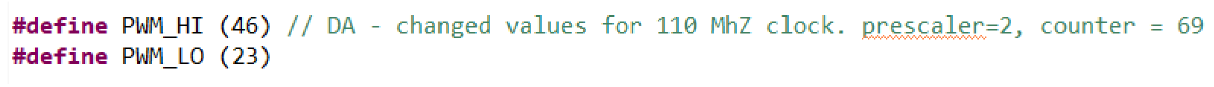
The largest piece of copied code is the SK6812 library (“sk6812.c”, “sk6812.h”) described in very helpful detail here: https://www.thevfdcollective.com/blog/stm32-and-sk6812-rgbw-led by “Frank from the VFD Collective.” This post also contains instructions to configure the required Timer, PWM and DMA settings correctly in STM32Cube IDE.

This library handles sending data to addressable RGB LEDs which use the WS2812 protocol. Since WS2812B LEDs have only a single data line connecting all the LEDs in series (as opposed to a two-wire Data/Clock protocol), they require precisely timed signals at a rate of about 800 kHz to be able to decode the pulses containing color information.

Each color information “bit” in the data stream starts with the voltage high, and the duration of the high voltage during the bit transmission interval (Tbit = 1/frequency = 1/800kHz = 1.25 uS) determines whether it is a “1” bit or a “0” bit. A “1” bit has V high for the first ⅔ of Tbit, while a “0” bit has V high for the first ⅓ of Tbit. Each LED in the chain requires 3 (8-bit) bytes to specify its RGB values.

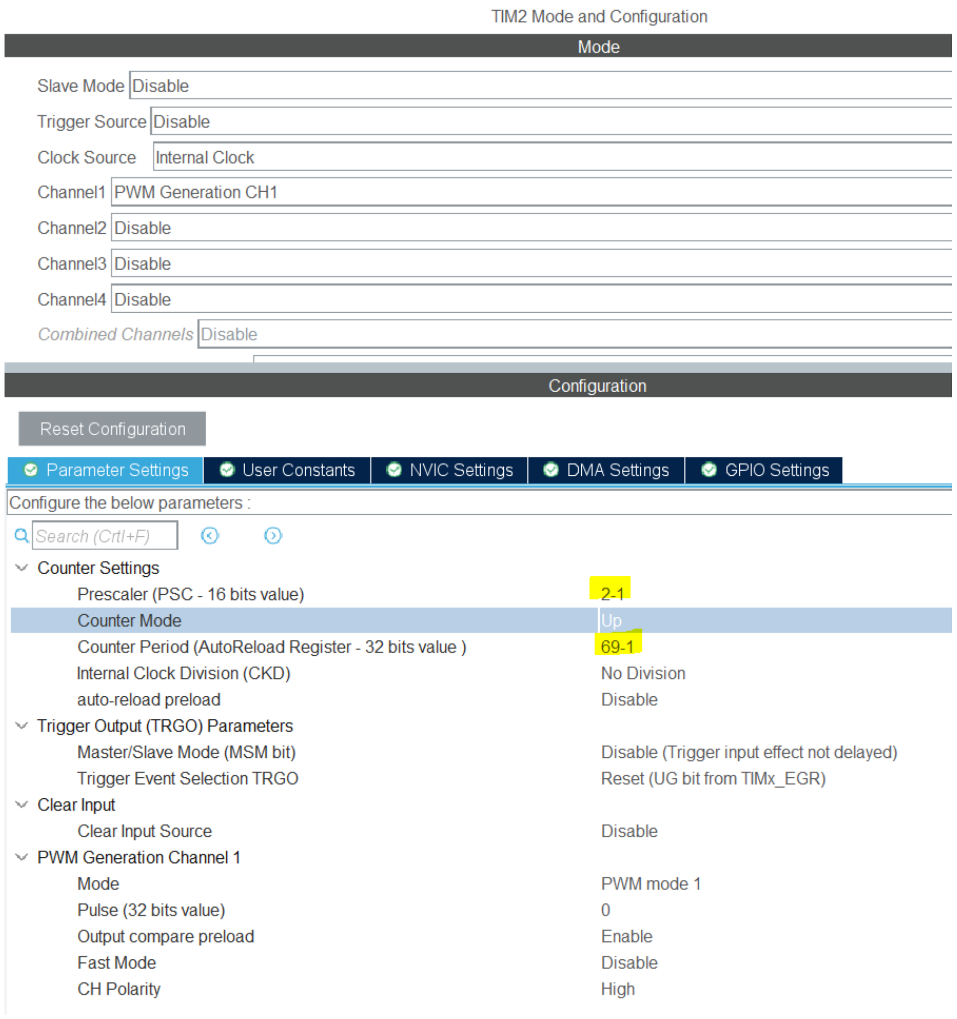
The SK6812 library uses a timer configured with PWM to transfer the contents of a data buffer containing RGB color information using DMA. The library code attempts to minimize the amount of memory reserved for DMA by using a circular buffer just large enough to hold the data for two RGB LEDs. When the first LED’s data is half-sent (triggering an interrupt), the code fills the other half of the buffer with the data for the next LED in the sequence, and continues to do this until data for all LEDs in the sequence have been sent. The library also takes care of sending a stream of zeroes to indicate the end of the data transmission for all the LEDs.

The library was developed for a different STM32 board, but required only minimal changes to work on the STM32L562E-DK. I used the same timer (TIM2) as the library’s creator to generate the PWM pulses, but the library was developed for a board with an 80 MHz clock, while my board’s default clock frequency is 110MHz. Therefore, in order to generate a PWM frequency of 80 KHz, I needed different Prescaler/Counter values. I chose a prescaler of 2 and a counter period of 69 for a net PWM frequency of 110 MHz/(2*69) = 797 KHz. As seen in the code snippet below, the Voltage transition timing from high to low occurs when the counter is at 46 (= 69 x ⅔) for a “1” data bit and when it is at 23 (= 69 x ⅓) for a “0” data bit.



```
#define PWM_HI (46) // DA - changed values for 110 MhZ clock. prescaler=2, counter = 69
#define PWM_LO (23)
```

Initially I had a timing error in using the library because I didn’t realize that a prescaler value of 2 had to be entered as a “1” in the Counter settings. Once sorted out, the library worked easily to control the LEDs.


TIM2 Mode and Configuration

Mode

Slave Mode	Disable
Trigger Source	Disable
Clock Source	Internal Clock
Channel1	PWM Generation CH1
Channel2	Disable
Channel3	Disable
Channel4	Disable
Combined Channels	Disable

Configuration

Configure the below parameters :

Counter Settings	
Prescaler (PSC - 16 bits value)	2-1
Counter Mode	Up
Counter Period (AutoReload Register - 32 bits value)	69-1
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable
Trigger Output (TRGO) Parameters	
Master/Slave Mode (MSM bit)	Disable (Trigger input effect not delayed)
Trigger Event Selection TRGO	Reset (UG bit from TIMx_EGR)
Clear Input	
Clear Input Source	Disable
PWM Generation Channel 1	
Mode	PWM mode 1
Pulse (32 bits value)	0
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High

CMSIS DSP/Complex Math

My code uses the CMSIS DSP library to perform a real FFT on the time-based audio data to transform it into the frequency spectrum. Since my development board has a FPU and is frankly, just really zippy, the FFT uses the floating point version of the real FFT algorithm (`arm_rfft_fast_f32`), rather than attempting to speed things up by using integer math. It also uses the DSP function `arm_cmplx_mag_f32` to compute the magnitude of the complex frequency vector values returned by the FFT.

Color Wheel Algorithm

The other piece of borrowed code is a simple cyclic RGB color generation algorithm from Adafruit (https://learn.adafruit.com/multi-tasking-the-arduino-part-3/utility-functions) used to easily generate a somewhat continuous RGB rainbow spectrum. Different colors on the spectrum are used to indicate different frequency values in the LED grid display.

Selective Elaboration on Application of Course Topics

Interrupts

Interrupts were used by several of the peripherals:

- DFSDM circular buffer generates interrupts when the buffer is half full and when it is completely full. These interrupts set code flags (`DmaRecHalfBuffCplt`, `DmaRecBuffCplt`) that let the main loop know when it is ok to copy the data out and process it.
- The push button generates an interrupt which sets a debouncing flag and starts a general purpose timer (TIM3) with its OWN interrupt
- The TIM3 Interrupt stops the timer, unsets the debouncing flag, and checks to see if the push button is still depressed. If so, it toggles the SSRA between autonomous and interactive modes
- UART RX receives a callback when the SSRA is in interactive mode and a character is sent via UART. It saves the character to a global variable and sets a flag so that the main loop can check to see if the character is a known command.
- UART TX has callbacks indicating completion of sending data and availability to send more. These call backs are a holdover from when the SSRA had no physical display element and transmitted all of its data via UART. Much less outgoing data travels over UART now, and these callbacks may not stay.

Timers

A number of different timers are used in this project:

- TIM2 is used in PWM mode to send data at regular intervals to the WS2812 LEDs (discussed in more detail in the SK6812 Library section of this report
- As noted under “Interrupts” above, TIM3 is used for debouncing the push button.
- TIM16 is a general purpose timer used to time sections of the code. Currently, it reports how long it takes to run the FFT routine. In writing up this report, I found that while I thought I had set a prescaler value of 110 (tick interval 1uS), I had actually forgotten to set the prescaler at all. So it turns out that my FFT routine actually runs a factor of **110** times faster than I had thought previously.
- The DFSDM peripheral, discussed in its own section below, has a clock output used to trigger recording of audio samples. I selected the 110 MHz system clock with a divider of 16 for the timer, however, the actual audio sampling rate depends on additional parameters.

DMA

DMA is used in three places in this project, and makes it much easier to perform continuous data acquisition and processing when data read/writes are non-blocking

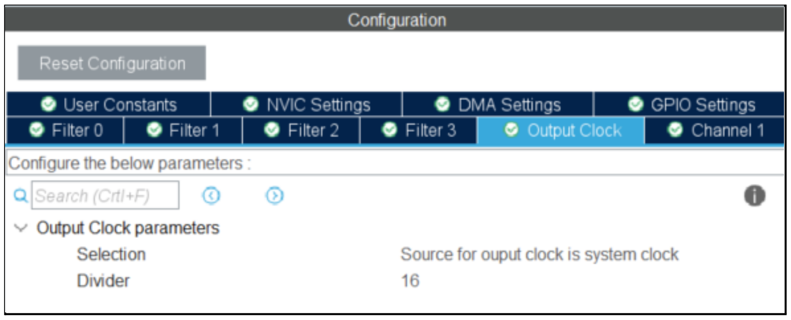
- The DFSDM peripheral uses DMA to write the audio data to a circular buffer

- Serial data sent through UART is sent via DMA in normal (vs. circular) mode
- The SK6812 Library uses DMA with PWM as discussed previously to send data for 1 LED at a time through the GPIO pin to the LED matrix.

DFSDM

The DFSDM is an interesting peripheral. According to ST, it is a generic ADC with additional interfaces and internal digital filters that is well suited for audio recording applications using MEMS microphones. I found information about using it with the STM32L562 processor in this presentation. DFSDM stands for “Digital Filter for Sigma-Delta Modulator. I don’t fully understand all of its uses, but it can apply various filters to the data, and offloading some processing from the CPU.

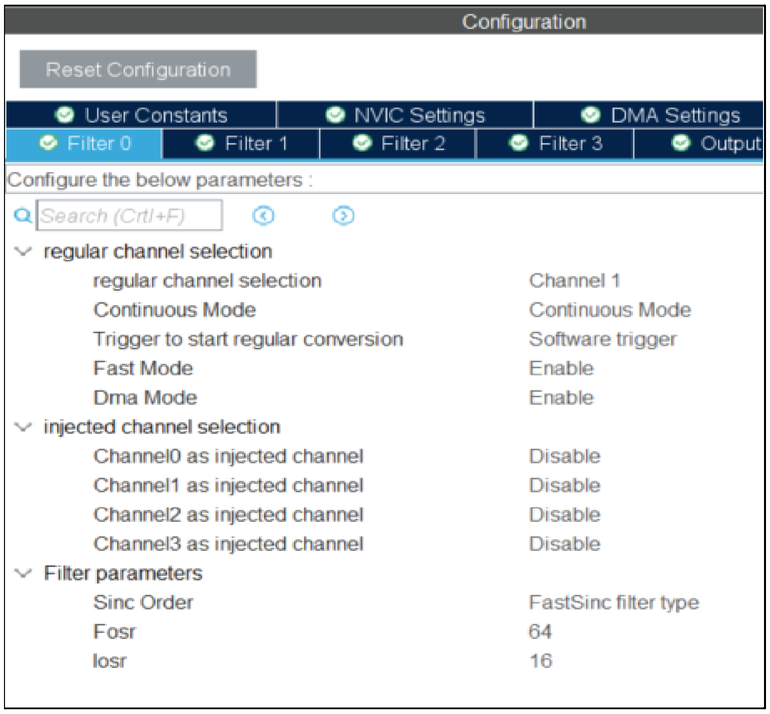
When using the DFSDM to record audio sampling, you must specify not only the input clock (System or Audio) and divider, but also an oversampling ratio (FOSR) and Integration Value (IOSR), as shown in the images below.



Configure the below parameters :

Output Clock parameters

Selection	Source for ouput clock is system clock
Divider	16



Configure the below parameters :

regular channel selection

regular channel selection	Channel 1
Continuous Mode	Continuous Mode
Trigger to start regular conversion	Software trigger
Fast Mode	Enable
Dma Mode	Enable

injected channel selection

Channel0 as injected channel	Disable
Channel1 as injected channel	Disable
Channel2 as injected channel	Disable
Channel3 as injected channel	Disable

Filter parameters

Sinc Order	FastSinc filter type
Fosr	64
Iosr	16

The integration just adds up multiple readings to average out random sensor value fluctuations. The net data measurement rate is given by Input data rate/(FOSR*IOSR). My code for the SSRA uses a routine called `getSamplingRate` to compute the rate of data measurement using these values.

Conclusion/Future Improvements

I've learned a great deal from the Embedded Systems course, and creating this final project has been an excellent way to make abstract lessons more concrete. I definitely appreciate the power and speed accessible with the programming tools and techniques I'm learning, though it will likely take me a long time to develop the patience to use them properly!

I would like to continue to develop the SSRA. As the ideal version of the SSRA is a small wearable real-time speech-rate measuring device, and v1.0 is fairly far from that, some of the future improvements are obvious.

I will need to take a deeper dive into signal processing techniques and speech detecting algorithms. To make the system into a battery-powered wearable, I will have to take a closer look at the system's power consumption (and probably ditch the 8x32 WS2812 LED matrix) . In the meantime, I am enjoying having my spectrum visualizer sitting next to me on my desk responding to the noises in my office.

Self-Grading

Criteria	**Self Grade (1-3)**	**Notes/Comments**
Meets Minimum Goals	2.5	(Docked myself a bit because the state machine is basic and the command line interface isn't comprehensive, though I'll rationalize that the project overcompensates in other areas to keep it from being a 2)
Completeness of Deliverables	2.5	Code could be better at being self-documenting, but otherwise I think I've documented the project reasonably well in this report and in the comments.
Clear Intentions/Working Code	3	Code works as intended, and while not yet structured perfectly, is simple enough to read and understand easily.
Reusing Code	2	No Licensing/Versioning included, but all external code has been attributed

Originality/Scope	3	Novel ✔ Awesome ✔ Beyond Requirements ✔
Bonus Points	None	Too busy making LEDs blink and fighting with STM32Cube IDE. Version control was implemented too late in the game to meet requirements for extra points.