*Blind Source Separation Using ICA*

Nayem Alam   
ECSE444 - Microprocessors  
McGill UniversityMontreal, Canada  
260743549

Tristan Bouchard  
ECSE444 - Microprocessors  
McGill UniversityMontreal, Canada  
260747124Alex Masciotra  
ECSE444 - Microprocessors  
McGill UniversityMontreal, Canada  
ID?

Thomas Philippon  
ECSE444 - Microprocessors  
McGill UniversityMontreal, Canada  
ID?

*Abstract*— This lab allowed us to deepen our knowledge of microprocessors by taking everything we learned from previous labs to implementing a source separation using the Fast Independent Component Analysis (ICA) technique. This experiment contains two parts: first, generating two sine waves, mixing the signals, store and read from flash and outputting mixed signals onto the oscilloscope; second, using FastICA to separate the signals, and outputting the separate signals onto the oscilloscope. The tools used were a microprocessor, namely the STM32 IOT Discovery kit, a breadboard, an oscilloscope and programmed in embedded-C using the Keil μVision5 IDE. Additionally, we used the CMSIS-DSP library functions (arm\_sin\_f32()) to generate samples of the sine waves, the HAL drivers to control the on-chip Digital to Analog Converter (DAC), and the Quad-SPI interface for storing samples onto the flash memory.

Keywords—microprocessor, STM32, FastICA, CMSIS-DSP, Quad-SPI, DAC

# Introduction

The purpose of this experiment was given to exercise the skills acquired from previous labs to finally build an audio application that employs Blind Source Separation (BSS) using the Fast Independent Component Analysis (FastICA) algorithm. There are many software and hardware limitations such as STM32 flash can only take 8 megabytes of information at a time or writing an algorithm to the DAC chip might be overwhelming. However, these limitations allow students to turn to strategies such as using the DMA, OS threads, interrupts and CMSIS-DSP library. Students were given the opportunity to explore the several effective functions they can use to minimize runtime, lines of code, to building a creative audio application. In order to initiate the project, a base code was provided for the Keil μVision5 IDE to guide the students on where to implement appropriate codes. However, the base code wasn’t enough to incorporate the QSPI (flash interface), as a result, the software CubeMX was used to modify the base code in order to use QSPI. Moreover, another tool that was provided was the STM32 kit with an audio jack converter. An oscilloscope was also used to display the sine waves. Pin numbers D13 and D7 were used to probe the channels (Channels 1 and 2 respectively) on the board for the sine wave. Students were given 2 weeks to deliver the initial demo and 2 weeks to deliver the final demo.

# The Design Problems

This section will work in tandem with Section III. As problems are introduced and explained, Section III will answer the problems accordingly.

## The Base Code

The first and foremost step to start this experiment is to ensure that the base code provided is fully functional with the functions that we’d need to implement. The issue that arose with this is that the base code provided didn’t include how to incorporate QSPI (flash interface).

## Generation of Sine Waves

In the previous labs we were required to generate triangular waves, but not sine waves. As a result, we had to read through the CMSIS-DSP library to see if there was a function that can generate sine waves. The CMSIS\_DSP library is a “suite of common signal processing for use on Cortex-M processor-based device” [5], which provides different functions to generate signals and perform complex mathematics.

To generate one sine wave signal deemed to be straightforward however another challenge was to generate two sine waves and we had to figure out at what frequency do two sine waves mix appropriately.

## Writing Sine Wave Samples to Flash

The next issue was how to access the Quad-SPI external interface of the board. The Quad-SPI chip is the external flash memory available on the STM32 microcontroller. As the sine waves generated from the “arm\_sin\_f32()” function takes up 32kB for a 2 second signal using 8-bit signal resolution, it was necessary to store the generated waves somewhere else than the processor SRAM due to their large size. The waves can then be read from memory to be mixed and written to the DAC, for example.

## Writing Sine Wave Sample to DAC and Audio Playback

This lab, like a few of the previous labs, required us to use the Digital to Analog Converter, called the DAC for short. This piece of hardware located on the board allows the user to create analog signals from digital inputs, such that these analog signals can be output to the digital pins of the microcontroller and to the external world.

We were given the option to output the samples through MATLAB or an oscilloscope to verify our sine waves. The problem faced with this was that we had to scale the signal to have a suitable amplitude for the DAC resolution (i.e. 8 bits) and also since sine waves have negative samples, we also needed to provide a DC offset the signal. Writing the sine wave to the DAC was very minor as we dealt with something like this in Lab 2.

# Approach to Problems

## The Base Code

After numerous run trials we concluded that the base project given was not sufficient enough to incorporate reading and writing to the flash memory. As a result, we used the program CubeMX which is an initialization code generator that helps to map out certain areas on the board such as pin location and will generate code based on the ones we’d like to map. What pins did we map out?

## Generation of Sine Waves

Generating sine wave signals were done using the formula (1) given in the lab handout. Initially, to test the implementation, only one sine wave was to be generated. After computing the value of the angle using the arithmetic shown within the arguments of the function, the sine wave signal was found using a simple CMSIS-DSP function [1]. This function is a fast approximation of the trigonometric sine function, with one argument, the angle in radians and returns sin(angle). Moreover, to generate two sine waves, we first needed to know at what frequency do two sine waves mix. To solve this, we used the frequency values C4 and G4 which was found using Table I, the corresponding values were 261 and 392 respectively.

In order to produce mixed sine waves, we converted the provided matrix in the documentation to 2 linear equations (2) and selected coefficients to be of value 0.5. This number was chosen to center the samples.





1. Frequencies for equal-tempered scale, A4 = 440 Hza

| Note | Frequency (Hz) | Wavelength (cm) |
| --- | --- | --- |
| C4 | 261.63 | 131.87 |
| G4 | 392.00 | 88.01 |

1. Extracted sample from Physics of Music – Notes (*https://pages.mtu.edu/~suits/notefreqs.html*)

## Writing Sine Wave Sample to DAC

We decided to display the sine waves onto the oscilloscope using the DAC output. This was a fairly simple solution as we implemented the same approach as Lab 2. We initiated channels 1 and 2 and used the HAL\_DAC\_SetValue [2] function which contains 4 parameters, a DAC handletype which is a pointer a DAC\_HandleTypeDef structure that contains the configuration information for the specified DAC, the DAC channel that we want to write to, the data alignment, and finally the output signals.

Furthermore, the second sine wave signal was offset with a DC value of 32000 (2\*16000) because it was required to generate 32000 samples. Both of our sine waves were sampled at a rate of 16000 samples/second, so we can generate samples worth 2 seconds of a sine wave. This would mean that we would be placing first sine wave iteratively form 0 to 31999, and the next sine wave offset by 32000. MAYBE ADD SOMETHING ABOUT SYSTICK IN HERE?

## Writing Sine Wave to Flash and Playback from Audio

We were provided with a document [3] that helped us instantiate and understand the external flash of the board. While computing each sine wave, we also wrote it to the QSPI. This was done using the function BSP\_QSPI\_Write [4] which takes in 3 parameters; pointer to data (sine wave) to be written, write starting address and the size of the data to write. This function writes to the QSPI memory. Next, in order to read values from the QSPI, we used the function BSP\_QSPI\_Read [4] and this allowed us to output sample to the audio interface using the flash memory.

The following software architecture visually explains our approach to the initial part of this experiment:

[INSERT BLOCK DIAGRAM HERE]

# Results

Two sine wave signals (mixed) were displayed on the oscilloscope pleasantly. The signal had fluctuating amplitudes, and this occurred because of \_\_\_\_\_. Furthermore, by plugging a headphone into the stereo jack we were able to hear two types of notes. This is because for this part of the experiment, we are listening to the mixed signals (x1(t) and x2(t)) and this is in coherence with our laboratory documentation mentioning that this is normal to hear as it is a linear combination of the original sine waves.

# Conclusion

The following lab allowed us to dive into a microprocessor and understand its components by building an audio application. The initial part of the lab report allowed us to incorporate different functions and libraries (namely QSPI and CMSIS-DSP) to output mixed sine waves. The biggest challenge faced during the initial process was how to read and write samples to the external flash memory. Through reading the link the teaching assistant provided us, we were able to determine what class to use and how. We were able to read from the flash memory and then output the number of samples on to the oscilloscope. The oscilloscope portrayed an appropriate mixed signal. Since we used coefficients of 0.5, to center the signal, we were able to clearly see the signals. Finally, while the oscilloscope was running, we also managed to get playback of x1(t) and x2(t), where each of them contained two notes that are a linear combination of the original signals (s1(t) and s2(t)). The next part of the lab will be to implement a FastICA that will separate the sine wave signals to recover the original signals s1(t) and s2(t).

##### References

1. arm\_sin\_f32.c File Reference. (2018). Retrieved from https://www.keil.com/pack/doc/CMSIS/DSP/html/arm\_\_sin\_\_f32\_8c.html
2. <https://www.st.com/content/ccc/resource/technical/document/user_manual/63/a8/8f/e3/ca/a1/4c/84/DM00173145.pdf/files/DM00173145.pdf/jcr:content/translations/en.DM00173145.pdf> [REF NOT COMPLETED]
3. <https://www.st.com/content/ccc/resource/technical/document/application_note/group0/b0/7e/46/a8/5e/c1/48/01/DM00227538/files/DM00227538.pdf/jcr:content/translations/en.DM00227538.pdf> [REF NOT COMPLETED]
4. STM32746G-Discovery BSP User Manual: STM32746G\_DISCOVERY QSPI Exported Functions - STM32746G-Discovery BSP Drivers Documentation. (2018). Retrieved from <https://documentation.help/STM32746G/group__STM32746G__DISCOVERY__QSPI__Exported__Functions.html>
5. "CMSIS DSP Software Library", *Keil.com*, 2018. [Online]. Available:http://www.keil.com/pack/doc/CMSIS/DSP/html/index.html. [Accessed: 19- Nov- 2018]