*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  
260746829

Thomas Philippon  
ECSE444 - Microprocessors  
McGill UniversityMontreal, Canada  
260747645

*Abstract*— This lab allowed us to deepen our knowledge of microprocessors by 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, storing and reading 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 the STM IoT node with a STM32L4 MCU based on ARM cortex M4 core, an oscilloscope and the Keil μVision5 IDE for Embedded-C programming. Additionally, we used the CMSIS-DSP library functions 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 into 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 building 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 the 128 Kbytes of SRAM available on the MCU as well as the synchronization of the DAC channels when outputting samples. However, these limitations allow students to turn to strategies such as using DMA, OS threads, interrupts and the CMSIS-DSP library. Students were given the opportunity to explore the several effective functions they can use to optimize their implementation and succeed in building a creative audio application. In order to initiate the experiment, a base project was provided for the Keil μVision5 IDE to guide the students on where to implement appropriate code. Everything required to generate, mix and output the sine wave was initialized in the base project. However, students might have to use the CubeMX software for the final implementation if OS threads and DMA are to be used. Moreover, another tool that was provided was the STM32 kit with an audio jack converter. An oscilloscope was used to display the sine waves. Pin numbers D13 and D7 were used to probe the channels 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.

##### PART 1

# 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.

## Generation of Sine Waves

It was pertinent to read through the CMSIS-DSP library to find a function that can generate sine waves. The CMSIS-DSP library is a collection of common signal processing for use on Cortex-M processor-based devices [1], which provides different functions to generate signals and perform complex mathematics.

## Writing Samples to the Quad-SPI Flash Memory

The on-chip SRAM memory of the MCU is restricted in terms of space (128kb), as a result we had to access the Quad-SPI interface of the board. The Quad-SPI chip is the external flash memory available on the STM32 microcontroller which provides an extra 64MB of flash memory. The lab handout indicated that the sine waves generated take up 32kB for a 2 second signal using 8-bit signal resolution. Thus, it was necessary to figure out how 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.

## Writing Sine Wave to DAC and Playback of Audio

Moreover, this lab required us to use the Digital-to-Analog Converter (DAC). This piece of hardware located on the board allows the user to create analog signals (i.e. voltage) from digital inputs.

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). Also, since sine waves have negative samples, we needed to provide a DC offset to the signal. Writing the sine wave to the DAC was very minor as we dealt with something similar in Lab 2, avoiding us having to work with UART and MATLAB.

# Approach to Problems

## Generation of Sine Waves

Generating sine wave signals were done using the formula given in the lab handout, as in (1). Initially, to test the implementation, only one sine wave was 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, arm\_sin\_f32() [2]. This function is a fast approximation of the trigonometric sine function, with one argument, the angle in radians and returns sin(angle) sample with a value in range [-1,1]. Moreover, we were required to sample the sine wave at 16000 samples per second, a sampling rate that achieves high quality audio. Therefore, in order to have two seconds worth of a sine wave, 32000 samples were created. Next, to prepare the signal to be written to the DAC, the sine wave needed to be offset by 1 to make a range [0,2]. We also set the resolution of the DAC to 8-bits, so the values that were being sent to the DAC were scaled to a baseline of 128 ranging from 0 to 255. A resolution of 8-bits was chosen to facilitate the implementation with the Quad-SPI memory unit, which is byte addressable and will explained in the following section.

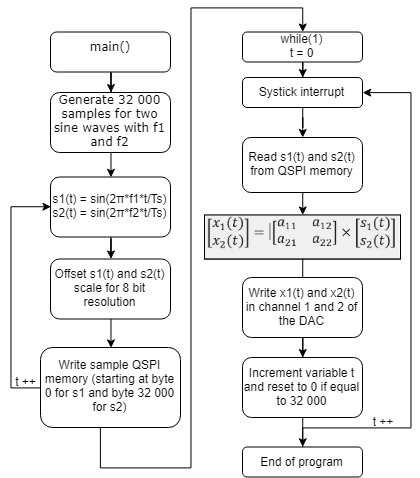
Moreover, to generate two sine waves that are pleasant to listen to when mixed, the corresponding frequencies 261Hz and 392Hz were selected [3]. The generation of the samples are shown in the left-hand side of Fig. 1, where the feedback connection between the bottom and the 3rd block represents the loop which generates all the samples.



## Writing Samples to the Quad-SPI Flash Memory

We were provided with a document [4] that helped us instantiate and understand the external flash of the board. While computing each sine wave, we also wrote it to the QSPI. First, the flash memory was initialized using the BSP\_QSPI\_Init, and then the flash chip was cleared using BSP\_QSPI\_Erase\_Chip. The function BSP\_QSP\_Write [5] was used to write to the flash, which takes in 3 parameters; a pointer to data to be written, the starting address and the number of elements to be written (in bytes). In our case, since the samples are written one at a time, the number of elements is set to 1 and the pointer to data is simply the address of the variable which holds the sample to be written. As specified in Fig. 1, the samples of the first sine wave are stored from address 0 to 31999. For the second sine wave, the samples were stored at address 32000 to 63999. Note that the addresses are in units of bytes.

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



1. Block Diagram of our code

## Writing Sine Wave to DAC and Playback of Audio

Everything that was mentioned so far is performed in the “main” function of the program, before the execution of the “while (1)” loop (represented on the right-hand side of *figure 1)*. The next part is to retrieve the two sine waves from memory, mix them and output them on the DAC channels. In order to write the sine samples to the DAC and perform audio playback, the samples need to be read from memory and output to the DAC at the right frequency, which is the frequency at which it was sampled. Using the Systick timer, we set the desired timer frequency to the clock source frequency (80MHz) divided by 16000, which gives an interrupt at a frequency of 16kHz. Then, the samples stored in the memory can be read with the function BSP\_QSPI\_Read() [5] and subsequently output on the DAC channels at a frequency of 16 kHz.

However, these samples must be mixed before being sent to the DAC channels*,* in order to produce a mixed sine wave. We used the matrix equation provided in the lab document. We selected the coefficients to be of value 0.5. This number was chosen to center the samples and avoid maxing out the DAC. Once mixed, each mixed sample was sent to the DAC and the process repeated each interrupt.

We initiated channels 1 and 2 using the HAL\_DAC\_SetValue() [6]. With the mixed sine wave signal being output on the DAC channels, we can either listen to it with headphones connected to the audio jack or display the sine waves onto the oscilloscope by probing the signal.

# Results and Further Implementation

Furthermore, by plugging a headphone into the stereo jack we were able to hear, for both channels, the mixed signal. 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.

When listening to the audio signal, we can hear a constant click every 2 seconds. This is because the first sample of the mixed sine wave does not align perfectly with the last sample when sampling the wave

The FastICA algorithm must be implemented for the final deliverable of this project. It will be challenging to implement the latter algorithm in Embedded-C. We will have to make extensive use of the CMSIS-DSP library, which contains several linear algebra functions suitable for matrix operations.

# Initial 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 to output mixed sine waves. 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. 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).

##### PART 2

The next part of this lab report required us to use a separation technique, FastICA to separate the signals. Then, similar to part one, we had to display the separated signals onto the oscilloscope.

# The Design Problem

## Memory Mapping

A big issue in this project is the management of memory. In part 1, we made use of the QuadSPI external flash memory to store the large mixed signal arrays. However, using the QuadSPI memory in part 2 of this project proved to be more complicated. This is due to the fact that we cannot write arrays straight into the direct memory.

Fast ICA implementation we have in C requires an array as an input. Originally, we had hoped to be able to use the internal flash memory but could not properly create arrays from the memory locations, nor read and write properly to the memory. The memory-mapped QuadSPI presented the same issues. Such, the signal had to be stored in an array, which must be in SRAM.

## Implementation of FastICA

Fast Independent Component Analysis (FastICA) is one of the most popular algorithms used for blind source separation. The algorithm takes as input the mixed observations and and works to estimate the mixing matrix (from part 1) via recursive gradient ascent (maximizes the signal) [7]. – verify

The biggest issue we faced was how to incorporate FastICA in Embedded-C language. However, in order to guide us towards the right direction, we were provided with a MATLAB file that contained about 600 lines of code. This seemed to be very daunting at first because it was difficult to interpret the lines of code in C. As a result, an updated file with streamlined code was given to us, from which we can base a C implementation. That was our starting point on learning how to implement FastICA.

Reading the lecture regarding Fast ICA [8] helped us understand the algorithm further. Since we now have two mixed signals generated, we can focus on separating the signals. The benefit of using this algorithm is that ICA strives to generate components as independent as possible by minimizing both the second order and higher-order dependencies in the given data [8].

# Approach to Problem

## Memory Mapping

As explained in section VI. A, the decision was made to store the signal in SRAM.Thus, we opted to create smaller portions of the signal, of length 1600 bytes, to be able to keep them in processor SRAM and perform Fast ICA on them easily. Not only does this design choice simplify the program by not having to read and write from the flash but its execution is also sped up, as the processor SRAM is much faster than the internal Flash memory.

## Implementation of FastICA

We created an initial program to help us better understand the FastICA algorithm. This program utilizes the QuadSPI memory, but instead of having the FastICA algorithm in Embedded-C on the microcontroller, we utilized a concept explored in an earlier lab experiment, namely UART transmission. Here, we create the sine waves and transmit them over UART to MATLAB, where a FastICA algorithm is applied to the samples, separating them. The separated samples were then retransmitted through UART to the board, where they were stored in the QuadSPI memory and played-back to the DAC in accordance to the Systick interrupt, as seen in part 1. This helped us understand FastICA better, such that we implemented it in Embedded-C.

The implementation of the Fast ICA algorithm in Embedded-C was done with the help of the CMSIS-DSP mathematics library, more precisely with the functions it provides for matrix and array arithmetic. The biggest issue here was to get a representative amount of the signal in an array to perform the ICA algorithm on it. As a result, we settled on arrays of length 1600 bytes, which is 1/10th of a second of the mixed signal. This design choice was made, as explained in section VI, to be able to have all the data in the processor SRAM.

Explain our implementation…

# Final Results

By plugging earphones in the headphone jack we heard sound that were a lot clearer as opposed to the mixed signals. What happened here? What did we see in the oscilloscope? How did the signals look? Can we tell that the signals were separated?

**You will need to provide some justification as to how you have determined that the original source signals have been successfully recovered (e.g. Compare original and estimated mixing matrices, measure frequency of unmixed signals via FFT, etc.)**

We cannot measure the frequency of the mixed signals due to its constant change in frequency (since we’re using two sine waves). However, we measured the frequency of unmixed signals via FFT and concluded that …

# Final Conclusion

During the initial report, we were required to generate two sine waves that are mixed, store them into the flash and output to the oscilloscope via the DAC chip. We were able to read from the flash memory, visualize the mixed waves on eh oscilloscope and managed to hear a playback of x1(t) and x2(t). Each of the signals contained two notes that were a linear combination of the original signals. In the next part we were required to separate the two sine wave signals and recover the original signals s1(t) and s1(t). By using a popular separation technique, FastICA, we were able to perform BSS on the mixed sine waves to separate the signals. These signals were then placed onto the QSPI flash memory and displayed on to the oscilloscope via the DAC chip. The signals seemed a lot nicer compared to the mixed signals. Finally, we were able to acquire audio playback of s1(t) and s2(t) which sounded a lot crisper and clearer compared to the mixed signals.

##### References

1. "CMSIS DSP Software Library", *Keil.com*, 2018. [Online]. Available:http://www.keil.com/pack/doc/CMSIS/DSP/html/index.html. [Accessed: 19- Nov- 2018]
2. arm\_sin\_f32.c File Reference. (2018). Retrieved from https://www.keil.com/pack/doc/CMSIS/DSP/html/arm\_\_sin\_\_f32\_8c.html
3. Frequencies of Musical Notes, A4 = 440 Hz. (2018). Retrieved from https://pages.mtu.edu/~suits/notefreqs.html
4. Quad-SPI (QSPI) interface on STM32 microcontrollers. (2018). Retrieved from 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
5. 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.htm
6. Description of STM32L4/L4+ HAL and low-layer drivers. (2018). Retrieved from 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
7. Meyer, B., & Shahshahani, A. (2018). *Project: Blind source separation using ICA*[Pdf] (p. 3). Montreal: McGill.