Implementation of FastICA on Cortex-M4

André Vallières
260742187

Umar Tahir
260681853

Ayden Aamir Malik
260627064

Ismail Faruk
26066352

Karl Godin
260726027

***Abstract*— Embedded systems have become increasingly popular and the interest has grown to develop and implement complex functions on low-cost systems, particularly relevant to digital signal processing. The final project consists of performing Blind Source Separation (BSS) using FastICA on a low-cost system containing a Cortex-M4 microcontroller. This final report expands upon how we implemented FastICA in addition to sine wave generation and storage on external flash using QSPI, as well as a simple User Interface (UI) via USART to provide control to a computer explained in the preliminary report. These key features form the basis of the project, and this final report will fully document the design decisions and choices made while also briefly touching on constraints and unexpected complications encountered.**

***Keywords*— DAC, DSP, USART, FastICA, BSS, CMSIS, Cortex-M4, QSPI, User Interface**

I. Introduction

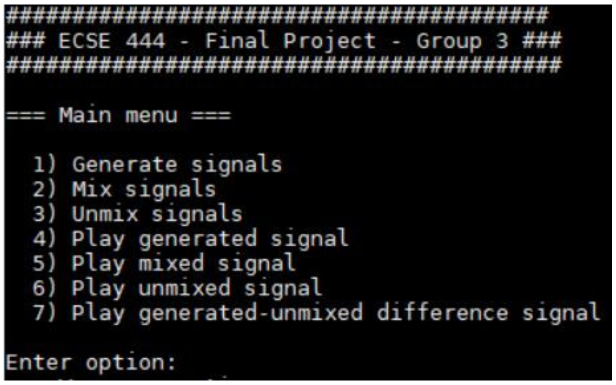
The final project consisted of three components: generating and playing two sine waves of a given frequency over the DAC channels, mixing these with a user-provided mixing matrix, and using the FastICA algorithm to perform Blind Source Separation on the mixed signals to obtain signals as close to the originally generated signals. To evaluate the accuracy of reconstructed signals, the original unmixed and reconstructed waveforms were compared in terms of their amplitude, frequency, and distortion. This was accomplished via an oscilloscope.

II. Overview

The generated signals are simple sine waves that get stored in the external flash due to on-chip memory constraints of the Cortex-M4 microprocessor. These signals are retrieved for mixing, and the mixed signals are also stored in the external flash. The use of external memory is crucial because the total size of the signals is greater than the available SRAM. The above processes are aided by a UI that allows the user to generate, mix and unmix sine wave signals, and then playback those signals. For each of these options, the user can enter parameters such as the frequencies of the signals to be generated or mixing coefficients for mixing signals. Furthermore, FastICA is performed on the stored mixed signal and the reconstructed signals are stored onto flash.

III. Functional Description

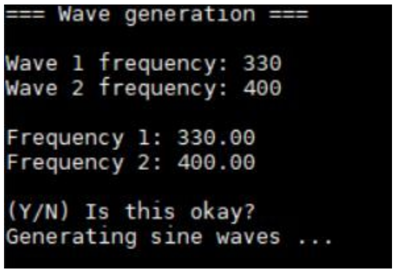
On startup, the flash is cleared to ensure it is prepared for data storage. Following that, the menu options are sent over UART and the user is prompted with seven choices. Figure 1 illustrates the options menu. If an invalid option is provided, the menu prompt is sent again. The options can be selected using numbered inputs provided via a keyboard.



```
########################################
### ECSE 444 - Final Project - Group 3 ###
########################################

=== Main menu ===

  1) Generate signals
  2) Mix signals
  3) Unmix signals
  4) Play generated signal
  5) Play mixed signal
  6) Play unmixed signal
  7) Play generated-unmixed difference signal

Enter option:
```

Figure 1: Start menu on boot up/Hard Reset

The first option prompts the user for two frequencies. The board then generates the respective samples based on the user parameters using the provided arm sine generation function. The output of this function, which maps to $[-1, 1]$ floats is then converted to map to $[0, 255]$ integers as required by the DAC. A temporary buffer of arbitrary size is used to accelerate the storing of the sine wave into flash. Instead of storing single bytes, the data block allows for storing multiple bytes at a time. The size of the buffer is 1000 and was chosen to achieve satisfactory behavior. Any arbitrary buffer size that would be a subset of the total SRAM capacity would have also worked. Fig. 2 illustrates the user interface for wave generation.



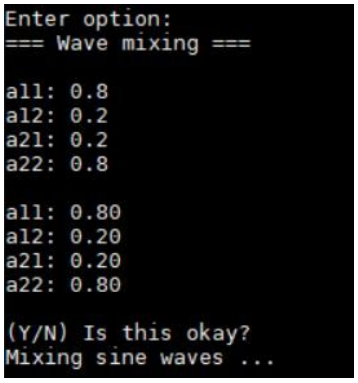
```
=== Wave generation ===

Wave 1 frequency: 330
Wave 2 frequency: 400

Frequency 1: 330.00
Frequency 2: 400.00

(Y/N) Is this okay?
Generating sine waves ...
```

Figure 2: Wave(s) generation menu

Selecting the second option prompts the user for the mixing matrix coefficients. The coefficients are normalized to make sure no overflow occurs. The wave samples are then retrieved from memory and the formula shown in Fig. 3 is used to compute the mixed signal. Fig. 4 illustrates the user interface for setting the mixing coefficients.

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} s_1(t) \\ s_2(t) \end{bmatrix}$$

Figure 3. Mixing Matrix

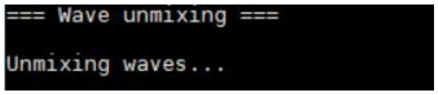


```
Enter option:
=== Wave mixing ===

a11: 0.8
a12: 0.2
a21: 0.2
a22: 0.8

a11: 0.80
a12: 0.20
a21: 0.20
a22: 0.80

(Y/N) Is this okay?
Mixing sine waves ...
```

Figure 4: Wave Mixing Menu

Notice that if $a_{i1} + a_{i2} > 1$, overflow occurs. Thus, the coefficients are normalized. This consists of dividing each coefficient by the sum of the row it resides in, effectively turning coefficients into percentages of mixing. The mixed signals are also stored in flash using a temporary buffer for the same reason as explained previously.

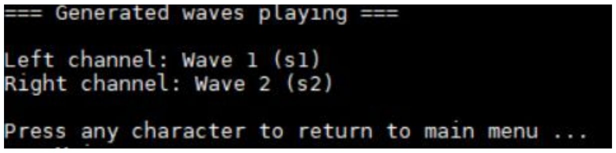
The third option unmixes the mixed waves. This is done by approximating the matrix coefficients displayed in Fig. 3 and reconstructing the original signal using the inverse matrix. This was done using FastICA. FastICA or Fast Independent Component Analysis is basically an algorithm that, for a signal coming from n sources, projects the mixed signal onto n orthogonal vectors, effectively separating the signal into n independent signals. In our case, n is 2, so the algorithm is at its simplest form. The implementation of this algorithm was based on the Python source code provided with the project guideline.



```
=== Wave unmixing ===

Unmixing waves...
```

Figure 5: Wave(s) unmixing UI

The fourth, fifth, sixth and seventh options are reserved for playback of signals. This is done by sending the selected signal to the DAC channel(s). A timer is used to generate interrupts to read from flash memory and send data onto the DAC channels at a frequency of 16 kHz. Playback is stopped once the user presses a character. Fig. 6 illustrates the user interface when a signal is being played with option 4, the generated signals, and Fig 7 illustrates the output on an oscilloscope.



```
=== Generated waves playing ===

Left channel: Wave 1 (s1)
Right channel: Wave 2 (s2)

Press any character to return to main menu ...
```

Figure 6: Example UI for playing a signal

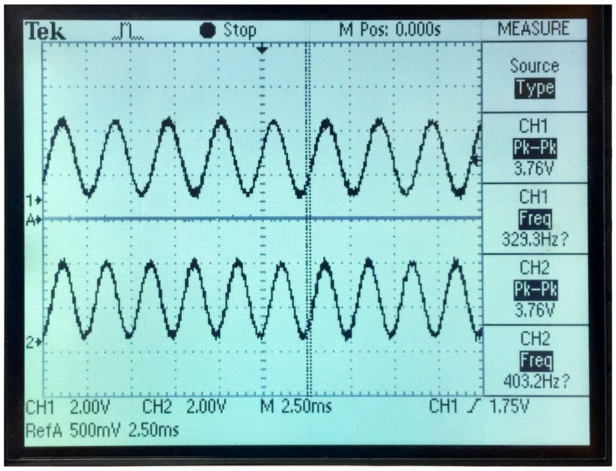
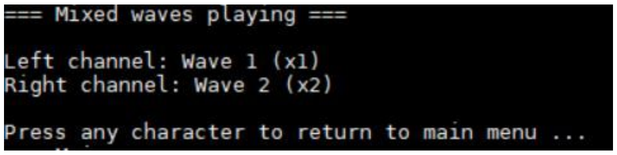


Figure 7: Example output for playing a signal

Fig. 8 illustrates the user interface when a signal is being played with option 5, the mixed signals, and Fig 9 illustrates the output on an oscilloscope.



```
=== Mixed waves playing ===

Left channel: Wave 1 (x1)
Right channel: Wave 2 (x2)

Press any character to return to main menu ...
```

Figure 8: Example UI for playing a mixed signal

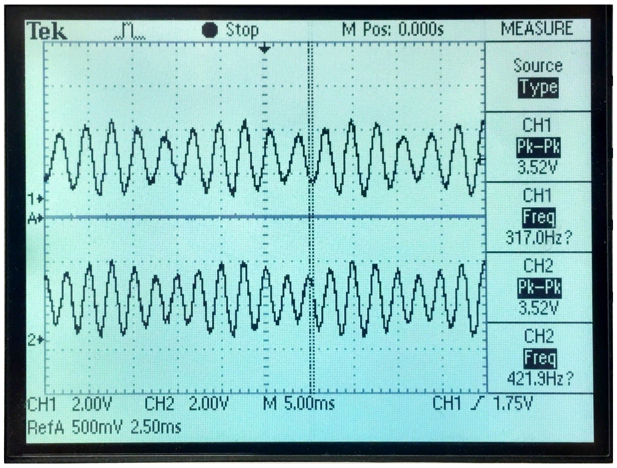
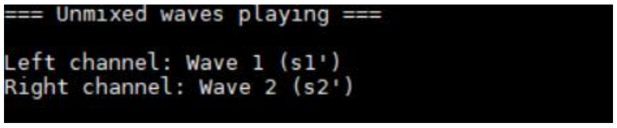


Figure 9: Example output for playing a mixed signal

Fig. 10 illustrates the user interface when a signal is being played with option 6, the signals unmixed from the mixed signals, and Fig 11 illustrates the output on an oscilloscope.



```
=== Unmixed waves playing ===

Left channel: Wave 1 (s1')
Right channel: Wave 2 (s2')
```

Figure 10: Example UI for playing the unmixed signal

Selecting the seventh option enables the differential mode which plays the difference of the generated and unmixed signals. It compares the generated signal and the unmixed signal to visualize the differences between both and validate the accuracy of the unmixing algorithm. It does so by subtracting the unmixed signal by the generated signal for each sample at a given time. The data is not preprocessed as the computations can be done within the timer interrupt. Furthermore, the data is not stored in flash as there is no preprocessed to data to store. Fig. 12 illustrates the user interface when a signal is being played with option 7, the difference between initially generated and later unmixed signals, and Fig 13 illustrates the output on an oscilloscope.

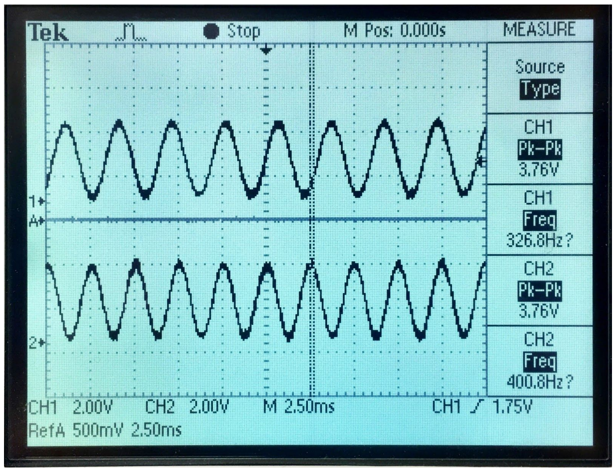
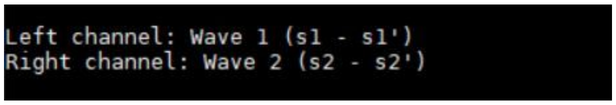


Figure 11: Example output for playing the unmixed signal



```
Left channel: Wave 1 (s1 - s1')
Right channel: Wave 2 (s2 - s2')
```

Figure 10: Wave Mixing Menu

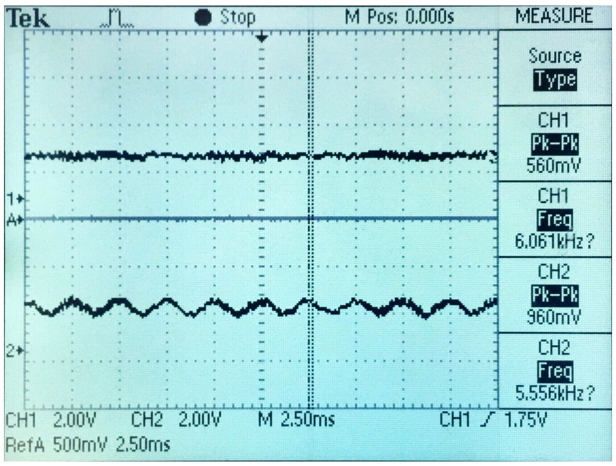


Figure 13: Example output of signal difference

IV. Parameters

Throughout the design, multiple parameters had to be set in order to achieve the prescribed behavior in the project definition. Although, not all behavior was outlined such as the resolution of the signal, so some design choices ought to be made.

A challenge was encountered early on in determining the base address locations for the generated and mixed waves. These had to be chosen so as to prevent any possibility of address overlap at any state of the program. Also, since flash is erased before writing to it, we made sure the generated and mixed waves were on different blocks so we could more effectively erase a portion of the flash. As of signal resolution, the system offers two options: 8-bit or 12-bit encoding. Initially, higher precision seemed better. However, the implementation of 12-bit would require much more processing and memory to accomplish. This is mostly due to the fact that the word size of the flash is 8-bits (byte addressable). In fact, using 12-bit precision would require two accesses per signal datum and some preprocessing to recover the generated data from flash, so we decided to use 8-bit because of its ease of use and because precision is not a main requirement of the project. Therefore, all interactions with flash and the DAC were done so with 8-bit long integers such as when converting the data and when sending it to the DAC.

Furthermore, a timer was used to send the generated wave over to the DAC. This was chosen to allow UI input while simultaneously sending data. According to the project specifications, the frequency of the generated signal should be 16 kHz. With a clock frequency for the processor of 80 MHz, the desired number of ticks between each timer interrupt is 5000 as shown in Eq. 1.

$$\frac{80\ MHz}{16\ kHz} = 5000\ ticks$$

Equation 1. Calculation for period of timer

Transmission of data was done in two ways, over UART and over pins. More precisely, the DAC signal was sent over the GPIO pins A4 and A5 for stereo output. UART was configured to transmit over the USB connection using a similar way to the labs. This corresponds to the GPIO pins B6 and B7. Furthermore, the baud rate was set to 115200 kbit/s for optimal transmission speeds.

V. Results

Ultimately, the system was implemented to perform simple sine wave generation, mixing, unmixing, and playback, as well as providing a user interface for interaction with the system. Furthermore, signal difference playback can be performed to assess the accuracy of the unmixing algorithm.

Although the project was completed according to project specifications, there are still a few ways to improve the system. For better maintainability, the project should be refactored to reflect better coding practices. In short, the majority of the code was found in the main file. By splitting the code into multiple files, the project would become maintainable and easy to understand.

For greater data precision and reduced rounding errors, the data size stored in flash and sent to the DAC could be increased from 8 to 12 bits. This would increase the resolution of the sound wave, thus creating higher quality signals and more precise unmixing. However, as mentioned previously, this would require significantly more processing time and memory space due to the byte alignment of data in flash.

To improve the processing speed for matrix manipulation, the CMSIS library matrix function can also be employed for signal mixing since the CMSIS library functions are much more optimized in most cases. Currently, simple multiplication was used for mixing the signal. The matrix functions would become useful for more complex processing. The FastICA implementation would also benefit from using CMSIS library matrix more intensively since the algorithm is easily parallelizable and this would effectively speed up the process. The main challenge for this is that complete signals do not fit in the SRAM, but it would certainly beneficial to perform to algorithm on big chunks of data at a time rather than on a single element.

Also, there can be slight clipping that happens every 2 seconds. This is caused by the frequency not being a whole number. A simple fix could be to round to the frequency, but that would affect the final generated frequency. A linear interpolation could also be done to blend the signal in between each replay. However, this could still generate signal artifacts.

Although a few improvements could be done, the system successfully served its purpose of generating and processing simple sine waves. This demonstrates the feasibility of basic signal processing on low-cost embedded systems.