Blind Source Separation Using ICA

Nayem Alam
ECSE444 - Microprocessors
McGill University
Montreal, Canada
260743549

Alex Masciotra
ECSE444 - Microprocessors
McGill University
Montreal, Canada
260746829

Tristan Bouchard
ECSE444 - Microprocessors
McGill University
Montreal, Canada
260747124

Thomas Philippon
ECSE444 - Microprocessors
McGill University
Montreal, Canada
260747645

***Abstract*— The goal of this lab was to build an audio application that performs blind source separation (BSS) using the Fast Independent Component Analysis (FastICA) algorithm. The project was implemented on the STM IoT node with a STM32L4 MCU, based on ARM cortex M4 core. This was done in Embedded-C programming using the Keil µVision5 IDE. Additionally, the CMSIS-DSP library, the HAL drivers, the Quad-SPI (QSPI) interface and MATLAB, were tools used to overcome the hardware and software difficulties encountered along the design process. The initial goal, which was to implement everything on the STM32L4 board was not satisfied. This was due to the fact that there were some memory allocation and software issues that were encountered. However, using the UART protocol to send the sine samples to MATLAB, performing FastICA in MATLAB and then sending the samples back to the STM32L4 board, proved to work for the un-mixing of the sine waves. This acted as an alternative approach which allowed the achievement of a functioning application.**

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

I. Introduction

The purpose of this experiment was to apply the skills acquired from previous labs to building a practical application, namely an audio application that performs BSS using the FastICA algorithm. This project was separated in two deliverables. The first one was to demonstrate that two sine waves of given frequencies could be generated, mixed, stored on the STM32L4 board and then output to the two DAC channels for audio playback. The second deliverable was to demonstrate that we can build an audio application that is capable of un-mixing the two sine waves generated in the first deliverable through the FastICA algorithm.

FastICA is one of the most popular algorithms used for blind source separation. The algorithm takes as input the mixed observations $x_1(t)$ and $x_2(t)$ and works to estimate the original mixing matrix via recursive gradient ascent [1]. The original mixing matrix is the 2 by 2 matrix that is represented in equation (1). For this project, the mixed observations $x_1(t)$ and $x_2(t)$ result from the mixing of the two sine waves $s_1(t)$ and $s_2(t)$ using equation (1).

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} s_1(t) \\ s_2(t) \end{bmatrix} \quad (1)$$

From an engineering perspective, this project was quite challenging due to several hardware and software limitations. For instance, to make the playback of two sine waves possible, at least 1 second worth of samples were needed to be stored on the STM32L4 board for each sine wave. At a sampling frequency of 16 kHz, which is the required sampling frequency for this project [1], it meant that 32,000 samples had to be stored on the board and accessible at any time. Knowing that there is only 1 MB of flash memory and 128 kB of SRAM available in the MCU, we had to think about alternatives for storing the samples [1]. Furthermore, design decisions had to be taken concerning the synchronization of the DAC channels, that is, the interrupt source. The Systick timer is an easy way to raise interrupts at a desired frequency, however, its usage is limited when a real-time operating system (RTOS) platform is utilized [2]. This report provides a description of our design approach and will describe our implementation of the audio BSS application, as well as the important decisions taken in order to succeed in the required task.

II. Challenges with FastICA

The biggest challenge encountered was the implementation of FastICA in Embedded-C. Due to the vectorized nature of the latter algorithm, extensive use of the CMSIS-DSP library was sought to perform the matrix operations. Heavy pre-processing of the data was required in order to prepare the signals for FastICA. The first step was centering the two signals, and the next step was to whiten the data. Consequently, a lot of time was spent trying to figure out how to make use of the library to perform complex operations, which includes determining the eigenvectors and eigenvalues of the signal samples to set up the matrix for FastICA.

Furthermore, memory allocation was also a limiting factor, as the sine samples were stored in the external QSPI flash memory (n.b. the choice of using the external QSPI flash memory will be discussed in section III.D). Since the QSPI memory unit is seen as an external device connected to the MCU with a SPI bus, we could not perform FastICA directly on the arrays that were stored into the QSPI memory. These arrays needed to be either loaded into a smaller SRAM (128 kb) memory or stored into the internal flash of the board (1 Mb), for operations to be performed directly on them.

A MATLAB file was provided to help with the implementation of FastICA [3]. Since we were running out of time and we wanted a functional project, the decision to use MATLAB was made as an alternative for the project. This way, the mixed signals could be sent to MATLAB via UART for FastICA to be performed. The unmixed samples could subsequently be sent back to the MCU and stored in the QSPI memory.

Our design process using MATLAB can be separated into 5 main steps, which is represented Fig. 1. The following section will cover the implementation of each step, as well as the evaluation and testing procedure used to verify the implementation in depth.

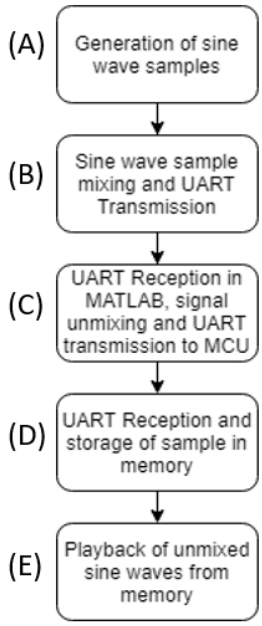

Fig. 1. *Software architecture of our BSS audio application*

III. Implementation of Project

A. Generating The Sine Waves

The first step of the project was to generate the samples required to output and play two sine waves of given frequencies. Generating sine wave signals was done using the formula given in the lab handout, as in (2). After computing the value of the angle using the arithmetic shown within the brackets of equation (2), the values of each sine wave sample were found using a function from the CMSIS-DSP library, arm_sin_f32() [4]. This function is a fast approximation of the trigonometric sine function, with one argument, the angle in radians. It returns sin(angle) sample with a value in range [-1,1]. Moreover, we were required to sample the sine wave at 16,000 samples per second, a sampling rate that achieves quality audio. Therefore, in order to have one second worth of a sine wave, 16,000 samples were computed iteratively within a loop. Moreover, to generate two sine waves that are pleasant to listen to when mixed, the corresponding frequencies 261Hz and 392Hz were selected [5]. These two were selected because they sound good when superimposed according to the lab handout.

$$s(t) = \sin(2\pi f \frac{t}{T_s}) \quad (2)$$

Even at such an early design stage, we needed to make sure the samples generated were exact, meaning that they could be used to generate an audio signal. We decided to output the samples to the DAC as they were generated. As a result, determining the accuracy of the two generated sine waves would be easy using the oscilloscope to measure their frequencies. 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. The reason why such a resolution was chosen will be explained in the next section (subsection C), as it relates to the usage of the external memory and the UART connection. We decided to use the Systick timer to output the samples to the DAC at the proper frequency, 16 kHz. The decision of using the Systick timer is strictly for simplicity, as the NVIC interrupt related to the latter timer was enabled in the provided base project. At this stage, we were aiming at a fast implementation because our goal was to test the accuracy of the two sine waves generated. Fig. 2 demonstrates the two original sine waves via DAC onto the oscilloscope.

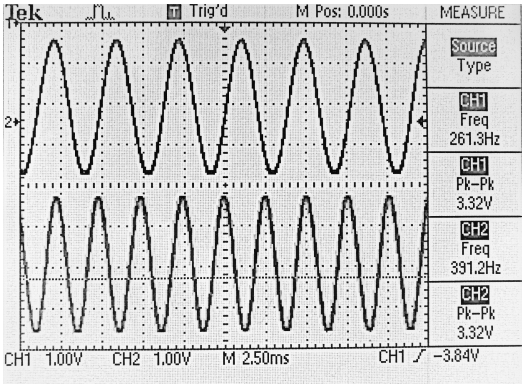

Fig. 2. *Two original sine waves generated, output to the DAC*

From Fig. 2, we can deduce that the frequencies in the two channels correspond to the frequencies 261 Hz and 392 Hz. Also, we can observe that our mapping of the samples are accurate since the peak to peak voltage of the 2 channels are approximately 3.3volts.

B. Mixing and Transmitting the Samples to UART

Mixing the signal was straight forward, as it was done using equation (1). The parameters inside the mixing matrix could be set to any value in the range [0,1]. We selected the coefficients randomly. Then we transmitted the mixed samples to UART. The HAL_UART_Transmit() function provided in the HAL drivers [6] can only send 8-bit unsigned integers at a time. Since the mixed samples were 32-bit floats, the decision taken was to convert the mixed samples to unsigned 8-bit integers. Thus, each HAL_UART_Transmit() function call sends a single sample that can be received by any device connected to the communication port. Since we had already written the code to convert the samples to 8-bits unsigned integers, we simply reused the code previously written. Fig. 3 shows the mapping of the mixed sine samples to a range of [0, 255] from line 248 to 253 and their conversion to unsigned 8-bit integers from line 256 to 257. Additionally, the two samples are transmitted to UART on line 260 and 261. The last argument of the HAL_UART_Transmit() function is the timeout value. It was set to a very large value to account for any possible delay introduced at the receiving end.

To make sure the samples were correctly sent to UART, we used MobaxTerm, which is a software for serial communication. A simple serial connection was set up with the MCU board, and we were able to observe the samples being printed in the serial window.

```
247     // Shift value of sample to get only positives
248     sine_sample= sine_sample + 1;
249     sine_sample2 = sine_sample2 + 1;
250
251     // Map to 8bits by multiply by 128, subtract 1
252     sine_sample = ((sine_sample * 255/2));
253     sine_sample2 = ((sine_sample2 * 255/2));
254
255     //convert from float to int to send over UART
256     data = (uint8_t)sine_sample;
257     data2 = (uint8_t)sine_sample2;
258
259     //send over uart
260     HAL_UART_Transmit(&huart1,&data, 1, 30000);
261     HAL_UART_Transmit(&huart1, &data2, 1, 30000);
```

Fig. 3. *Mapping and transmitting mixed sine samples over UART*

C. Unmixing the Sine Waves in MATLAB via FastICA

After the sine samples were sent to MATLAB, we applied the FastICA algorithm to the mixed signals, which was the crux of this project. As the implementation of the C algorithm proved to be too time-consuming using the CMSIS-DSP library, we opted to send the samples to MATLAB using the UART serial interface. To do this, we used the provided code containing the MATLAB implementation of the FastICA algorithm and modified it to handle a UART connection.

We began by receiving the 8-bit samples sent by the MCU and converted them to double precision floating point numbers, which the default variable type in MATLAB. Then, we reversed the scaling that we had done in the MCU (Fig. 3) which re-centers the mixed signal amplitudes to the range [-1, 1]. Next, we concatenated both signals into a 2x16000 matrix, which we then passed on to the "fastica" function in MATLAB [2].

Following the execution of the algorithm, we were left with two separated signals. Since the amplitude of the signals were not necessarily [-1,1], we had to determine the max and min for each signal, to identify their amplitude and be able to map them to 8-bit integers. To transmit these samples over UART back to the MCU, we first scaled them to [0, 255] range to be able to fit within 8-bit unsigned integers (as they were initially sent). We then initiated the transfer of the data, by sending a sample of each signal sequentially. At the receiving end, the samples can immediately be stored in the QSPI external flash memory, as the QSPI memory is byte-addressable and the samples are 8-bit values. Therefore, it is simply a matter of storing each sample in sequential addresses.

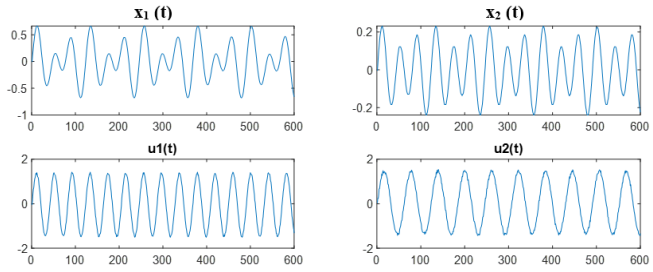

Fig. 4. *Mixed $x_1(t)$ and $x_2(t)$ and Unmixed $u_1(t)$ and $u_2(t)$ generated from the MATLAB FastICA algorithm*

D. Receive and Store the Samples in the Memory

Receiving the unmixed samples was possible by using the function HAL_UART_Receive(). The parameters stayed the same as the HAL_UART_Transmit(), except the timeout. The timeout for the Receive was set to 3,000,000. The timeout was set to this number in order to compensate for the time MATLAB takes to perform FastICA and transmit the unmixed signals back over UART.

As each sample was received, it was stored into the external QSPI flash memory. We decided to use the external QSPI flash memory, as it was suggested in the lab document. Additionally, the QSPI consists of a lot of memory, meaning that we would not run into any memory allocation problems along the process. The idea of using the internal flash was raised, as it is possible to access the internal memory using the HAL drivers. It is also possible to perform operations directly on arrays stored in internal flash. However, since there is 1 MB of memory available in the internal flash, it is not enough to hold the entire set of samples. Therefore, there were no benefits with using the internal flash since the design decision was made to perform FastICA in MATLAB, and to do so, we needed to store 32,000 samples of 1 byte.

To use the QSPI memory, first it was initialized using the BSP_QSPI_Init, and then the flash chip was cleared using BSP_QSPI_Erase_Chip. The function BSP_QSP_Write [7] 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 to. As UART transmitted 8-bit characters, each sample was able to fit into exactly one address in the QSPI memory. The samples of the first unmixed sine wave are stored from address 0 to 15,999. For the second sine wave, the samples were stored at address 16000 to 31999, which is byte addressable.

E. Writing the Unmixed Waves to the DAC

The next part was to retrieve the two sine waves from memory, mix them and output them on to the DAC channels. In order to write the sine samples to the DAC and perform audio playback, the samples needed to be read from memory and outputted to the DAC at the right frequency, which is the frequency at which it was sampled. Using the Systick timer and interrupts, we set the desired timer frequency to the clock source frequency (80MHz) divided by 16,000, which gives an interrupt at a frequency of 16kHz. Then at each interrupt, the samples stored in the memory can be read one by one with the function BSP_QSPI_Read() and subsequently output on the DAC channels at a frequency of 16 kHz. The BSP_QSPI_Read() takes the same arguments as the Write() function. The pointer to the data was used to temporarily hold each sample and give that same address to the DAC.

We then initiated and wrote to both channels 1 and 2 using the HAL_DAC_SetValue(). The decision was made to use 8 bit resolution as opposed to 12 bit. This is because as we stored 8-bit integers in memory, when reading from the memory, no further computations were needed in order to write to the DAC. This convenience allowed us to read one byte of memory at a time at every Systick interrupt and write straight to the DAC. We can either listen to the unmixed samples with headphones connected to the audio jack or display the sine waves onto the oscilloscope by probing the signal as seen in Fig. 5.

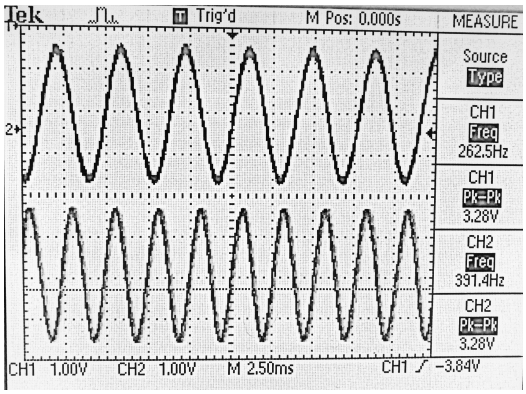

Fig. 5. *Unmixed sine waves, output to the DAC after performing FastICA*

As seen from Fig. 5, the resulting samples were properly unmixed and returned to their original base frequencies of 261Hz and 392Hz. They were also mapped to the DAC resolution properly as the peak to peak voltage is approximately 3.33 volts.

Final Project and User Interface

As mentioned in the previous sections, different evaluation procedures were used throughout the development of the project to make sure each stage was working correctly. However, the last, and probably most important step was putting everything together and designing a user-friendly application allowing the playback of the unmixed sine waves. Synchronizing the MATLAB code with the UART transmission from the MCU board was challenging in this stage of the design. For instance, we had to clear the QSPI memory during the initialization, which usually takes about 50 seconds while making sure the MALTAB function would receive all the samples through UART without timing out. This section is devoted to the description of the overall system and the implementation of the user interface.

Step A, B, C and D of Fig. 1 were executed in the main, before the while(1) loop. However, before generating the samples and transmitting them to UART, the QSPI memory had to be cleared, as stated previously. We decided to make the built in LED of the board blink during the latter process. It would stop blinking to notify the user that the generation of the sine waves can start. However, the MATLAB code must be running when the sine waves' samples are generated, mixed and transmitted to UART. For this reason, the code pauses when the LED stops blinking. The status of the pushbutton is then polled, waiting for the user to start the generation of the sine waves by pushing the button. Once the blue pushbutton is pressed, the generation of the two sines waves starts. The samples, which are mixed and mapped to 8-bit integers, are subsequently transmitted to UART while the MATLAB code is running. MATLAB then waits to receive 16,000 samples for each since wave and performs FastICA. Meanwhile, the MCU board waits to receive the unmixed signals' samples, which are retransmitted through UART when MATLAB is done processing. The MCU receives the unmixed signals two samples at a time, one for each wave and stores them in the QSPI memory. Once the two unmixed signals are stored, step E of Fig. 1 is executed. The code enters the while(1) loop, and outputs the samples on the DAC for audio playback of the two unmixed sine waves.

The results are shown on Fig. 5. The two unmixed signals had frequencies of 262.5 Hz and 391.4 Hz. We were also able to clearly distinguish between the two frequencies in each DAC channel by using our earphones. Knowing that the original frequencies were equal to 262 Hz and 391 Hz, we can conclude that our application was able to achieve the goal of this project.

Conclusion

The following lab allowed us to dive into a microprocessor and understand its components by building an audio application. The first part of the project allowed us to incorporate different functions and libraries to output mixed sine waves. We were able to read from the QSPI flash memory and then output the mixed sine waves to the DAC channels. Finally, while the oscilloscope was running, we also managed to get playback of $x_1(t)$ and $x_2(t)$, where each of them contained two notes that are a linear combination of the original signals ($s_1(t)$ and $s_2(t)$). For the next part, by using a separation technique, FastICA, we were able to perform BSS on the mixed sine waves to separate the signals via MATLAB. We then displayed the unmixed signals successfully on the oscilloscope via the DAC channels (Fig. 5). The final frequencies of 262.5 Hz and 391.4 Hz, which are very close to the original frequencies of 262 Hz and 291 Hz were observed on the oscilloscope. Finally, we were also able to acquire audio playback of the two unmixed sine waves using earphones. Unfortunately, the sound quality was not as expected, as the sound was crispy and metallic. This is probably due to the 8-bit resolution of the DAC channels. A possible improvement for the sound quality would be to use a 12-bit resolution for the DAC. Another way to improve the sound quality would be to include a proper signal amplifying circuit, as the MCU is unable to supply sufficient current to the earphones to get clear sound.

Although we were able to complete the project with a working implementation, there are some areas that could have been improved. For one, implementing the whole FastICA algorithm in C using the CMSIS-DSP library. This would remove the burden of synchronizing the MATLAB FastICA code with the C code. More importantly, it would make a more legitimate project, as the motivation behind using an MCU is to design standalone applications. However, nowadays in the IoT world, our project model can be useful for heavy computational tasks that cannot be performed on embedded systems alone.

References

[1] Meyer, B., & Shahshahani, A. (2018). *Project: Blind source separation using ICA*[Pdf] (p. 3). Montreal: McGill.

[2] Mbed OS Documentation | Reference. (2018). Retrieved from https://os.mbed.com/docs/v5.8/reference/rtos.html

[3] (2018). Retrieved from http://research.ics.aalto.fi/ica/fastica/

[4] arm_sin_f32.c File Reference. (2018). Retrieved from https://www.keil.com/pack/doc/CMSIS/DSP/html/arm__sin__f32_8c.html

[5] Frequencies of Musical Notes, A4 = 440 Hz. (2018). Retrieved from https://pages.mtu.edu/~suits/notefreqs.html

[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] 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