Laboratory 2

Exploring MAC/SIMD/Buffers using the STM32F411E DISCO

1. PREREQUISITES

- You have the ENCM 515 lab kit (the STM32F411E DISCO board, a USB A to mini-B cable
- You have your own set of headphones
- You have successfully installed the STM32CubeIDE and the STM32Cube Programmer[1]
- Check D2L for other materials that might be required

2. OUTLINE

In this lab, the aim is to explore some of the concepts around optimizing the C code based on understanding what is happening on the processor and then trying to use the features available on the processor. Unlike the previous lab, this activity will have a bit less handholding.

By the end of this lab, you should be:

- Comfortable with using the ITM port and SWV Trace Log to profile your code
- Able to use the STM32CubeProgrammer to load data in and out of memory
- Be able to optimize filter code using a variety of techniques, including the MAC and SIMD instructions of the Arm Cortex-M4, as well as implement circular buffers.

Q> There are several Questions throughout the lab for you to answer. You should collate your answers/observations/screenshots in a Word document (or equivalent), such as the lab sheet available on D2L. If there are demo components, you should show these to an Instructor/TA. If you are doing this lab remotely, you will need to screen-record/video the demo components.

Submit your completed Lab 2 sheet as a pdf together any other added/modified .c or .h files to a zip archive and upload to D2L.

PART ONE

3. PREP

Using STM32CubeProgrammer, download `noisy.bin` to 0x08020000 to your STM32F411E DISCO, as we will use the data of noisy.bin as our input data for this lab. This file is actually audio data, sampled at 8000 Hz, and stored in the well-known `wav` format.

➢ *Use this margin to make notes!*

You should check out this link to get an idea of what the wav format entails, but it is, essentially, 44 bytes of header data, followed by the remaining audio data (in our case, stored as 16-bit PCM). This audio file contains both left and right channel data, so you can imagine it as something like:

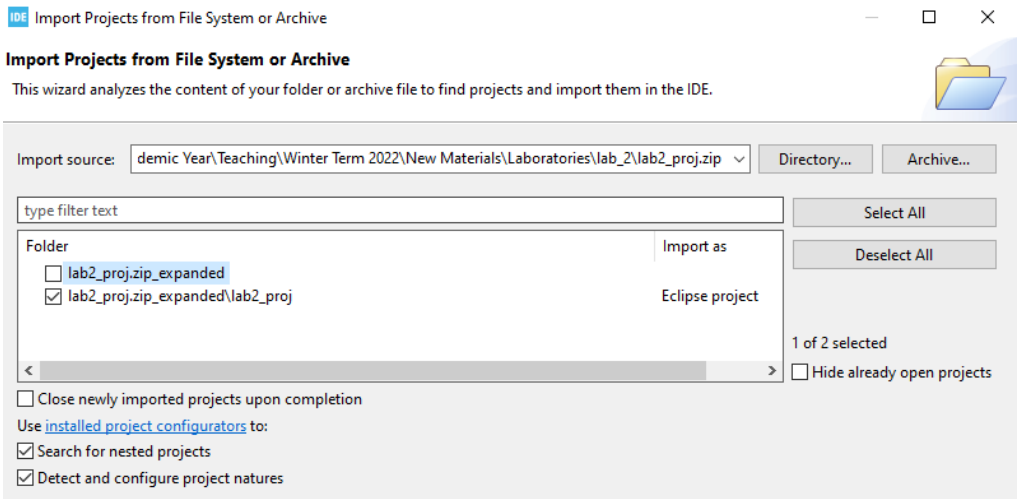
L data[0]	R data[0]	L data[1]	R data[1]	L data[2]	R data[2]	….

4. IMPORT EXISTING PROJECT

Open STM32CubeIDE and import the **lab2_proj** project into your workspace. To do this, go to File > Import, and follow the prompts. You should be able to build the project without any issues after importing. The project has a number of additional files that you can play around with outside the lab.

Note: This project is has some custom code for initializing peripherals that isn't the same as the autogenerated code that is produced by modifying the .ioc. DO NOT MODIFY THE .IOC OR AUTO-GENERATE NEW CODE.

[1] https://www.st.com/en/development-tools/stm32cubeprog.html



Spend a bit of time examining the code that we've provided, focusing on **main.c**. Of particular interest are the various `#defines`, global variables, and so on.

The main fun happens line 158 onwards, in the `while(1)` loop of `int main()`.

Out of the box, you should see the following code:

```
if (sample_count < 64000) {
	newSampleL = (int16_t)raw_audio[sample_count];
	newSampleR = (int16_t)(raw_audio[sample_count] >> 16);
	sample_count++;
} else {
	sample_count = 0;
}
```

This code iterates through `raw_audio` (notice that `raw_audio` is an `int32_t*` pointing to 0x802002C). The code pulls out 2x 16-bit data at a time, separating them into `newSampleL` and `newSampleR`. In this lab, we'll focus mostly on processing `newSampleL`.

Below this is our processing code. Note that the `if` statement suppresses the filter output until we've collected enough input data.

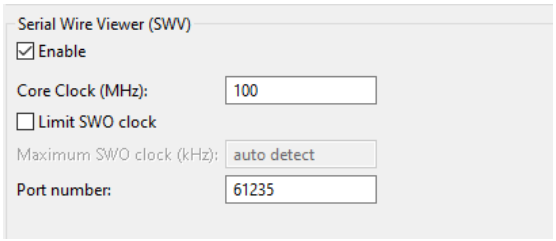
```
filteredSampleL = ProcessSample(newSampleL,history_l); // "L"
new_sample_flag = 0;
if (i < NUMBER_OF_TAPS-1) {
    filteredSampleL = 0;
    i++;
} else {
    if (bufchoice == 0) {
        filteredOutBufferA[k] = ((int32_t)filteredSampleL << 16) +
(int32_t)filteredSampleL; // copy the filtered output to both channels
    } else {
        filteredOutBufferB[k] = ((int32_t)filteredSampleL << 16) +
(int32_t)filteredSampleL;
    }

    k++;
}
```

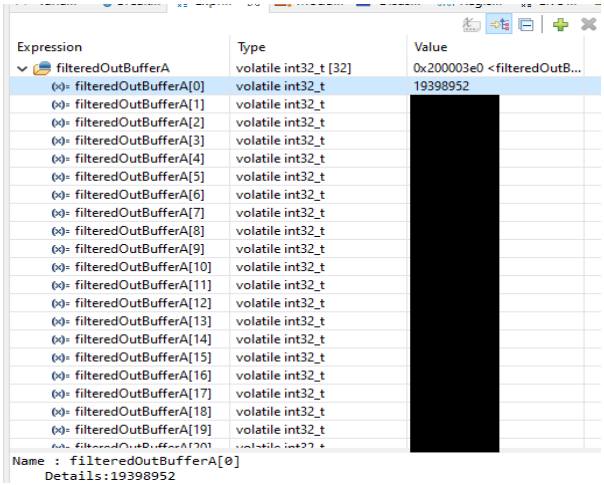
Q1> How many seconds of audio is there in the input data? Show your working.

As mentioned earlier, we're only interested in processing the L data at the moment, and we currently store the filtered output in buffers (note that we duplicate the 16-bit filtered L data to fill each entry of the filtered output buffers, where the duplicate L data is standing in for filtered R data).

Let's debug the design. Create a debug configuration as required (note that the System Core clock is configured to 100 MHz in this project). Open up the SWV Trace Log and Configure the Serial Wire Viewer settings to enable ITM Stimulus Ports 31, 30, and 0.



Q2> Run the debugger until filteredOutBufferA is full. Take a screenshot of the contents of the buffer, as below. Do you get the expected output?



Now, let's try something different. **Comment out** line 32: `#define FUNCTIONAL_TEST 1.` This will change the way our program operates, where instead of having the new data come in in the while(1) loop, the data is instead "produced" by a periodic interrupt. Check out

`void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` near line 302.

This should trigger this call back function every 125 us. As you can see, it works essentially the sane as the code in main but sets the `new_sample_flag` to 1. When the interrupt has been serviced (i.e., at the conclusion of the callback), the processor resumes its execution, probably somewhere in the `while(1)` loop of `int main()`. At some point, the `ProcessSample()` code will run because the `new_sample_flag` is now 1.

Q3> Run the debugger again until filteredOutBufferA is full. Take a screenshot of the contents of the buffer, as below. Do you get the expected output?

If we take a look at the SWV trace log, you should be able to notice that there are a lot of writes to ITM Port 30.

1	ITM Port 30	10	23800	238.000000 µs
2	ITM Port 30	10	48799	487.990000 µs
3	ITM Port 30	10	73798	737.980000 µs
4	ITM Port 30	10	98797	987.970000 µs

Take a look again at the timer callback. There following code runs if new_sample_flag is 1 when the timer callback is triggered – i.e., this is notifying us that we have missed a sample!

```
if (new_sample_flag == 1) {
	ITM_Port32(30) = 10;
}
```

PART TWO

5. Lab Tasks

This sets the scene for us – we want to be able to be able to filter the samples in a timely fashion! The rest of the lab is as follows:

Q4> How much time/how many cycles are required for each sample, with the original ProcessSample function? Take a screen shot of the generated assembly code and explain where you might think the main bottlenecks are. Screenshot/copy your function and explain the changes that you made.

Q5> Create a new ProcessSample2 function, making use of the appropriate MAC instruction. You might remember from class that we used in-line assembly for this. How much time is required for each sample? Do we satisfy timing requirements now? Screenshot/copy your function and explain the changes that you made.

Q6> Create a new ProcessSample3 function, making use of the appropriate SIMD instruction. You might remember from class that we might be able to use an intrinsic for this. How much time is required for each sample? Do we satisfy timing requirements now? Screenshot/copy your function and explain the changes that you made.

Q7> Create a new ProcessSample4 function, this time, treating history_l as a circular buffer. How much time is required for each sample? Do we satisfy timing requirements now? Screenshot/copy your function and explain the changes that you made.

For each of these questions/tasks, you will need to be mindful **of checking that your code remains functionally correct**, so you might want to alternate between `#define FUNCTIONAL_TEST 1` being commented-out or defined. You can add new variables/functions as you see fit**. Be sure to add to your lab sheet an explanations of what you have added or modified.**

Q8> Write a reflection of what you have observed and learned in this lab, including an explanation of how you checked that your code optimizations preserved functional correctness.

To submit:

- **Add your lab sheet and any other added/modified .c or .h files to a zip archive and upload to D2L.**