Laboratory 1

Exploring DSP concepts with STM32CubeIDE and 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 successfully installed the STM32CubeIDE and the STM32Cube Programmer[1]
- You have set up the supporting Jupyter Notebook in Google Colab or locally
- You should download the STM32CubeF4 package before the lab, if possible[2]
- Recommended programs: Audacity
- Check D2L for other materials that might be required

2. OUTLINE

In this lab, the aim is to explore some of the concepts around implementing DSP algorithms in an embedded platform. This lab comprises two major parts: (1) exploring debugging, fixed vs. floating point, and (2) investigating the differences between fixed-point and floating-point implementations of digital filters.

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
- Able to implement and characterize a floating point and fixed point FIR filter

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 1 sheet as a pdf together with your main.c, in a single zip archive, on D2L.

PART ONE

➢ *Use this margin to make notes!*

3. CREATE A NEW PROJECT

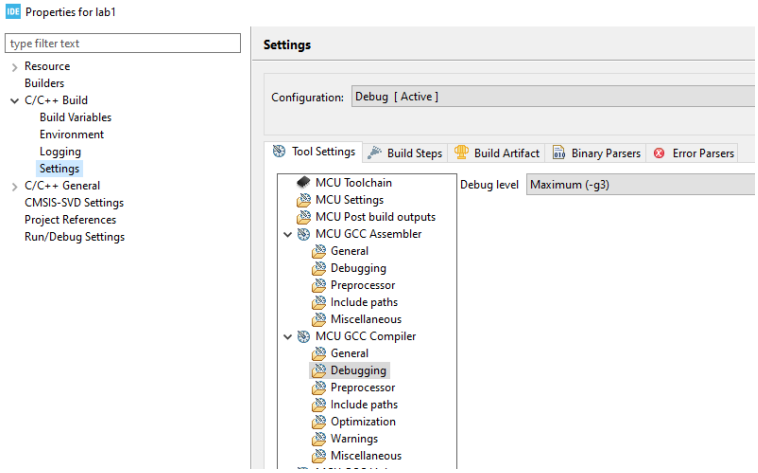
Create a new STM32CubeIDE project for the STM32F411E DISCO (follow the Lab 0 instructions as required). Don't forget to change the settings for the RCC Mode and Configuration (i.e., set the HSE to use the Crystal/Ceramic Resonator). Once your project is created, you should **add the necessary code** and **change the required settings** so that you can see the output of `printf()`'s on the SWV ITM Data console.

In this lab session we will make use of the ITM stimulus ports, so let's add the following handy macro. Later on, this will let us write values to the n^{th} port.

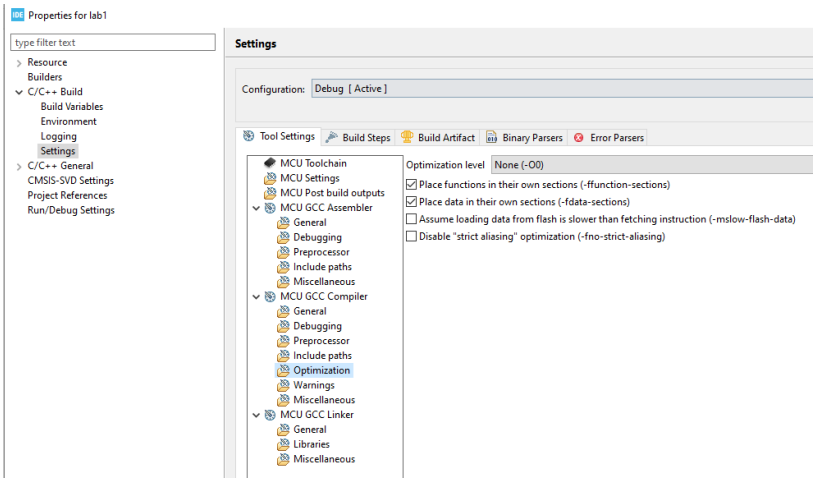
```
#define ITM_Port32(n)     (*((volatile unsigned long *)(0xE0000000+4*n)))
```

Because we are using this lab to explore some of what's happening under the hood, we should make sure that we are compiling with debug information and that we are not letting the compiler do too much optimization for us at this stage. Right click on your project in the Project Explorer and select properties.

[1] https://www.st.com/en/development-tools/stm32cubeprog.html
[2] https://github.com/STMicroelectronics/STM32CubeF4 --- BTW, I've uploaded a copy of this to D2L



Make sure Debug level is set to Maximum.



And that optimization level is None.

Note that, in practice, you will change these settings depending on where you are in the design flow. For instance, when it's time to send your code into production, it might not be necessary to include debug information (and in some cases, you might want to remove the debug information for intellectual property reasons). Changing the debug level can help improve code size. In main(), after the initialization stuff, add `HAL_SuspendTick();`

Q1 > Once you have set up your project, add a `printf()` to `main()` that prints your name, the date, and a short joke to the SWV ITM Data Console. Take a screenshot of the console output after running your code on the STM32F411E Disco.

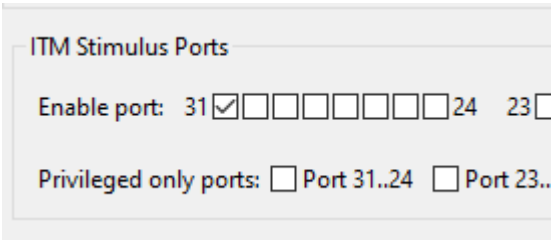
4. INSTRUMENTING CODE

Now, let's try to get a sense of how long it takes for some code to execute on this platform. **Profiling** the performance of software is an important part of any design process as it allows us to get a sense of potential bottlenecks and inefficiencies.

First, **remove** the `printf()` that you added for Q1. Write a function `void BasicLoopTest(void)` that contains a for loop that loops 2,000,000 times. Call this function in `main()` (somewhere in the `/* USER CODE BEGIN 2 */` section). Write to ITM port 31 before and after calling the function, like so:

```
  ITM_Port32(31) = 1;
  BasicLoopTest();
  ITM_Port32(31) = 2;
```

Build and then start debugging. Before resuming, open the SWV Trace Log view, and make sure you enable port 31 in the trace configuration. Resume the program.



Take a look at the SWV Trace Log. You should notice that there are two entries for ITM Port 31.

Q2 > Make a note of the time stamps and cycles for the two ITM Port 31 entries. How many clock cycles are there between the two writes? How much time has elapsed? What is the average amount of time taken for each loop iteration?

After running it once through, restart the program and instead switch to **Instruction Stepping mode**, when the processor starts executing `BasicLoopTest`. Make a note of which instructions are being executed.

5. BASIC OPERATIONS ON FLOATING/FIXED POINT NUMBERS

To get a sense of the differences between floating point and fixed point operations on this platform, let's try out some arithmetic in C.

Add the function prototypes for two new functions, called `void FloatingExperiment(void)` and `void FixedExperiment(void)`.

5.1. Floating Point Experiments

Let's start with the following:

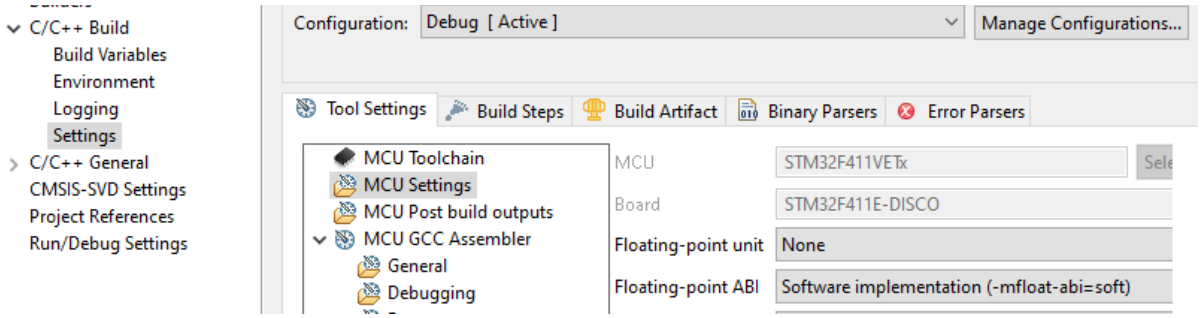
```
void FloatingExperiment(void) {
	float a = 0.5;
	float b = 0.125;
	float c = a * b;
}
```

Call this function in `main()` and surround the function call with writes to `ITM_Port32(31)` so we can get the cycle count before and the cycle count after. Start the debugger and add a breakpoint at the start of this function. Then, when you're in the function, take a look at the instructions being executed as well as the different contents of the floating point registers.

Q3 > How many clock cycles are taken to perform the floating point function? What happens if you change the value of a and b to something that is not a "clean" power of 2. Try a few different values (maybe 5), and record the time taken. Include situations where the numbers are different magnitudes (like 10-point something multiplied by 0-point something…)

Note, if you want to be able to print floats to the console (i.e., `printf("%f\n", some_float);`), you will need to add a special flag to the linker. To do this, go into the Project Properties, select C/C++ Build > Settings > Tool Settings > MCU GCC Linker > Miscellaneous, and then add `-u _printf_float` to the Other flags.

For our next trick, let's tell the compiler to pretend we don't have a floating point unit, by changing the project settings.



Q4 > How many clock cycles are taken to perform the floating point function now?

Once you've done this experiment, be sure to change the project settings back to what they were before.

5.2. Fixed Point Experiments

5.2.1. General Fixed Point

Now, let's try a similar experiment, but with Fixed Point numbers. Let's use Q1.15 format for now. Because there's no "built-in" data type for Q numbers (more on this later), we'll use int16_t.

```
void FixedExperiment(void) {
	int16_t a = ???; // should be 0.5;
	int16_t b = ???; // should be 0.125;
	int16_t c = a * b; // leave this as is initially
}
```

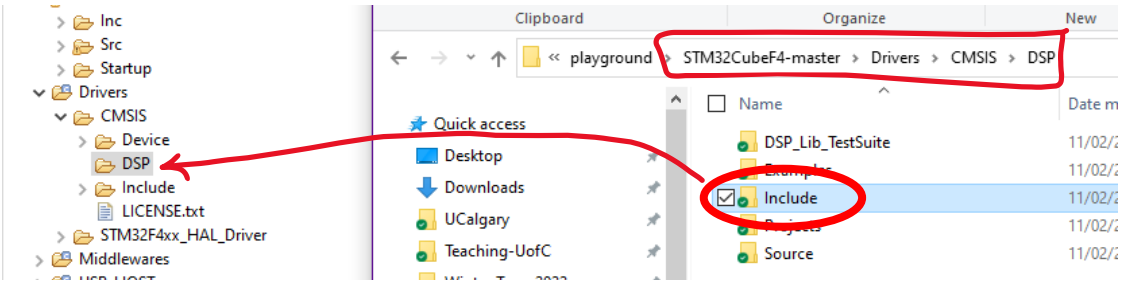
Q5 > Fix the code snippet so that the correct values are loaded in a and b. First, leave the multiplication as is. Debug the program and observe what's happening when you step through the instructions. You will probably get a wrong/unexpected value for c – modify the statement so that you get the correct result in the int16_t. How many cycles does this take?

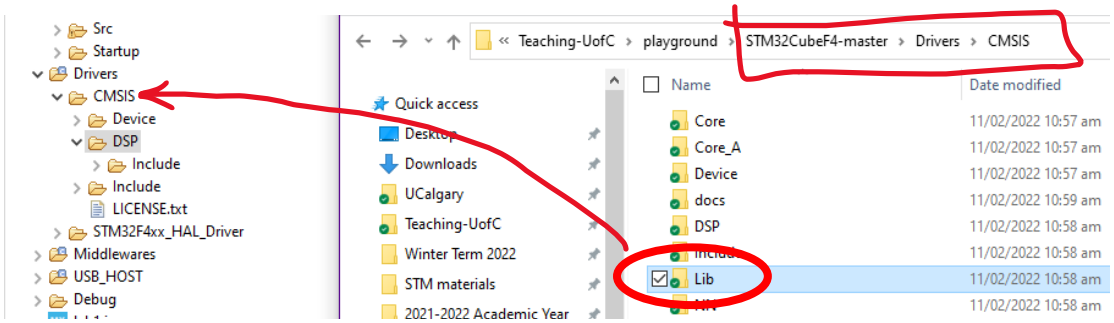
5.2.2. CMSIS

By now we should be able to infer that ARM intends their designs to do DSP, and to that end, they provide a helpful library that helps us implement algorithms faster and in a more portable way. The ARM Common Microcontroller Software Interface Standard (CMSIS) is a vendor-independent abstraction layer. In this course, we will pay particular attention to the CMSIS DSP Library.[3] Let's download STMicroelectronic's version for their STM32F4 family from here if you haven't already. Once we've downloaded it, we can integrate the DSP libraries into our project.[4]

- Create a folder under Drivers\CMSIS called DSP
- Copy <STM32Cube_Repository>\STM32Cube_FW_F4_V.X.XX.X\Drivers\CMSIS\DSP\Include and paste in in the created folder
- Copy <STM32Cube_Repository>\STM32Cube_FW_F4_V.X.XX.X\Drivers\CMSIS\Lib and paste it under ..\Drivers\CMSIS.

[3] https://arm-software.github.io/CMSIS_5/DSP/html/index.html
[4] Use the official instructions here: https://community.st.com/s/article/configuring-dsp-libraries-on-stm32cubeide or by just following my condensed version in this manual





- Now, go to the Project properties and open the C/C++ build > Settings
  - in the Tool Settings tab, select the Include paths in the MCU GCC Compiler part, and add `../Drivers/CMSIS/DSP/Include` which you can find by selecting Workspace….
- Then, go to the MCU GCC Linker part and select Libraries.
  - In the Libraries part, add `arm_cortexM4lf_math`.
  - In the library search path, click add, and navigate to `../Drivers/CMSIS/Lib/GCC` in the workspace.
- Finally, click back to MCU GCC Compiler, this time selecting Preprocessor.
  - Add `ARM_MATH_CM4` as a symbol to the project. You should now be able to add `#include "arm_math.h"` to your file and build without any issues.

Note a few things. First, including the CMSIS library[5] provides us with some handy helper types, like `q7_t, q15_t, float32_t`. It also provides a few helpful functions, some of which we'll consider in future labs. For now, let's check out:

void	**arm_float_to_q15** (const **float32_t** *pSrc, **q15_t** *pDst, uint32_t **blockSize**)
	Converts the elements of the floating-point vector to Q15 vector. More...

Create a new function, `void CMSISExperiment(void);`. In this function, you should declare three arrays: an array of 10 different floats, and empty array of q15_t, and an empty array of float32_t. Pick a few random floats in the range [-1,1) and a few outside. Call the above function with the q15_t array as the destination Using this result, call a function to go back to float.

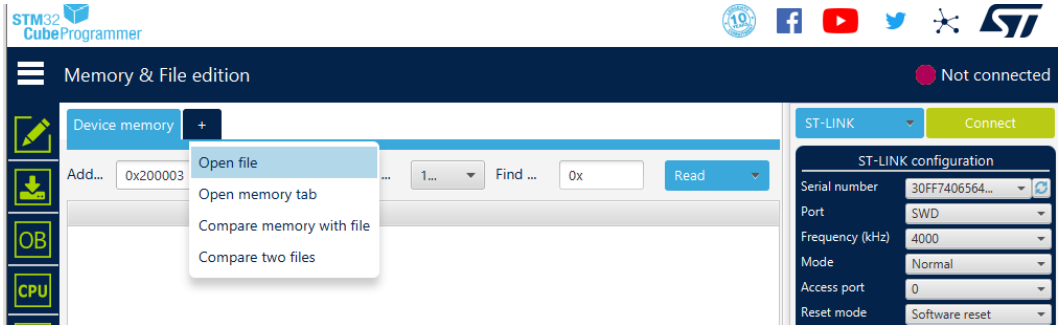
Q6> Take a screenshot of the Variables window after calling these functions. What do you observe?

[5] You should do an online search to find the documentation for CMSIS-DSP

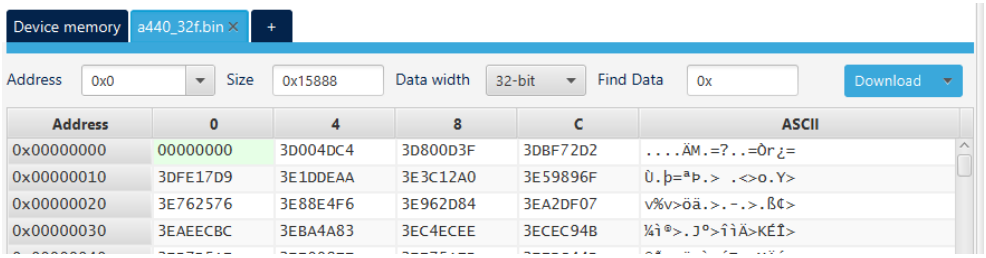
PART TWO

6. PREPARATION

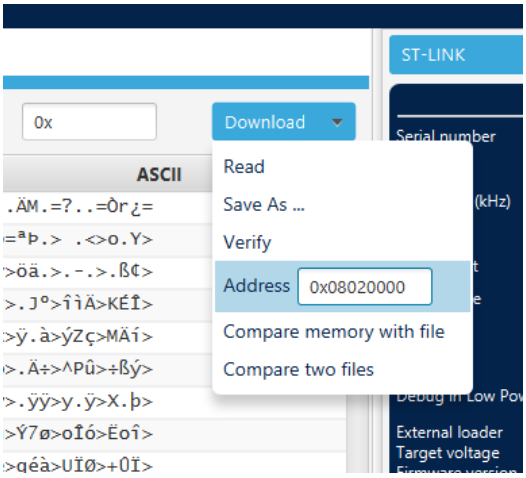
In this part of the lab, we are going to run 0.5s of a 440 Hz sine wave sampled at 44100 Hz. For our floating point experiments, we are going to use the sine wave stored as **32-bit** floats. What we need to do is get the data into our microcontroller. To do this, we will use the STM32CubeProgrammer. First, let's open the file **a440_32f.bin.**



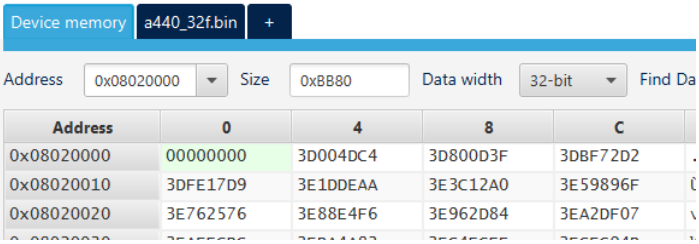
You should see the contents of that file in the window.



We are going to copy this data to a part of the microcontroller's flash memory (which should persist on reset). I've selected **0x08020000** as it's unlikely at this stage for our code to be so large as to reach this part. Select the downward arrow/triangle on "Download" and enter the address.



Now, let's connect to the board. This will only work if **you are not** currently debugging. Hit Connect on the top right. If all goes well, you board should have connected. When you hit "Download", the data should be copied to the microcontroller. You can check this by going to the Device Memory tab, and putting **0x08020000** as the address.

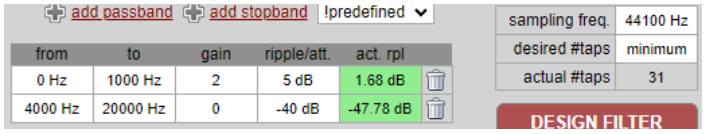


This should match up with the .bin contents!

Hit the Disconnect button on the top right (basically, we want to free up the STLink connection to the board so that the debugger in the IDE can later connect). We can now proceed to Section 7.

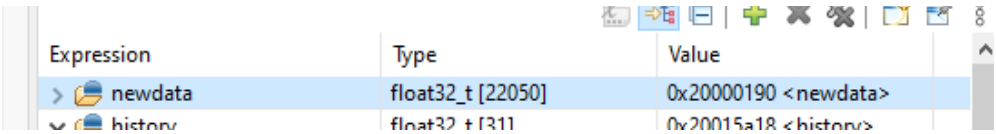
7. FIR FILTER – FLOATING POINT

For this lab, I've used TFilter to design an FIR filter with the following characteristics.

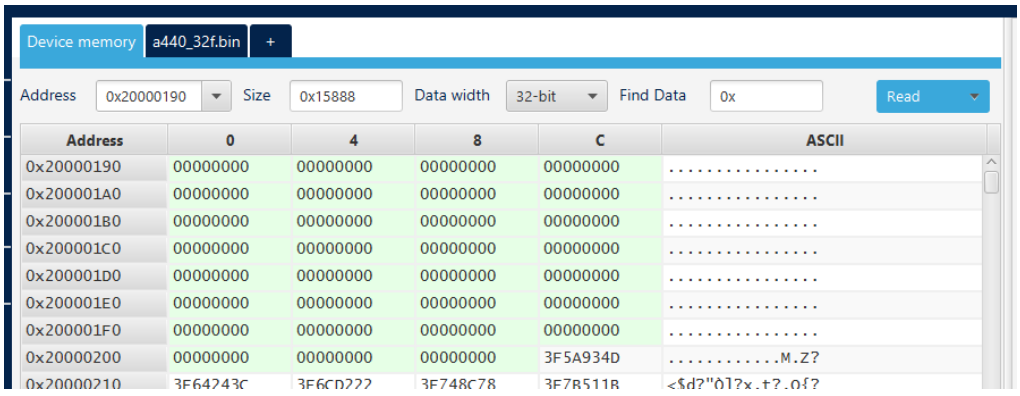


Let's run our sine wave through the filter! To do this however, we will need to write the filter function in C. There are several ways to write the C code for a filter, but in this lab, I will recommend a fairly naïve version, where the filter coefficients and input delayed samples are stored as global variables.[6] To help you along, check out **snippets.c** for some code fragments that will help start you off.

Once you have completed your implementation, you will need to run the debugger to find the memory location of the filtered signal. For example, my implementation has the "newdata" **at 0x20000190.**



Stop your debugging session. Let's now get the data processed data **out** of the microcontroller, again using the STM32CubeProgrammer. Connect the microcontroller. In device memory, this time set the address to where your filtered signal will end up. Set the size of memory to read accordingly. Reset the microcontroller with the black button on the board, wait a little bit, and then hit read. You should see some data! Click the down arrow on "Read" to "Save as…", extracting the transformed signal to something like filtered440_float.bin.



[6] You are more than welcome to write your own version, provided you can show that you are getting the correct answers, numerically!

Q7> Implement the FIR filter (floating point) and extract the filtered signal using the STM32CubeProgrammer. Load the binary data into the Lab1_Support Jupyter notebook. Plot the first 250 samples. Plot the spectrum. Screenshot these plots. How much time is taken to filter the signal? Is this filtering going to be "good enough" for our sample rate, if we were receiving new audio signals in real-time?

8. FIR FILTER – FIXED POINT

This time, you will implement the same FIR filter, but using the Q1.15 (or int16_t) data type. Follow the same process as before, **except**, I am not going to help you as much.[7] You should be able to use the model a fixed point implementation on the floating point implementation that you have just completed.

A few hints:

- You should probably comment out the floating-point related stuff from Section 7 when you embark on this part of the lab. I wouldn't delete the existing code you've written, but instead write new functions
- You will need to use STM32CubeProgrammer to load a different input file, this time, **a440_16-bit.bin**. Note that the binary file size is different to the float version. This is still 0.5s of a 440 Hz sine wave sampled at 44100 Hz.
- For this lab, do not worry about overflow/underflow.
    - However, if you have time at the end after taking the required measurements, write code to help you count the number of times the filter output overflows/underflows

Q8> Implement the FIR filter (fixed point) and extract the filtered signal using the STM32CubeProgrammer. Load the binary data into the Lab1_Support Jupyter notebook. Plot the first 250 samples. Plot the spectrum. Screenshot these plots. How much time is taken to filter the signal? Is this filtering going to be "good enough" for our sample rate in this case, if we were receiving new audio signals in real-time?

Q9> Write a reflection of what you have observed and learned in this lab

To submit:

- **Add your lab sheet and your main.c to a zip archive and upload to D2L.**

[7] Hint, check the lecture slides?