Workshop:
Using ARM CMSIS DSP Library for filtering and FFT functions

- Introduction to CMSIS
- Opening a project in STM32CubeIDE
- Connecting inputs and outputs to the Discovery board
- Filtering your input signal
- Implement a filter using MATLAB
- What is an FFT?
- Calculate FFT magnitudes

Introduction to CMSIS

CMSIS stands for Common Microcontroller Software Interface Standard https://arm-software.github.io/CMSIS_6/latest/General/index.html. It offers a variety of libraries and tools for ARM processors like those in the STM32 family.

CMSIS-DSP is an open source library of DSP functions optimized for ARM processors https://arm-software.github.io/CMSIS_5/DSP/html/index.html.

Opening a project in STM32CubeIDE

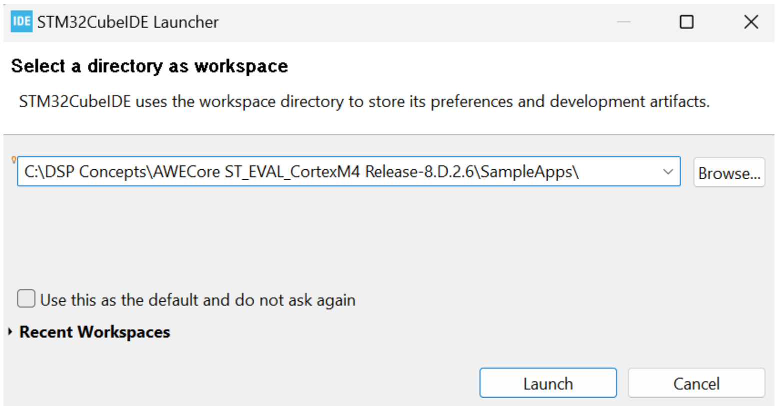
Connect a mini USB cable to one end of the Discovery board, and plug into your PC.

Open STM32CubeIDE by clicking on the STM32CubeIDE 1.17.0 icon.

For the workspace directory, enter:

`C:\DSP Concepts\AWECore ST_EVAL_CortexM4 Release-8.D.2.7\SampleApps`

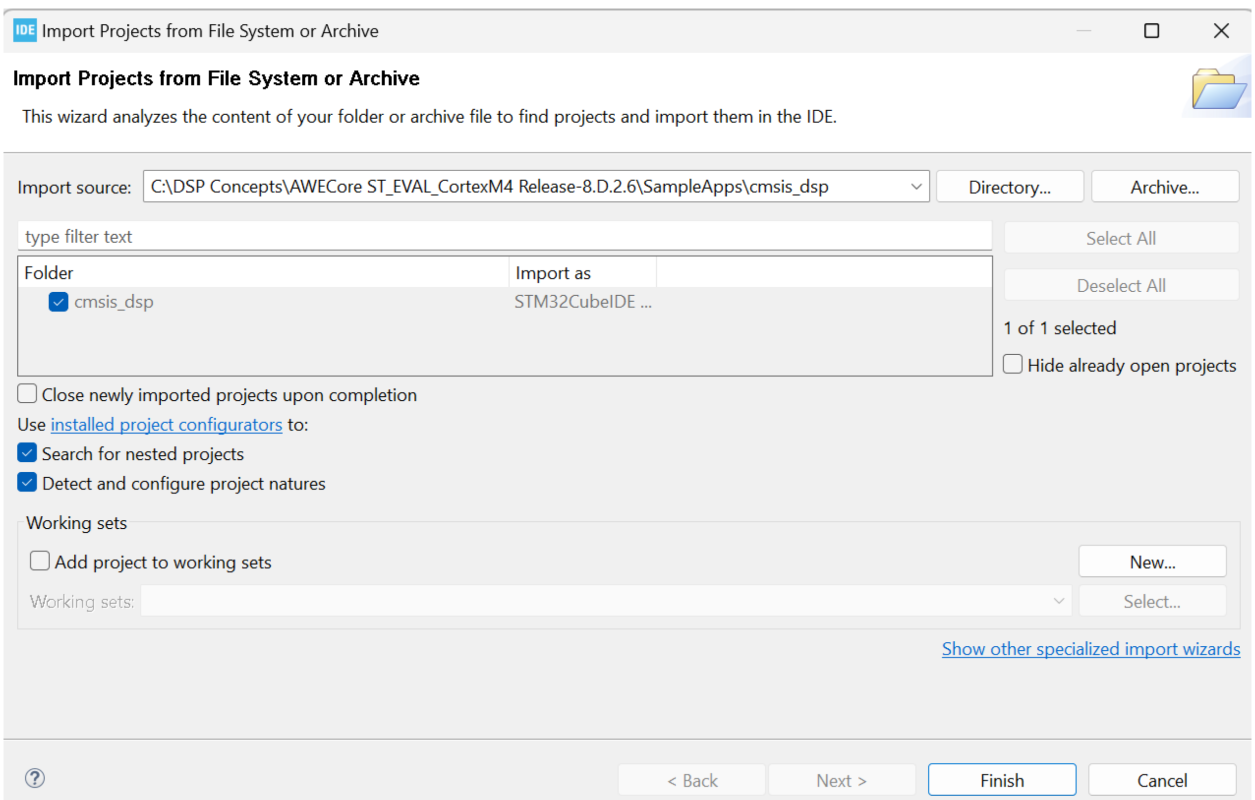
Click Launch.



Choose File – Open Projects from File System. For Import Source, choose:

`C:\DSP Concepts\AWECore ST_EVAL_CortexM4 Release-8.D.2.7\SampleApps\cmsis_dsp`

Click Finish.



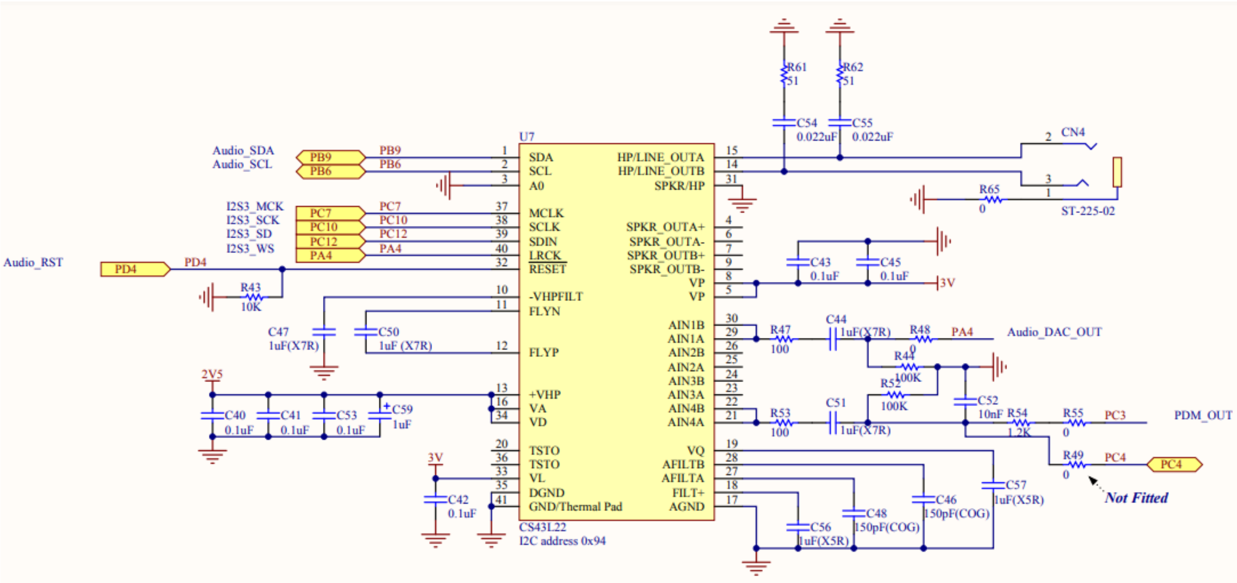
Double-click on the project name `cmsis_dsp` in the Project Explorer to expand its contents. Double-click on `cmsis_dsp.ioc` to open the configuration pinout.

Notice the pins marked in green. These pins are active connections for the project.

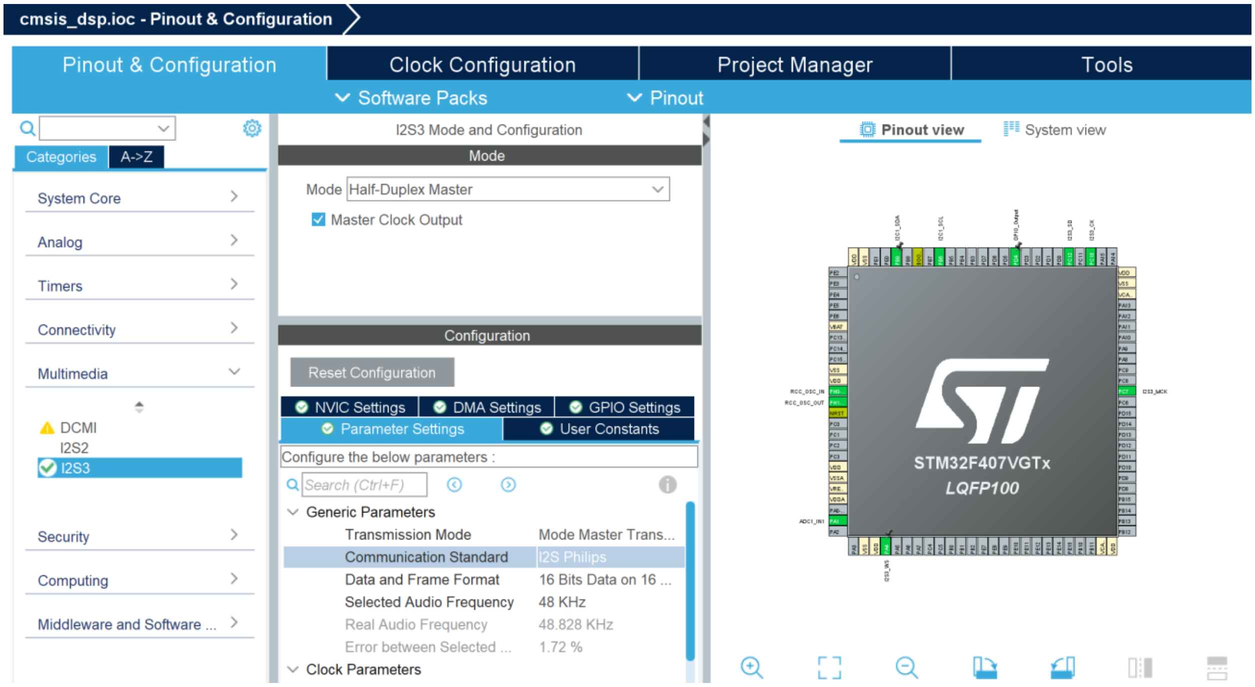
Find the ADC pin, and verify that it is assigned to pin PA1. This means that input voltages must be connected between pin PA1 and ground GND on the Discovery board. (Many other ADC input pins are available.)

You will also notice pins for I^{2}C. These pins are used to set up the DAC which is responsible for sending audio output to the headphone on the Discovery board. A set of I^{2}S pins are used to transmit the actual audio output data.

The CS43L22 DAC sends its output signal to the headphone jack on the Discovery board, as may be confirmed on the circuit schematic. The I2S3 inputs to the DAC can be seen at the left. The pin numbers on the schematic correspond to the pins on the microcontroller pinout in STM32CubeIDE.



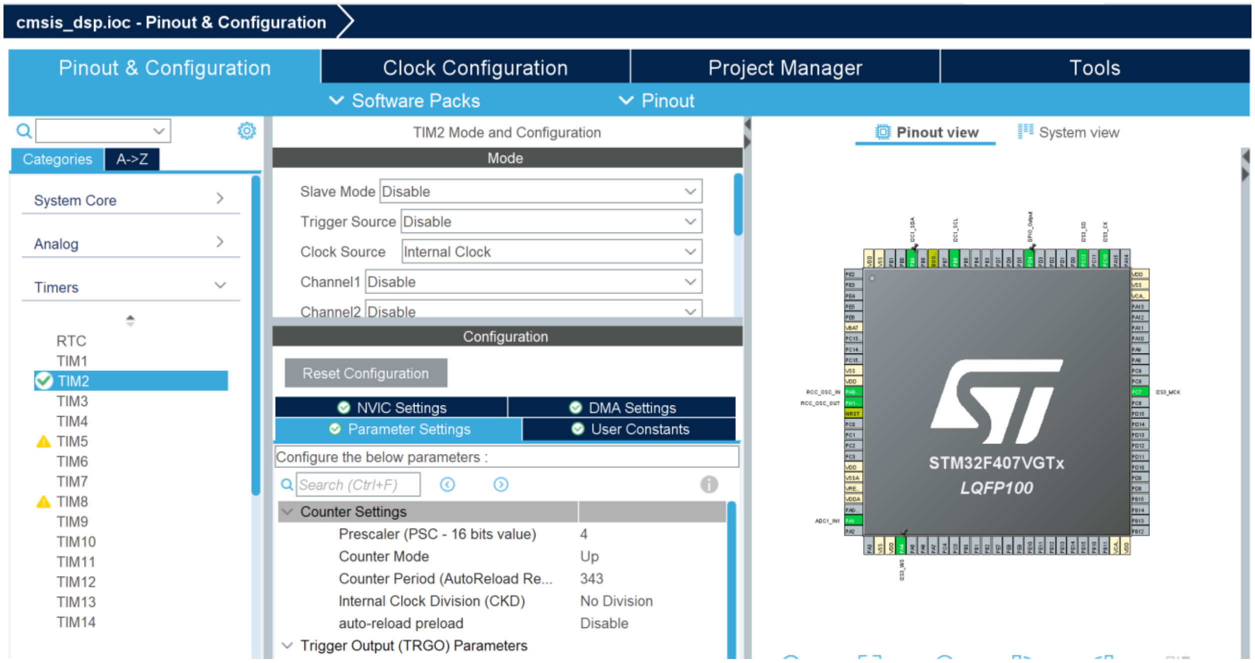
In STM32CubeIDE, under Multimedia, click on I2S3 (already checked off). Among its Parameter Settings, an audio frequency of 48 kHz is selected for the audio output data to the headphone. Notice that the Real Audio Frequency is 48.828 kHz, not exactly 48 kHz. Your project offers the closest available sampling frequency using the available clock frequency and clock divisors. Appendix 1 suggests options for obtaining a real sampling frequency that is closer to 48 kHz, if desired.



Click on Timers and TIM2. This timer controls the sampling frequency for the ADC input. The goal is to match the frequency of the I2S output of 48.828 kHz. The timer sampling frequency is determined by:

$$\text{sampling frequency} = \frac{\text{APB2 peripheral clock frequency}}{(\text{Prescaler} + 1)(\text{Counter Period} + 1)}$$

In this project, the APB2 peripheral clock frequency is 84 MHz (verify this on the Clock Configuration tab). With a Prescaler of 4 and a Counter Period of 343, the expected sampling frequency is 48.837 kHz, very close to the I2S sampling frequency.



In the Project Explorer, open the `Core/Src` folder and double-click on `main.c` to open it.

Much of the initialization code in `main.c` is produced automatically from the configuration file. Any parts of the code that lie in `USER CODE` areas are those produced by the user. Any user code that is entered outside a `USER CODE` area is removed when code is generated.

Connecting inputs and outputs to the Discovery board

To provide input to the Discovery board, you will use an online tone generator such as:

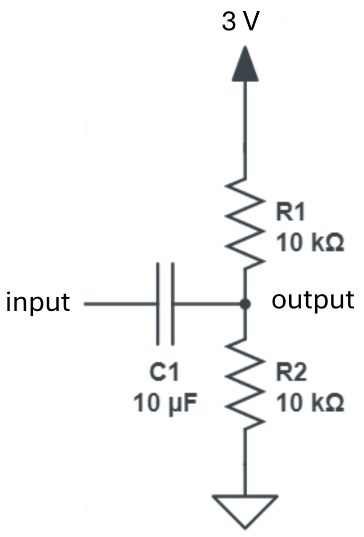
https://onlinetonegenerator.com/multiple-tone-generator.html

Select a frequency and click On/Off to hear the tone.

Frequency	Waveform	Volume	Panning (Left/ Right)	On/ Off	
453	Sine				Remove

Connect a stereo mini to flying leads cable to your PC headphone jack. The black jumper is ground and should be connected to a GND pin on the Discovery board. The other two jumpers are left and right stereo channels. Connect either one to pin PA1, which is the ADC input to the board.

The analog-to-digital converter on the Discovery board expects non-negative voltage inputs between 0 and 3 V. These voltages are mapped to 12-bit values from 0 to 4095. The audio from your PC has a zero mean voltage, and the negative-going voltages will be chopped off by the ADC. The workshop exercises will still work reasonably well despite this, but in general audio from the PC should be applied to a circuit that shifts the DC bias from 0 V to 1.5 V. Such a circuit is illustrated here.



If a function generator is available, a 3 Vpp sine wave with a DC offset of 1.5 V, can also be used as input to the Discovery board, connected between pins PA1 and GND.

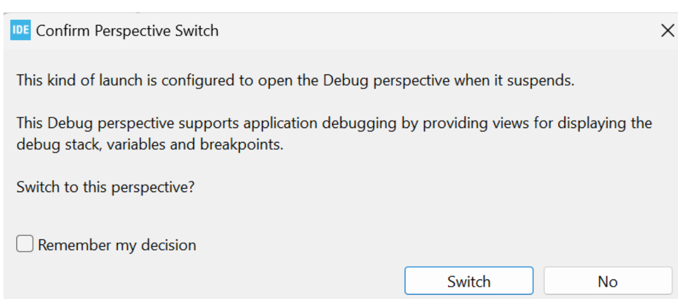
Connect a headphone to the headphone jack of the Discovery board.

In STM32CubeIDE, locate the project `cmsis_dsp` in the Project Explorer and open `main.c` from `Core/Src`. Ensure that `TALK_THROUGH` is selected.

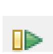
```
#define TALK_THROUGH 1
#define FIR_FILTER 0
#define CALC_FFT 0
```

Click Run – Debug (or click on the picture of a bug ).

Agree to Perspective Switch.



Once the download has been verified successfully, click Play/Resume:

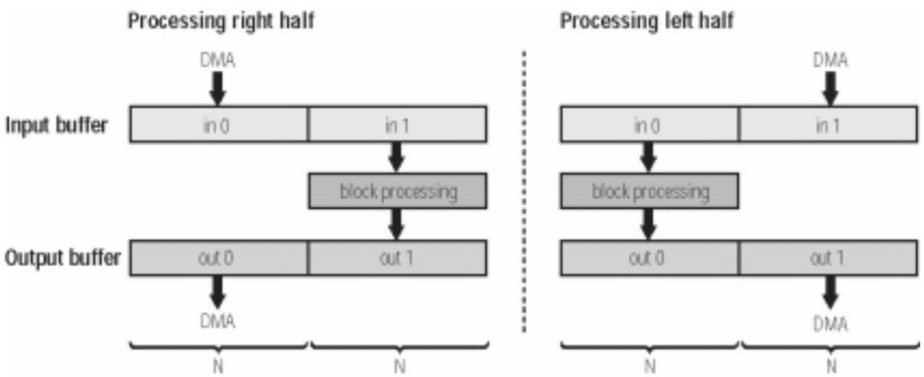


You should hear the sine wave in your headphones. Vary the frequency of the sine wave on your PC (or function generator) and hear the result, from 0 to 5 kHz.

Stop by clicking the red square.

Overall program structure

Direct Memory Access (DMA) is used for both input and output in the `cmsis_dsp` project. Using DMA means that accepting inputs and sending outputs is offloaded from the main processor and handled by a DMA controller. Input and output DMA buffers are defined. While one half of a DMA input buffer is accepting input samples, the other half is being processed; while one half of the DMA output buffer is filling up with processed samples, the other half is being transmitted.



To implement this, the project contains four "callback" functions, which each set a flag with a buffer is half or completely full:

```
void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef *hadc)
{
//    // first half of buffer is full
      adc_callback_state = 1;
}

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
      // second half of buffer is full
      adc_callback_state = 2;
}

void HAL_I2S_TxHalfCpltCallback(I2S_HandleTypeDef *hi2s)
{
      i2s_callback_state = 1;
}

void HAL_I2S_TxCpltCallback(I2S_HandleTypeDef *hi2s)
{
      i2s_callback_state = 2;
}
```

The ADC callback flags are used in the `while(1)` loop. Whenever half of the ADC DMA buffer is full, its values are copied and a pointer to the values is prepared for `process_DSP`:

```
        if(adc_callback_state != 0){

              if (adc_callback_state == 1) { //buffer half full
                    for(int i = 0; i < N; i++){
                          input_buffer[i] = (float32_t) adc_val[i];
                    }

                    input_buffer_ptr = &input_buffer[0];
                    output_buffer_ptr = &output_buffer[0];

              }
              else if (adc_callback_state == 2) { //buffer full
                    for(int i = N; i < 2*N; i++){
                          input_buffer[i]= (float32_t) adc_val[i];
                    }

                    input_buffer_ptr = &input_buffer[N];
                    output_buffer_ptr = &output_buffer[N];

              }
              process_DSP();
              adc_callback_state = 0;

        }
```

The I^2S protocol is used to send audio outputs to the headphone jack using DMA. The two halves of the I^2S DMA buffer are transmitted out in turn. The `audio_buffer_out` array has 4N positions, twice as many as for other DMA buffers, to account for left and right stereo channels.

```
        if(i2s_callback_state != 0){
              if(i2s_callback_state == 1){

                    for(int i = 0; i < N; i++){
                              audio_buffer_out[2*i] = (uint16_t)
output_buffer[i];
                              audio_buffer_out[2*i+1] = (uint16_t)
output_buffer[i];
                    }
              }
              else if(i2s_callback_state == 2) {

                    for(int i = N; i < 2*N; i++){
                              audio_buffer_out[2*i] = (uint16_t)
output_buffer[i];
```

```
                                audio_buffer_out[2*i+1] = (uint16_t)
output_buffer[i];
                          }
                }
                i2s_callback_state = 0;

          }
```

Filtering your input signal

Digital filters are defined by lists of numbers call filter coefficients. These filter coefficients determine how a filter will behave – low pass, high pass, band pass, band stop. In this workshop, you will use an FIR (finite impulse response) filter. For this kind of filter, the filter output $y[n]$ is computed from filter inputs using a difference equation.

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + \cdots + b_Mx[n-M]$$

where n is the current sample number, the b_k values are the filter coefficients, and $x[n-k]$ refers to the input sample k steps in the past.

In `main.c`, you will see a list of filter coefficients `filter_coeffs`. The function `arm_fir_init_f32` initializes the filter using `filter_coeffs`.

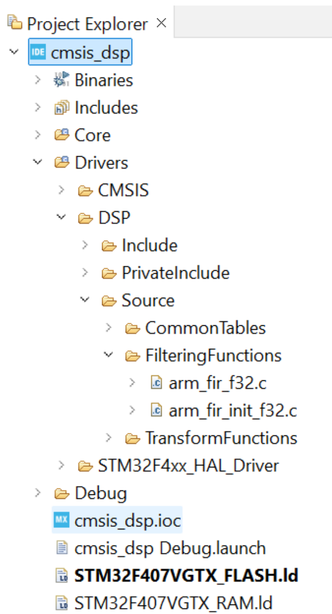
Near the top of `main.c`, select `FIR_FILTER`.

```
#define TALK_THROUGH 0
#define FIR_FILTER 1
#define CALC_FFT 0
```

In the `process_DSP` function, the filter code will now be active. The CMSIS-DSP library uses "block processing," which means that it filters a group of N input samples at once. In your project, N is defined to be 1024 near the top of `main.c`.

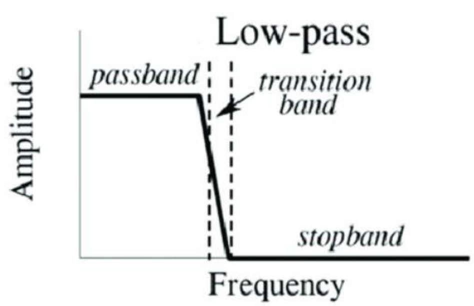
```
void process_DSP(){
    #if TALK_THROUGH
        for(int i = 0; i < N; i++){
            *(output_buffer_ptr + i)= *(input_buffer_ptr + i);
        }

    #elif FIR_FILTER
        arm_fir_f32(&lpfilter, input_buffer_ptr, output_buffer_ptr, N);

    #elif CALC_FFT
        for(int i = 0; i < N; i++){
            *(output_buffer_ptr + i)= *(input_buffer_ptr + i);
        }
        //  your code goes here

    #endif
}
```

The files needed to initialize and run the filter are in the `Drivers/DSP/Source/FilteringFunctions` folder of your project. Appendix 2 describes where to obtain the required code from the ST32Cube Repository.



Run – Debug your project, and when download is successful, click Play/Resume. Ensure your online tone generator is running.

The filter provided in the `cmsis_dsp` project is a low pass filter, meaning lower frequencies pass through the filter and higher frequencies are blocked. The cutoff frequency for the filter is about 3 kHz. Select sine wave frequencies to test your filter. Frequencies below 3 kHz should be clearly audible; frequencies above should be more attenuated. This behaviour is quite different from your previous talk-through experiment.



Implement a filter using MATLAB

This is the MATLAB code that produced the coefficients originally in the `cmsis_dsp` project.

```
b=fir1(60,0.125);
id=fopen('coeffs.txt','w');
for i = 1:length(b)-1
    fprintf(id,'%f,  ',b(i));
    if(mod(i,5) == 0)
```

```
        fprintf(id,"\n");
    end
end
fprintf(id, '%f ',b(length(b)));
fclose(id);
```

The first line designs the filter and puts the coefficients in the variable `b`, and the rest of the lines print the coefficients in a convenient way. The command:

```
b=fir1(num_coeffs-1, cutoff_ratio);
```

produces `num_coeffs` coefficients for a low pass filter, so `fir1(60,0.125)` creates 61 coefficients. The cut-off frequency for the filter equals the cut-off ratio times half the sampling frequency so a cut-off ratio of 0.125 for a sampling frequency of 48 kHz gives a cut-off frequency of (0.125)(48kHz/2) = 3 kHz.

Design your own filter in MATLAB. Tips:

(1) Do not exceed 80 coefficients, because very long filters may take too long to compute for the lengths of DMA buffers in use.
(2) Choose a cut-off ratio that will give you a cut-off frequency you can easily hear, perhaps between 200 Hz and 4 kHz.
(3) If you wish to create a high-pass filter, the syntax is:

```
b=fir1(num_coeffs-1, cutoff_ratio,'high');
```

In this case, the higher frequencies pass through the filter and the lower frequencies are attenuated.

For a bandpass filter, two ratios must be provided, one for each edge of the pass band. Frequencies within the band pass, and frequencies outside the band are attentuated:

```
b = fir1(num-coeffs-1, [edge1_ratio edge2_ratio]);
```

Design your filter. To look at its shape in Matlab, type:

```
freqz(b,1);
```

The magnitude response in the top half of the plot shows the gains of the filter in dB, across frequency. Along the horizontal axis, you will see “normalized frequencies” from 0 to 1. Multiplying a normalized frequency by $f_S/2$ Hz gives the equivalent frequency in Hz.

In MATLAB, type the command `pwd` to learn the present working directory. Using your File Explorer, navigate to this directory and open the file `coeffs.txt` using a text editor. Copy the coefficients and paste them into `main.c` in place of the original coefficients.

Edit the number of filter coefficients to reflect your design. Remember that the number of filter coefficients is one greater than the value you used in your call to `fir1` in Matlab.

```
#define FILTER_NUM_COEFFS 61
```

Run – Debug and Play/Resume. Test your filter with tones from your PC's online tone generator. Verify that the filter output signal is present and absent where you expect.

What is an FFT?

FFT stands for Fast Fourier Transform. This is an exceedingly important tool in engineering. Supplied with samples of a digital signal, the FFT produces the spectrum of the signal, meaning the frequency composition of the signal.

The FFT accepts as input N samples of a signal, and produce N complex-valued outputs. The magnitudes of these complex number outputs are used to construct the magnitude spectrum of a signal, while the phases of these outputs are used to construct the phase spectrum of the signal. The magnitude spectrum is usually of greatest interest, and is the spectrum displayed on oscilloscopes.

For example, the magnitude spectrum and the phase spectrum for a 9600 Hz sine wave sampled at 48 kHz are shown below. A total of 1024 samples of the sine wave are used as input to the FFT, and as a result the FFT produces 1024 complex outputs. These are numbered as FFT index values from 0 to 1023. An FFT index can be mapped to a frequency f in Hz using the equation:

$$\mathrm{f} = \mathrm{k}\frac{\mathrm{f_S}}{\mathrm{N}}$$

where k is the FFT index (running from 0 to N-1), f_S is the sampling frequency, and N is the number of points.

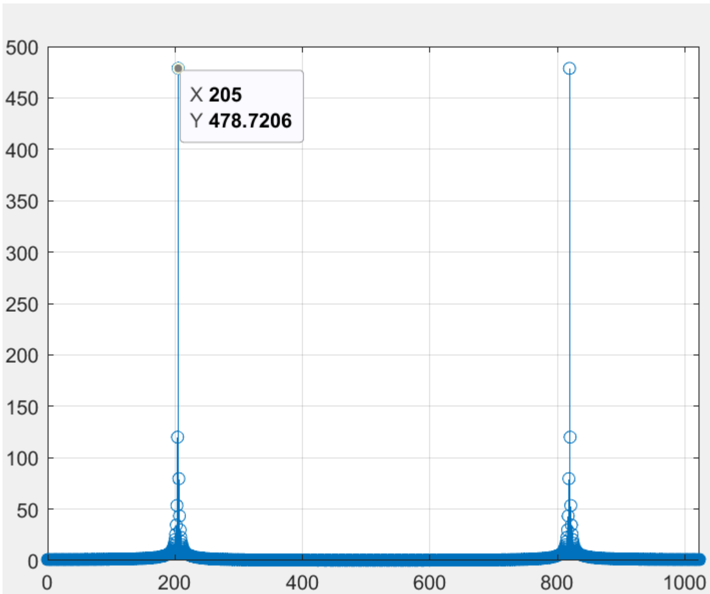
Notice that the magnitude spectrum contains two sharp peaks. This is somewhat unexpected, since a sine wave is known to contain a single frequency only. The second peak is what is called an image, and there an infinite number of other images in the magnitude spectrum, which could be seen if the plot were widened. The portion of the magnitude spectrum that is useful to look at is the region for k = 0 to k = N/2, because at FFT index k = N/2, the frequency is k $\mathrm{f_S}$/N = (N/2) ($\mathrm{f_S}$/N) = $\mathrm{f_S}$/2, which is the Nyquist frequency. If you are not familiar with sampling theory, the Nyquist theorem states that you must sample at least twice the maximum frequency in your signal. This means that the maximum frequency in your signal cannot exceed half the sampling frequency, so for a sampling frequency of 48 kHz, the maximum relevant frequency is 24 kHz, which is located at k = 512 on the magnitude spectrum, at the halfway point. Thus, there is only one true peak in the magnitude, consistent with the sine wave input.

A second thing to notice about the magnitude spectrum is the location of the first peak. It occurs at FFT index 205. This gives:

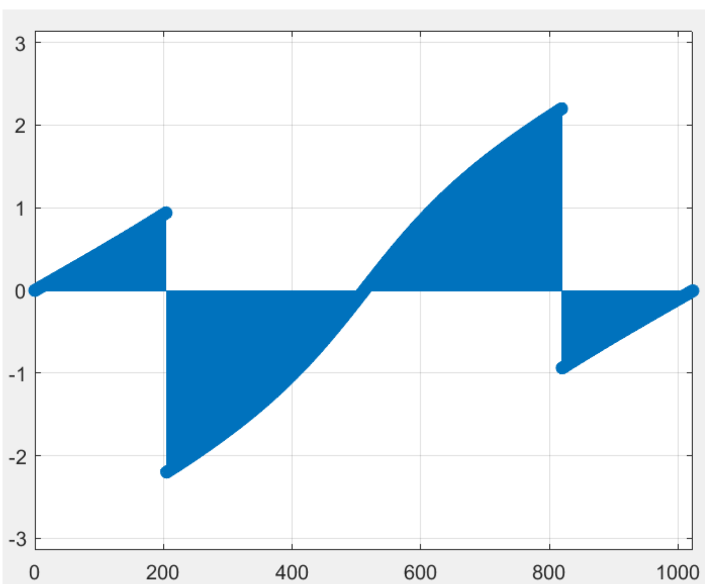
$$\mathrm{f} = \mathrm{k}\frac{\mathrm{f_S}}{\mathrm{N}} = 205\,\frac{48000}{1024} = 9609.375\ \text{Hz}$$

This value does not exactly match the known frequency of 9600 Hz, but is the closest frequency possible given that the FFT can only report for integer values of k.

Magnitude spectrum (FFT magnitude vs FFT index)

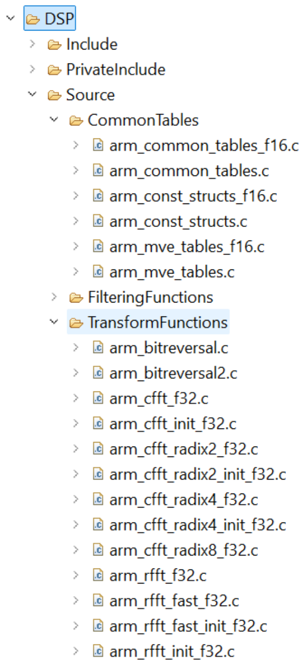


Phase spectrum (FFT phase vs FFT index)



Calculate FFT magnitudes for a signal

Using the CMSIS-DSP FFT functions needs additional CMSIS DSP folders (CommonTables and TransformFunctions). These have already been added to the `cmsis_dsp` project. Appendix 2 explains where to find the necessary code.



At the start are declarations for an instance of an FFT object called `fft`, an array for complex FFT outputs called `fft_out`, and an array for magnitudes `mag`. These declarations are entered within the `USER CODE` Private Variables area, after the declaration of the filter coefficients.

```
arm_rfft_fast_instance_f32 fft;
float32_t fft_out[2*N];
float32_t mag[N];
```

In `USER CODE` area 2, the FFT object is initialized:

```
arm_rfft_fast_init_f32(&fft, N);
```

In the `process_DSP` function, the first few lines for the `CALC_FFT` option send the input signal straight to the headphone output, so you can confirm by listening that signal is present. Then an FFT is computed for an N-sized block of input and FFT magnitudes are calculated for FFT index values from 0 to 512. In general, the magnitude of a complex number $\mathrm{Re} + \mathrm{jIm}$ would be calculated as $\sqrt{\mathrm{Re}^2 + \mathrm{Im}^2}$. To understand code, it is important to know that, in the array `fft_out`, real and imaginary parts of the complex number outputs alternate. The only exceptions are the first two locations in the FFT output buffer, which as described at [CMSIS-DSP: Real FFT Functions](), hold the FFT outputs for DC and the Nyquist frequency.

```
#elif CALC_FFT
	for(int i = 0; i < N; i++){
		*(output_buffer_ptr + i)= *(input_buffer_ptr + i);
	}
```

```
	arm_rfft_fast_f32(&fft, input_buffer_ptr, fft_out, 0);
	mag[0] = fft_out[0];

	mag[N/2] = fft_out[1];

	for(int i=1; i < N/2; i++){

		mag[i] = sqrt(pow(fft_out[2*i],2) + pow(fft_out[2*i+1],2));
	}
#endif
```

Finally, select `CALC_FFT` near the top of `main.c`.

```
#define TALK_THROUGH 0
#define FIR_FILTER 0
#define CALC_FFT 1
```

Set your online tone generator to generate a sine wave. For the frequency you choose, calculate the FFT index where you expect the biggest magnitude by using the equation that gives the FFT frequencies:

$$\mathrm{f} = \mathrm{k}\frac{\mathrm{f_S}}{\mathrm{N}}$$

At this point, it is important to recall that the real sampling frequency in use is 48.828 kHz. To calculate where you expect a spike for a 440 Hz sine wave input, solve with $\mathrm{f_S}$ = 48828 and N = 1024 to give:

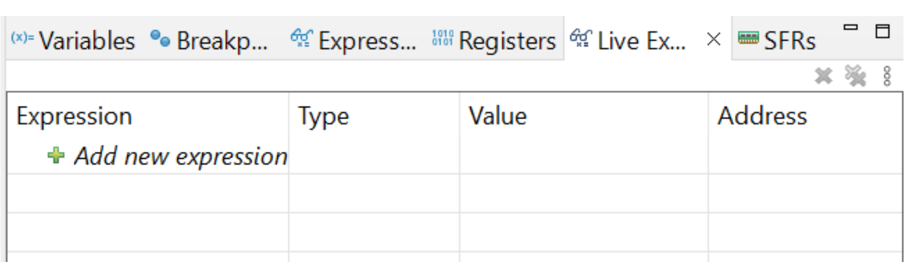
$$440 = k\frac{48828}{1024}$$

$$k = 9.2$$

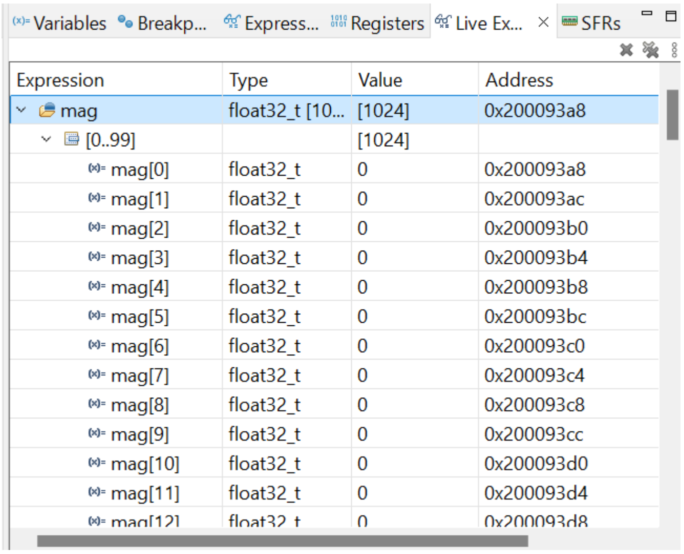
This means the highest peak in the magnitude spectrum will be at the closest integer, 9.

Run – Debug your project.

In Debug mode, find the Live Expressions tab in the window at the upper right.



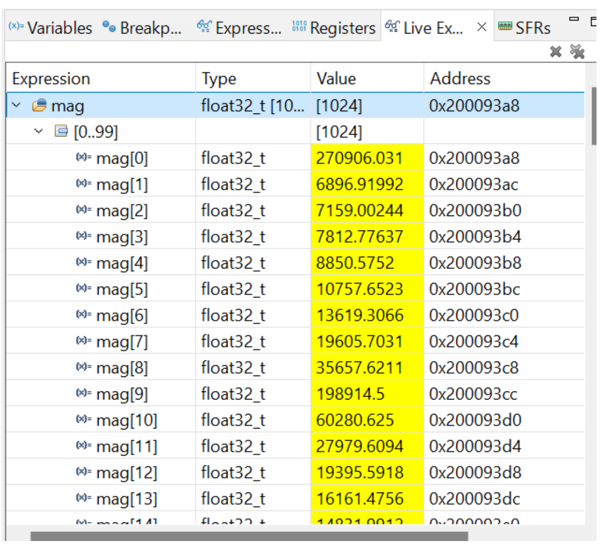
Type `mag`, the name of the array containing the FFT magnitudes, in the Expression column and type Enter. Click beside the arrow next to `mag`. Click beside the arrow next to [0..99]. It may take a moment for the array to open. Live expressions can be used for any global variable in your project.



Variables Breakp... Express... Registers Live Ex... SFRs

Expression	Type	Value	Address
mag	float32_t [10...	[1024]	0x200093a8
[0..99]		[1024]	
mag[0]	float32_t	0	0x200093a8
mag[1]	float32_t	0	0x200093ac
mag[2]	float32_t	0	0x200093b0
mag[3]	float32_t	0	0x200093b4
mag[4]	float32_t	0	0x200093b8
mag[5]	float32_t	0	0x200093bc
mag[6]	float32_t	0	0x200093c0
mag[7]	float32_t	0	0x200093c4
mag[8]	float32_t	0	0x200093c8
mag[9]	float32_t	0	0x200093cc
mag[10]	float32_t	0	0x200093d0
mag[11]	float32_t	0	0x200093d4
mag[12]	float32_t	0	0x200093d8

Click Play/Resume. You should see numbers populate the `mag` array in the Live Expressions window. Click Pause . Search for the FFT index that you calculated for the frequency of your sine wave. Is the FFT magnitude especially high for this FFT index, in comparison to its neighbours?



Variables Breakp... Express... Registers Live Ex... SFRs

Expression	Type	Value	Address
mag	float32_t [10...	[1024]	0x200093a8
[0..99]		[1024]	
mag[0]	float32_t	270906.031	0x200093a8
mag[1]	float32_t	6896.91992	0x200093ac
mag[2]	float32_t	7159.00244	0x200093b0
mag[3]	float32_t	7812.77637	0x200093b4
mag[4]	float32_t	8850.5752	0x200093b8
mag[5]	float32_t	10757.6523	0x200093bc
mag[6]	float32_t	13619.3066	0x200093c0
mag[7]	float32_t	19605.7031	0x200093c4
mag[8]	float32_t	35657.6211	0x200093c8
mag[9]	float32_t	198914.5	0x200093cc
mag[10]	float32_t	60280.625	0x200093d0
mag[11]	float32_t	27979.6094	0x200093d4
mag[12]	float32_t	19395.5918	0x200093d8
mag[13]	float32_t	16161.4756	0x200093dc

Click Play/Resume. Change the frequency of your tone from the online generator and verify that the location of the large magnitude has shifted to the expected new location.

At the online tone generator, you can add a tone so multiple tones play at once. Experiment if you like. There should be multiple FFT index values with large magnitudes for multiple tones.

Besides Live Expressions, STM32CubeIDE offers a multitude of debugging features such as breakpoints, stepping, and code completion. With some small modifications, STM32 projects can also use `printf` functions.

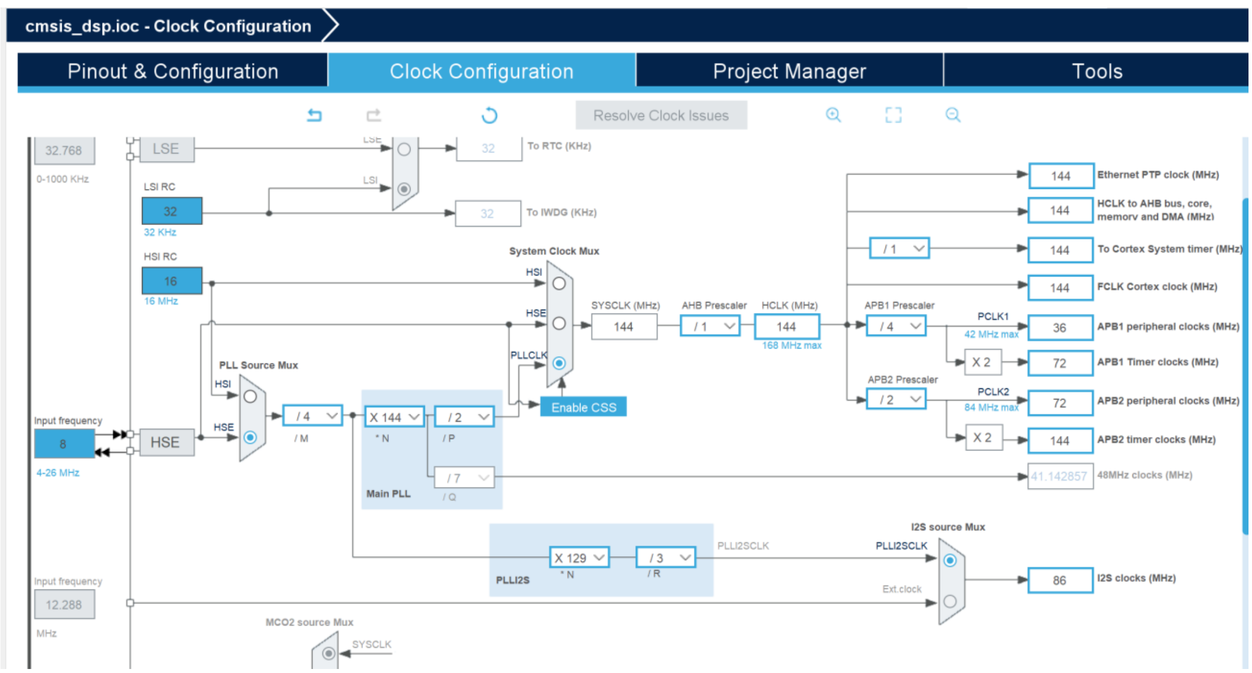
Appendix 1 Setting sampling frequency for I2S interfaces

Reference manual: [STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCUs - Reference manual]

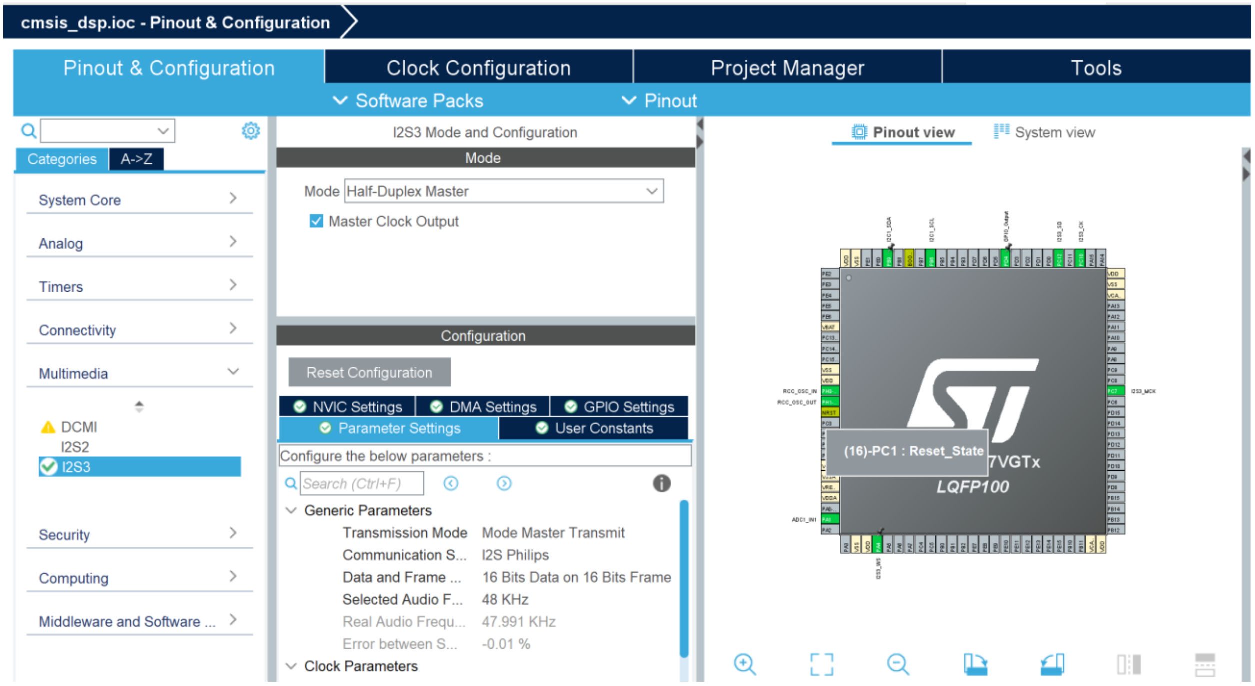
Section 28.4.4 of the reference manual for the STM32F407 provides details for I2S sampling frequency calculation. The most important thing to know is that your project will choose clock divisors to get as close as possible to the sampling frequency you have selected.

In this project, a real sampling frequency of 48.828 kHz was available when a sampling frequency of 48 kHz was selected. This has repercussions for interpreting FFT results.

It is possible to make changes on the clock configuration tab to get closer to 48 kHz. One such solution is shown.

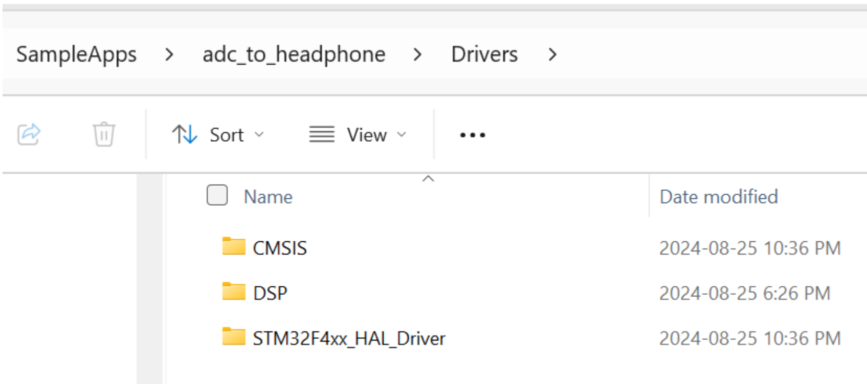


With the above clock settings, the real audio frequency becomes 47.991 kHz.


cmsis_dsp.ioc - Pinout & Configuration
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Software Packs
Pinout
Categories
A->Z
System Core
Analog
Timers
Connectivity
Multimedia
DCMI
I2S2
I2S3
Security
Computing
Middleware and Software ...
I2S3 Mode and Configuration
Mode
Mode Half-Duplex Master
Master Clock Output
Configuration
Reset Configuration
NVIC Settings
DMA Settings
GPIO Settings
Parameter Settings
User Constants
Configure the below parameters :
Search (Ctrl+F)
Generic Parameters
Transmission Mode Mode Master Transmit
Communication S... I2S Philips
Data and Frame ... 16 Bits Data on 16 Bits Frame
Selected Audio F... 48 KHz
Real Audio Frequ... 47.991 KHz
Error between S... -0.01 %
Clock Parameters
Pinout view
System view
(16)-PC1 : Reset_State
7VGTx
LQFP100

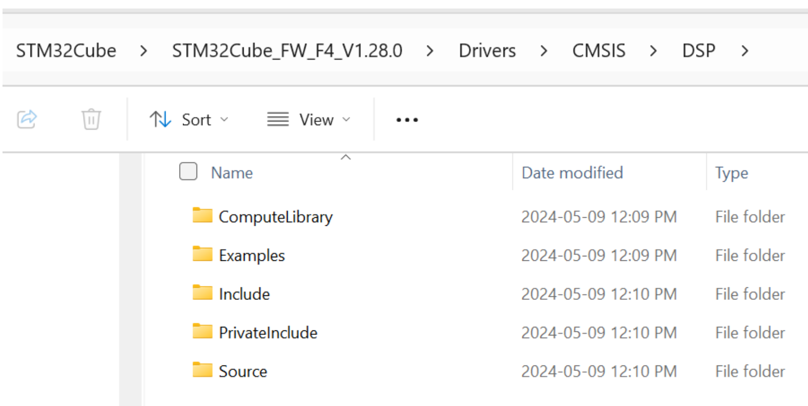
Appendix 2 Adding CMSIS DSP functions to your project

Go to your File Explorer. In the `Drivers` folder of your project, create a folder called `DSP`.



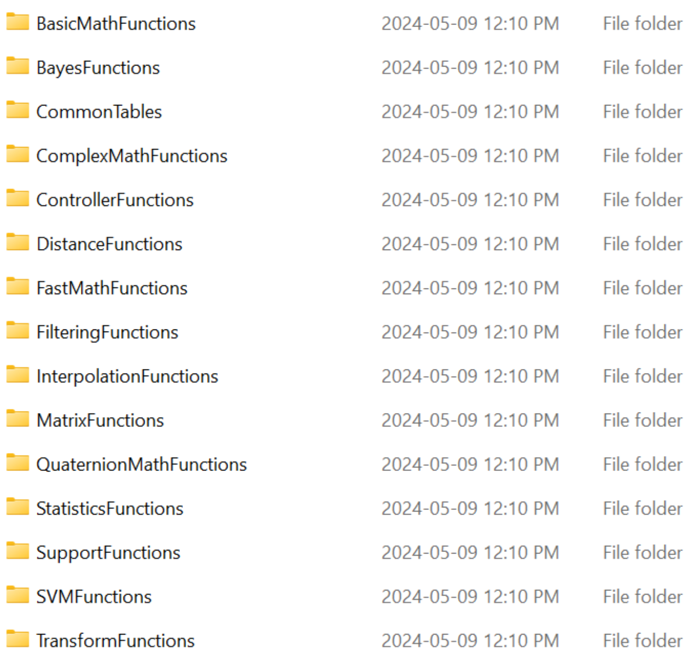
Go to `C:\Users\your_user_name\STM32Cube\Repository\STM32Cube_FW_F4_V1.28.0\Drivers\CMSIS\DSP`

This directory contains the folders shown:

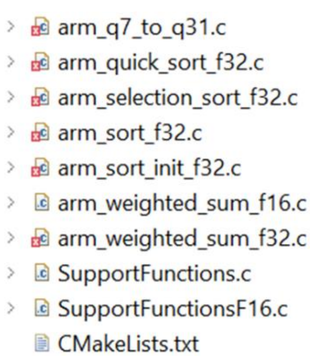


From this directory, copy the `Include`, `PrivateInclude`, and `Source` folders into the `DSP` folder you created.

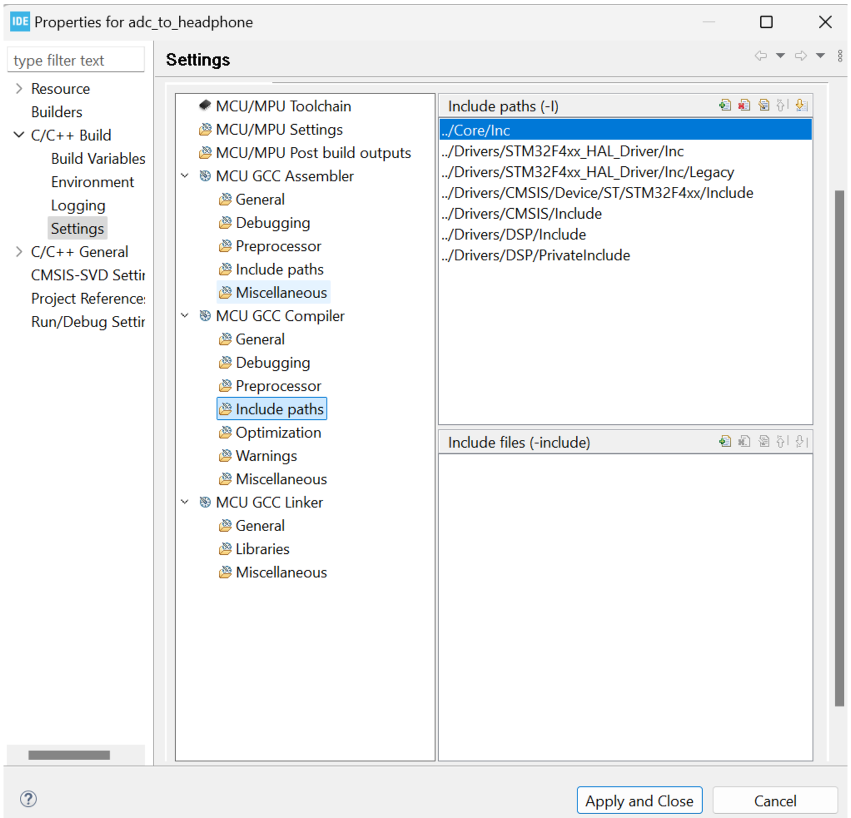
Open the `C:\DSP Concepts\AWECore ST_EVAL_CortexM4 Release-8.D.2.7\SampleApps\cmsis_dsp\Drivers\DSP\Source` directory. Delete any folders that your project does not need. For the remaining folders, delete any functions that your project does not need.



One at a time, open each folder that remains in the `DSP/Source` folder your project. Each folder contains a file called `<foldername>.c`, and many also include a file called `<foldername>F16.c`. For example in `DSP/Source/SupportFunctions`, the last two files are `SupportFunctions.c` and `SupportFunctionsF16.c`. Delete these two functions to avoid multiple declaration errors. Do the same for each folder in the `DSP/Source` directory.



Go to STM32CubeIDE. In the Project Explorer, right-click on your project name and click on Properties. In C/C++ Build – Settings – MCU GCC Compiler – Include paths, click on the final entry, and then click the green plus. Add the path `../Drivers/DSP/Include`. Repeat to include the path `../Drivers/DSP/PrivateInclude`. Click Apply and Close.



In your Project Explorer, go to `Core/Src` and double-click on `main.c` to open it. To use the CMSIS DSP code, you must add **`#include`** `"arm_math.h"` to the `USER CODE` Private includes area.

```
/* Private includes ------------
/* USER CODE BEGIN Includes */
#include "CS43L22.h"
#include <math.h>
#include "arm_math.h"
#include <stdio.h>
#include <stdlib.h>
/* USER CODE END Includes */
```