Mini STM32 Speaker

(with DSP)

**Overview**

Earlier this year I started a project converting the LEGO guitar amplifier of set no. 21329 into a functional, portable speaker for my desk. The project involved minor modifications to the set to support features such as a power switch, power indication LED, volume control knob, auxiliary input, and an output speaker, as well as a custom-designed PCB powered by USB-C.



Source: Author

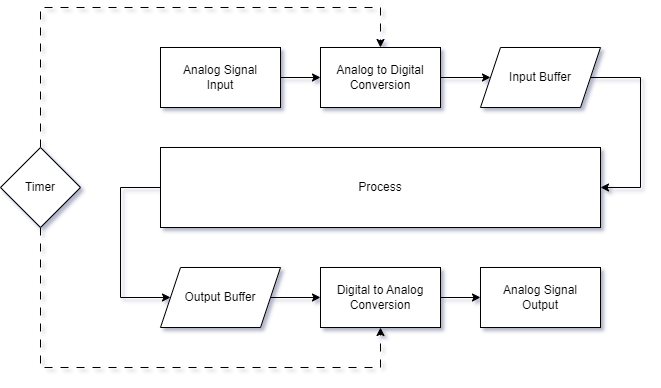
In an effort to extend its flexibility and longevity as an audio device, I have begun the process of integrating an STM32F-series microcontroller into the project. Converting an analog signal to digital provides the ability to program filters and effects long after the hardware has been manufactured.

I have chosen to use an STM32F-series microcontroller for its cost, availability of resources, and performance – including an integrated floating-point unit (FPU) for precise DSP calculations which can be made using the CMSIS DSP library.

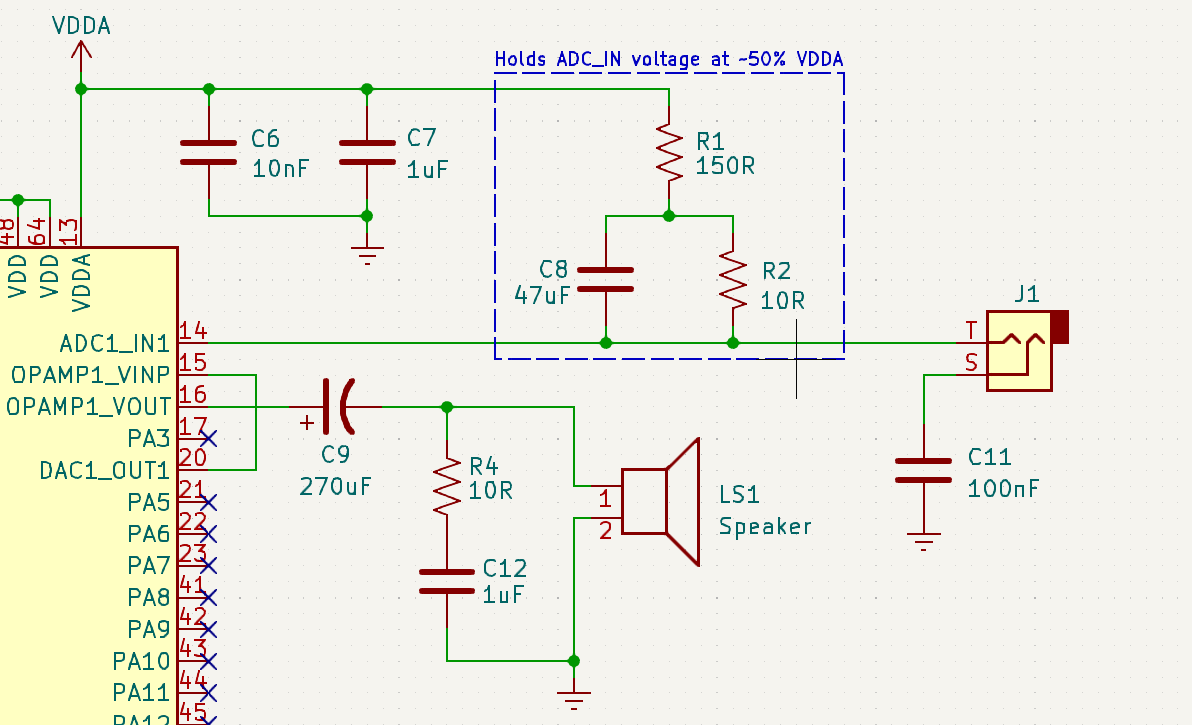
**Design**

A brief outline of the intended approach is modeled in the diagram below:

First, the analog input signal needs to be acquired and converted from a continuous stream to discrete values. The discrete values can then be modified within the program before being converted back into a continuous analog stream. The conversions can be triggered by the same timing source to help ensure conversion integrity and consistency.

The input signal will be acquired using a mono (TS) auxiliary jack, which will be biased to ~50% of the analog voltage to facilitate complete signal acquisition. The signal will also be filtered of extreme frequencies before it is registered by the A/D converter to prevent aliasing. The signal will then be processed as desired using an FIR filter to further attenuate unwanted frequencies. Additionally, other effects can be applied, but care must be taken not to overload the limited resources of the microcontroller. Passive components should be placed strategically to ensure consistent delivery of the processed signal after D/A conversion. To further increase the signal strength and integrity, an integrated OpAmp will be utilized to drive the output.

Source: Author

Pictured below is the analog circuit as I have described:

Source: Author

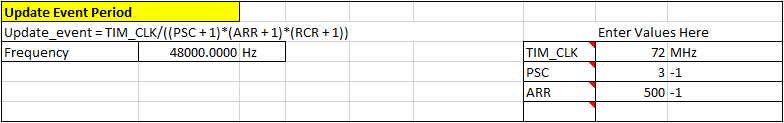
The signal is acquired at Pin 14 via the mono audio jack. The circuit, outlined in blue, pulls the input signal up to ~50% VDDA to help ensure complete and accurate sampling of positive and negative components of the signal. C6 and C7 are decoupling capacitors to reduce VDDA noise. Pin 20 is the D/A converter output which is shorted to the OpAmp input at Pin 15. Pin 16 is the OpAmp output, which drives through a polarized aluminum electrolytic capacitor on its way to the output speaker. R4 and C12 provide additional signal filtering and stabilization.

**STM32**

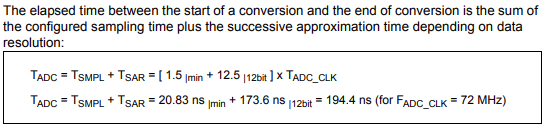
ST provides some great software solutions for development, and I like to use most of them. CubeMX provides an intuitive experience and supports in-depth customization for STM32 microcontrollers. Because of its portability, the HAL (Hardware Abstraction Layer) library will be utilized.

**Timer Configuration**

As illustrated in the block diagram above, the A/D and D/A conversions should be triggered by the same clock source. Using the maximum clock frequency allowed by the STM32F303RE, 72MHz, the desired sampling rate of 48kHz can be achieved with a few simple calculations.

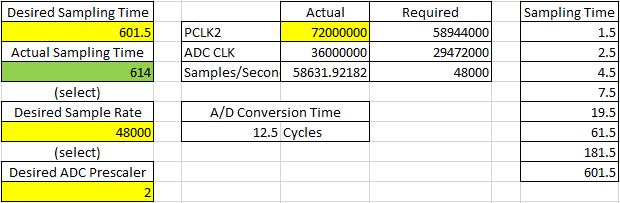
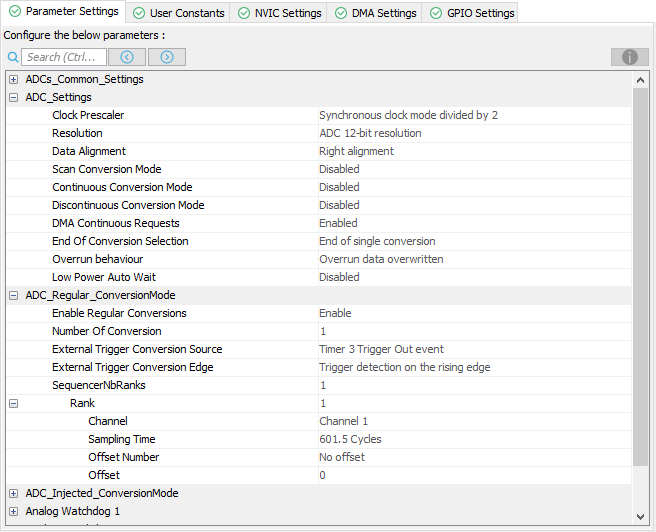
**A clock pre-scaler of 3 (-1 to account for zero value) with an auto-reload register value of 500-1 sets Timer 3 to ‘tick’ 48,000 times per second (48kHz), which is the desired sampling rate.

Source: Author

Source: STMicroelectronics

With global interrupts enabled, every ‘tick’ of Timer 3 triggers an ADC conversion, and each conversion takes 12.5 + x cycles to complete. To convert 48k samples per second with the highest possible resolution, the ADC is configured as defined below:

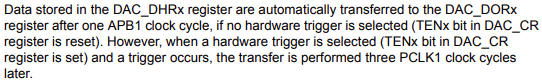
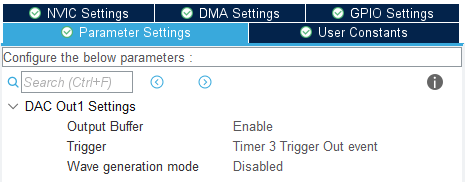
**ADC Configuration**

The ADC is oversampled by ~20%, with the extra samples being overrun, leaving ~48,000 samples per second to be modified and set to the output.

Source: Author

Source: Author

**DAC Configuration**

To synchronize with the A/D conversion, the D/A conversion will be triggered by Timer 3, which triggers at a rate of 48,000 times/sec. While each A/D conversion takes 12.5 + x (x = 601.5, in this case) cycles as outlined above, each D/A conversion only takes 3 cycles when triggered by hardware (Timer 3, in this case), leaving plenty of time to spare between each of the 48,000 conversions per second.

Source: Author

Source: STMicroelectronics

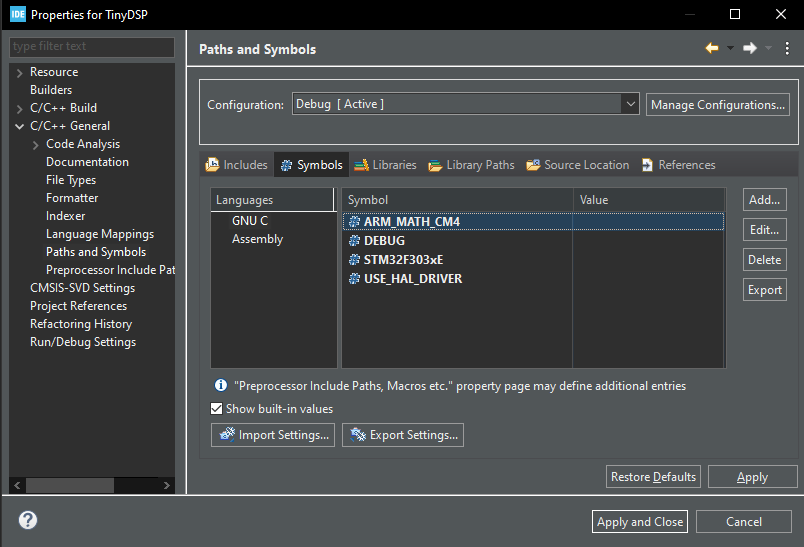
**DMA Configuration**

To reduce CPU load, Direct Memory Access (DMA) will be leveraged by the A/D and D/A conversion channels. Each will implement a circular buffer to support the continuous stream of data into and out of the system. The ADC and DAC both support up to 12-bit resolution, so the DMA data width doesn’t need to be any longer than a half word (16 bits). Though there aren’t many processes with which to compete, the priority should be set “Very High” for both channels to ensure timely delivery.

**OpAmp Configuration**

After the signal is processed, it will need to be amplified, if only a little bit. Utilizing an internal OpAmp in “Follower” eliminates the need for an external OpAmp, thereby effectively reducing the PCB footprint and associated cost.

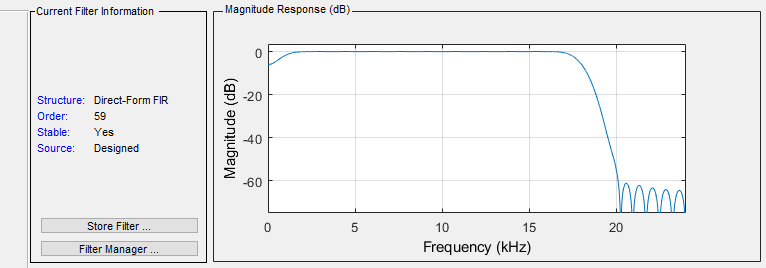
**DSP Library Configuration**

To use the supported CMSIS DSP library, it will be imported into the project using CubeMX’s Middle- and Software configuration menu. Though the library supports many Cortex cores, the STM32F303xE is a Cortex M-4 processor, so it should be defined within the project settings which set of instructions to leverage.

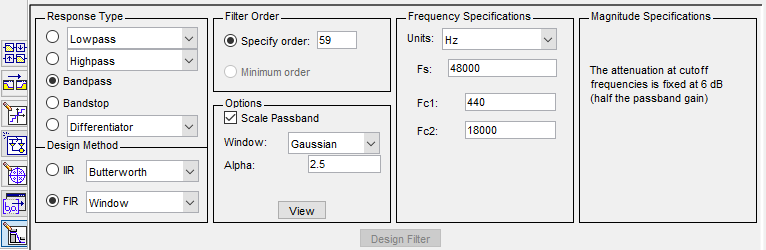
Source: Author

**FIR Filter**

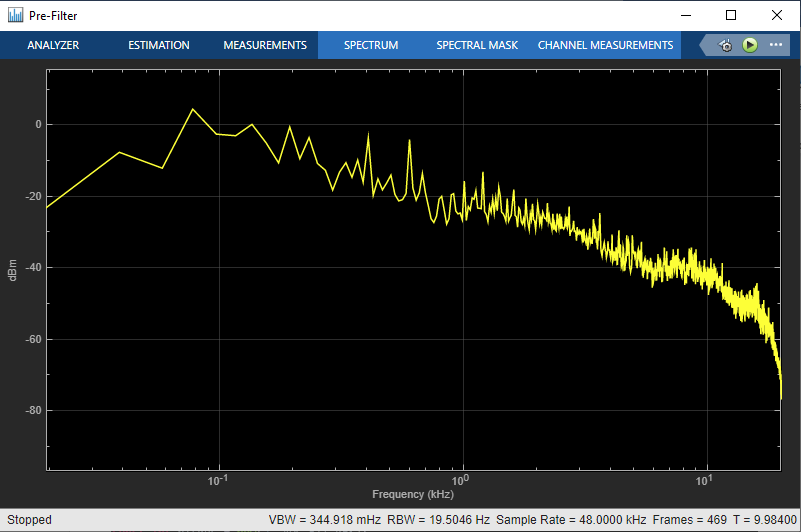
Before effects are applied, the input signal will be digitally filtered using a bandpass filter to attenuate frequencies beyond the desired range. The highest frequencies this application requires are no more than 20kHz, which is well below the Nyquist frequency of 24kHz (48kHz / 2), so aliasing will be avoided. The frequencies above 18kHz will be rolled off, and those below 440Hz will be slightly attenuated to reduce signal muddiness. To reduce computational demand (at the cost of filter resolution), the Order of the FIR filter will be kept as low as possible without sacrificing attenuation requirements. Designed in MATLAB’s Filter Designer, the filter is shown below:



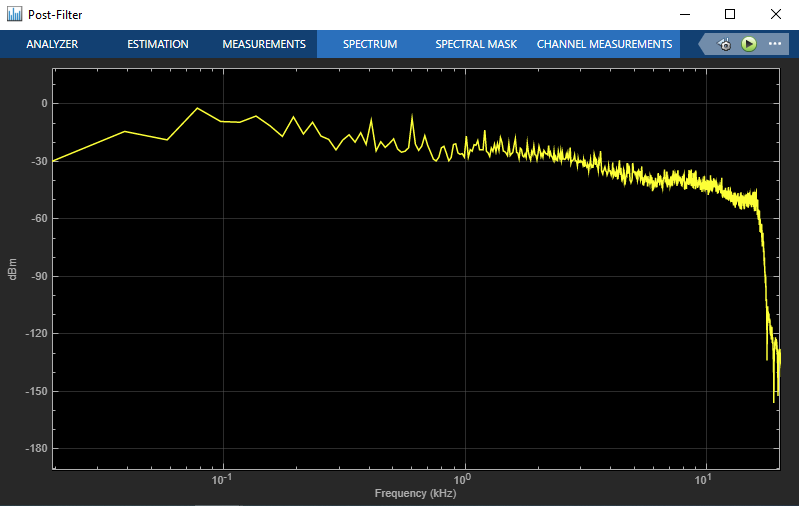
Source: Author

Source: Author

To verify that the filter will produce the intended results, a brief simulation is conducted within Simulink.



Above is the frequency response before being filtered, and below is the frequency response after being filtered. The frequencies above ~18kHz are rolled off sharply, and the low frequencies below 440Hz are attenuated by ~5dB. Looks good!



**Code**

First, important variables are defined using calculations made earlier in the project, which will be referenced throughout main.c:

**#define** **ADC\_1\_BITS** 12

**#define** **ADC\_2\_BITS** 8

**#define** **BLOCK\_SIZE** 1024

**#define** **FIR\_BLOCK\_SIZE** 128

**#define** **NUM\_TAPS** 60

ADC\_1\_BITS and ADC\_2\_BITS represent the bit resolution of each ADC, respectively. BLOCK\_SIZE and FIR\_BLOCK\_SIZE are the sizes of in/out buffers and FIR buffers, respectively. NUM\_TAPS is the number of taps as defined by the FIR filter designed using MATLAB – this should be updated relative to the filter being used.

**const** **int** adc1Res = **pow**(2, ADC\_1\_BITS);

**const** **int** adc2Res = **pow**(2, ADC\_2\_BITS);

**const** **int** numBlocks = BLOCK\_SIZE/FIR\_BLOCK\_SIZE;

**const** **int** dacAlignment = DAC\_ALIGN\_12B\_R;

**const** **int** dacChannel = DAC\_CHANNEL\_1;

**const** **float** gain = 2.0f;

// Coefficients for FIR filter

**static** **float32\_t** firTaps[NUM\_TAPS] =

{

-0.0002885172435,-0.001160182524,-5.817951387e-05,-0.001331895357,-0.001663194387,

-9.533889533e-05,-0.003539981088,-0.001330590458,-0.001582383644,-0.006341828499,

9.335886716e-05,-0.006326230243,-0.007433256134, 0.000799603411, -0.01474349387,

-0.003697932465,-0.003903942881, -0.02298442833, 0.0053132209, -0.0193928685,

-0.0218609143, 0.01309455186, -0.04651772976,0.0007144561969, 0.00369216525,

-0.07755808532, 0.06292685121, -0.0654515177, -0.09863256663, 0.5688271523,

0.5688271523, -0.09863256663, -0.0654515177, 0.06292685121, -0.07755808532,

0.00369216525,0.0007144561969, -0.04651772976, 0.01309455186, -0.0218609143,

-0.0193928685, 0.0053132209, -0.02298442833,-0.003903942881,-0.003697932465,

-0.01474349387, 0.000799603411,-0.007433256134,-0.006326230243,9.335886716e-05,

-0.006341828499,-0.001582383644,-0.001330590458,-0.003539981088,-9.533889533e-05,

-0.001663194387,-0.001331895357,-5.817951387e-05,-0.001160182524,-0.0002885172435

};

**arm\_fir\_instance\_f32** fir;

**float32\_t** firState[NUM\_TAPS + FIR\_BLOCK\_SIZE - 1];

**uint16\_t** lineBias = adc1Res \* 0.5f;

**uint16\_t** adc1DataIn[BLOCK\_SIZE]; // Line audio data from ADC1 input

**float32\_t** adc1DataOut[BLOCK\_SIZE];

**float32\_t** dacDataIn[BLOCK\_SIZE];

**uint16\_t** dacDataOut[BLOCK\_SIZE];

**uint8\_t** adc2Data; // Volume control data from ADC2 input

**uint16\_t** \*adc1DataInPtr;

**float32\_t** \*adc2DataOutPtr;

**float32\_t** \*dacDataInPtr = &dacDataIn[0];

**uint16\_t** \*dacDataOutPtr = &dacDataOut[0];

**float32\_t** \*inF32, \*outF32;

ADC data resolution is calculated, and variables for filtering are defined. Input and output buffers and their respective pointers, as well as filter buffers and their respective pointers are defined. Filter tap values are taken from MATLAB and defined here.

// Initialize filter pointers

inF32 = &adcDataOut[0];

outF32 = &dacDataIn[0];

// Initialize filter

**arm\_fir\_init\_f32**(&fir, NUM\_TAPS, &firTaps[0], &firState[0], FIR\_BLOCK\_SIZE);

**HAL\_TIM\_Base\_Start**(&htim2);

**HAL\_TIM\_Base\_Start**(&htim3);

**HAL\_OPAMP\_Start**(&hopamp2);

**HAL\_ADC\_Start\_DMA**(&hadc1, (**uint32\_t**\*)adcDataIn, BLOCK\_SIZE);

**HAL\_ADC\_Start\_DMA**(&hadc2, (**uint32\_t**\*)&ctrlData, 1);

**HAL\_DAC\_Start\_DMA**(&hdac1, dacChannel, (**uint32\_t**\*)dacDataOut, BLOCK\_SIZE, dacAlignment);

In the main() function, operations are started, and the filter and its relevant pointers are initialized. Because the ADC is making continuous DMA requests, the input buffer is quickly filled and converted, triggering relevant callback functions:

// Sets pointers to first half of ADC buffer; moves first half of data from ADC in to ADC out

**void** **HAL\_ADC\_ConvHalfCpltCallback**(**ADC\_HandleTypeDef**\* hadc)

{

adc1DataInPtr = &adc1DataIn[0];

adc2DataOutPtr = &adc1DataOut[0];

**ADC\_ProcessHalf**();

}

// Sets pointers to second half of ADC buffer; moves second half of data from ADC in to ADC out

**void** **HAL\_ADC\_ConvCpltCallback**(**ADC\_HandleTypeDef**\* hadc)

{

adc1DataInPtr = &adc1DataIn[BLOCK\_SIZE / 2];

adc2DataOutPtr = &adc1DataOut[BLOCK\_SIZE / 2];

**ADC\_ProcessHalf**();

}

At each half of each A/D conversion, pointers are swapped between the first and second half of the buffers to support a consistent data stream. After they are set, data coming from the ADC is converted to float and set to the buffer for processing. The data is also adjusted to account for bias.

// Moves data from ADC in to ADC out; compensates for input bias

**void** **ADC\_ProcessHalf**(**void**)

{

**for** (**int** i = 0; i < (BLOCK\_SIZE \* 0.5f); i++)

{

adc2DataOutPtr[i] = (**float**)adc1DataInPtr[i] - bias;

}

}

In the infinite while(1) loop within the main() function, data is continuously filtered and set to the DAC output.

// Apply filter and move DAC data from in to out

**for** (**int** i=0; i<numBlocks; i++)

{

**if** (i == numBlocks-1)

{

// Move data from in to out

**DAC\_Process**();

}

// Apply filter in blocks

**arm\_fir\_f32**(&fir, inF32 + (i \* FIR\_BLOCK\_SIZE), outF32 + (i \* FIR\_BLOCK\_SIZE), FIR\_BLOCK\_SIZE);

}

To further support consistent data stream, the filter is applied in blocks. As defined in the variables earlier, FIR\_BLOCK\_SIZE is 128 bits, which means there will be 8 blocks of 128 bits to process (8 \* 128 = 1024 = BLOCK\_SIZE). When the filter has been applied to all of the blocks, the data is set to the DAC. The data is also adjusted for volume, gain, and to account for bias.

// Moves data from DAC in to DAC out, applies volume and gain calculations to samples; compensates for bias

**void** **DAC\_Process**(**void**)

{

**float** vol;

**for** (**int** i = 0; i < BLOCK\_SIZE; i++)

{

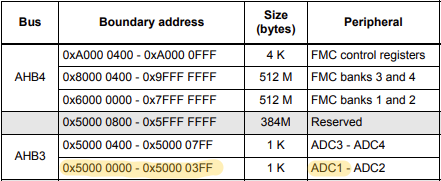
vol = (**float**)adc2Data / adc2Res; // Value between 0 and 1; multiplier for volume

dacDataOutPtr[i] = (**int**)((dacDataInPtr[i] \* vol) \* gain) + bias; // Apply volume multiplier and compensate for bias

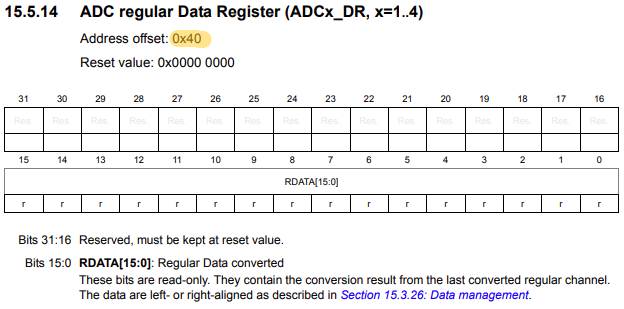
}

}

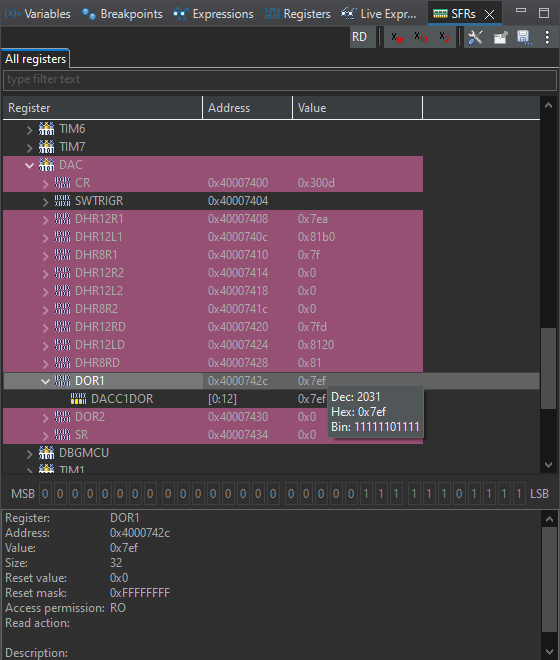
**Signal Monitoring**

Now that setup is complete and data is streaming from input to output, the signals can be monitored for accuracy in real-time using CubeMonitor. All that’s needed are the register addresses for the data to be monitored. The two registers of interest are the ADC input and the DAC output, which can be found in the reference manual.

Source: STMicroelectronics



Source: STMicroelectronics

So, the address for the ADC1 Data Register is 0x50000040. The address for the DAC can be found the same way, or with the help of a quick debug session:

Source: Author

The address for the DAC Data Out Register is 0x40000742c. Both of these values can be observed in real time with the Live Expression Viewer in STM32CubeIDE, or they can be plotted on a graph in CubeMonitor to provide additional context. Below are the data in CubeMonitor before and after accounting for input bias:

Sampled at 12 bits, ADC1 has a resolution of 4096:1. Before adjusting for bias in code, ADC1’s input can be observed at a continuous value of ~2060, which is approximately 50.3%, reflective of the analog bias circuit on the input. meaning the VDDA and the DAC output sits at ~24% VDDA. The adjustment to the ADC input is not reflected in the second graph, because it is applied to the input *after* it is registered. The DAC output sits at ~2048, which is exactly 50% VDDA, as expected. When applied to some music, the effect of the FIR filter on the DAC output data is apparent, as shown in the graph below.

Source: Author

Source: Author

Source: Author

While its capabilities may pale in comparison to hardware devices with advanced audio CODECs, the STM32F-series microcontroller proves to be capable of performing advanced DSP algorithms on its own.