IIR Filter MATLAB Design and Simulation Plus C Implementation using the CMSIS DSP Library for STM32 Multichannel ADCs

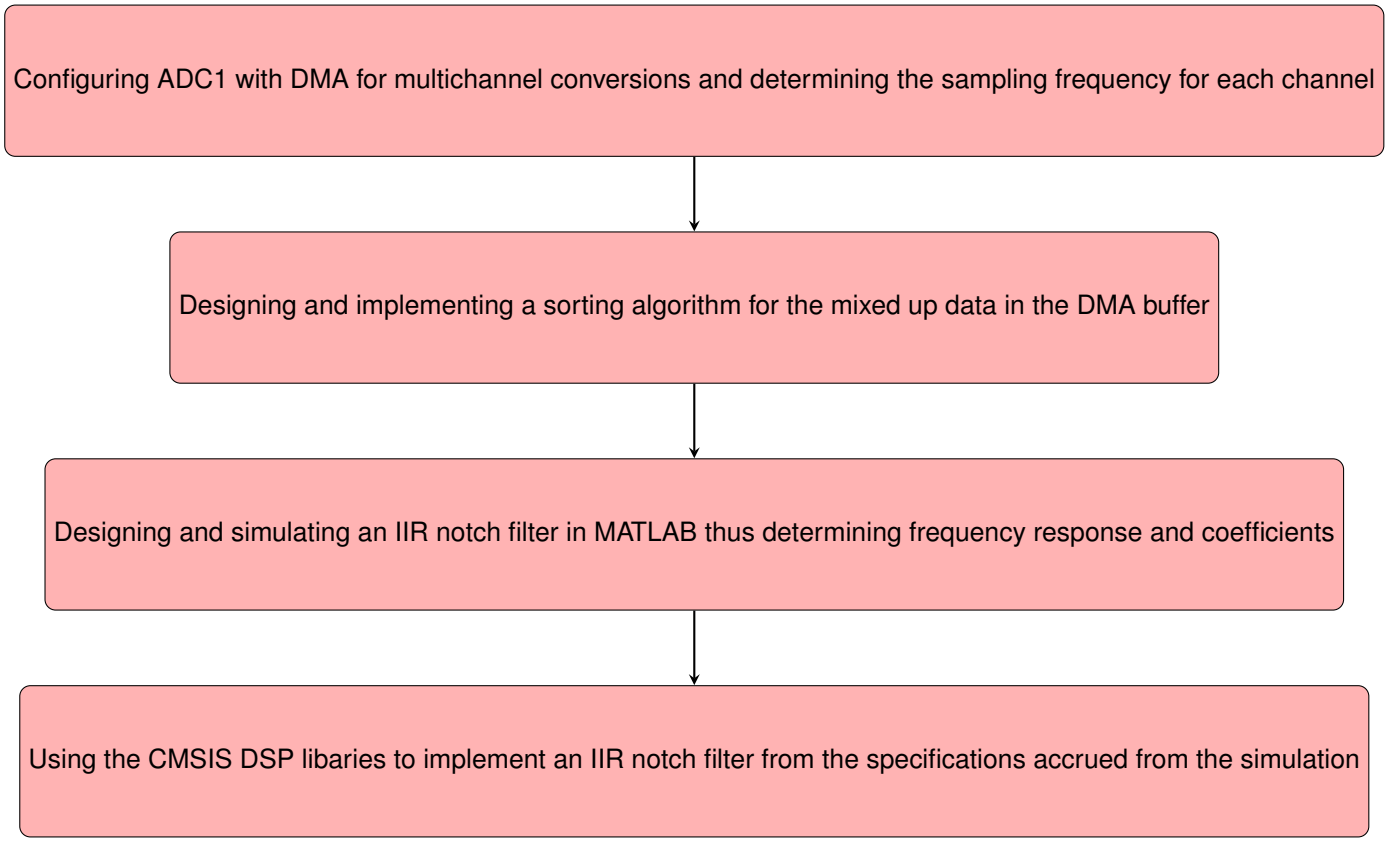
Mwangi Alex.W

September 19, 2024

Introduction

This article is an effort to explain and illustrate how I went about in designing an embedded software based IIR notch filter using the CMSIS DSP library on an STM32G4 to suppress 50Hz power line interference for a biosignal acquisition board I designed in Flux. The board's role is to collect, process and transmit muscle signals otherwise referred to as sEMG signals from the surface of key muscles of both human arms. Such signals are susceptible to 50Hz powerline interference from the mains hence measures to mitigate the interference's effect are paramount otherwise the data accrued from the board would be populated by inaccurate values. IIR filters offer the advantage of assured stability and ease of implementation compared to FIR filters and FFT techniques. A notch filter is able to attenuate signals of a minimal bandwidth. In other words, we say that a notch filter is a high frequency specificity bandstop filter. In this article , I will focus on one of the ADCs I utilised—ADC1. I enabled 2 out of the available 16 channels for ADC1. Conversions in each ADC are conducted in ranks through the scan conversion mode feature of STM32 MCUs. That means that if an ADC is DMA enabled (as is with our case), data is converted and stored in the DMA buffer in turns for each of the two ADC channels. For ADC1 of the STM32G4A1VET6, I chose channel IN1 with rank 1 and channel IN2 with rank 2. The conversion frequency for ADC1 was chosen to be 10.471kHz by prescaling timer 6 and setting it's autoreload register to a supposed value as the ADC used timer triggered conversions. This conversion frequency can however be altered according to one's design requirements. Here, I assume the reader has already chosen and configured a conversion frequency that suits his/her application as the conversion frequency parameter is a key specification we require to commence filter design. Since we are utilizing two channels, the frequency of operation for each channel is half the ADC conversion frequency (5.24kHz for our case). Data from both channels is loaded in a pre-initialized DMA buffer that holds IN1 channel data in its even numbered buffer locations and IN2 channel data in its odd numbered buffer locations. I therefore designed a sorting algorithm to ensure that data for each channel is loaded from the DMA buffer and stored in respective

buffers where data can hence be accessed for processing. What follows the design of the sorting algorithm is the design of an IIR notch filter simulated using MATLAB and implemented using the CMSIS DSP libraries to attenuate 50Hz powerline interference. Simulation is necessary to ensure that the filter suits our frequency response expectations and also achieve the filter coefficients and state variables that are required to run the notch filter application. MATLAB supports built-in functions necessary to design and simulate any kind of filter with ease hence abstracting all the complexities we would incur by designing the filter mathematically. After we have confirmed the filter's frequency response is as required and having achieved the filter coefficients and state variables needed as arguments to the CMSIS DSP functions, we jump into developing the filter. The filter and sorting algorithm was developed on STM32Cube IDE with the C programming language. The flow chart below outlines the design flow I followed to design and develop the filter.



The Sorting algorithm

You have configured your STM32 MCU's ADC to operate in a particular frequency of conversion depending on your application's requirements and enabled two or more channels for either or all ADCs. To determine the frequency of conversion for each channel, we divide the ADC's conversion frequency by the number of channels. Here, we configure ADC1 to operate at 10.471kHz with two channels—IN1 and IN2. That means that the frequency of conversion for each channel is half the ADC's conversion frequency—5.24kHz. When conversions occur in either of the two channels of ADC1 (IN1 or IN2), data is stored in a DMA buffer as 32bit

integer types (modifiable). This occurs in turns such that the even number DMA buffer register locations store data from the rank 1 channel (IN1) while the odd number DMA buffer register locations store data from the rank 2 channel (IN2) as the conversions are stored in the DMA buffer successively. This means that that the DMA buffer holds mixed up data at any given time and that half of the data is of IN1 channel while the other half is of IN2 channel when the buffer is filled. It is therefore necessary to implement a sorting algorithm to load each data component in the DMA buffer and store it in a buffer that holds data of only one particular channel. That way, we can access each channel's data independently from a buffer and process it accordingly. To implement the algorithm on STM32Cube IDE, I created a header file and a corresponding .c file upon which required variables, buffers, pointers and functions would be declared, initialised and defined. In the header file I initialised a struct that would hold members that would be accessed through a pointer to run the sorting algorithm. The header file is as shown on Listing 1.

Listing 1: Header file for the sorting algorithm

```
/*
 * _DMA_Sort.h
 *  Author: Mwangi Alex. W
 *
 *  This header file holds all the variables and buffers within
 *  a struct that are neccessary to implement the sorting
 *  algorithm and all fucnction declarations for the same
*   algorithm. The sorting algorithm is responsible
 *  for scanning the DMA buffer that stores data for all
 *  channels (IN1 and IN2 for ADC1 ) loading each value from
 *  the buffer, and storing it in buffers that now hold data for
 *  respective channels
 */

#ifndef INC__DMA_SORT_H_
#define INC__DMA_SORT_H_

// MACROS
#define ADC_DMA_BUFFERSIZE 2500
#define ADC_DMA_HALFBUFFERSIZE 1250
#define ADC_DMA_QUATERBUFFERSIZE 625
#define ADC_DMA_EIGHTHBUFFERSIZE 312


//INCLUSIONS
#include "stm32g4xx_hal.h"  // Replace 'g4' with your STM32 series

//STRUCTS, VARIABLES AND POINTERS
typedef struct //Struct for ADC1
```

```
{
    uint32_t ADC1_DMA_bfr[ADC_DMA_BUFFERSIZE]; // Buffer that stores the ADC conversions.
        Source buffer
    uint32_t ADC1_DMA_mon; // Variable to monitor one value from the DMA buffer

    uint32_t ADC1_IN1_bfr[ADC_DMA_HALFBUFFERSIZE]; // Buffer that stores sorted IN1 data
    uint32_t ADC1_IN1_mon; // Variable to monitor one value from the IN1 data buffer

    uint32_t ADC1_IN2_bfr[ADC_DMA_HALFBUFFERSIZE];  // Buffer that stores sorted IN2 data
    uint32_t ADC1_IN2_mon; // Variable to monitor one value from the IN2 data buffer

} ADC1_DMA_sort;

extern ADC1_DMA_sort*ADC1_DMA_sort_ptr; // Pointer to the struct

//FUNCTION DECLARATIONS
/* Loads data from the upper half of the DMA buffer for ADC1 (When a half complete call back
    is fired) and stores it to two buffers; one for channel 1 data and another for the channel
     2 data such that they are stored and can be accesed independently */
void ADC1_DMA_sort_uhb (void);

/* Loads data from the lower half of the DMA buffer for ADC1 (When a complete call back is
    fired) and stores it to two buffers; one for channel 1 data and another for the channel 2
    data such that they are stored and can be accesed independently */
void ADC1_DMA_sort_lhb (void);

#endif /* INC__DMA_SORT_H_ */
```

Now that we have all the resources in form of variables, buffers, pointers and function declarations that we need to implement our sorting algorithm. I present to you the .c file on Listing 2 that defines the functions responsible for sorting the data in the DMA buffer.

Listing 2: .c file for the sorting algorithm

```
/*
 * _DMA_Sort.c
 *  Created on: Sep 14, 2024
 *  Author: Mwangi Alex. W
 *
 *  Defines all the functions that implement the sorting algorithm for all ADCs
 */
#include "_DMA_Sort.h"

```

```
void ADC1_DMA_sort_uhb (void)
{
    for(uint32_t m=0; m<ADC_DMA_HALFBUFFERSIZE-1; m++) //Scans the upper half of the DMA
        buffer
    {
        if(m==0 || (m%2==0)) // Checks that the buffer register location is even and loads the
             data
        {
            for(uint32_t n=0; n<ADC_DMA_QUATERBUFFERSIZE-1; n++) // Stores data on the upper
                half of IN1 channel data buffer
            {
             ADC1_DMA_sort_ptr->ADC1_IN1_bfr[n]=ADC1_DMA_sort_ptr->ADC1_DMA_bfr[m];
            }
        }
        else // Checks that the buffer register location is odd and loads the data
        {
            for(uint32_t p=0; p<ADC_DMA_QUATERBUFFERSIZE-1; p++)
            // Stores data on the upper half of IN2 channel data buffer
            {
             ADC1_DMA_sort_ptr->ADC1_IN2_bfr[p]=ADC1_DMA_sort_ptr->ADC1_DMA_bfr[m];
            }
        }
    }

};

void ADC1_DMA_sort_lhb (void)
{
    for(uint32_t m=ADC_DMA_HALFBUFFERSIZE; m<ADC_DMA_BUFFERSIZE-1; m++) // Scans the lower
        half of the DMA buffer
    {
        if(m==0||(m%2==0)) // Checks that the buffer register location is even and loads the
            data
        {
            for(uint32_t n=ADC_DMA_QUATERBUFFERSIZE; n<ADC_DMA_HALFBUFFERSIZE-1; n++) //
                Stores data on the lower half of IN1 channel data buffer
            {
             ADC1_DMA_sort_ptr->ADC1_IN1_bfr[n]=ADC1_DMA_sort_ptr->ADC1_DMA_bfr[m];
            }
        }
        else // Checks that the buffer register location is odd and loads the data
        {
            for(uint32_t p=ADC_DMA_QUATERBUFFERSIZE; p<ADC_DMA_HALFBUFFERSIZE-1; p++)  //
                Stores data on the lower half of IN2 channel data buffer
            {
             ADC1_DMA_sort_ptr->ADC1_IN2_bfr[p]=ADC1_DMA_sort_ptr->ADC1_DMA_bfr[m];
```

```
            }
        }
    }

};
```

To implement the algorithm henceforth, we only need to call this functions in the main.c file. Listing 2 shows how I called the functions.

Listing 3: Calling the functions in main.c

```

/* USER CODE BEGIN Includes */
#include "_DMA_sort.h"
/* USER CODE END Includes */

 /* USER CODE BEGIN 2 */

//ADC1 is started using timer 6 tiggered conversions
  HAL_StatusTypeDef HAL_TIM_Base_Start(TIM_HandleTypeDef *htim6);
    ADC_status=HAL_ADC_Start_DMA(&hadc1, ADC1_DMA_sort_ptr->ADC1_DMA_bfr,ADC_DMA_BUFFERSIZE);

  /* USER CODE END 2 */


/* USER CODE BEGIN 4 */
void                    HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef *hadc) // Fires when
    the upper half of the DMA buffer is filled
{
     if (hadc->Instance == ADC1)
       {
            ADC1_DMA_sort_uhb(); // Handles the upper half of the DMA buffer

            ADC1_DMA_sort_ptr->ADC1_DMA_mon=ADC1_DMA_sort_ptr->ADC1_DMA_bfr[
                ADC_DMA_QUATERBUFFERSIZE];  // monitoring one of the DMA buffer registers in
                the upper half of the buffer

            ADC1_DMA_sort_ptr->ADC1_IN1_mon=ADC1_DMA_sort_ptr->ADC1_IN1_bfr[
                ADC_DMA_EIGHTHBUFFERSIZE]; // monitoring one of the IN1 data buffer registers
                in the upper half of the buffer

            ADC1_DMA_sort_ptr->ADC1_IN2_mon=ADC1_DMA_sort_ptr->ADC1_IN2_bfr[
                ADC_DMA_EIGHTHBUFFERSIZE]; // monitoring one of the IN2 data buffer registers
                in the upper half of the buffer
```

```
27          }
28
29  }
30
31  void                    HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc) // Fires when the
        lower half of the DMA buffer is filled
32  {
33        if (hadc->Instance == ADC1)
34          {
35              ADC1_DMA_sort_lhb(); // Handles the lower half of the DMA buffer
36
37              ADC1_DMA_sort_ptr->ADC1_DMA_mon=ADC1_DMA_sort_ptr->ADC1_DMA_bfr[
                    ADC_DMA_HALFBUFFERSIZE + ADC_DMA_QUATERBUFFERSIZE]; // monitoring one of the
                    DMA buffer registers in the lower half of the buffer
38
39              ADC1_DMA_sort_ptr->ADC1_IN1_mon=ADC1_DMA_sort_ptr->ADC1_IN1_bfr[
                    ADC_DMA_QUATERBUFFERSIZE + ADC_DMA_EIGHTHBUFFERSIZE]; // monitoring one of the
                     IN1 data buffer registers in the lower half of the buffer
40
41              ADC1_DMA_sort_ptr->ADC1_IN2_mon=ADC1_DMA_sort_ptr->ADC1_IN2_bfr[
                    ADC_DMA_QUATERBUFFERSIZE + ADC_DMA_EIGHTHBUFFERSIZE]; // monitoring one of the
                     IN2 data buffer registers in the lower half of the buffer
42          }
43  }
44  /* USER CODE END 4 */
```

The sorting algorithm is now implemented. We now have individual buffers that hold data for channels IN1 and channel IN2. The data is continuously uploaded since the DMA buffer is a circular buffer. This means that after every successive half complete call back (when sorting starts to occur), we have fresh data in all the buffers involved in the sorting. We now shift our focus to designing the IIR notch filters for each channel.

IIR Filter Design and Simulation using MATLAB

The objective of IIR filter design is to determine the digital filter's numerator and denominator coefficients for its Z-transform to satisfy filter specifications such as pass band gain and stop band attenuation as well as cutoff frequency or frequencies for the lowpass, high pass, band pass, and bandstop filters. In our case we are designing a high specificity bandstop filter in the form of a notch filter. The design flow we follow here can however be adopted in the design of any other IIR filter. To design a digital filter, we first of all need its specifications. Its specifications depend on the kind of filter we intend to design. For our case, we need digital notch filter design specifications. A notch filter has three frequency parameters; the centre frequency which has the maximum attenuation and the cutoff frequencies which characterize the bandwidth of the filter

and have 3dB attenuation. Since we intend to attenuate 50Hz powerline interference, the centre frequency for the notch filter is chosen to be exactly that. The cutoff frequencies on the other hand are chosen to be equidistant from the centre frequency. For example, if the lower cutoff frequency is 50-x, the higher cutoff frequency has to be 50+x so as to exhibit symmetry. After choosing the digital filter specifications depending on our performance requirements, we now undertake what we refer to as frequency warping. Frequency warping involves converting the given digital frequency specifications to analog ones in order to achieve a corresponding analog filter for our digital filter by invoking the analog specifications on a normalised lowpass analog prototype filter of our choice. The formula for warping digital frequencies specifications to analog frequency specifications is as given in equation 1.

$$\omega_a = \frac{2}{T}\tan\left(\frac{\omega_d T}{2}\right) \tag{1}$$

Where ω_a is the warped analog frequency, T the period of operation for each channel $\left(\frac{1}{f}\right)$, and ω_d the digital frequency specification.

Frequency warping is done to every digital filter specification. Another parameter of importance is the bandwidth of our notch filter (W). Its bandwidth is the difference between the higher cutoff frequency and the lower cutoff frequency. Hence to find the bandwidth of the filter in the analog domain, we warp the cutoff frequencies and find their difference. The equation for the analog bandwidth of the filter is as shown on equation 2.

$$W_a = \frac{2}{T}\tan\left(\frac{\omega_{2d} T}{2}\right) - \frac{2}{T}\tan\left(\frac{\omega_{1d} T}{2}\right) \tag{2}$$

Where W_a is the analog bandwidth of the filter, T is the period of operation for each channel $\left(\frac{1}{f}\right)$, ω_{2d} is the digital higher cutoff frequency, and ω_{1d} is the digital lower cutoff frequency of the notch filter.

To achieve the digital centre frequency from the two digital cutoff frequency specifications, we use the formula;

$$\omega_{od} = \sqrt{W_{1d} W_{2d}} \tag{3}$$

Where ω_{od} is the digital centre frequency of the filter

To achieve the corresponding analog centre frequency, we warp the digital centre frequency. Hence,

$$\omega_{oa} = \frac{2}{T}\tan\left(\frac{\omega_{od} T}{2}\right) \tag{4}$$

Where ω_{oa} is the analog centre frequency for the filter of the filter

After having chosen and determined our digital and warped analog frequency specifications, we embark on choosing a normalised lowpass filter prototype in the form of a Laplace transform transfer function that will be

used to determine the corresponding analog filter's Laplace transform for our filter. The prototype we choose determines the order of our filter. The higher the order of the filter the better the performance but at the cost of computational resources. That trade off should therefore be optimised. The order of the digital filter is raised by two from the order of the analog lowpass prototype filter we choose. This means that we should choose an analog low pass prototype which is of 2 orders less than the the order we intend for our digital filter. In our case, we intend to design a 4th order digital filter which means that our lowpass prototype needs to be of order 2. A second order normalised analog lowpass prototype filter has the form;

$$H_p(s) = \frac{1}{s^2 + \sqrt{2}s + 1} \tag{5}$$

The next step is to convert the lowpass prototype to a bandstop filter with the specifications of the warped analog frequencies. Luckily MATLAB provides a function that does exactly that. Its input arguments are the numerator and denominator coefficients of the prototype filter, the warped analog centre frequency and the warped analog bandwidth of the filter. The function is given in Listing 4. The outputs of that function are numerator and denominator coefficients of the Laplace transform for the corresponding analog filter for our intended digital filter. With our corresponding analog filter at hand, we now convert it to the digital domain using the bilinear transform method such that we achieve the Z-transform for our intended digital filter. MATLAB provides a function that does exactly that. The function is given in Listing 4. Its inputs arguments are the numerator and denominator coefficients of the corresponding analog filter and the sampling frequency of each channel while its output arguments are the numerator and denominator coefficients for our intended digital filter Z-transform. This coefficients are our ultimate goal. Nonetheless, we need to simulate the filters frequency response and confirm that it matches our expectations. We therefore implement a built in MATLAB function that computes the frequency response. The function is shown in Listing 4. Now that I have explained the workflow, I present to you Listing 4 below which shows all the MATLAB functions necessary to design and simulate the filter. The code is well commented hence all the necessary functions and parameters are traceable.

Listing 4: MATLAB script for designing and simulating an IIR notch filter

```
%% A MATLAB script to design and simulate an IIR notch filter that
% attenuates 50Hz powerline interference for a biosignal acquisition board
%% Frequency and period constants
T=1.91e-4; % Period of each channel
fs=1/T; %=5.2355kHz. Frequency of operation for each channel
W1d=2*pi*(47); % Digital filter 3db lowerside cuttoff frequency specification
Wnd=2*pi*(53); % Digital filter 3db upperside cuttoff frequency specification
W=(((2/T)*(tan((Wnd/2)*T)))-((2/T)*(tan((W1d/2)*T)))); % Warped analog filter frequency
    bandwidth specification
Wd0=sqrt(W1d*Wnd); % Digital filter centre frequency
Woa=((2/T)*tan((Wd0/2)*T)); % Warped analog filter centre frequency

%% Lowpass prototype to lowpass analog filter
[B,A]=lp2bs(1,[1 sqrt(2) 1], Woa, W); % This function converts a lowpass prototype filter to a
     band stop filter with the warped frequency specifications.Functions for converting to
    lowpass, highpass, and bandpass are also available. B and A are the coefficients of the
    corresponding analog filter laplace transform for our intended digital filter

%% Bilinear transform
disp('Numerator coefficients b and Denominator coefficients a of the 4th order digital notch
    filter Z-tranform:');
[b,a]=bilinear(B,A,fs);%Implements a bilinear transform using B and A. b and a are the
    coefficients of the intended digital filter z-transform
disp(['b: ', num2str(b)]);
disp(['a: ', num2str(a)]);

%% Frequency response plots (magnitude and phase)
[hz, f]=freqz(b,a,512,fs); % Computes the frequency and phase response of the filter
magdB=20*log10(abs(hz)); %Converting magnitude to dB attenuation or gain. +ve dB=gain while -
    ve dB=atenuation
subplot(2,1,1), plot(f,magdB),grid; % Plots the magnitude response
axis([0 fs/50 -20 5]);  % Adjusting view according to filter characteristics
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
title('Frequency Response in dB')

phi=180*unwrap(angle(hz))/pi; %Converts the angles to radians
subplot(2,1,2), plot(f, phi); grid; % Plots the phase response
axis([0 fs/2-100 0]); % Adjusting view according to filter characteristics
xlabel('Frequency (Hz)')
ylabel('Phase (degrees)')
title('Phase Response')
```

Listing 4 achieves the objective of designing and simulating the filter. It displays the coefficients of the 4th order digital notch filter on the command window and frequency response plots. The coefficients will be shown

in the next section of the article as we develop C code for the filter using CMSIS DSP libraries. The Frequency response plots of the filter are as shown on figure 1

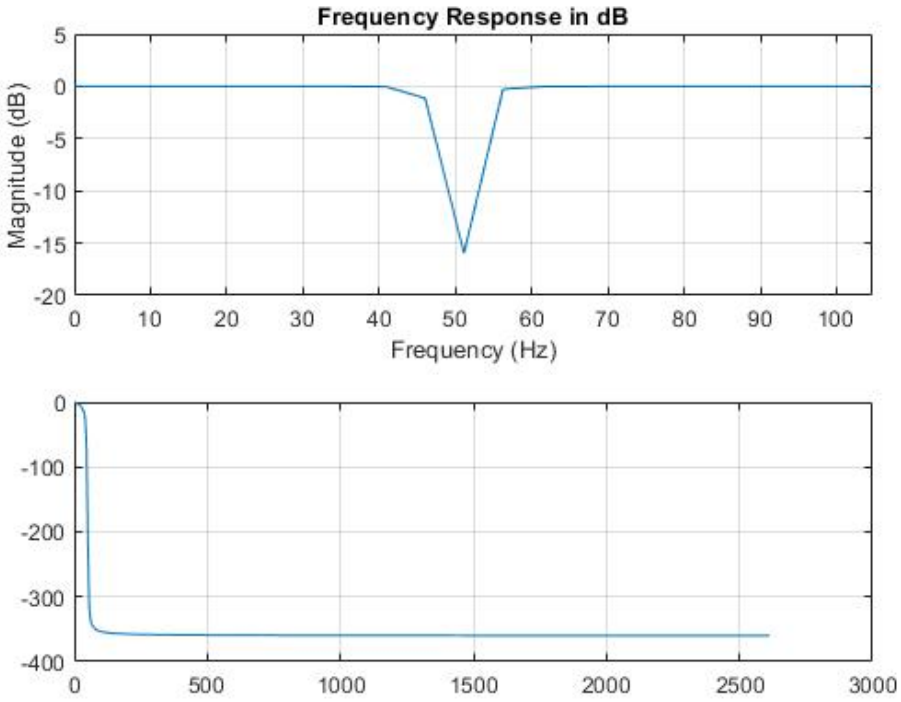
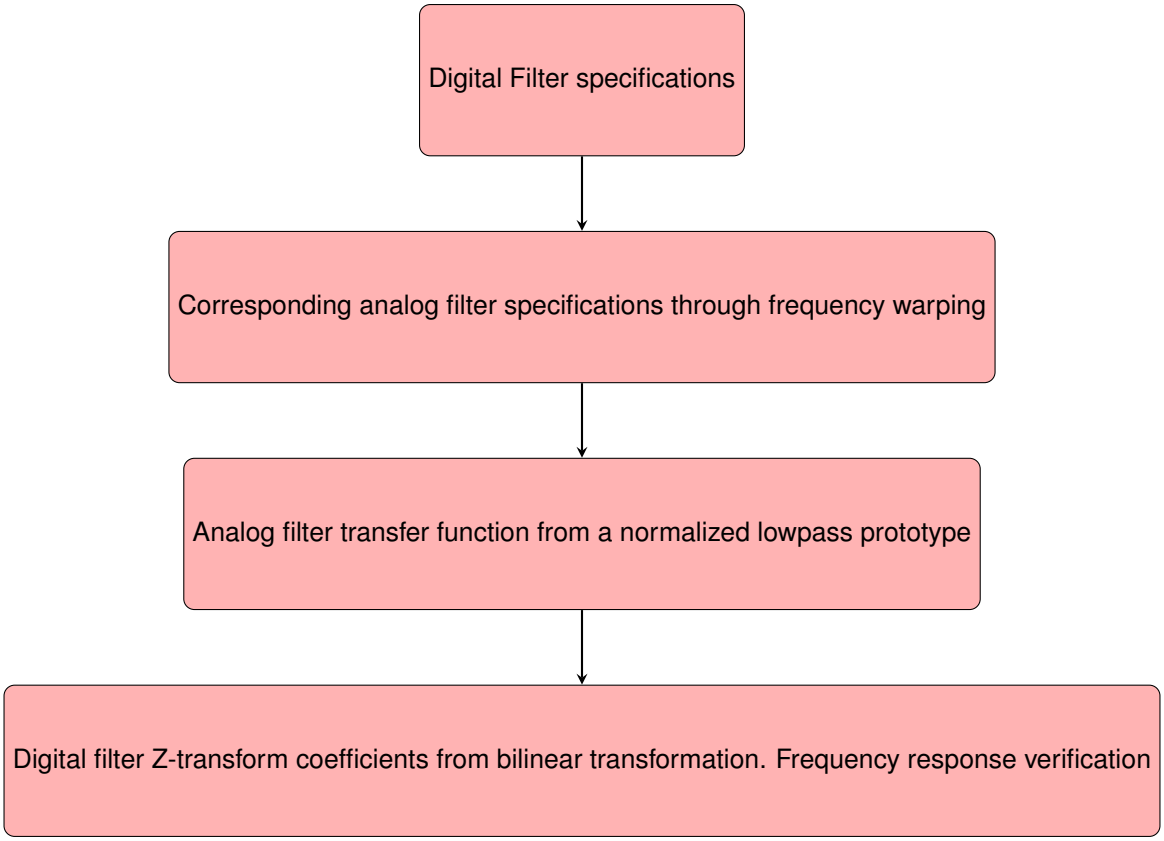


Figure 1: Magnitude and phase response plot of the filter. The figure shows a centre frequency of approximately 51Hz which is near our intended 50Hz centre frequency. 50Hz is attenuated by about 13dB which will suffice for our application. The inaccuracy can be addressed by using better lowpass prototype filters and using Chebyshev or Butterworth filter design techniques

The flow chart below shows the design flow for designing the filter.



Implementing the Filter using CMSIS DSP Libraries on an STM32

The key prerequisites to implement an IIR filter are the numerator and denominator coefficients, the order of the filter and the form of implementation. The coefficients and order for the filter were already achieved in the design phase. Its the form of implementation that remains. Forms of implementation come in two ways—Direct form I and Direct form II. One main advantage of the Direct Form II filter over Direct Form I is that it requires half the number of state variables. Generally, Direct Form I implementations are preferred for fixed-point implementations (better numerical behavior) while Direct Form II is preferred for floating-point implementations (fewer state variables). Since Cortex M4 has a capable floating point unit, Direct form II implementations are preferred. The CMSIS libraries provide functions to implement IIR filters in the form of biquad cascaded functions. A biquad filter is an IIR filter of order 2 and is the building block used for designing any kind of IIR filter. A biquad cascaded function thus is a function that enables us to scale a biquad filter in order to achieve an IIR filter of a higher order. A deeper dive into how Direct Form II filters are implemented and the biquad cascaded functions necessary to implement them can be found on the CMSIS DSP website (Link to the page). Primarily, given that our filter is of order 4, we need to implement two cascaded biquad filters of Direct Form II by invoking the functions that CMSIS DSP provides for implementing cascaded biquad filters. However, before using this functions, they need to be integrated into the STM32CubeIDE environment. Guidance on how to configure the necessary Include and library files can be found here. Since the functions are explained in the CMSIS DSP site, I will go ahead to present the header and .c files I developed to implement the filter.

Listing 5: Header file for the IIR notch filter implementation

```
1
2  /*
3   * _ADC1_INx_NF.h
4   *
5   * Created on: Sep 14, 2024
6   * Author: Mwangi Alex. W
7   *
8   * This header file consists of all the variables and pointers that
9   * are to be fed to the CMSIS DSP functions for IIR biquad filter
10  * implementation for a notch filter that attenuates 50hZ powerline
11  * interference noise. The variables and pointers are declared in
12  * the form of a structure upon which every member of the struct is
13  * accessed as a pointer. That way, the input arguments of the DSP
14  * functions are fed as members of a struct.
15  */
16
17 #ifndef INC__ADC1_INX_NF_H_
18 #define INC__ADC1_INX_NF_H_
19
20 // MACROS
21 #define NOOFCOEFFICIENTS 5 // The number of coefficients of one biquad filter stage are 5.
        Total number of coefficients = number of stages X number of coefficients per stage
22 #define NOOFS_VARIABLES 2  // The number of variables for each biquad filter are 2. Total
        number of variables = number of stages X number of state variables per stage
23
24
25 //INCLUSIONS
26 #include "_DMA_Sort.h" // Header file that caters for the sorting of mixed up channel data
        from the DMA buffer into buffers that contain the data from respective ADC channels
27 #include "arm_math.h" // Header file neccessary for running CMSIS DSP functions
28
29 //STRUCTS, VARIABLES AND POINTERS
30 // The struct that holds all the variables and pointers that are needed to to run the biquad
        DSP function
31
32 typedef struct // Handles IN1 channel data
33 {
34     uint8_t ADC1_IN1_numstages; // The number of biquad stages for the notch filter. One
          biquad is of the order 2. Using two stages means our filter is of order 4. Declaration
           with initialization would result in an error
35
36     float32_t  ADC1_IN1_coeffs[2*NOOFCOEFFICIENTS]; // The coefficients for each biquad stage.
           The coefficients are declared as an array of size-->Total number of coefficients
37
38     const float32_t *ADC1_IN1_pcoeffs; // Pointer to the coefficients array
```

```
39
40      float32_t ADC1_IN1_States[2*NOOFS_VARIABLES]; // A buffer of the size-->Total number of
            states, is declared and all variables initialized to zero later in the .c file
41
42      float32_t *ADC1_IN1_pState; //Pointer to the states array
43
44      float32_t *ADC1_IN1_psrc; // Pointer to the input or source buffer. The source buffer in
            this case is one that stores sorted data from channel 1 of ADC1
45
46      float32_t _NF_ADC1_IN1_bfr[ADC_DMA_HALFBUFFERSIZE]; // The destination or output buffer.
            Stores data from channel 1
47
48      float32_t *ADC1_IN1_pdst; // Pointer to the destination buffer
49
50      uint32_t ADC1_IN1_Blocksize; // Number of samples to be processed. Equal to the source
            buffer size
51
52      const arm_biquad_cascade_df2T_instance_f32 S1; //Instance that is fed to the CMSIS DSP
            functions
53
54  } ADC1_IN1_NF;
55
56  extern ADC1_IN1_NF ADC1_IN1_NF_arg; // Variable used initialize the struct in the .c file
57
58
59
60  typedef struct   // Handles IN2 channel data
61  {
62      uint8_t ADC1_IN2_numstages; // The number of biquad stages for the notch filter. One
            biquad is of the order 2. Using two stages means our filter is of order 4. Declaration
             with initialization would result in an error
63
64      float32_t  ADC1_IN2_coeffs[2*NOOFCOEFFICIENTS]; // The coefficients for each biquad stage.
             The coefficients are declared as an array of size-->Total number of coefficients
65
66      const float32_t *ADC1_IN2_pcoeffs; // Pointer to the coefficients array
67
68      float32_t ADC1_IN2_States[2*NOOFS_VARIABLES]; // A buffer of the size-->Total number of
            states, is declared and all variables initialized to zero
69
70      float32_t *ADC1_IN2_pState; //Pointer to the states array
71
72      float32_t *ADC1_IN2_psrc; // Pointer to the input or source buffer. The source buffer in
            this case is one that stores sorted data from channel 2 of ADC1
73
74      float32_t _NF_ADC1_IN2_bfr[ADC_DMA_HALFBUFFERSIZE];  // The destination or output buffer.
```

```
        Stores data from channel 2
75
76      float32_t *ADC1_IN2_pdst; // Pointer to the destination buffer
77
78      uint32_t ADC1_IN2_Blocksize; // Number of samples to be processed. Equal to the source
            buffer size
79
80      const arm_biquad_cascade_df2T_instance_f32 S2; //Instance that is fed to the CMSIS DSP
            functions
81
82  } ADC1_IN2_NF;
83
84  extern ADC1_IN2_NF ADC1_IN2_NF_arg; // Variable used initialize the struct in the .c file
85
86
87
88
89  //FUNCTION DECLARATIONS
90  void init_ADC1_IN1_struct(void); // Funtion that initializes the pointer to the source buffer
        at runtime for channel 1
91
92  void init_ADC1_IN1_FO_biquad_filter(void); // The CMSIS DSP function that initializes the
        filter coefficients, variables and buffer sizes before filtering commences for channel 1
93
94  void update_ADC1_IN1_FO_biquad_filter(void); // The optimised CMSIS DSP function that handles
        all the filtering operations and uploads the updated values to the destination buffers for
         channel 1
95
96
97  void init_ADC1_IN2_struct(void); // Funtion that initializes the pointer to the source buffer
        at runtime for channel 2
98
99  void init_ADC1_IN2_FO_biquad_filter(void); // The CMSIS DSP function that initializes the
        filter coefficients, variables and buffer sizes before filtering commences for channel 2
100
101 void update_ADC1_IN2_FO_biquad_filter(void); // The optimised CMSIS DSP function that handles
        all the filtering operations and uploads the updated values to the destination buffers for
         channel 2
102
103 #endif /* INC__ADC1_INX_NF_H_ */
```

Listing 6: .c file for the IIR notch filter implementation

```c

/*
 * _ADC1_INx_NF.c
 *
 *  Created on: Sep 14, 2024
 *  Author: Mwangi Alex. W
 */

//INCLUSIONS
#include <_ADC1_INx_NF.h>

// GLOBAL BUFFER DECLARATIONS
float32_t ADC1_IN1_coeffs[2 * NOOFCOEFFICIENTS] =
{
    // First Biquad Stage coefficients
         /*b0-b2*/ 0.9660f,  -3.9793f,  5.9654f, /*a1-a2*/ 3.9860f,  -5.9653f,

    // Second Biquad Stage coefficients
         /*b0-b2*/ 1.0000f,  -3.9793f,  0.9966f, /*a1-a2*/ -3.9725f,  -0.9932f
}; // Coefficients for the notch filter as obtained in MATLAB for channel 1

float32_t ADC1_IN1_States[2 * NOOFS_VARIABLES] = {0.0f};  // Initialize state variables to 0

ADC1_IN1_NF ADC1_IN1_NF_arg = // Initializes the struct declared in the header file
    {
        .ADC1_IN1_numstages = 2,  // Number of stages in the biquad filter

        .ADC1_IN1_coeffs=  // This will copy the coefficients to the local array within the
            struct
        {
            0.9660f,  -3.9793f,  5.9654f,  3.9860f,  -5.9653f,
            1.0000f,  -3.9793f,  0.9966f,  -3.9725f,  -0.9932f
        },

        .ADC1_IN1_pcoeffs = ADC1_IN1_coeffs,  // Pointing to the global coefficient array

        .ADC1_IN1_States = {0.0f},  // Zero-initialize the state variables

        .ADC1_IN1_pState = ADC1_IN1_States,  // Pointing to the global state array

        .ADC1_IN1_psrc = NULL,  // Initialize to NULL, assign at runtime

        .ADC1_IN1_pdst = ADC1_IN1_NF_arg._NF_ADC1_IN1_bfr,  // Initialize destination buffer
            pointer

```

```
        .ADC1_IN1_Blocksize = ADC_DMA_HALFBUFFERSIZE, // Processed samples are equal in number
             to the source buffer size

        .S1 = {0},  // Initialize the CMSIS DSP biquad filter instance (Initialized properly
            at runtime)
    };



float32_t ADC1_IN2_coeffs[2 * NOOFCOEFFICIENTS] =
{
    // First Biquad Stage coefficients
         /*b0-b2*/ 0.9660f,  -3.9793f,  5.9654f, /*a1-a2*/ 3.9860f,  -5.9653f,

    // Second Biquad Stage coefficients
         /*b0-b2*/ 1.0000f,  -3.9793f,  0.9966f, /*a1-a2*/ -3.9725f,  -0.9932f
};  // Coefficients for the notch filter as obtained in MATLAB for channel 2. Equal to those
    of the channel 1 as the filter is a duplicate

float32_t ADC1_IN2_States[2 * NOOFS_VARIABLES] = {0.0f};  // Initialize state variables to 0

ADC1_IN2_NF ADC1_IN2_NF_arg = // Initializes the struct declared in the header file
    {
        .ADC1_IN2_numstages = 2,  // Number of stages in the biquad filter

        .ADC1_IN2_coeffs=  // This will copy the coefficients to the local array within the
            struct
        {
            0.9660f,  -3.9793f,  5.9654f,  3.9860f,  -5.9653f,
            1.0000f,  -3.9793f,  0.9966f,  -3.9725f,  -0.9932f
        },

        .ADC1_IN2_pcoeffs = ADC1_IN2_coeffs,  // Pointing to the global coefficient array

        .ADC1_IN2_States = {0.0f},  // Zero-initialize the state variables

        .ADC1_IN2_pState = ADC1_IN2_States,  // Pointing to the global state array

        .ADC1_IN2_psrc = NULL,  // Initialize to NULL, assign at runtime

        .ADC1_IN2_pdst = ADC1_IN2_NF_arg._NF_ADC1_IN2_bfr,  // Initialize destination buffer
            pointer

        .ADC1_IN2_Blocksize = ADC_DMA_HALFBUFFERSIZE,

        .S2 = {0},  // Initialize the CMSIS DSP biquad filter instance (Initialized properly
```

```
            at runtime)
85      };
86
87
88  // ASSUMING THE ADC1_DMA_sort_ptr IS ALREADY INITIALIZED ELSEWHERE IN YOUR CODE
89  extern ADC1_DMA_sort*ADC1_DMA_sort_ptr; // Reinitializes the pointer to the struct responsible
        for sorting data from the DMA buffer to the source buffer
90
91
92  // ASSIGNING THE POINTER TO THE SOURCE BUFFER AT RUNTIME
93  void init_ADC1_IN1_struct(void)
94    {
95      //Ensure that ADC1_DMA_sort_ptr is initialized
96      ADC1_IN1_NF_arg.ADC1_IN1_psrc = (float32_t *)(ADC1_DMA_sort_ptr->ADC1_IN1_bfr);  //
            Typecasting---Converts the data in IN1 data buffer to floating point values. Assigns
            the source buffer at runtime
97    }
98
99
100
101 void init_ADC1_IN1_FO_biquad_filter(void)
102   {
103     arm_biquad_cascade_df2T_init_f32
104     (
105             (arm_biquad_cascade_df2T_instance_f32 *) &ADC1_IN1_NF_arg.S1,  // Pointer to the
                    instance within the struct
106             ADC1_IN1_NF_arg.ADC1_IN1_numstages,                                    // Number
                    of stages (2 in this case)
107             ADC1_IN1_NF_arg.ADC1_IN1_pcoeffs,                                      // Pointer
                     to coefficients array
108             ADC1_IN1_NF_arg.ADC1_IN1_pState                                        // Pointer
                     to the state buffer
109     ); // CMSIS DSP function responsible for initializing the coefficients and variables for
            channel 1
110   }
111
112 void update_ADC1_IN1_FO_biquad_filter(void)
113    {
114     arm_biquad_cascade_df2T_f32
115       (
116           &ADC1_IN1_NF_arg.S1, // Pointer to the instance within
117           ADC1_IN1_NF_arg.ADC1_IN1_psrc, // Pointer to the source buffer
118           ADC1_IN1_NF_arg.ADC1_IN1_pdst, // Pointer to the destination buffer
119           ADC1_IN1_NF_arg.ADC1_IN1_Blocksize // Number of samples to be processed. Equal to
                the source buffer size
120       ); // This is the CMSIS DSP function responsible for implementing the cascaded Direct
```

```
          Form II filter. The arguments are clearly commented
121
122    }
123
124
125
126 void init_ADC1_IN2_struct(void)
127   {
128     //Ensure that ADC1_DMA_sort_ptr is initialized
129     ADC1_IN2_NF_arg.ADC1_IN2_psrc = (float32_t *)(ADC1_DMA_sort_ptr->ADC1_IN2_bfr);  //
          Typecasting---Converts the data in IN1 data buffer to floating point values. Assigns
          the source buffer at runtime
130   }
131
132
133
134 void init_ADC1_IN2_FO_biquad_filter(void)
135   {
136      arm_biquad_cascade_df2T_init_f32
137      (
138             (arm_biquad_cascade_df2T_instance_f32 *) &ADC1_IN2_NF_arg.S1,  // Pointer to the
                   instance within the struct
139             ADC1_IN2_NF_arg.ADC1_IN2_numstages,                                    // Number
                   of stages (2 in this case)
140             ADC1_IN2_NF_arg.ADC1_IN2_pcoeffs,                                      // Pointer
                    to coefficients array
141             ADC1_IN2_NF_arg.ADC1_IN2_pState                                        // Pointer
                    to the state buffer
142      ); // CMSIS DSP function responsible for initializing the coefficients and variables for
           channel 2
143   }
144
145 void update_ADC1_IN2_FO_biquad_filter(void)
146    {
147     arm_biquad_cascade_df2T_f32
148       (
149           &ADC1_IN2_NF_arg.S1, // Pointer to the instance within
150           ADC1_IN2_NF_arg.ADC1_IN2_psrc, // Pointer to the source buffer
151           ADC1_IN2_NF_arg.ADC1_IN2_pdst, // Pointer to the destination buffer
152           ADC1_IN2_NF_arg.ADC1_IN2_Blocksize // Number of samples to be processed. Equal to
                  the source buffer size
153       ); // This is the CMSIS DSP function responsible for implementing the cascaded Direct
           Form II filter. The arguments are clearly commented
154
155    }
```

We have now created the header file and defined the necessary functions for our IIR notch filter in a .c file. The task remaining is to call this functions in the main.c file. Listing 7 shows how I called the functions in the main.c function.

Listing 7: Calling the functions in main.c

```
1
2  /* USER CODE BEGIN Includes */
3  #include <_ADC1_INx_NF.h>
4  /* USER CODE END Includes */
5
6   /* USER CODE BEGIN Init */
7    init_ADC1_IN1_struct(); // Pointer to the source buffer is initiaized at runtime
8    init_ADC1_IN1_FO_biquad_filter(); // Initializes the filter coefficients and variables
9
10   init_ADC1_IN2_struct(); // Pointer to the source buffer is initiaized at runtime
11   init_ADC1_IN2_FO_biquad_filter(); // Initializes the filter coefficients and variables
12
13   /* USER CODE END Init */
14
15    /* Infinite loop */
16   /* USER CODE BEGIN WHILE */
17   while (1)
18   {
19       update_ADC1_IN1_FO_biquad_filter(); // Filters channel 1 data
20       update_ADC1_IN2_FO_biquad_filter(); // Filters channel 2 data
21
22    /* USER CODE END WHILE */
```

Parting Shot

We have designed, simulated and developed an IIR notch filter for an STM32G4 using MATLAB for design and simulation and STM32CubeIDE for development. The part remaining is to provide a proof of concept for the filter which I will be presenting soon. Nonetheless, the article provides a walk through of how to come up with IIR filters without adopting traditional mathematical and programming rigour yet maintaining optimum performance. It comes in handy for anyone whose intent is to develop and implement digital IIR filters in STM32 cortex M4 MCU's. Any edits to the article can be tracked through my public GitHub repository here.