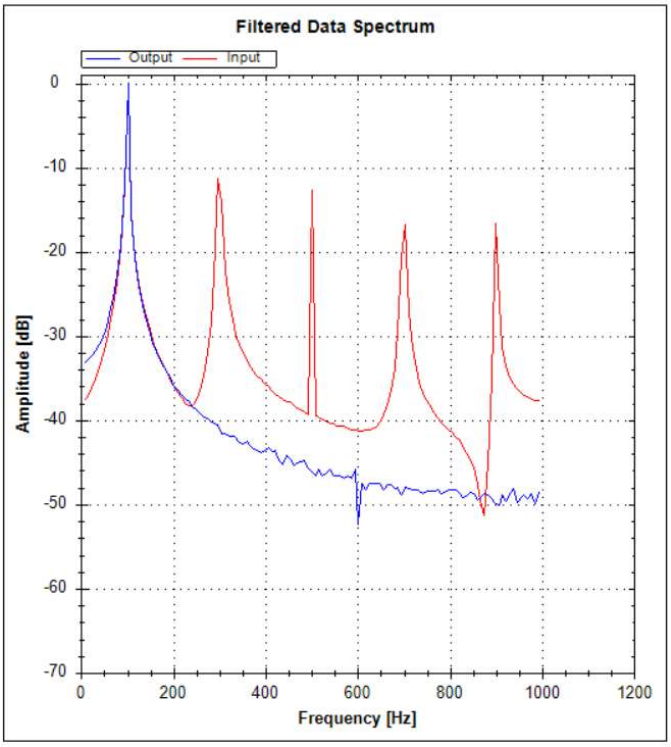
Real-Time Digital Harmonic Extractor with Window-Sinc Filter Designer Tool

Siong Moua
ECE 522 Final Project
December 20, 2021



Saint Cloud State University
Electrical Engineering Department

Acknowledgement

Thank you to Professor Zheng for this superior guidance and knowledge in making this great project available to all his ECE 422/522 students.

Table of Contents

Objective

Design a real-time digital filter system that will allow the user to visualize the input and output signal in both time and frequency domain. Use PIC24FV16KM202 MCU as the filter engine and a C# Windows Form application for the control and data visualization.

System Design Specification

1. PIC24 must be able to digitize a square wave of three different frequencies 10, 50, and 100 Hz.
2. The sampling frequency for the three different square wave input must be 2kHz, 1kHz, and 200 Hz.
3. A C# Window Form GUI application must be developed for data visualization and user control of the system.
4. System must have three different FIR LPF with corner frequency at 10, 50, and 100 Hz.
5. System must have three different FIR BPF with center frequencies at 30, 150, and 300 Hz.
6. Both FIR filter types must have a minimum of 64 taps (coefficients).
7. Both FIR filter type must be 16-bits.
8. The C# GUI application must allow the user to download new filter coefficients to PIC24 in real-time.
9. The C# GUI application must display both the sampled data and filtered data on the same graph.
10. The C# GUI application must display both the spectrum of the sampled data and filtered data on the same graph.
    Note: System must use FFT or DFT to get spectrum of the sampled/filtered data.
11. All axis for all the graphs used must be labeled with the correct and proper units (time, frequency, voltage, amplitude, etc.).
12. The C# GUI must have a combo box to allow the user to select different sampling frequency.
13. The C# GUI must have different textbox to display maximum voltage, minimum voltage, and frequency.
    Note: These measurements are for the filtered data
14. The C# GUI application must have different trackbars for control over triggering, shifting left/right, zooming in/out, and enlarging up/down.
15. The C# GUI application must also have a start and stop button.
16. Additional features can be added.

Materials Required

1. PIC24FV16KM202
2. CCS C compiler
3. Visual Studio (Windows Form Application .Net Framework)
4. Bluetooth module RN42

5. ICD-U64 programmer
6. Function generator
7. Power supply (5V)

Procedures

1. Understand how convolution works by reading the content of Appendix E. This is important as a circular buffer is used in this project to implement the convolution between the sampled data and the filter coefficients. The mathematics performed is what produces the filtering of the data.
2. Real and understand Appendix D on why and how normalization is performed on the output filtered data.
3. Read and understand Appendix A, B, and C. These three sections explained some very important code key features used in both the PIC24 MCU and the C# GUI application.
4. Suggested but not required to read Appendix F on how the filter designer tool works.
5. Build the circuit as shown in Figure 2.
6. Open CCS C compiler and create a new project targeting PIC24FV16KM202 with an internal oscillator at 32 MHz.
7. Utilize the knowledge gained from studying the important given code and as well as the suggested reading above to help develop the required system based on the specifications.
8. Open Visual Studio and create a new project.
9. Utilize the provided code and visual content of this document and create the required control system.

Note: *If needed please contact me directly for the complete code at smoua@stcloudstate.edu*

Design Methods

In this design PIC24FV16KM202 is operating with a clock frequency of 32 MHz. The PIC24 holds the job as a real time filter engine and act as a slave to a C# GUI application that is running on a laptop. At the request of the C# GUI application for more data, PIC24 will gather 256 of its sampled data, and 256 of its filtered output data and send it over to the C# GUI application. The PIC24 uses a RN42 Bluetooth module with the UART (Universal Asynchronous Receiver Transmitter) protocol with a baud rate of 115200 bits/second to send its two arrays of data to the C# GUI application. Figure 1 below shows the overall system diagram.

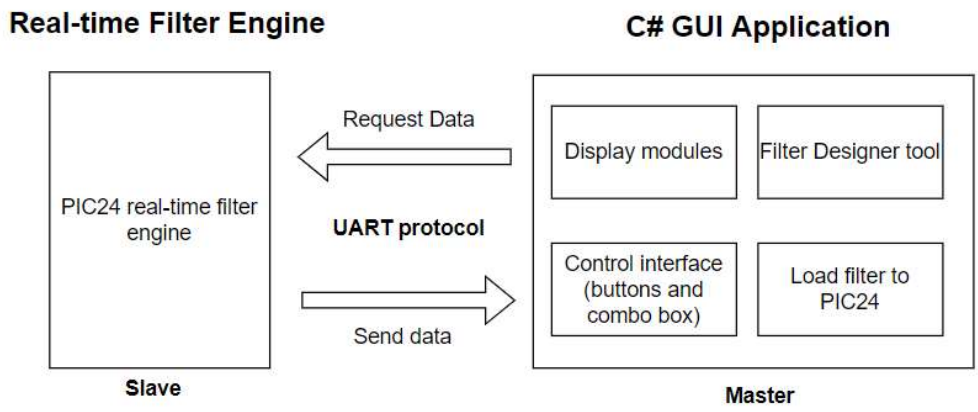


Figure 1: Top system block diagram.

Notice that the C# GUI application serve as a master to the PIC24 and as a control and data visualization interface for the user. The user can use the C# GUI application to control the different features of the system such as stopping, starting, changing trigger value, changing sampling frequency, zooming in/out, enlarging up/down, and shifting left/right.

One very nice feature of the C# GUI application is its ability to design FIR filter in real-time using the Window-Sinc method. This Window-Sinc Filter Designer Tool allows the user to analyze the designed filter performance by plotting the designed filter impulse response to a ZedGraph. It will also generate the required coefficients and allow the user to download it in real time to PIC24 for implementation. The different filters that this Window-Since Filter Designer Tool can design is LPF (low pass filter), HPF (high pass filter), and BPF (bandpass filter). Detailed description of these features will be presented later.

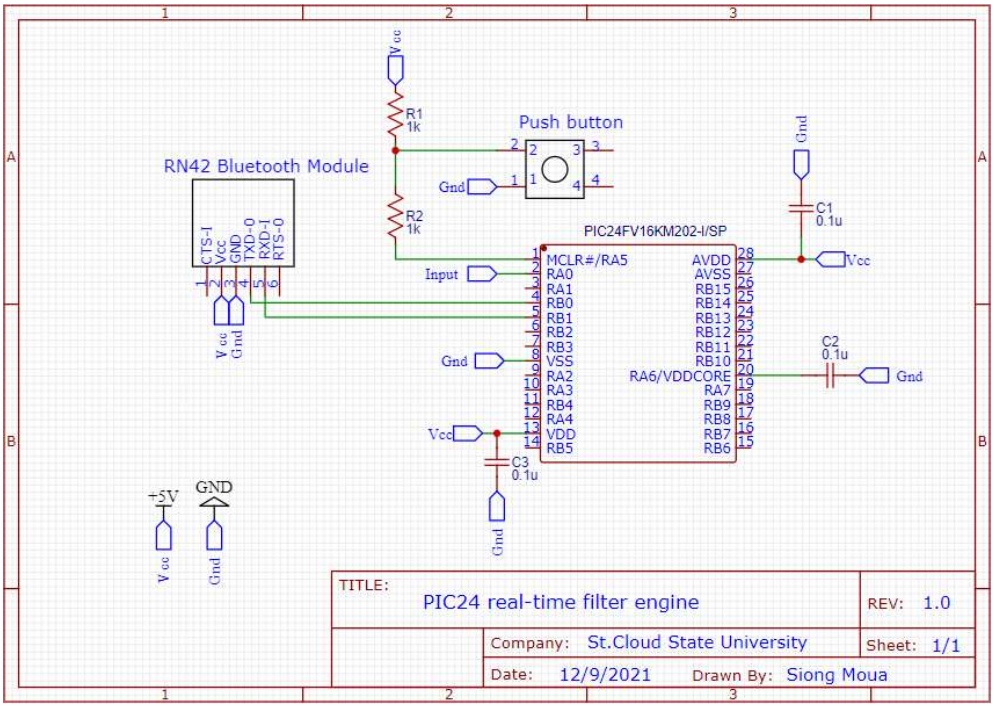


Figure 2: Hardware layout for the PIC24 real-time filter engine module.

Shown above in Figure 2 is the hardware layout for the PIC24FV16KM202 module.

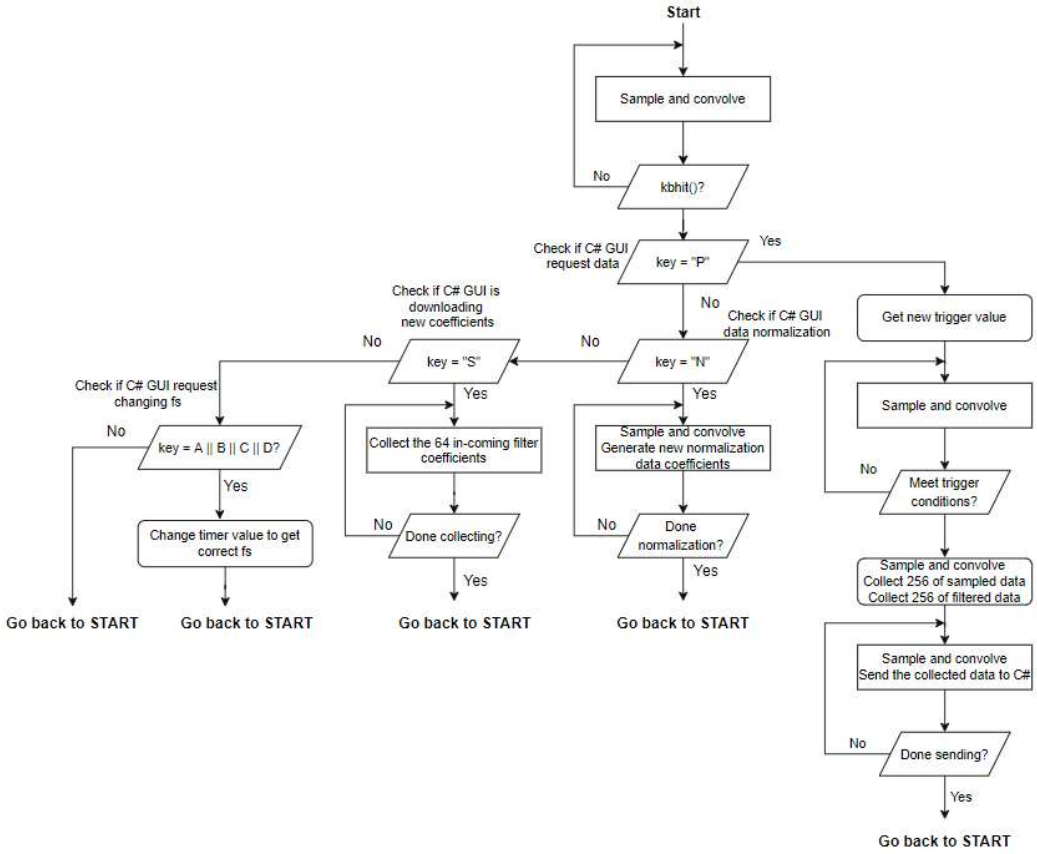


Figure 3: Flow chart of logic implemented in PIC24.

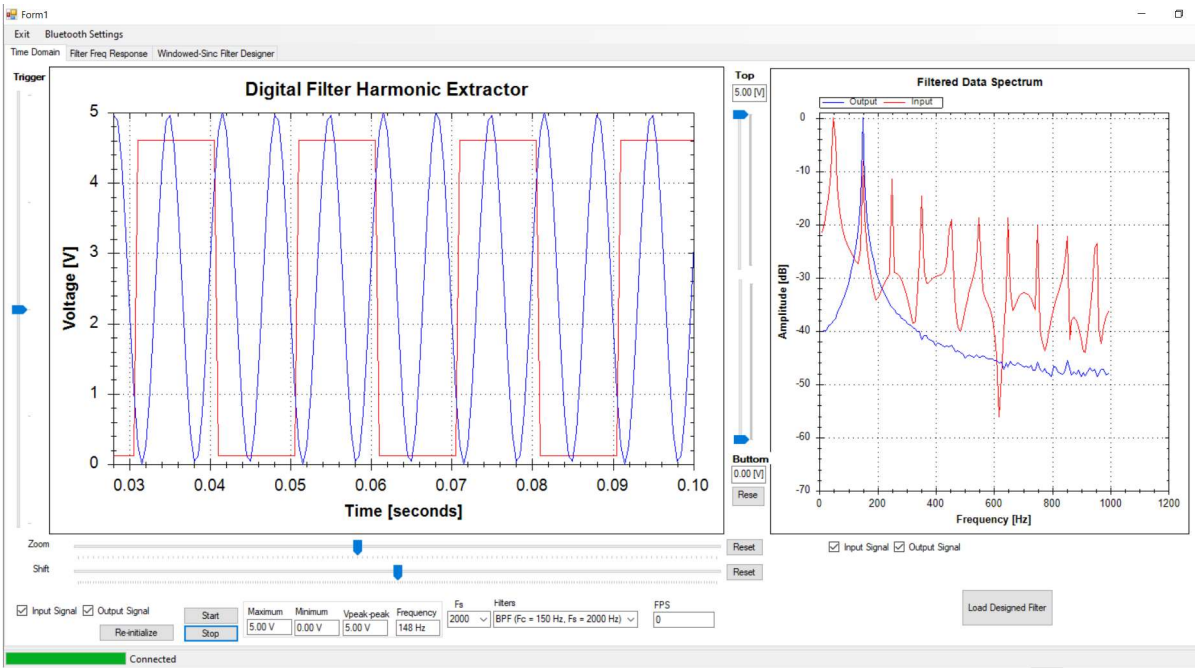


Figure 3: The designed C# GUI application used for control and data visualization.

The C# GUI application was designed using Visual Studio Windows Form Application .Net Framework. Again, the purpose of this C# GUI application shown in Figure 3 is for data visualization and control of the system. It have two ZedGraph display grides, one for displaying both the sampled data signal and the filtered data signal. The second ZedGraph on the right is to display both the frequency spectrum of the sampled input signal and the filtered data signal.

At the bottom of this C# GUI application, it also has text boxes showing the frequency of the filtered signal, maximum and minimum of the filtered signal, peak-to-peak voltage of the filtered signal, and the FPS (frame per second, display rate of the system).

It must also be noted that there are 6 pre-designed FIR filter coefficients. These 6 pre-designed FIR filters were designed using MATLAB filter designer tool. This mean that the user can select any of these 6 pre-designed FIR filters coefficients and download it to PIC24 without having to design it again. These 6 FIR filters has a dedicated window tap called 'Filter Freq Response' which can be seen at the top left corner of Figure 3. This tap allow the user to visualize the impulse response the different pre-designed FIR filters in real-time. The different visualization of these 6 filters will soon be discussed.

Table 1: The 6 pre-designed FIR filters and its design parameters

Filter number	Type of filters	Design parameters	Design method
1	LPF	Fc = 100 Hz, Fs = 2 kHz	Hamming
2	BPF	Fc1 = 280 Hz, Fc2 = 320 Hz, Fs = 2 kHz	Gaussian
3	LPF	Fc = 50 Hz, Fs = 2 kHz	Haning
4	BPF	Fc1 = 145 Hz, Fc2 = 155 Hz, Fs = 2 kHz	Blackman
5	LPF	Fc = 10 Hz, Fs = 2 kHz	Hamming
6	BPF	Fc1 = 25 Hz, Fc2 = 35 Hz, Fs = 200 Hz	Blackman

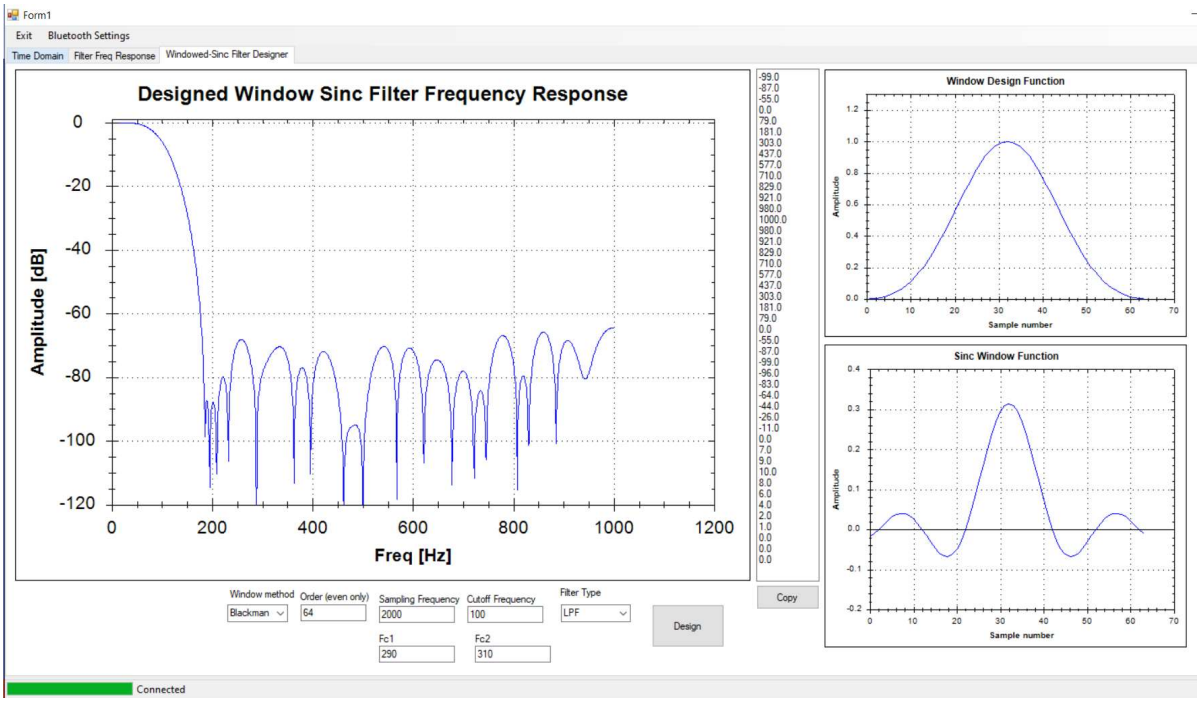


Figure 4: Window-since filter designer application for coefficient generation.

In Figure 4 shows the filter designer tool that was developed and embedded within the C# GUI application. This filter designer tool uses the window-sinc filter design method which allows the user to design different type of FIR filters in real time. The user can also select different type of window methods (Blackman, Hamming, Haning, and No window), order, filter types (LPF, HPF, and BPF), and cutoff frequency. This designer tool also allow the user to visualize and analyze the performance of the filters by plotting its impulse response on a ZedGraph. This tool will also allow the generated coefficients to be download to PIC24 for implementation. For the mathematic theory of how this window-sinc filter designer method work please refer to Appendix F.

Results and Analysis

In this section the follow results and performance of the designed system will be discussed in detail.

It is very important to point out that the input signal used to test the performance of this system is all square wave signal with amplitude of 5V and frequency of (10 Hz, 50 Hz, and or 100 Hz). And all the 6 pre-designed LPF/BPF was designed using MATLAB filter designer tool with 64 orders with fixed-point coefficients (16-bit numerator word length and 15-bit numerator facture length).

Table 2: Handshake protocol between PIC24 and C# GUI

Control Character from C# GUI to PIC24	Operation and meaning
0	Trigger: min = 0 and max = 50
1	Trigger: min = 50 and max = 150
2	Trigger: min = 100 and max = 150
3	Trigger: min = 150 and max = 200
4	Trigger: min = 200 and max = 250
P	C# requesting data from PIC24
N	C# requesting PIC24 to normalize its data
S	Notify PIC that the next incoming data is an array of coefficients
E	The received coefficient is negative
D	The received coefficient is positive
A	Sampling frequency = 200 Hz
B	Sampling frequency = 1 kHz
C	Sampling frequency = 2 kHz

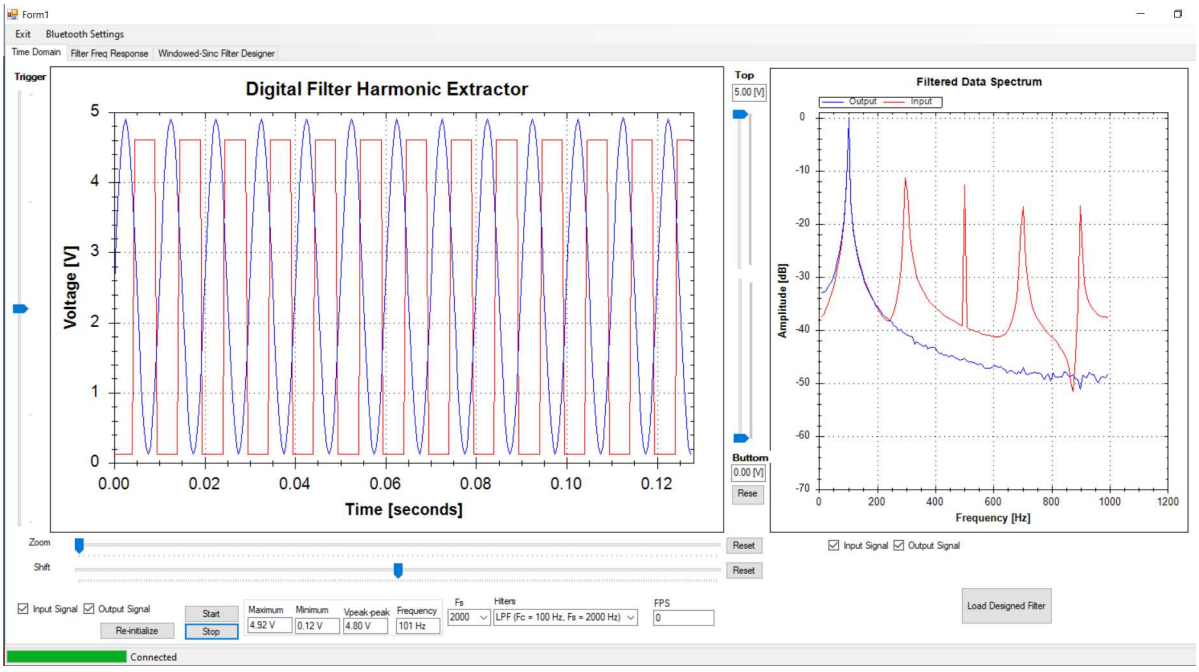


Figure 5: Digitize a 100 Hz square wave signal and use the pre-designed LPF filter (filter number 1 of Table 1) to extract the 1st harmonic (100 Hz).

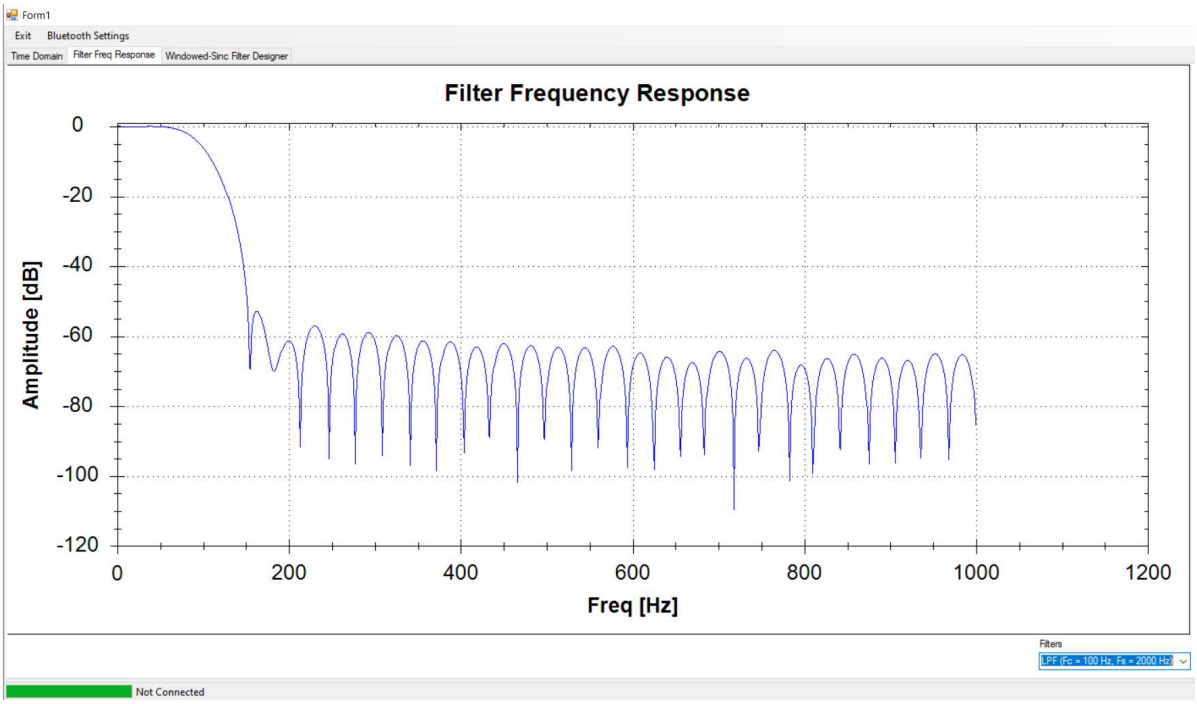


Figure 6: The impulse response of the pre-designed LPF (filter number 1 of Table 1) with Fc = 100 Hz and Fs = 2kHz.

The LPF used to produce the results shown in Figure 5 have a cutoff frequency of 100 Hz and sampling frequency of 2 kHz as presented earlier in Table 1. This LPF uses the window hamming method to generate its coefficients. The impulse response of this filter is shown above in Figure 6.

This LPF filter was selected for this application since at the third harmonic (300 Hz) the amplitude is attenuated down to about -60 dB. This mean that the magnitude of the third harmonic is attenuated to 1/1000 of its original amplitude. Table 3 below listed some of the magnitude and its corresponding dB scale values.

In theory at the third harmonic (300 Hz) the amplitude should be attenuated down to -60 dB but based on the result of Figure 5 it only gets attenuated to about -42 dB. This mean that in practice it only gets attenuated down to about 7/1000 of its original amplitude. This is still a good attenuation as the other harmonics including the third harmonic no longer exist in the time plot (meaning only a pure sine wave of 100 Hz is seen). This as well can be seen in the frequency plot as shown in Figure 5 (meaning no other harmonic is observed other than the 100 Hz harmonic).

Table 3: Magnitude and dB scale

Magnitude [unit less]	dB
1	0
.707	-3
.501	-6

.316	-10
.1	-20
.031	-30
.01	-40
.003	-50
.001	-60
.0003	-70
.00009	-80
.00003	-90
.000009	-100
.000003	-110
.000001	-120

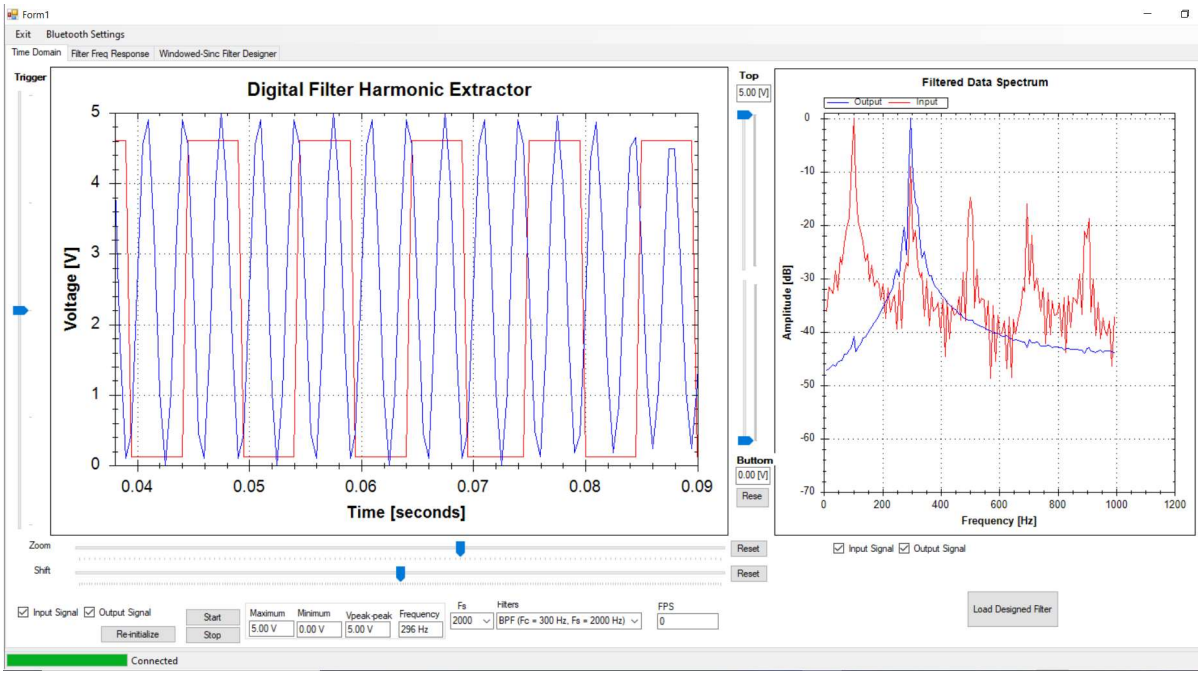


Figure 7: Digitize a 100 Hz input square wave signal and use the pre-designed BPF filter (filter number 2 of Table 1) to extract the 3rd harmonic (300 Hz).

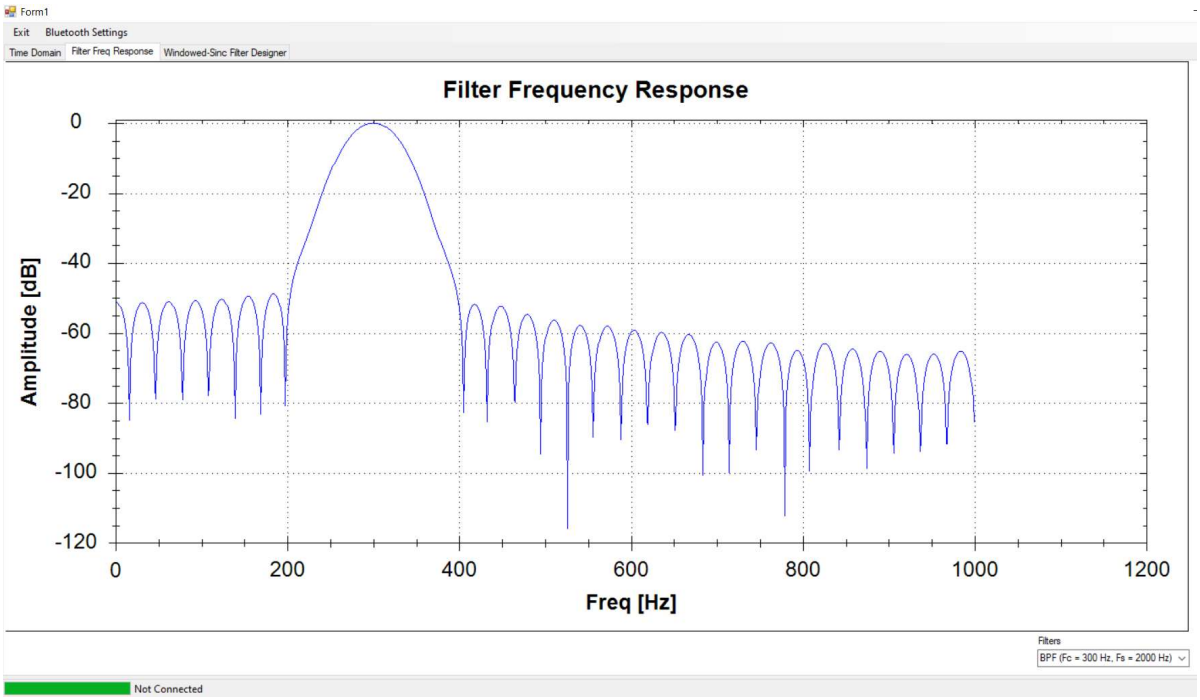


Figure 8: The impulse response of the pre-designed BPF (filter number 2 of Table 1) to extract third harmonic of the 100 Hz input square wave signal.

In Figure 7 a BPF is used to filter the 100 Hz square wave input signal and attempt to extract the 3rd harmonic (300 Hz). The BPF uses the Window Gaussian design method to generate the required 64 coefficients. The impulse response of this BPF is shown in Figure 8. This filter was selected as the candidate for this job as it has an attenuation of at least -50 dB at its stop band. With reference to Table 3 -50 dB at the stop bands mean that all other harmonic except for the 3rd harmonic gets attenuated down to at least 3/1000 of its original amplitude.

In practice the stop band attenuation at the nearest harmonics (1st and 5th harmonic) have at most an amplitude of about -38 dB. Though it is still good enough to extract and filtered to show only the 3rd harmonics as shown in Figure 7 of the time domain plot.

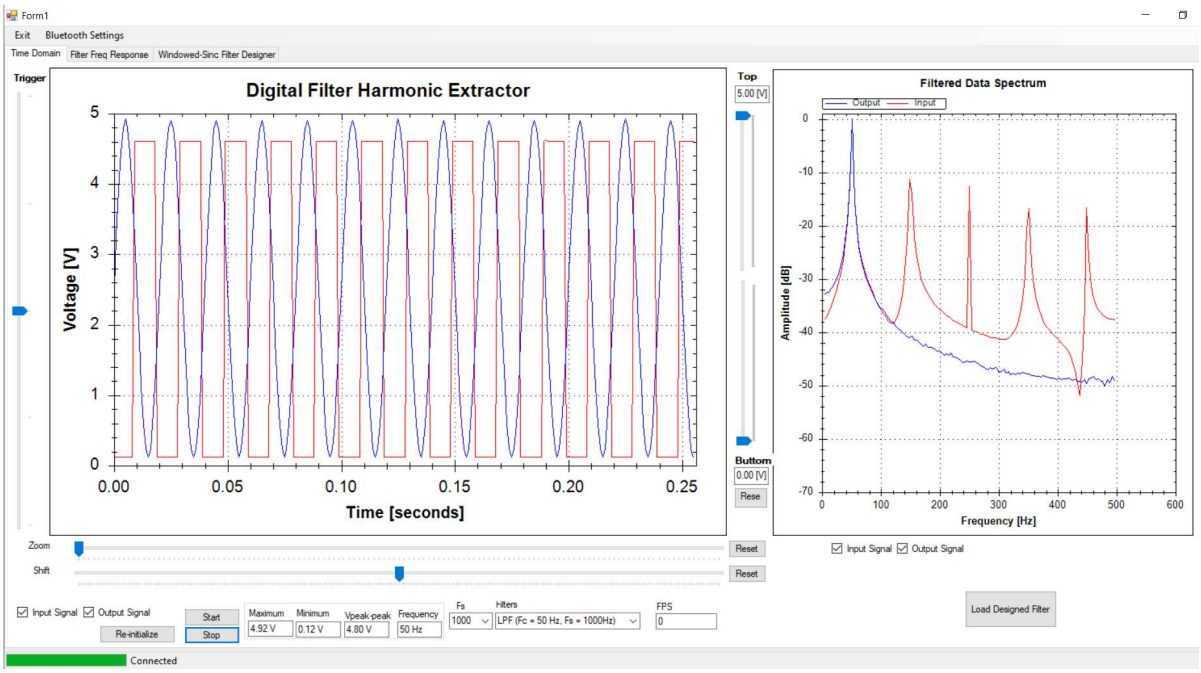


Figure 9: Digitized a 50 Hz square wave input signal and use the pre-designed LPF filter (filter number 3 of Table 1) to extract the 1st harmonic (50 Hz).

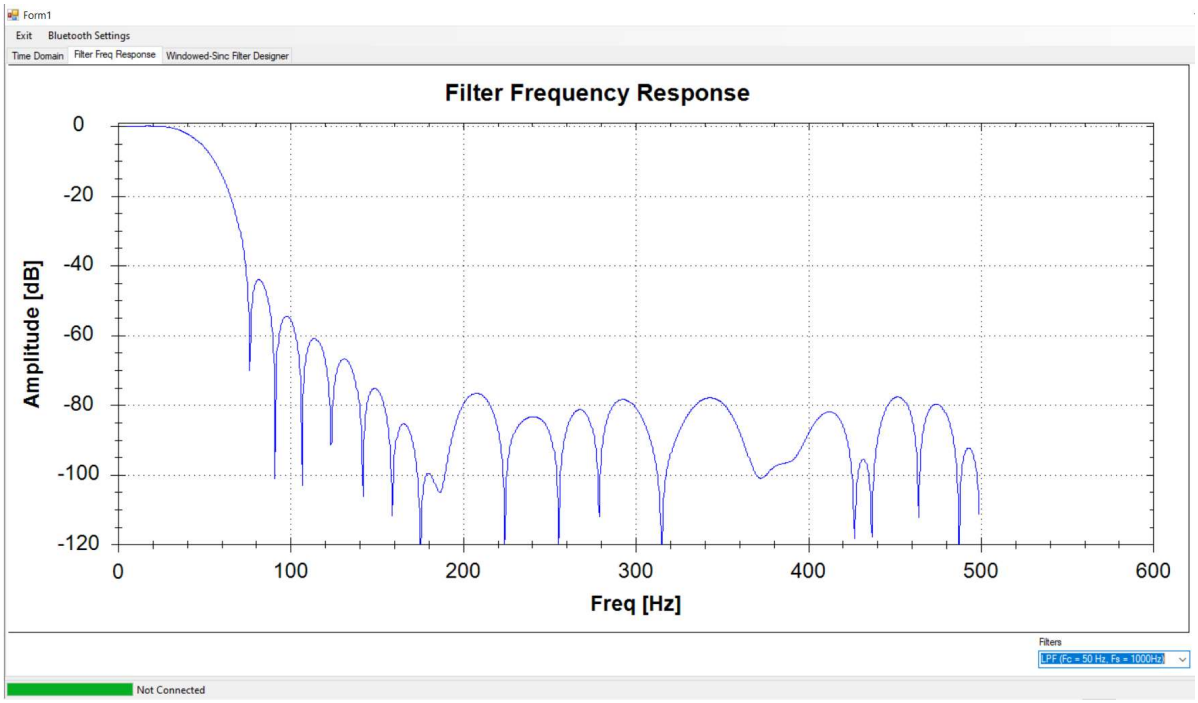


Figure 10: The impulse response of the pre-designed LPF (filter number 3 of Table 1) to extract the 1st harmonic of the 50 Hz input square wave signal.

In Figure 9 a LPF filter is used to extract the 1st harmonic of a 50 Hz input square wave signal. The LPF was designed using the Window Haning method with a cutoff frequency at 50 Hz and a sampling frequency of 1 kHz. The impulse response of this LPF is shown in Figure 10. Looking

at the impulse response of this LPF and knowing that the input frequency of the square wave will be 50 Hz, the next harmonic is at 150 Hz (3rd harmonic). At the 3rd harmonic the attenuation is about -57.5 dB, which mean that the 3rd harmonic amplitude will get attenuated down to .0013 of its original amplitude after filtering. When applying this LPF the actual attenuation at the 3rd harmonic is about -42 dB, and this can be seen in Figure 9 on the right frequency spectrum plot.

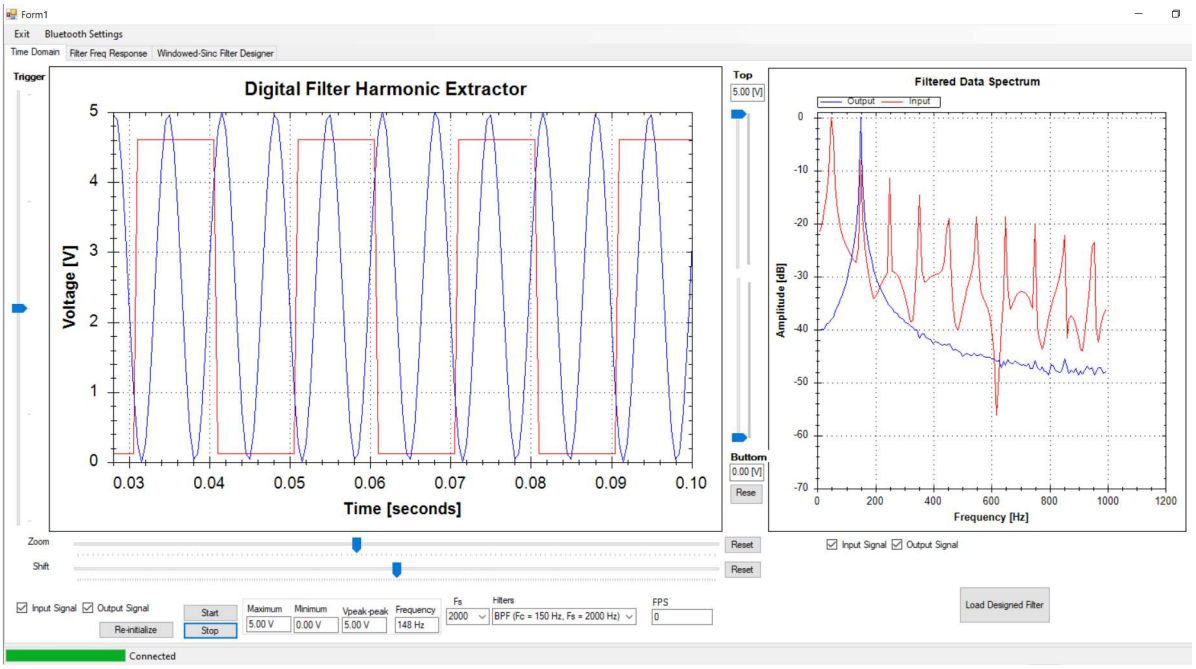


Figure 11: Digitize a 50 Hz square wave signal and use the pre-designed BPF (filter number 4 of Table 1) to extract third harmonic (150 Hz).

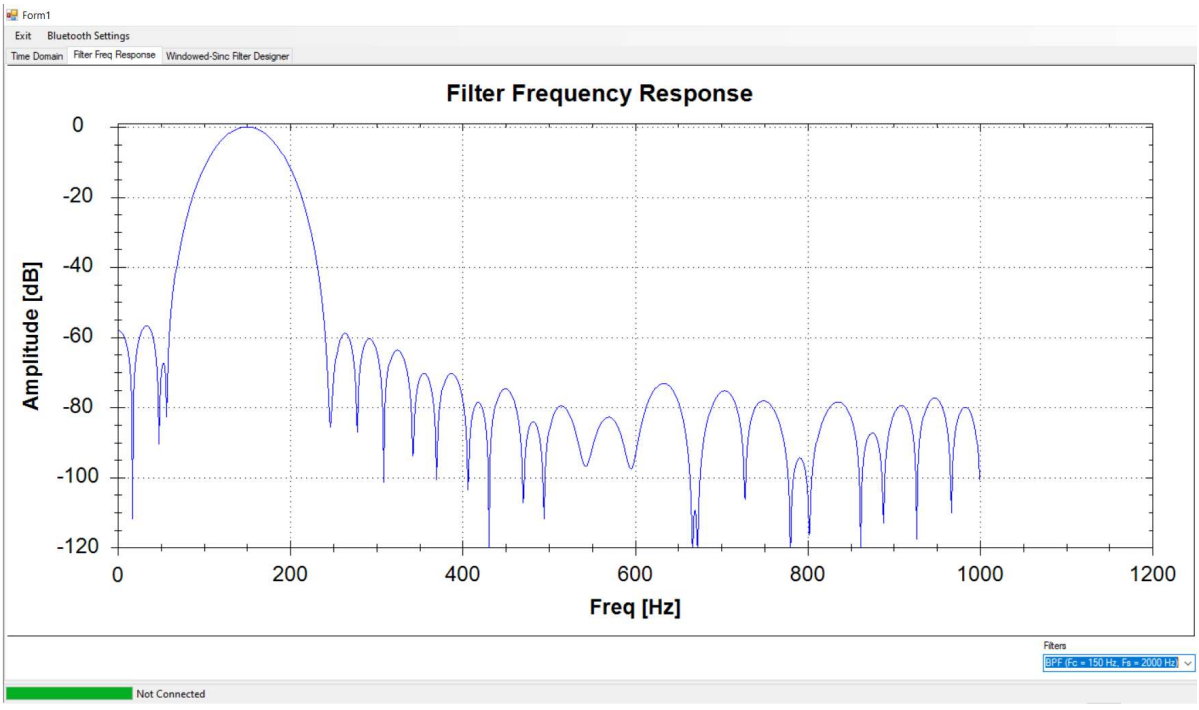


Figure 12: The impulse response of the pre-designed BPF (filter number 4 of Table 1) to extract the 3rd harmonic of the 50 Hz input square wave signal.

In Figure 11 a BPF is used to extract the 3rd harmonic of a 50 Hz input square wave signal. This mean that the 3rd harmonic is located at 150 Hz. This filter uses the Window Blackman design method. The impulse response of this BPF is shown in Figure 12 above. The attenuation at the stop bands is about -55 dB. This mean that after filtering the 50 Hz input square wave with this BPF filter, all other harmonics except for the 3rd harmonics will get attenuated down to .0017 of its original amplitude. But looking at the filter in action as shown in Figure 11, the stopband attenuation is only at -40 dB. Even though the performance is not as the theory suggested, the attenuation is still good enough to fully extract the 3rd harmonic (150 Hz) and attenuate all other harmonics to where it cannot be seen anymore.

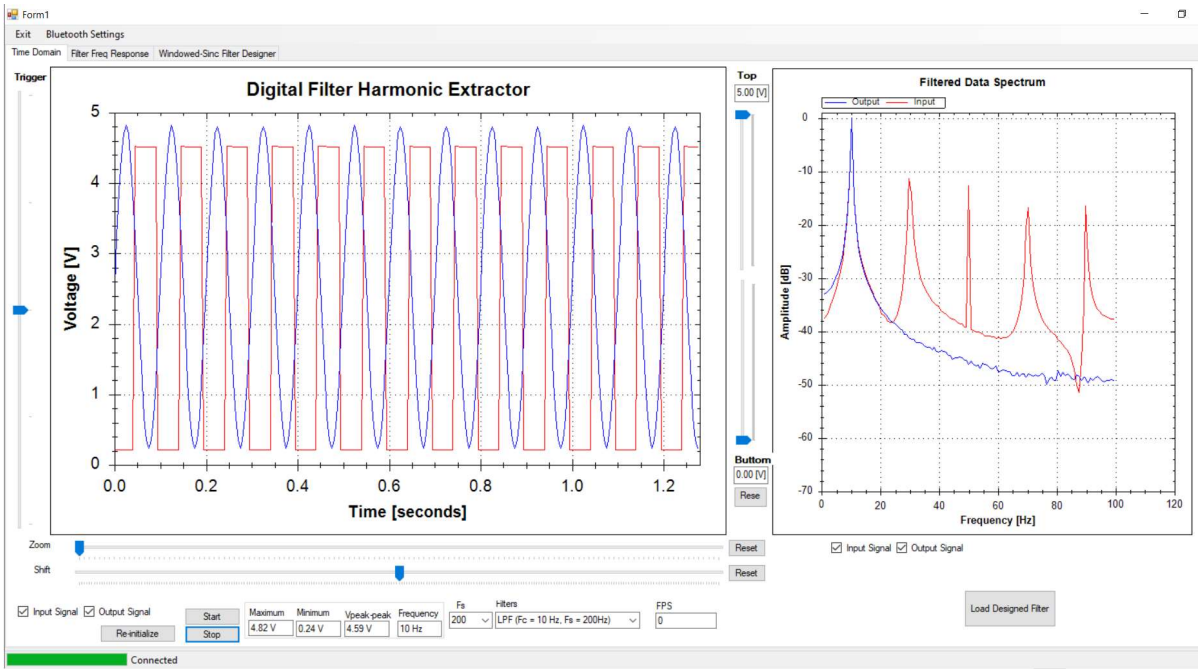


Figure 13: Digitize a 10 Hz square wave signal and use the pre-designed LPF (filter number 5 of Table 1) to extract first harmonic.

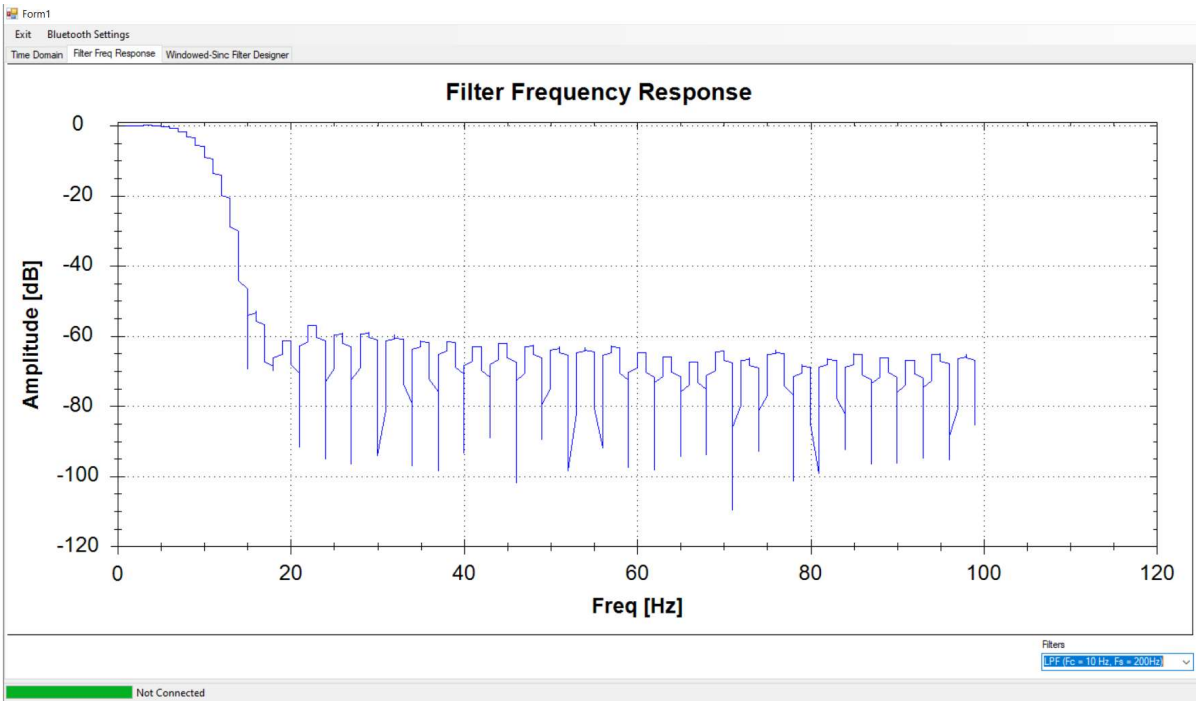


Figure 13: The impulse response of the pre-designed LPF (filter number 5 of Table 1) to extract the 1st harmonic of a 10 Hz input square wave signal.

In Figure 12 a LPF is used to extract the first harmonic of a 10 Hz square wave input signal. This mean that after filtering the output should be a sine wave of 10 Hz. This LPF was designed using the Window Hamming design method with a cutoff frequency at 10 Hz and sampling frequency of 200 Hz.

The impulse response of this LPF is shown in Figure 13. Note that the attenuation for all other harmonics is at least -60 dB. In practice as shown in Figure 13 where this LPF is being used, it only have an attenuation of -42 dB. With this attenuation it is enough to filter out all other harmonics and left only the 1st harmonic visible.

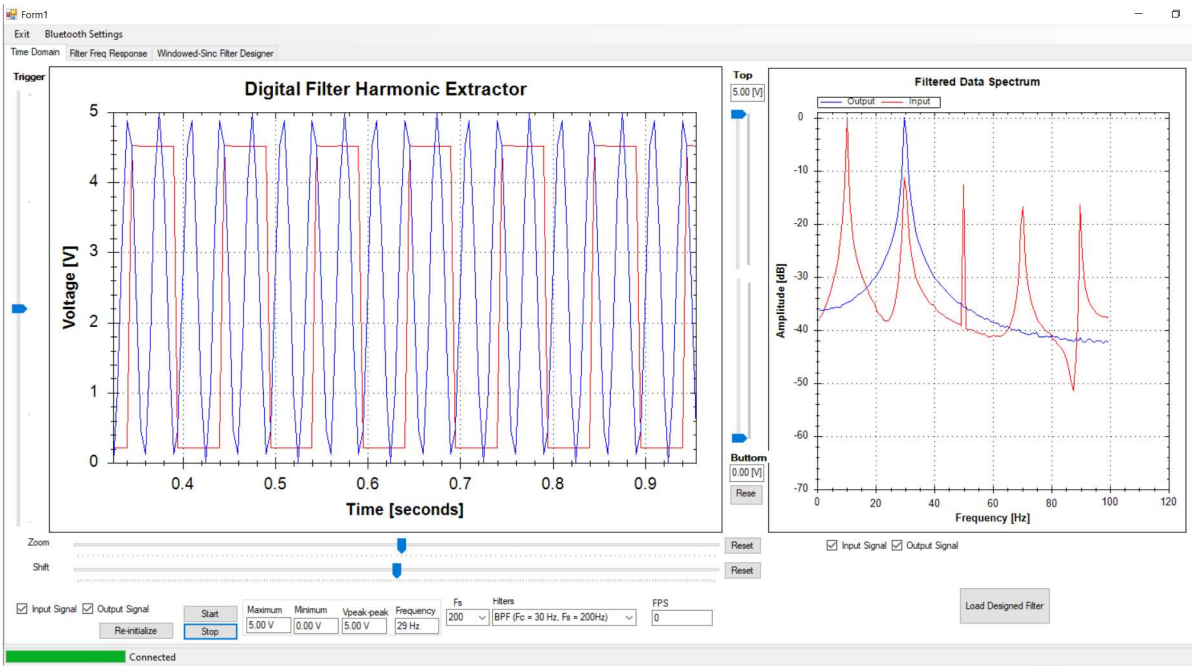


Figure 14: Digitize a 10 Hz square wave signal and use the pre-designed BPF (filter number 6 of Table 1) to extract third harmonic.

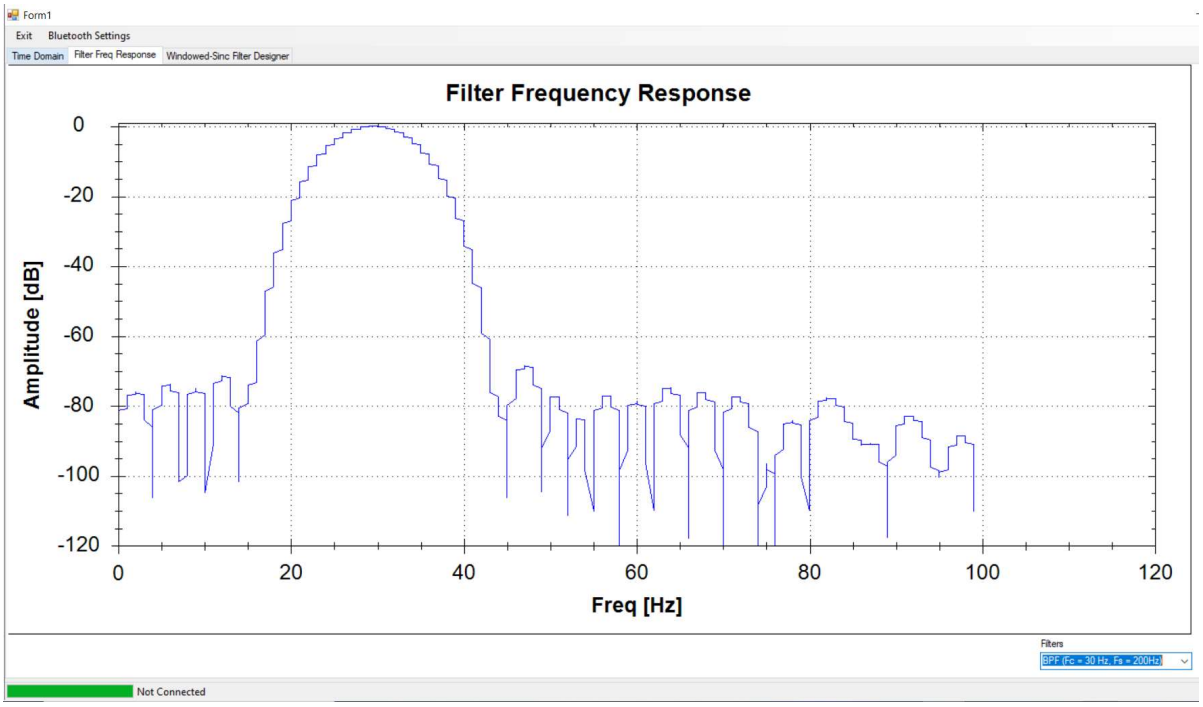


Figure 15: The impulse response of the pre-designed LPF (filter number 2 of Table 1) to extract the 3rd harmonic of a 10 Hz input square wave signal.

If the task is to extract the 3rd harmonic of a 10 Hz square wave signal, then it mean that a BPF centered at 30 Hz is needed. This is exactly what Figure 14 is showing. A BPF was designed using the Window Black design method to have a center frequency of 30 Hz and sampling frequency of 200 Hz. By looking at the filtered data frequency spectrum on the right plot of Figure 14 it is obvious that all harmonics except for the 30 Hz get attenuated down to at least -38 dB. Though the impulse response of this BPF is shown in Figure 15 and suggesting that it should get attenuated to at least -68 dB at the 5th harmonic.

Table 4: Summary of the 6-predesigned filter performance in theory and practice

Filter number	Type of filters	Design parameters	Design method	Theory Attenuation [dB]	Actual Attenuation [dB]
1	LPF	Fc = 100 Hz, Fs = 2 kHz	Hamming	-60	-42
2	BPF	Fc1 = 280 Hz, Fc2 = 320 Hz, Fs = 2 kHz	Gaussian	-50	-38
3	LPF	Fc = 50 Hz, Fs = 2 kHz	Haning	-57.5	-42
4	BPF	Fc1 = 145 Hz, Fc2 = 155 Hz, Fs = 2 kHz	Blackman	-55	-40
5	LPF	Fc = 10 Hz, Fs = 2 kHz	Hamming	-60	-42
6	BPF	Fc1 = 25 Hz, Fc2 = 35 Hz, Fs = 200 Hz	Blackman	-68	-38

A summary of the performance of the 6-predesigned filter is shown in Table 4 above. It can be concluded that the performance in theory did not match of the actual performance, and this is true for all the 6 filters. Even without problem, the attenuation of other harmonics is still good enough where the other harmonics is not present at the filtered output data.

Filter Designer tool and its implementation

In this section the developed Window-Since Filter Designer Tool embedded within the C# GUI application is used to generate different filters and coefficients to be downloaded to PIC24 for implementation. Therefore, the following analysis will cover the performance of 2 real-time designed FIR filters. Listed below in Table 3 is the two designed FIR filters and its design parameters using the Window-Sinc Filter Designer Tool.

Table 5: The designed filter in real-time and its parameters.

Filter number	Type of filters	Design parameters	Design method
1	LPF	fc = 100 Hz, fs = 2 kHz	Hanning
2	BPF	fc1 = 290 Hz, fc2 = 310, fs = 2 kHz	Blackman

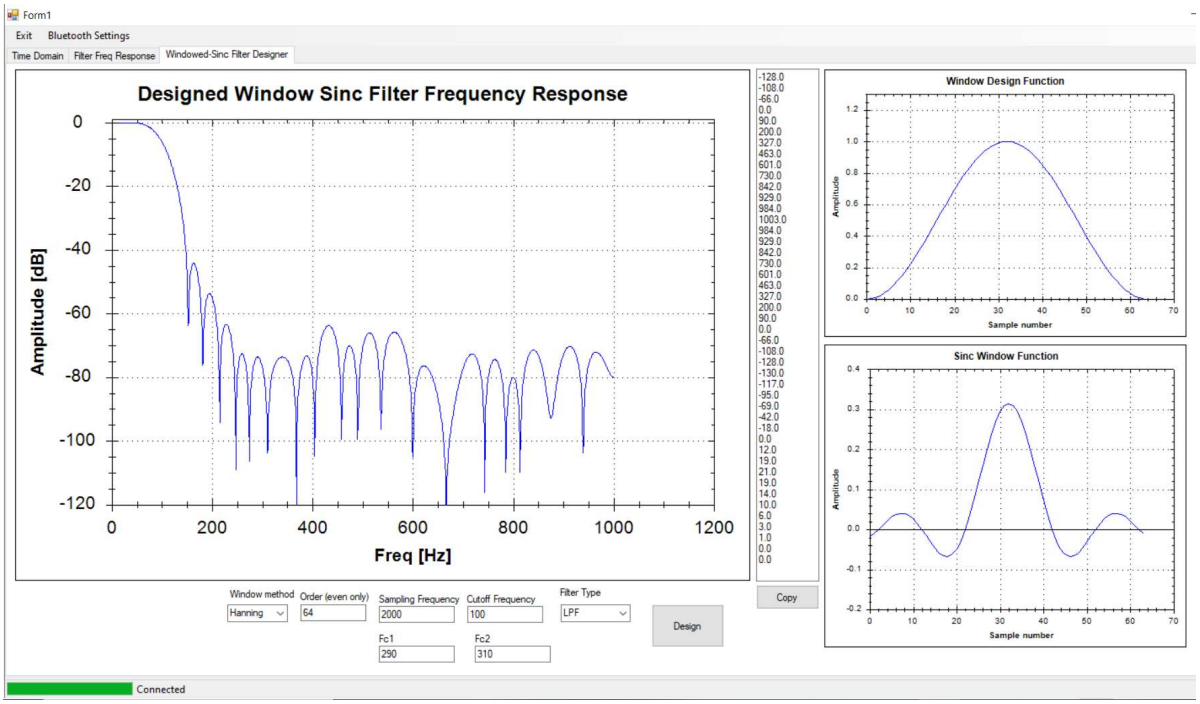


Figure 16: Impulse response of the design LPF (filter number 1 in Table 3).

The first filter designed using the Window-Sinc Filter Designer tool is a LPF (filter number 1 in Table 3). This LPF was designed as shown above in Figure 16 using the Hanning window method with a cutoff frequency at 100 Hz and sampling frequency of 2 kHz. The performance of this LPF can be analyzed by looking at the impulse response as shown in Figure 16. All other harmonics other than the 100 Hz will have an amplitude attenuation of at least -65 dB.

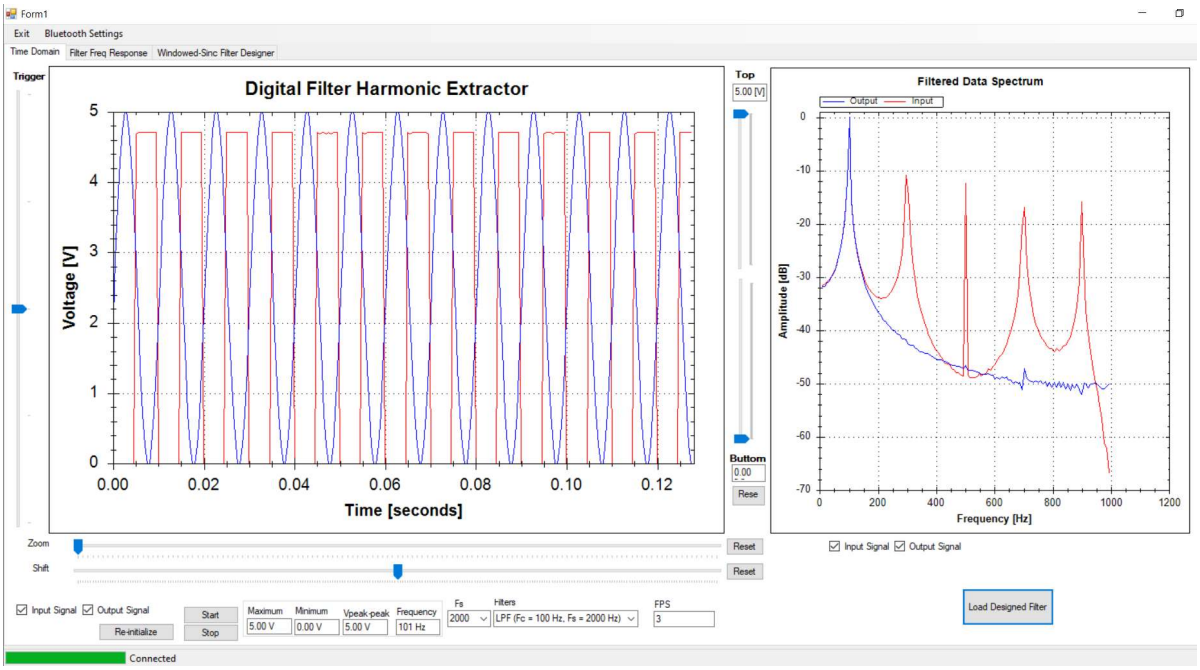


Figure 17: Implementation of the designed LPF (filter number 1 in Table 3).

The implementation of the designed LPF (filter number 1 in Table 5) using the developed Window-Sinc Filter Designer Tool is as shown in Figure 17. Looking at the frequency spectrum of the filtered data shown in Figure 17 all other harmonics other than the 1st (100 Hz) gets attenuated down to about -42 dB.

Even though the performance of the designed filter compared to its application result didn't match, the filtering is still good enough to only extract the 1st harmonic while attenuated all other harmonics.

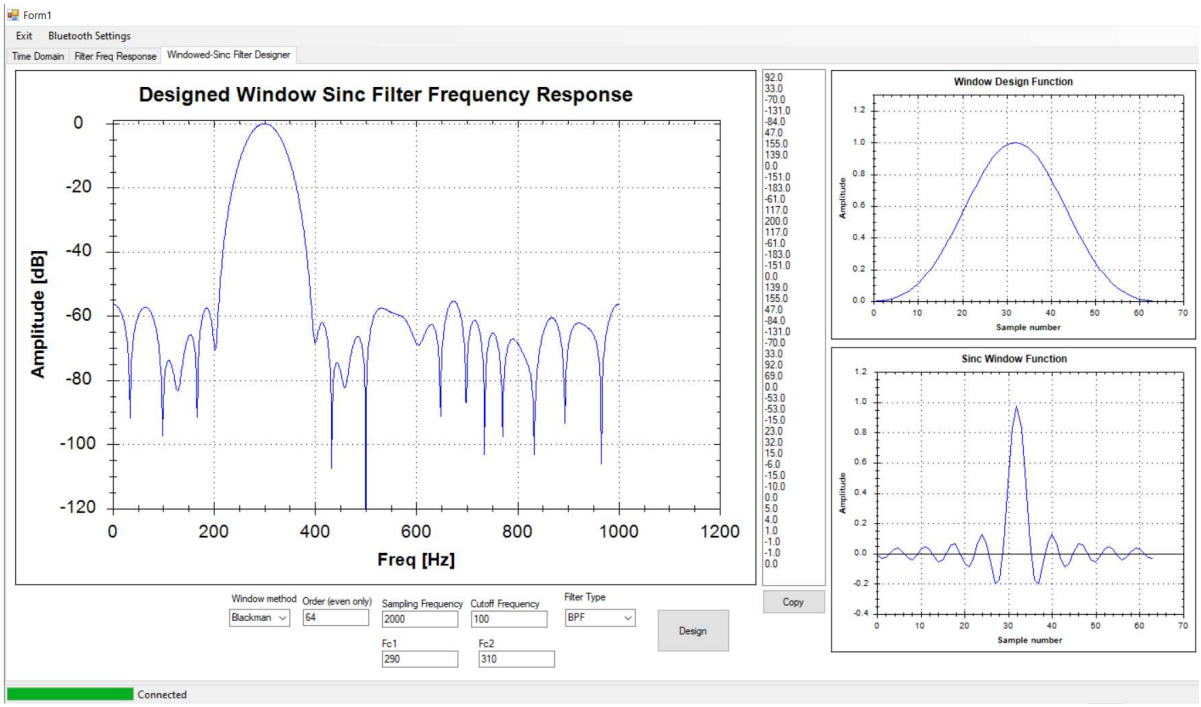


Figure 18: Impulse response of the design LPF (filter number 2 in Table 3).

The second filter designed using the window-sinc designer filter tool is a BPF (filter number 2 in Table 3). This BPF was designed as shown above in Figure 18 using the Blackman window method with a center frequency at 300 Hz and sampling frequency of 2 kHz. The performance of this BPF can be analyzed by looking at the impulse response as shown in Figure 18. All other harmonics other than the 300 Hz will have an amplitude attenuation of at least -55 dB.

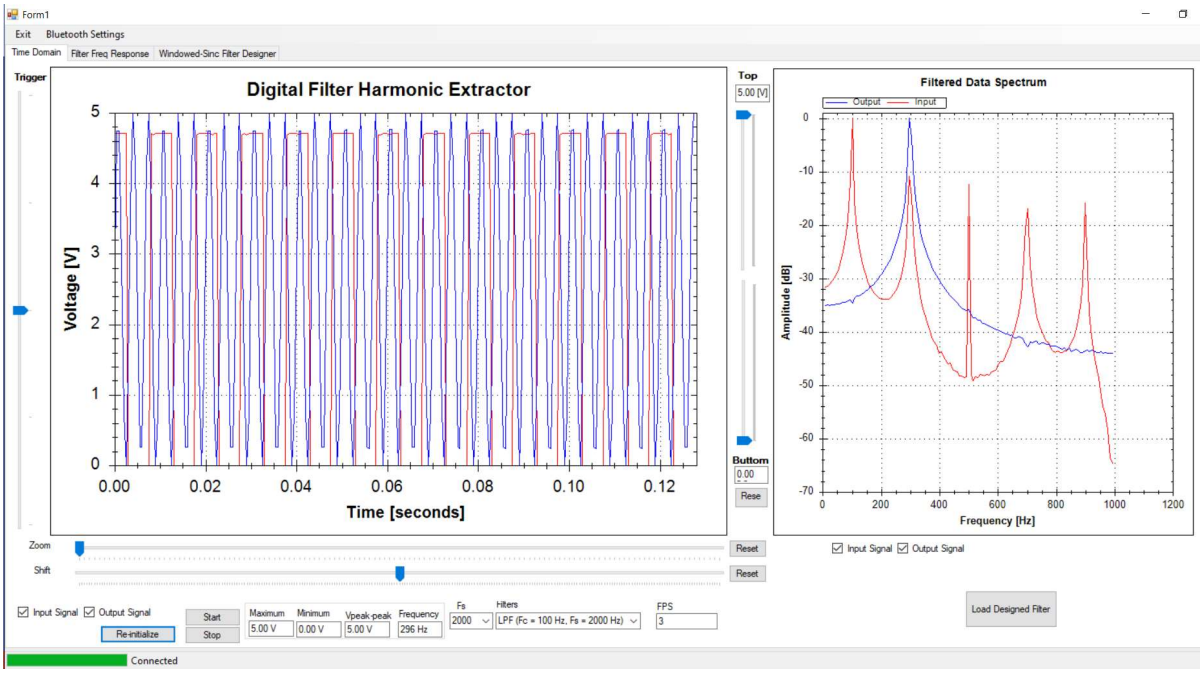


Figure 19: Implementation of the designed BPF (filter number 2 in Table 3).

The implementation of the designed BPF (filter number 2 in Table 3) using the developed window-sinc filter designer tool is as shown in Figure 19. Looking at the frequency spectrum of the filtered data shown in Figure 19 all other harmonics other than the 3st (300 Hz) gets attenuated down to about -35 dB. Even though the performance of the designed filter compared to its application result didn't match, the filtering is still good enough to only extract the 3rd harmonic while attenuated all other harmonics.

Table 6: The designed filter in real-time and its parameters.

Filter number	Type of filters	Design parameters	Design method	Theory Attenuation [dB]	Actual Attenuation [dB]
1	LPF	fc = 100 Hz, fs = 2 kHz	Hanning	-65	-42
2	BPF	fc1 = 290 Hz, fc2 = 310, fs = 2 kHz	Blackman	-55	-35

Show in Table 6 is the summary of the performance of the two filters designed using the developed C# window-sinc filter designer tool. It can be concluded that the theory attenuation for both filters are at least -55 dB, but in actual implementation the worst-case attenuation is -35 dB. Even though -35 is not as the theory suggested, it is still good enough attenuation to extract the desired harmonic.

Discussion

In this project the designed system consists of PIC24 serving as a slave to a C# GUI application running on a laptop capable of Bluetooth connectivity. PIC24 will quantize an input square wave signal and filters the sampled data in real-time. At the request of the C# GUI application (the

master) PIC24 (the slave) will capture a portion of the data it is processing and sent it over via a RN42 Bluetooth module. The C# GUI application also can load new filters coefficients to PIC24. Mainly the C# GUI application is used for data visualization and user control of the system.

The developed system met all the required design specifications. This mean that the PIC24 module can change its sampling frequency depending on the command given from the C# GUI application. PIC24 also can accept new filter coefficients from the C# GUI application. The user also have control over the triggering point of the filtered data. The user can also control features such as zoom in/out, zoom enlargement, shifting left/right, and starting/stopping.

The display interface in the C# GUI application also have textboxes displaying maximum voltage, minimum voltage, frequency, and peak-to-peak voltage. The system also have drop down combo boxes allowing the user to select the desired sampling frequency, and as well desired filters coefficients to be download to PIC24.

The designed system also incorporated a Window-Sinc Filter Designer Tool allowing the user to design filters in real-time and generate the required filter coefficients. This filter designer tool also allow the user to load the designed filter coefficients to PIC24 for implementation. This Window-Sinc Filter Designer Tool allow the user to design three different types of filters including LPF, HPF, and BPF. The user can input different design parameters including sampling frequency, cutoff frequency, windowing methods, filter orders, and filter types. Most importantly it provided a ZedGraph for performance visualization of the designed filter in real-time. This will allow the user to analyze if the designed filters meets specification or not.

In this project 6 pre-designed FIR filters (3 LPF and 3 BPF) was designed using MATLAB filter designer tool. The different filters and its design parameters is presented earlier in Table 1 with each holding the job of extracting a certain harmonic of a square wave signal.

These 6 pre-designed filters was tested and its performance is summarized in Table 4 above. It can be concluded that the performance of the 6 pre-designed filters did not perform as the theory suggested, but it is still good enough to extract the desired harmonic and attenuate all others to a point where it is not presented in the filtered data.

Two filters (LPF and BPF) was designed using the Window-Sinc Filter Designer Tool. These two filters have its design parameters listed in Table 5 and the actual filter performance listed in Table 6. Same as for the 6 pre-designed filters, the two filters design using the Window-Sinc Filter Designer Tool produce performance that does not match the theory performance. Though it is good enough to extract the desired harmonics while attenuated all other harmonics to a degree where it no longer can be seen in the time or frequency domain plots.

Conclusion

In this project a real-time digital filter system was developed around a PIC24FV16KM202 MCU and a Window Form C# GUI application. The PIC24 serve as a filter engine and slave to the C# GUI application. The C# GUI application provide the data visualization and control interface between the user and the PIC24 system. A window-sinc filter designer tool was also developed and embedded with the C# GUI application to allow the user to design and generate filter

coefficients in real-time. The system was developed and tested and met all the design specifications. Therefore, this project was implemented successfully and met all design specifications.

Appendix A

Handshakes between PIC24 and C# GUI control application

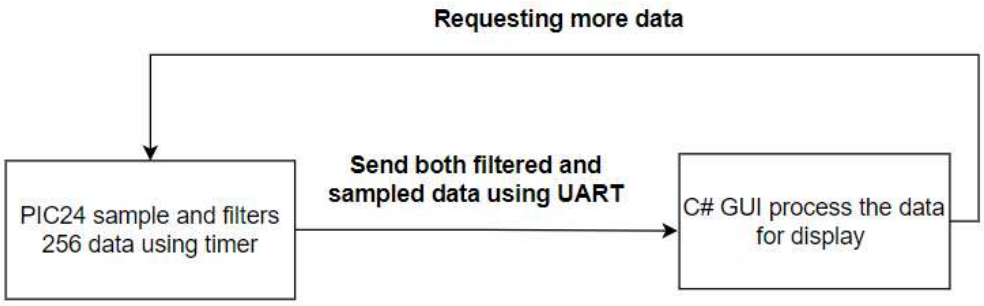


Figure A1: Handshake between PIC24 and C# GUI application.

At the start of the system, the user must use the C# GUI application to pressing 'Start' to send a character 'P' followed a triggering value (0, 1, 2, 3, and or 4). Each of these trigger value represent a set of intervals in the full range of 0 to 255 and is shown below in Table A1. Once PIC24 received this it will start collect the outputs of the filter and put in an array of 256. At the same time, PIC24 will also put the sampled signal (square wave) into an array of 256. When both arrays is filled it will start to send these data to the C# GUI application. Note that while it is sending timer is not disabled, and so the PIC24 filter engine is not stopped but continue to run. It will not push any more data into any of the two arrays until C# request for it to do so. Once C# received all both arrays of 256 data it will start to process the data. After C# finished processing and displaying the data accordingly, it will automatically send the request flag 'P' followed by the trigger value back to PIC24. This process will repeat again and again and in effect it looks like real time in the C# GUI visualization application. Though the user can stop the process anytime and can always start it again.

Table A1: Meaning of trigger values

Control Character from C# GUI to PIC24	Operation and meaning
0	Trigger: min = 0 and max = 50
1	Trigger: min = 50 and max = 150
2	Trigger: min = 100 and max = 150
3	Trigger: min = 150 and max = 200
4	Trigger: min = 200 and max = 250

Appendix B

Important code implementation in PIC24

How to setup ADC in PIC24 for digitization of input signal

```
1  #include <24FV16KM202.h>
2  #device ADC = 8
3  #device ICSP=3
```

Even though PIC24 have 12-bit ADC, it will not be used in this project due to its computational power as it will take longer for it to sample. For our application we are going to use the 8-bit ADC so that we can compute all the mathematic required for the sampled data in the circular buffer with the 64 coefficients within the required timing, though our resolution in digitization has decreased dramatically. In line 3 we are telling PIC24 to do exactly this, to digitize the input signal with 8-bit resolution. ***It is important to note that the place where this command is placed is also important, therefore it needs to be placed right at the beginning section of the program.***

```
314                                                    // Setup ADC
315     setup_adc(ADC_CLOCK_DIV_2 | ADC_TAD_MUL_4);
316     setup_adc_ports(sAN0 | VSS_VDD);
```

In line 316 PIC24 is commanded to use sAN0 (AN0 also mean pin 2) as the ADC pin and use the Vss and Vdd (0V and ~5 V) as the reference to the ADC module.

Line 315 tells PIC24 which clock to use to drive its internal ADC module. In this case we are dividing the 16 MHz clock by 2, and the aquations time of the ADC is set to 4 times the input clock. Meaning that we will give the ADC module some time (4*(1/8M) = .5 us) to tried to correctly determine the input voltage before it convert the reading to the digital domain. It is important to note that if we don't give the ADC module enough time to quantize the input signal, the reading of the ADC will not be accurate. In other words, we can tell the ADC module to digitize (1/16M = 62.5 ns) at its maximum speed but don't expect to get accurate reading!

Appendix C

How to read an array of data from C# GUI serial buffer

In the C# program, we must not read the data right away when we received it. Instead, we should wait until we finally received 256*2 data in our received buffer. This method will allow our system to run and display the waveform at a faster rate as opposed to reading one character on every received interrupt from the serial port in C#.

```
845         if (serialPort1.BytesToRead >= (data_length * 2))
846         {
847             frame_per_second++;
848
849             serialPort1.Read(data_received, 0, data_length);
850
851             serialPort1.Read(data_received_sample, 0, data_length);
852
```

In line 845 above, the program will not do anything (no reading the buffer) until there is 256*2 data in the received buffer. Now when the serial received interrupt fires, and it is the 512th data in the serial received buffer, the program will read all 512 data into the 'data_received' variable.

How to invoke a method when program is inside C# serial received method

When the 512 data are received, C# will start to process these data and it needs to display this information to the C# GUI for user to analyze. Therefore, we must be able to display this information after we finished processing the data. This mean that inside the C# serial received method after we have read all the 512 data and have finished processing, we must call other methods from within the C# serial received method. Though the C# serial received method runs at a different thread, and so we cannot just display information directly to a textbox from within inside this C# serial received method. We must invoke other methods, otherwise it will throw some 'cross threading' exception, and the program will stop and fail. Therefore, below will allow us to work around this 'cross threading' issue.

```
900             //-----------------------------------------Get max
901             this.Invoke(new EventHandler(get_max));
```

Call the desired function as show above from within the C# serial received method.

```
1196    private void get_max(object sender, EventArgs e)
1197    {
1198        max_voltage = data_received_double[0];
1199
1200        for (int i = 1; i < (data_length - 1); i++)
1201        {
1202            if (max_voltage < data_received_double[i])
1203                max_voltage = data_received_double[i];
1204        }
1205
1206        textBox2.Text = String.Format("{0:0.00} V", max_voltage);
1207    }
```

Define the method as shown above. This will allow the program to halt from within the C# serial received method and jumps to another method. In this example we are searching for the maximum value within the filtered output data set (256 data).

Appendix D

How normalization is implemented

Normalizing is a very important concept in this project. Since the filter coefficients is fixed point and is 16-bit, this mean that the coefficients will have large coefficients or very small coefficients within the range of (-32767 to +32767). The output that is desired is within 0 to 255, meaning 0V to 5V. Since the coefficients are fixed point and not normalized, the coefficients can still be use but normalization must be used for the output data to be within the desired range. The below equation is the normalization that is used for data output normalization.

$$Normalized_{data} = \frac{Data - Average\ of\ Data}{Max - Min} * 255 + 127 \qquad [Eq.D1]$$

The following figures shows the filtered data output for 'Not Normalize' and 'Normalized'. Note that the normalized data uses the equation Eq. D1 presented above.

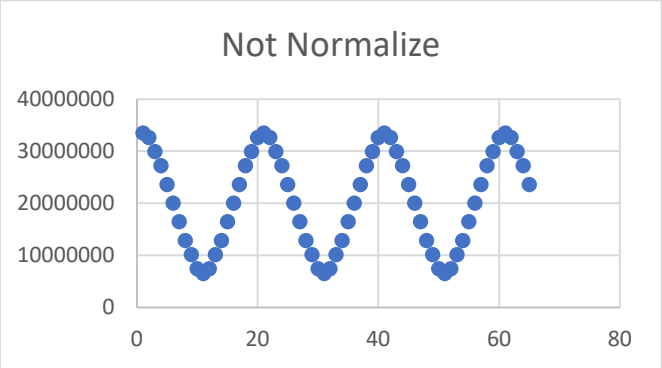


Figure D1: Example of un-normalized data.

The filtered data output is very large in magnitude if normalization is not applied. This can be seen in Figure D1.

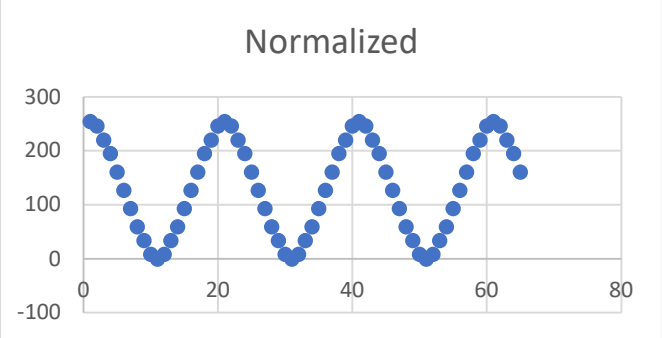


Figure D2: Example of normalized data.

The filtered data output is within the 0 to 255 range after normalization.

Therefore, it is very important that the filtered data be normalize within the 0 to 255 range (0V to 5 V) since the square wave input signal will be in the 0V to 5V range only.

Appendix E

How convolution works and applied to this project (circular buffer)

Convolution is the most important concept in DSP (digital signal processing). If without convolution the digital filter used in this project would have not been possible. Convolution in the time domain is multiplication in the frequency domain and vice versa.

There are two approaches to understanding convolution:

1. **Input side algorithm**
2. **Output side algorithm**

It is important to understand the **input side algorithm** first before understanding the **output side algorithm.** Therefore, the **input side algorithm** will first be analyzed.

Understanding the input side algorithm

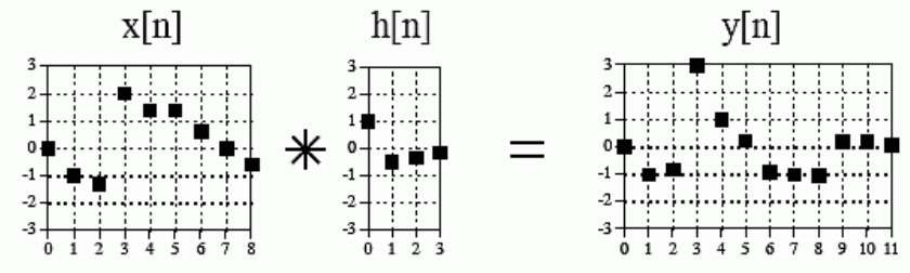


Figure B1: Example of 9-point convolution with 4-point impulse response. [Smith 112]

Show above in Figure B1 is a time discrete time signal convolving with the impulse response of a system. Notice that x[n] have 9 samples, and h[n] have 4 taps, and y[n] have 12 samples. The output y[n] will always a length equal to the number of x[n] sample plus number of h[n] sample minus one. In this example it is (9 + 4) – 1 = 12.

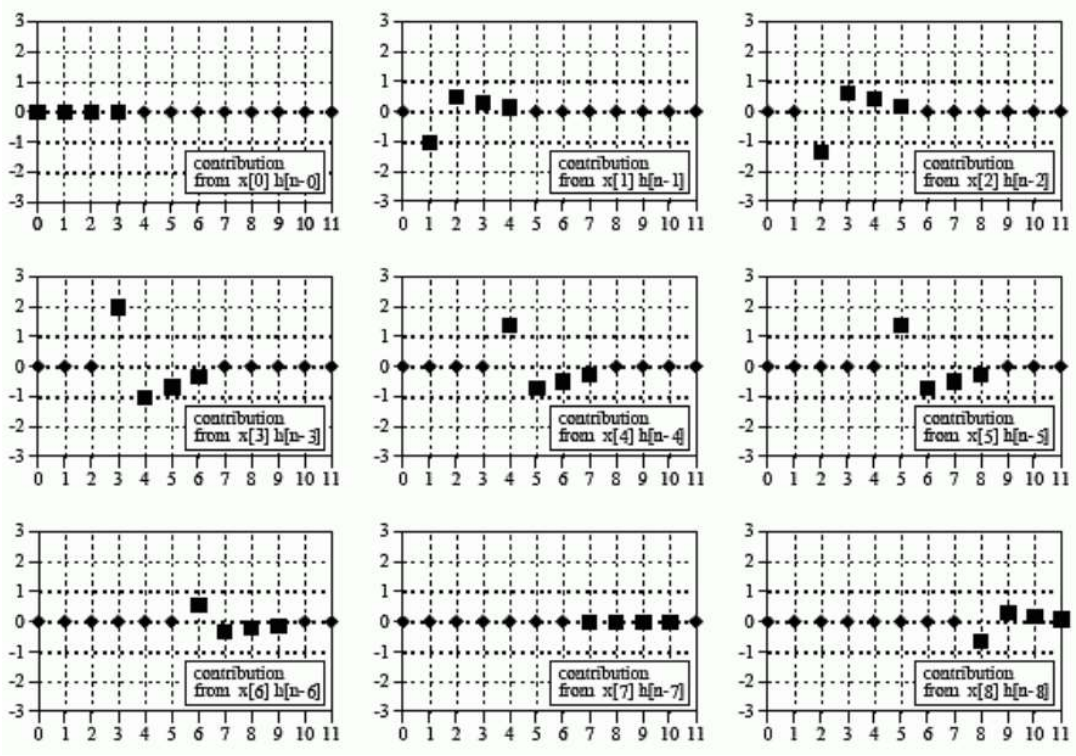


Figure B2: Output signal components contribution for the convolution shown in Figure B1 using the input side algorithm. [Smith 113]

In the Figure B1, there is a 9-point signal convolving with a 4-point impulse response (the filter). When using the input side algorithm to analyze this convolution, the output is as shown in Figure B2 above.

Notice that in Figure B2 there is 9 different plots, due to the 9 different points of the signal. Also notice that the length of each plot in Figure B2 have length of 12 (0 to 11). Again, recall that the filtered data will have a length of 12 from earlier due to (9+4-1 = 12).

Let us analyze the first top left plot in Figure B2. First start with all 12 points with amplitude of 0. Second look at the original 9-point signal and uses the amplitude of the first data point to multiply with the data h[x] impulse response. Since the first point in the 9-point data is 0, the multiplication of 0 with h[x] will produce an all zero amplitude for the 4 impulse points. Therefore, there exist 4 black boxes at the start of the first top left plot.

This process is continued for all the 9-point data of the input signal with the impulse response h[x]. Once this operation is performed on all the input signal data points, the output can be determined by adding the columns of each plot directly to produce an output of length 12 as shown in the right most plot if Figure B1 as y[n].

With this input side algorithm convolution method, one might think and ask how does this algorithm applicable for real time data processing if all the input data is first needed before hand?

The answer is that it is applicable for real time data processing, but further analyzing needs to be done to prove this point. This can be shown and prove if we analyze the plots of Figure B2 closely. We first need to analyze and see which plots (of all the 9 plots) contributed to the final output. Let us take data point number 6 as an example. The output of point number 6 in y[x] in Figure B1 is contributed by only the x[3], x[4], x[5], and x[6] plots shown in Figure B2. All the other plots have no contribution to y[6] at all. So, in total there is only 4 plots that contributed to y[6] as shown;

$$y[6] = x[3]h[3] + x[4]h[2] + x[5]h[1] + x[6]h[0] \qquad Equation\ [1] \qquad \text{[Smith 116]}$$

Notice that what is being contributed to y[6] is only the sampled data from the past and **NOT** the future data. Analyzing equation [1], we can observe that going from right to left the index of the impulse response h[x] is counting upward. Therefore, it would make sense to **flip** the impulse response h[0] for left to right before even performing this input side algorithm. And by doing this we can just simply multiply the h[n] coefficients directly with the current sampled data and past sampled data (otherwise the mathematics will need to be in a christ cross order). By analyzing equation [1] this way it become clear and self-evident that we don't need to know all the signal data x[n] beforehand, and all is needed is just the past 4 past signal data to perform the input side algorithm. This is what leads to the second convolution method called the **output side algorithm.** Also note that this kind of data processing using convolution produces the filtering effect.

Understanding Output Side Algorithm

The output side algorithm is the algorithm that we all comes to use and been taught in class. Why the h[n] is flipped in the diagram below one must first understand the input side algorithm introduced earlier.

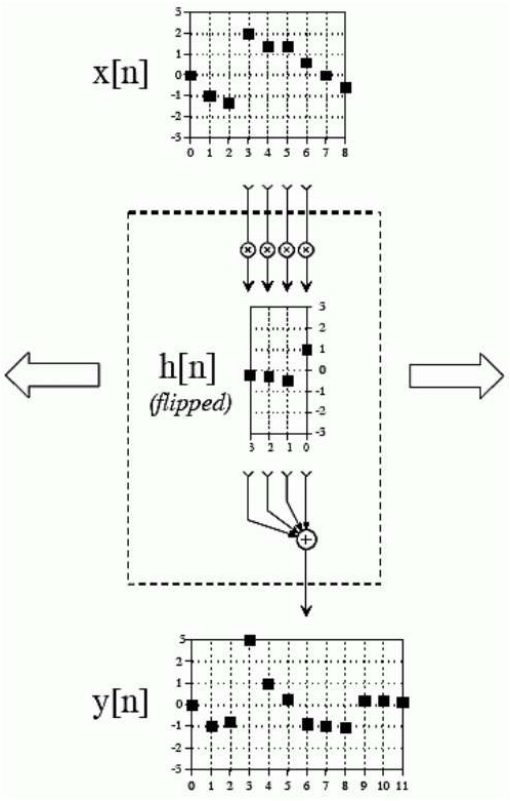


Figure B3: The convolution machine. [Smith 117]

Now by looking at the output side algorithm shown in Figure B3 (also called the convolution machine) we can finally see that the convolution algorithm between two data set (input data x[n] and impulse response h[n]) is applicable for real-time applications as it is required to know past data only. Again, remember that convolution in effect is a filtering process.

In Figure B3, the impulse response h[n] is flipped left for right, and to produce the output y[6]. It uses the past 4 data signal and multiplied with the impulse response and summed up to produce y[6].

Here is the equation that describes the discrete convolution process. Note that it describe convolution in term of the input sided algorithm that is introduced earlier (no flipping is done to h[j]).

$$y[i] = \sum_{j=0}^{M-1} h\,[j]x[i-j] \qquad Equation\ [2] \qquad Convolution\ summation\ \ [Smith\ 120]$$

How the convolution circular buffer is used

In this project a circular buffer of length 64 is used to implement the convolution between in coefficients and the sampled data. The convolution method used here is the second method presented earlier called the output side algorithm. Show below in Figure B4 is the diagram depicting how the circular buffer is used to implement the output side algorithm as a convolution machine as shown earlier in Figure B3.

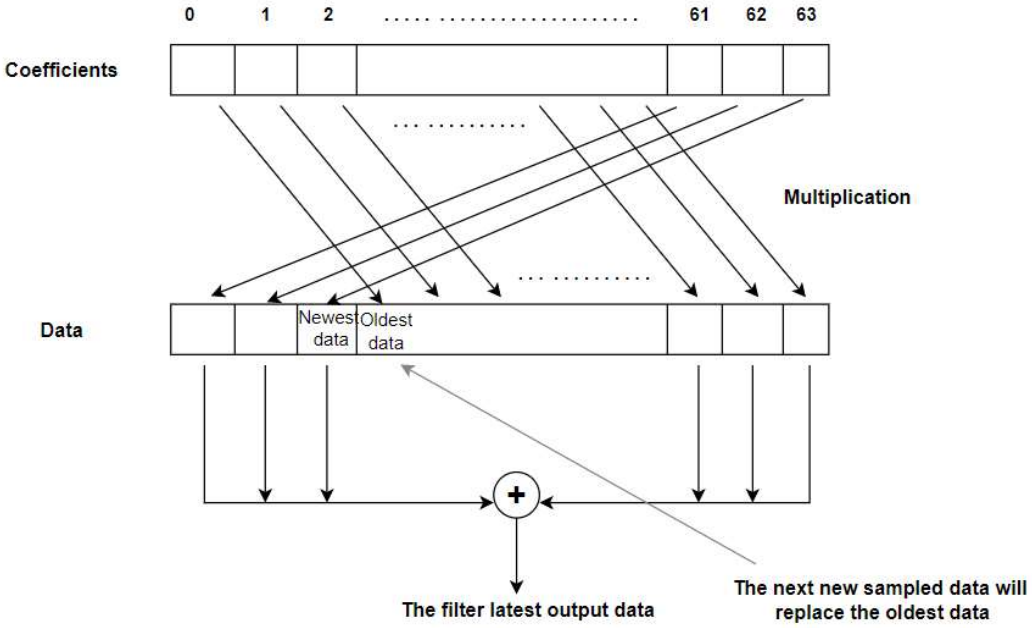


Figure B4: The circular buffer used to implement the output side algorithm.

The circular buffer have a length of only 64 (same length as the number of coefficients). Since this circular buffer have a finite number of lengths this mean that at the start of the system PIC24 will initialize the data array of 64 to be filled with all zeros. PIC24 will start to sample the input signal and process the data using the convolution machine as discussed earlier in Figure B3. Note that for every sampled data PIC24 put into the data circular buffer, the convolution machine is performed to produce a filtered output data. It will continue to do so until all the array of 64 that is holding the sampled data is full. Once the array of 64 sampled data is full the next sampled data will replace the oldest sample (in this case it will replace index position 0). This process with the circular buffer implements the output side algorithm using the convolution machine in real-time and in effect it produces the filters the sampled data.

Appendix F

Window-Sinc Filter Designer Tool theory

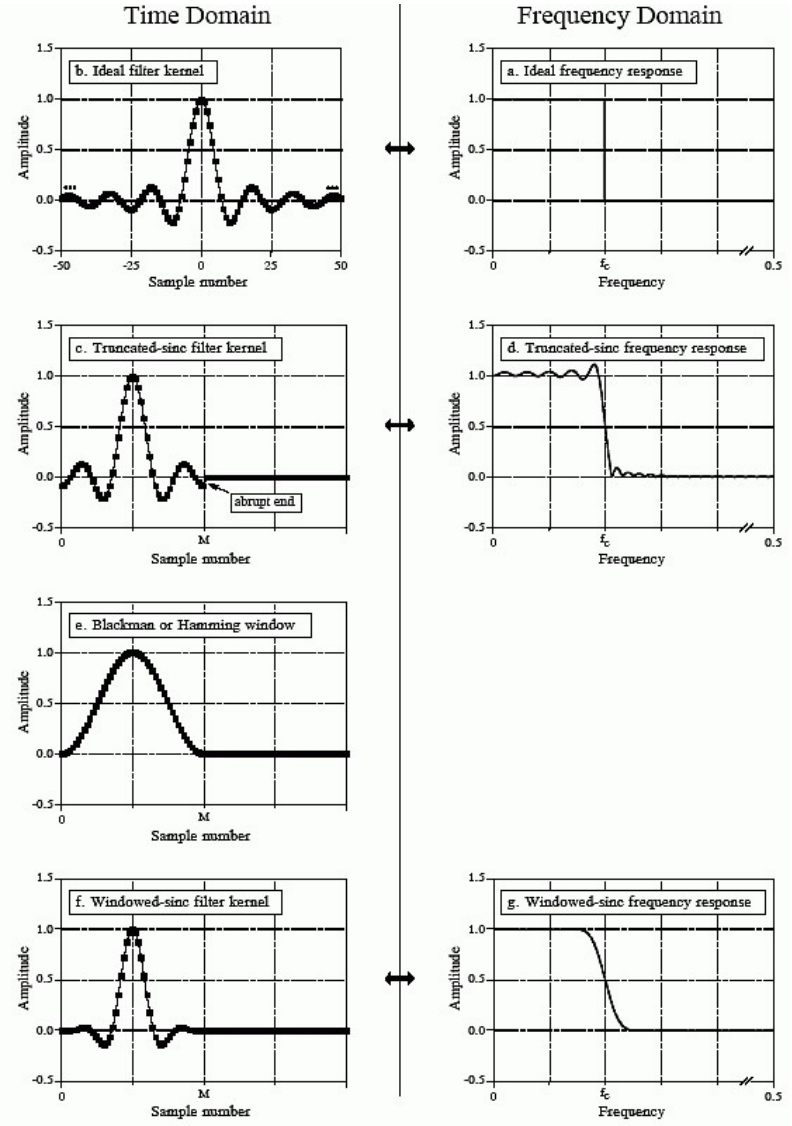


Figure F1: How the Window Sinc Filter design method works [Smith 287].

In Figure F1 plot b, it shows the sinc function in the time domain (sample number). The sinc function in the frequency domain is the ideal low pass filter and it is show in Figure F1 plot a.

Notice that in plot b of Figure F1 the sinc function goes to positive and negative infinity. Though computer cannot store an infinity amount of data. Therefore, to use this sinc (ideal LPF) we must truncate it. In Figure F1 plot c shows the truncation of the sinc function to a finite amount of data only. Truncating the sinc function this way produces the abrupt effect and the ideal LPF of the sinc function no longer is ideal and is shown in Figure F1 plot d. Due to the abrupt truncation of the sinc function in the time domain, the LPF in the frequency domain have ripples in its pass and stop bands.

How to reduce these ripples one might ask? The answer is to use a window. There are different windows that can be used to help smooth out the ripples. In this example the Blackman and Hamming window can be use. In Figure F1 plot e shows a window. Notice that the window have the same length as the length of the truncated sinc function.

Recall that multiplication in the time domain is the same as convolution in the frequency domain and vice versa. Therefore, the truncated sinc function is multiplied with the window function to produce the final filter kernel. This can be seen in Figure F1 e, f, and g.

There are many available different windows but listed below are the three common windows equations;

Hamming Window $w[i] = 0.54 - 0.46\cos\left(\frac{2\pi}{M}\right)$ [$Smith$ 286]

Blackman Window $w[i] = 0.42 - 0.5\cos\left(\frac{2\pi i}{M}\right) + 0.08\cos\left(\frac{4\pi}{M}\right)$ [$Smith$ 286]

Hanning Window $w[i] = 0.5 - 0.5\cos\left(\frac{2\pi i}{M}\right)$ [$Smith$ 288]

Note that for the three equations above the variable M must be an even integer. The variable i goes from 0 to M.

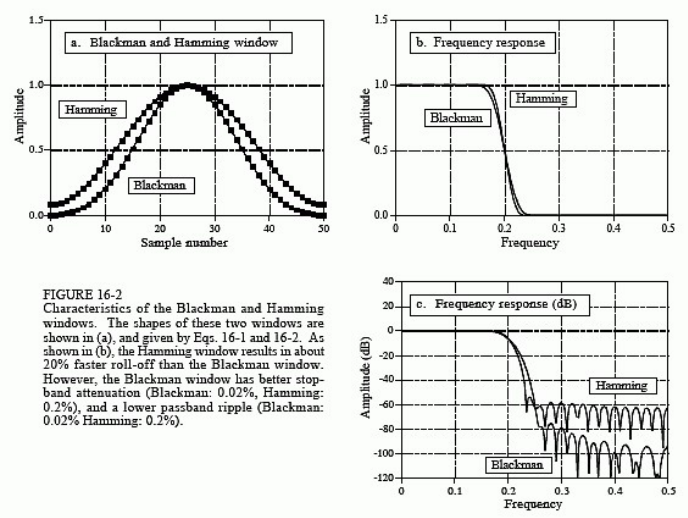
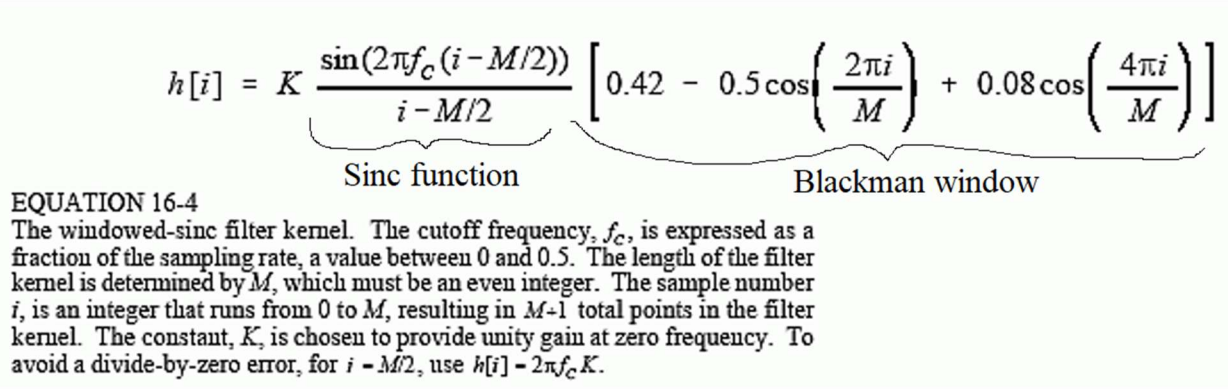


FIGURE 16-2
Characteristics of the Blackman and Hamming windows. The shapes of these two windows are shown in (a), and given by Eqs. 16-1 and 16-2. As shown in (b), the Hamming window results in about 20% faster roll-off than the Blackman window. However, the Blackman window has better stopband attenuation (Blackman: 0.02%, Hamming: 0.2%), and a lower passband ripple (Blackman: 0.02% Hamming: 0.2%).

Figure F2 : Performance difference between the Blackman and the Hamming window. [Smith 288]

Presented in Figure F2 is the performance between the Blackman and the Hamming window. The Blackman is the better window as it have faster roll-off and better stopband attenuation than the Hamming window. Therefore, the Blackman should be the first choice between the two.



EQUATION 16-4
The windowed-sinc filter kernel. The cutoff frequency, f_c, is expressed as a fraction of the sampling rate, a value between 0 and 0.5. The length of the filter kernel is determined by M, which must be an even integer. The sample number i, is an integer that runs from 0 to M, resulting in $M+1$ total points in the filter kernel. The constant, K, is chosen to provide unity gain at zero frequency. To avoid a divide-by-zero error, for $i = M/2$, use $h[i] = 2\pi f_c K$.

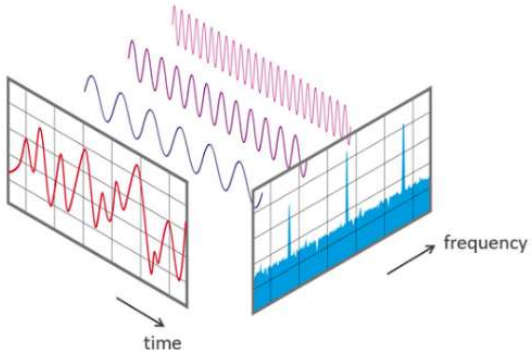
Figure F3: Equations used to generate the filter coefficients. [Smith 291]

In Figure F3 shows the equations used to generate coefficients for implementing the designed filter. Notice this equation consist of the sinc function and the Blackman window. Both the sinc and the Blackman window are multiplied together, implies that in the frequency domain both their impulse response are convolved together. To avoid dividing by zero for when i = M/2, the user must let h[i] = 2*fc*k for when i = M/2. The constant K is to make sure that at the zero frequency there is unity gain. Therefore, ignore K during the calculations of the filter coefficients as it is used for normalization only and should be applied last. Though K must be chosen such that the sum of all the coefficients is equal to 1. One might ask how to pick K such that the sum of all the coefficients equal 1? It is an easy task, all you need to do is perform a running sum on all the calculated coefficients, and then divided each coefficient by this total running sum. This will guarantee that the sum of all the coefficients is equal to 1, and therefore have unity gain at the DC.

Appendix G

Understanding DFT and C# code implementation

The DFT (discrete Fourier Transform) is a mathematical tool that transform a signal from the time domain into the frequency domain. The figure shown below demonstrated this concept well. In essence the DFT is correlating the time domain signal with each of the basic functions (sinusoidal). Correlation is the detection of how likely a signal exist in another signal on comparison. In other words, it is a way (indeed the optimal way) of detecting a known signal inside another signal. And so, when we are performing the DFT on a time signal we are trying to detect which basic functions (sinusoidal functions of different frequency) exist inside the time domain signal.



Source: https://makeabilitylab.github.io/physcomp/signals/FrequencyAnalysis/index.html

The mathematical definition of the DFT is shown below;

- The DFT of N real data values x[n]: $x[n] \leftrightarrow X(k)$

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}kn}, \quad k = 0, 1, \ldots, N-1$$

$$= \sum_{n=0}^{N-1} x(n)\left[\cos\left(\frac{2\pi}{N}kn\right) - j\sin\left(\frac{2\pi}{N}kn\right)\right]$$

$$= \sum_{n=0}^{N-1} x(n)\cos\left(\frac{2\pi}{N}kn\right) - \sum_{n=0}^{N-1} x(n)\, j\sin\left(\frac{2\pi}{N}kn\right)$$

Source: Borrowed from Dr. Zheng's slide show (ECE 522 lecture 18).

x(n) = discrete time signal (sample)

k = the number of the frequency domain point we are trying to calculate

N = number of total time domain samples (constant)

n = index that loops through all the time domain samples to produce 1 single frequency domain point

From the equation above, we can write program that can implement this DFT equation. By analyzing the DFT equation, it can be concluded that 2 nested FOR loop will be needed to implement this.

Since the DFT equation required a lot of computations, it is implemented in C# as PIC24 will take a long time to implement this.

The below figure shows the use of the DFT formula in C#.

```
369         //calculate dft
370         for (int k = 0; k < data_length; k++)
371         {
372             for (int n = 0; n < data_length; n++)
373             {
374                 dftReal[k] += (double)data_received[n] * Math.Cos(2 * Math.PI * k * n / N);
375                 dftImag[k] -= (double)data_received[n] * Math.Sin(2 * Math.PI * k * n / N);
376             }
377         }
```

Note that this program of the DFT was borrowed from Dr. Zheng's lecture 18 slide show.

How to find the frequency after performing DFT? Simply find the magnitude of the real and imaginary part, and loops through the 256 data points to determine which amplitude have the highest amplitude and keep tracks of the index of that highest amplitude. Then multiply the index by the frequency resolution to determine the frequency of the input signal.

Frequency resolution = fs/N; where fs is the sampling frequency

and N is the number of time samples (constant)

It is important to note that the DFT will produce a mirrored image of the frequency spectrum of the input signal. Therefore, we only need to use the first half of the magnitude array.

This also points out a very important concept of DSP regarding the sampling frequency. From the sampling theorem, we must have a sampling frequency twice that of the input signal that we are trying to digitize. So, if our sampling frequency is 10 kHz, and the input signal is 8 kHz the DFT will not be able to correctly identify this input signal's frequency. In this example the DFT can only successfully and correctly identify frequency that is below or equal to 5 kHz.

Reference

[1] Smith, Steven W. “Convolution.” *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing , San Diego, California , 1997, pp. 107–122.

[2] Smith, Steven W. “Windowed-Sinc Filter.” *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing , San Diego, California , 1997, pp. 285–296.

[3] https://makeabilitylab.github.io/physcomp/signals/FrequencyAnalysis/index.html