2004

# PIC 18F452 implementation of digital filters

Vikram A. Bose-Mullick
*West Virginia University*

# PIC 18F452 IMPLEMENTATION OF DIGITAL FILTERS

Vikram A Bose-Mullick

Thesis Submitted to
the college of Engineering
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Electrical Engineering

**Lane Department of Computer Science and Electrical Engineering**

**Morgantown, West Virginia**
**2004**

Keywords: Microchip PIC, 18F452, FIR filter, LMS, Adaptive Filter, Noise Cancellation, Echo Cancellation, Filter Design

# ABSTRACT

PIC 18f452 implementation of digital filters

Vikram A Bose-Mullick

This research hopes to explore the computational limits of the PIC18f452 chip by encompassing the designing and implementation of two types of filters for the PIC 18F452 microcontroller. The main purpose of this research is to implement a floating-point least mean square (LMS) error adaptive filter and its secondary goal is a fixed-point implementation of finite impulse response (FIR) filter. FIR filters are specified via a graphical user interface (GUI) and upon demand, optimized C-language code is generated for the popular CCS PIC C-Compiler. In is the intent of this research to learn whether FIR filters can be made computationally viable on the PIC18 chips, can they run stably with reliable and repeatable performance? What is the minimum execution time possible at the processing limits of the chip? And how is filter attenuation affected when taps are scaled down from floating-point to fixed point? For the floating point LMS filter it desired to explore the relationship between sampling-rate and filter order and to develop a hardware optimized floating point library for general use. The minimum execution time for the LMS filter achieved during this research is 26.7 μs per order. The FIR filter code generation software developed during this study allows graphical specification, inspection of response curves. It ultimately presents three options for automatic code generation — program-space efficient code (uses minimum code space), data-memory efficient code (uses minimum RAM) and speed-efficient code (optimized for quickest execution), thereby allowing up to a 75th order FIR filter with the best execution time of 800ns per MAC cycle achieved at the bit-depth of 8-bit samples and 8-bit taps. The filter tap conversion from floating-point format to 8-bit fixed point reduced the attenuation by an average of 28%. In general, both filters gave a strong performance with consistent, reliable and repeatable results.

# DEDICATION

Both small and large, to everyone that made a difference. Above all, I dedicate my work to my kind and loving family.

# ACKNOWLEDGEMENT

I wish to express my deepest possible thanks to Dr Powsiri Klinkhachorn my advisor and dear friend, for his excellent guidance and boundless friendship. Even though I've never received an 'A' from him, I have no regrets because I have learned more in his classes than all my 'A's put together.

A special thanks to Dr. Roy Nutter for bringing microprocessors into my depth of field through his innovative lectures, inspiring chats and for showing us the value of journals and periodicals. They contained the very first sparks that led to my intense love of the subject.

I express my deep appreciation for to Dr. McConnell from whom I learned the fundamentals for analog and digital without which nothing is possible.

Thank you gentleman for your generosity of spirit, for your infinite kindness you're your constant support and finally for making the time to serve in my committee.

A special thanks to my friend Tal Gottesman for filling in and picking up the slack during during zillions of hours that were spent putting this document together.

Finally my thanks to Braxton Lewis and my boss Dr. McCawley for all their constant help and support.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF NOMENCLATURE

| | |
|---|---|
| 1. MCU | Microcontroller Unit. |
| 2. DSP | Digital Signal Processor. |
| 3. ADC | Analog to Digital Converter. |
| 4. DAC | Digital to Analog Converter. |
| 5. PWM | Pulse Width Modulation. |
| 6. LMS | Least Mean Squared. |
| 7. FIR | Finite Impulse Response. |
| 8. IIR | Infinite Impulse Response. |
| 9. ALU | Arithmetic Logic Unit. |
| 10. MIPS | Millions of Instructions per Second. |
| 11. MAC | Multiply and Accumulate. |
| 12. SMD | Surface Mount Technology. |
| 13. RAM | Random Access Memory/Read Write Memory. |
| 14. EEPROM | Electrically Erasable Programmable Read Only Memory. |
| 15. FFT | Fast Fourier Transform. |
| 16. GUI | Graphical User Interface. |
| 17. ISR | Interrupt Service Routine. |
| 18. EOB | End of Buffer. |
| 19. BOB | Beginning of Buffer. |
| 20. FSR | File Select Register. |

# CHAPTER 1: INTRODUCTION

## 1.1 Personal Motivation

My fascination with single chip microcontrollers began with my undergraduate days and has remained consistent ever since. In a world where minimalism is the catchword, they fit the role perfectly, being a cost effective way to elegantly solve complicated problems, thereby making so many aspects of electronics and software accessible to engineers and students as well. During my undergraduate and graduate years I participated in projects involving electronics and circuit design and always enjoyed finding new opportunities for microcontroller based solutions. With circuits getting more and more complex, filters must be installed to control noise and dealing with filters meant having to look for the right capacitors and the right resistors, op amps and repeating the same tedium all over again, especially if it was determined that a new set of filtering specifications were required. The alternative to a true digital-filter is to use a switched capacitor filter but those are usually not as clean as analog filters and require a clock signal that adds switching noise; with an additional circuit component drawing power, occupying space and incurring an explicit monetary cost.

It is here that the Finite Impulse Response (FIR) filters step in, being an attractive alternative to using analog filters and switched capacitor digital filters. I learned about them in theory and conducted a couple of Matlab simulations before realizing that these are ideal for use with microcontroller projects. They impose no additional monetary cost upon the circuit; can be easily reconfigured by changing code, without any lag in performance with time. The challenge is to do a very efficient implementation for the PIC 18 architecture so it becomes possible for the filter to function as a supplementary application, thereby, providing an intuitive graphical interface that will allow anyone to easily generate these filters using a simple point and click system. However, not all noise problems can be solved by using FIR filters.

Sometimes due to the nature of the noise, especially if it is correlated, it is impossible for an ordinary (fixed-band) filter to remove it, because both the signal and noise occupy the same frequency range. For instance, if the echo of the signal was the source of noise, then the echo could not be removed simply by suppressing its frequencies, because the echo and the source have a strong correlation. In cases like these, adaptive filters are used to reduce noise. The least mean squared (LMS) error is a commonly used adaptive noise cancellation algorithm that is ideal for this purpose because it is a good compromise between computational complexity and performance.

.

## 1.2 Signal Processing using the PIC 18F452 Microcontroller

Microcontrollers such as PIC chips which run at speeds up to 10 MIPS (million instructions per second) are useful for gaining valuable practical experience with low bandwidth signal processing ideas. What makes them so convenient is the wealth of built in hardware, which can sample signals, perform ADC conversions and contain multiple timers for accurate timing. Moreover, there are a number of low cost compilers making the package available under $6.00 per chip [4] and as low as $175 for a C-Compiler and an in-circuit programmer for $75 [2], making it feasible cost-wise as well.

## 1.3 Digital Filters Vs Analog Filters

Digital filters have several advantages and disadvantages over their analog counterparts. The main advantage of digital filters is that they occupy no physical space as they are implemented completely in software and operate by applying a mathematical algorithm designed to produce the filtering effect. Since digital filters need no physical components i.e., capacitors and resistors, their performance does not degrade with age or respond to ambient environmental conditions. Another major advantage is that some digital filter (FIR filters) can have a unique property called linear phase response, which is critical in many communications applications. Analog filters presently, have a much greater dynamic range however, than digital filters because they are not limited by factors like sampling rate and computation speed [5].

## 1.4 PIC 18 Microcontroller Family

Microchip Technologies manufactures a popular line of micro controllers known as Peripheral Interface Controller or PIC chips. The PIC 18F452, released in May 2002, is currently one of their fastest chips [3]. At the core of this chip is an 8-bit RISC based ALU that can process 10 MIPS at 40 MHz. Its design is based on Harvard architecture, allowing it to have separate data and program memories. Its memory is divided into 32 KB of flash based program memory and 1.5 KB of volatile data memory (RAM) as well as 256 Bytes of EEPROM. PIC chips have a RISC based instruction set consisting of a small yet seminal set of instructions, most of which are single cycle, thereby making them fast executing and easy to program. Other valuable devices such as analog to digital converters, pulse width modulation, multiple timers, I/O Ports are all integrated within the same chip that also contain hardware support for several popular serial communication protocols such as $I^2C$, SPI and UART. Running at 40 MHz, it takes the 18F452 chip 100ns to multiply two bytes and compute a 16-bit result. The other noted feature that makes this chip viable for signal processing applications is that, it contains multiple hardware pointers that allow very fast access to data stored within the chips' RAM.

## 1.5 Detailed Research Objectives and Contributions

The main focus of this research will be to test and validate the PIC chips' ability to implement a real-time floating-point LMS based Adaptive filter, which is a very useful way to deal with noise that is too closely related to the signal for conventional band compensating filters to handle.

1. A suitable general-purpose, adaptive noise cancellation circuit will be designed, that is both cost effective and customizable to serve several different applications. The circuit will be tested using test signals generated by a PC sound card as shown in Figure 1.5.1. This will allow the modeling of different types of noise and to test various signal to noise ratios. The circuit will process the signals in real-time and the results will be measured using a data acquisition system. Analysis of the recorded data should reveal the effective noise reduction versus noise reductions predicted via simulations.

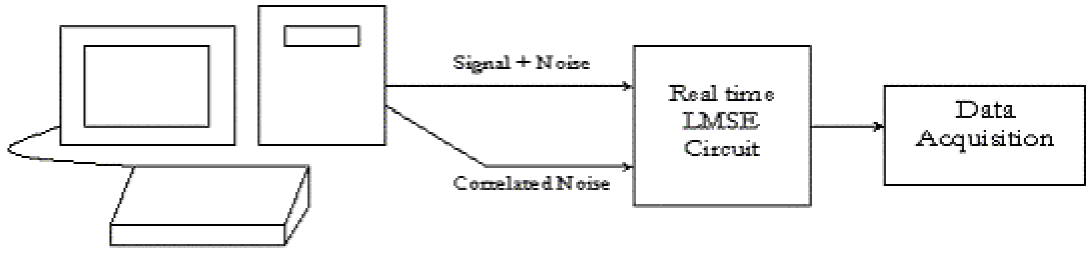


Figure 1.5.1 Topology for Real-Time LMS Circuit Testing

2. During the course of this research, software will be developed to benefit the users of the popular CCS PIC Compiler. The software will include a modular library for PIC 18XXX with optimized floating-point math support. Although the compiler is inherently capable of handling floating-point data, it performs common floating-point operations such as addition, subtraction, and multiplication at an alarmingly slow rate rendering it unpractical for real-time applications. Therefore, another aim of this research will be to develop a modular library that will provide a faster alternative to the compilers built in floating-point system.

3. The secondary focus of this research will be to test and validate the PIC chips' ability to implement a real-time fixed-point FIR filter, which is a very practical idea, because it can be seamlessly used in countless applications where noise and the signal of interest occupy separate frequency bands.

4. A Graphical User Interface (GUI) will be developed that allow users to design various types of FIR filters, such as Low-Pass, High-Pass, Band Pass, Notch or any combination of the above, in short, multi-band filters. The user may design the filter by taking a point and click approach to specifying band-edges, attenuations, sampling rate etc. and the software will show users the respective frequency and phase response graphs. Once the user is satisfied with the filter they have designed, the software will present them with several realization options, thereby allowing them to decide whether they want the filter optimized for execution speed, or conservative RAM usage or conservative program-memory usage. Ultimately, optimized C language source code is generated that is ready to be compiled for either PIC 18F452 chip or the smaller PIC 18F252 chip or easily adapted for the remaining chips in the PIC 18FXXX family by a moderately experienced programmer. Finally, the GUI will generate a diagram of the test circuit needed to install the filter code.

5. Each type of FIR filter created by the software will be evaluated independently by applying a constant-power frequency sweep generated by a filter test program. The real-time output of the filter will be recorded by a data acquisition system and its performance will be analyzed though PC based data analysis tools such as FFT.

## 1.6 Organization

Chapter two will cover a literature review and theoretical background of existing techniques for digital filter implementation for both LMS and FIR filters. Chapter three will constitute the implementation details for both the filters. Chapter four will present results and analysis and chapter five will contain recommendation for future work. An appendix is provided that contains all codes written during this exploration and a user's manual for the filter design of the GUI.

# CHAPTER 2: LITERATURE SURVEY

## 2.1 Classification of Filtering Methods

The earliest filters were analog filters. In recent years, digital filters have gained popularity due to the lowering cost of microprocessors and the increased level of convenience and flexibility offered by digital filters. Advances in technology allowed them to function at a faster speed and now they are rapidly approaching the large dynamic range of analog filters [5]. A broad classification of Digital filters is presented in Figure 2.1.1.

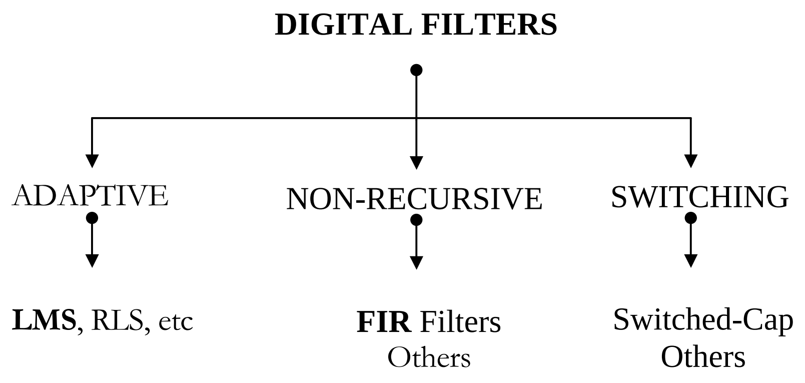


Figure 2.1.1 Classification of Signal filtering methods

## 2.2 Digital Filters

A digital filter is a discrete-time linear system that operates on an input sequence, modifies it, and produces the output sequence. The input sequence is usually obtained by digitizing a signal, thereby converting it into discrete time, with the output sequence being transformed back into an analog signal through an appropriate digital to analog process. The steadily reducing cost of portable computation is thereby making a direct contribution to the rise of popularity of digital filters.

## 2.3 Non-Recursive Type Digital Filters

The most commonly used Non-Recursive filter is the FIR filter. The weights of this type of digital filters are constant and are computed at design time. Since the weights remain constant, the stability of FIR filters can be guaranteed. However, they can have several topologies — the transversal topology as shown in Figure 2.3.1 being the most common type and the one used for this research [9].

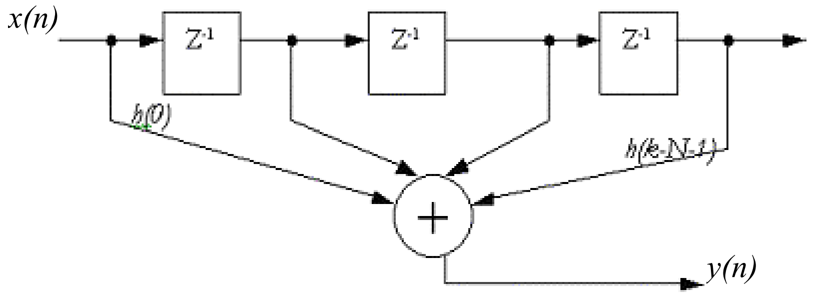


Figure 2.3.1 The transversal topology of the FIR filter

The transversal FIR filter is characterized by the following equation.

$$y(n) = \sum_{k=0}^{N-1} h(k) * x(n-k) \tag{2.1}$$

Where,

*x(n)*: discrete time elements of the sampled signal
*y(n):* is the computed output of the FIR filter
*h(k):* are the coefficients of the filter also knows as filter-taps

Linear convolution of the filter coefficient with the sampled signal produces the filtering effect. Since multiplication and addition are the only mathematical operations involved with the FIR filter, this process is ideally suited for use within the PIC 18F452 microcontroller. The clear advantage of using FIR filters is the radical alteration in its frequency compensation, which can be achieved by simply providing the system with a new set of filter coefficients. Another interesting property of FIR filters is that, they are the only type of filter that can have a true linear phase response. Since this research deals exclusively with the implementation aspects of FIR filters, it is assumed that the coefficients of the filter have already been computed. For more theoretical details regarding obtaining filter coefficients refer to [7].

## 2.4 Switching type digital filters

The switched capacitor filter is a common type of switching filter. Switching type digital filters are a convenient alternative to using high order analog filters. They are packaged for convenient use and typically require a clock signal and power to operate. Most are strictly low-pass filters; others can be programmed by additional resistors, to be used as band pass and notch filters. However, this convenience comes at the expense of additional monetary cost and components and having to deal with the inescapable incurrence of switching noise [1].

## 2.5 Adaptive Filters

One of the most successful adaptive algorithms is the LMS filter developed by Widrow [14]. LMS, sometimes known as LMSE is excellent for dealing with correlated noise where noise and the signal are too much alike to be filtered using ordinary band-compensating filters such as low-pass, band-pass etc. Such filters are commonly referred to as adaptive filters and they are used in applications such as, echo-cancellation over communication lines, noise-cancellation, Electro-cardiogram (ECG) in pregnant mothers, suppressing machine noises in mines and countless other applications.

## 2.6 Least Mean Squared Error (LMS)

The LMS filter is based on the steepest decent algorithm where the weight vector is updated from sample to sample as follows:

$$W_{k+1} = W_k - \mu \nabla_k \tag{2.2}$$

where,

Wk: Is the weight vector

$\nabla_k$*: Is* the true gradient vector

$\mu$*:* Rate of convergence also referred to as learning rate

The LMS algorithm is a practical method of obtaining estimates of the filter weights Wk in real time. The Widrow-Hopf LMS algorithm for updating weights from sample to sample is given by:

$$W_{k+1} = W_k + 2\mu e_k X_k \tag{2.3}$$

where,

$$ek = y_k - W_K^T X_k \tag{2.4}$$

*ek*: Is the error term

$X_k$: Is the correlated noise vector

LMS algorithm above does not require prior knowledge of the signal statistics, but instead uses instantaneous estimates to tune the filter. The weights obtained by the LMS algorithm only estimates, but these estimates improve gradually with time as the weights are adjusted and the filter adapts itself to the characteristics of the signals. Eventually, the weights converge. The condition for convergence is,

$$0 < \mu < \frac{1}{\lambda \max} \tag{2.5}$$

where,

*λmax:* Is the maximum Eigen value of covariance matrix.

The main objective in adaptive noise cancellation is to produce an optimum estimate of the correlated noise in the contaminated signal. This is done by the simultaneous sampling of two signals — one being the signal of interest to be filtered and the other being the source of correlated noise, referred to as the reference. The adaptive filter in Figure 2.6.1 uses the reference to predict the degree of contamination in the signal of interest by the process of correlation.

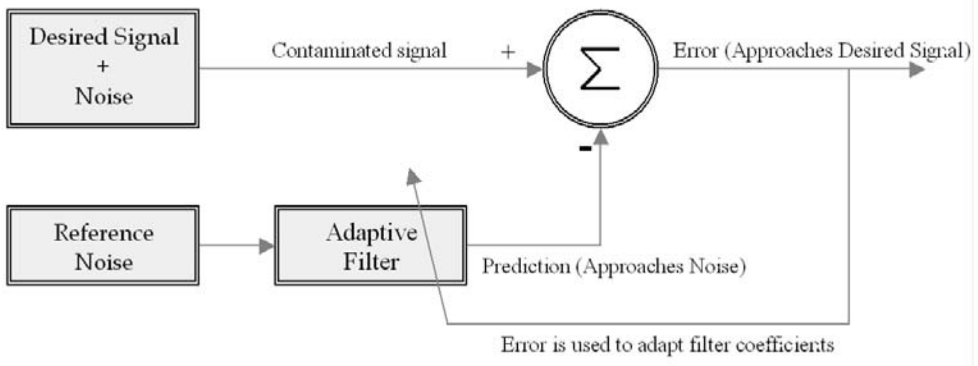

Desired Signal
+
Noise
Contaminated signal
+
Σ
Error (Approaches Desired Signal)
-
Reference Noise
Adaptive Filter
Prediction (Approaches Noise)
Error is used to adapt filter coefficients

Figure 2.6.1 LMS filter Topology

The adaptive filter attempts to predict the amplitude and phase of the noise present in the contaminated signal by correlating the reference with the contaminated signal. The prediction of the adaptive filter constantly approaches the actual noise present in the contaminated signal. With the error signal continuously being used to tune the filter, it gradually approaches the desired signal. Figure 2.7.1 shows a flow diagram for the LMS filter algorithm.

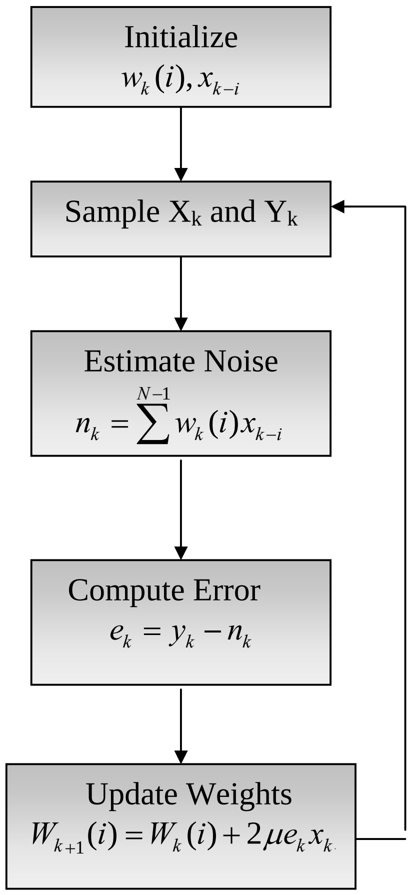
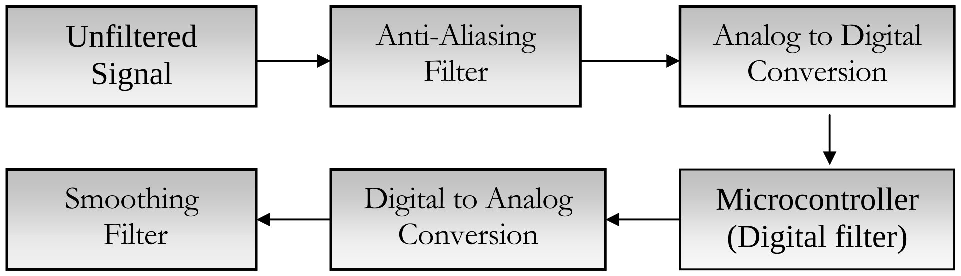


Figure 2.7.1 Standard Implementation of LMS Filter

## 2.8 Implementation of a digital filter

Digital filters are a natural choice for circuits that are interfaced to or controlled by a microcontroller. Part of the microcontroller's computing power may be dedicated to filtering the sampled input signals. Figure 2.8.1 is a block diagram for typical digital filter implementation.



Unfiltered Signal
Anti-Aliasing Filter
Analog to Digital Conversion
Microcontroller (Digital filter)
Digital to Analog Conversion
Smoothing Filter

Figure 2.8.1 Standard Implementation of Digital Filter

The Anti-Aliasing filter is a low-pass filter designed with a cutoff that is at least half the sampling rate of the analog to digital converter (ADC). This is used to prevent sampling of frequencies above Nyquist rate [6]. A smoothing filter is another low-pass filter that is used to reduce the harmonic distortion resulting from the quantization process. An illustration of the described process is presented in Figure 2.8.2.

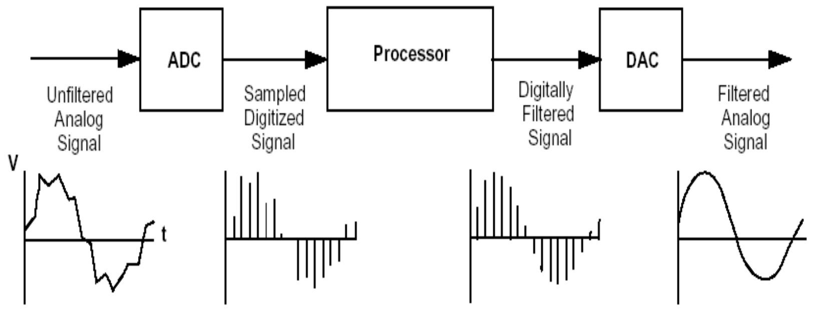


Figure 2.8.2 Digital Signal Processing overview

# CHAPTER 3: IMPLEMENTATION DETAILS

Implementation details for the fixed-point FIR filter are discussed first followed by the implementation details for the more complicated floating-point LMS filter.

## 3.1 Finite Impulse Response (FIR)

Development of a FIR filter generally involves two distinct phases. The first one is the design phase and the other is the realization phase. The design phase involves specifying filter characteristics such as band-edges, frequency-response and phase-response etc. and finally derives the filter coefficients for the intended filter. There are several ways to obtain filter coefficients. For this research the Matlab filter design toolbox was used to generate them. In FIR filters, the same hardware can be used to realize many different types of filters. It can be seamlessly reconfigured from a low-pass to band-pass to notch or a combination of all of them by simply supplying a new set of coefficients. The implementation discussed in this study is optimized for the PIC 18F452 instruction set although it is flexible enough to be easily adapted to other inexpensive microcontrollers with similar hardware.

The software developed for building FIR filters includes a program that allows users to visually specify the filter parameters. Once the filter has been finalized, optimized code will be automatically generated for the PIC 18F452 processor. Since the filter is usually used as a supplementary application, it must be designed to co-exist with a main application. The proposed implementation uses only a fraction of the microcontrollers' total computational capacity and the remaining cycles are reserved for the main application. Additionally, the implementation scheme is easy to reconfigure without making changes in hardware.

The second phase is the realization phase. This involves the selection of an appropriate platform upon which the filter will be implemented. In this case the platform desired is the PIC 18F452 chip. Real-time implementation involves three distinct processes — firstly the analog-to-digital conversion of a signal; followed by mathematical processing by the filtering algorithm; and finally, if needed the obtained results have to be transformed back into an electrical signal using a suitable digital-to-analog conversion technique. All three processes mentioned above must be performed within a proper time constraint or the result becomes invalid. For instance, if we are sampling a signal at 4000Hz then our worst-case time is 1/4000Hz or 250us. All filter computations must be completed within the time window of 250us. The block diagram of the FIR filter is presented in Figure 3.1.1.

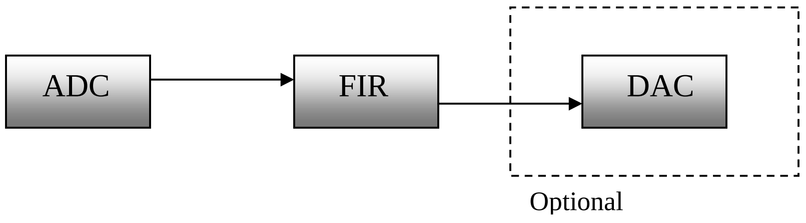


Figure 3.1.1 FIR filter block Diagram

## 3.2 Implementation Background

Three different implementation strategies are provided to the user as options, each with its advantages and drawbacks. They are minimum RAM implementation, minimum program memory implementation and minimum execution time implementation. Each will be discussed in detail in the following sections.

Implementation aims to take advantage of the PIC chip's hardware architecture and instruction sets. The PIC 18F452 chip has certain features in its hardware that makes it a good choice for filtering applications. The following restrictions were used while implementing the FIR filter algorithm in order to maximize the filter throughout.

1. Multiplication operations are restricted to unsigned integer data only. The Table 3.2.1 is a summary of manufacturer published multiplication-performance for the PIC 18F452 chip [10]. Table 3.2.1 outlines the speed gain from using the hardware multiplier and by favoring unsigned-multiplication operations instead of signed multiplication operations. $\text{Time}_H$ is the time needed performing hardware multiplication and $\text{Time}_S$ is the time needed to perform software multiplication.

*Table 3.2.1 Multiplication speeds for PIC18452*

| ROUTINE | METHOD | $\text{Time}_H/\text{Time}_S$ | Speedup |
|---|---|---|---|
| 8x8 Unsigned | Hardware/Software | 100ns/6.9μs | 6900% |
| 8x8 Signed | Hardware/Software | 600ns/9.1μs | 1500% |
| 16x16 Unsigned | Hardware/Software | 2.4μs/24μs | 1000% |
| 16x16 Signed | Hardware/Software | 3.6μs/25.4μs | 1400% |

2. The analog to digital converter is used with 8-bit resolution. Even though the built in ADC on the PIC chip is capable of sampling up to 10-bit resolution, the PIC memory and ALU are both 8-bit wide. It is therefore most efficient in handling 8-bit data. Hence, all filter coefficients and ADC data will be restricted to 8-bit resolution.

3. All memory references are made using indirect addressing. The PIC 18F452 chip contains three hardware pointers. FSR0, FSR1, FSR2, each being 12 Bits and capable of covering the entire RAM size for the PIC 18 family (up to 4096 bytes for PIC18f2515). By shortening the range of these pointers to 8-bits we can gain efficiency at the expense of smaller memory coverage. The pointer space will be restricted to 8-bits to cover

256 bytes of RAM or a single bank of RAM. This means that all our buffers and other dynamically allocated areas of RAM have to be confined to 256 bytes of memory.

## 3.3 FIR Filter Implementation

FIR filter implementation scheme on the PIC 18F452 chip can be categorized using the following major steps shown in Figure 3.3.1.

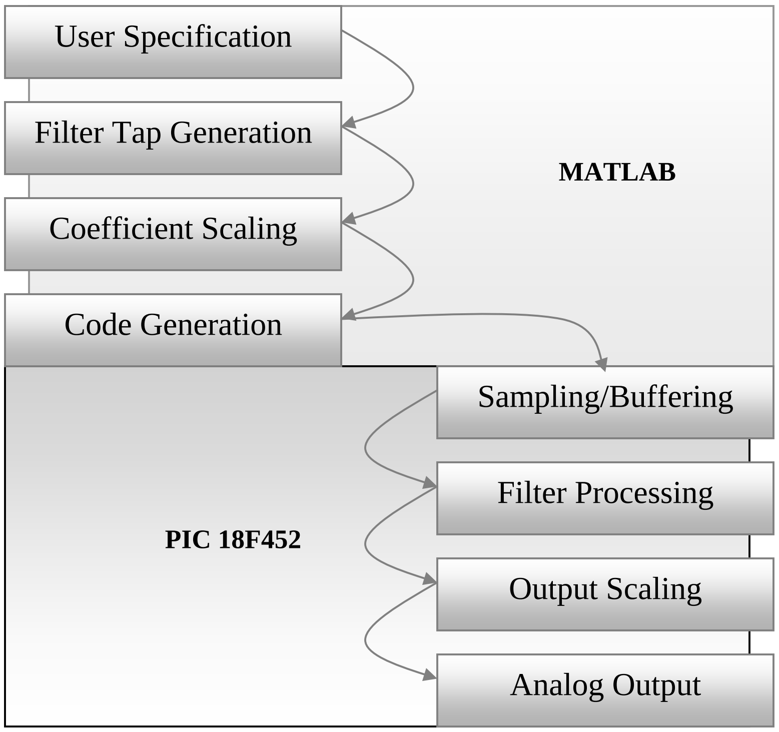
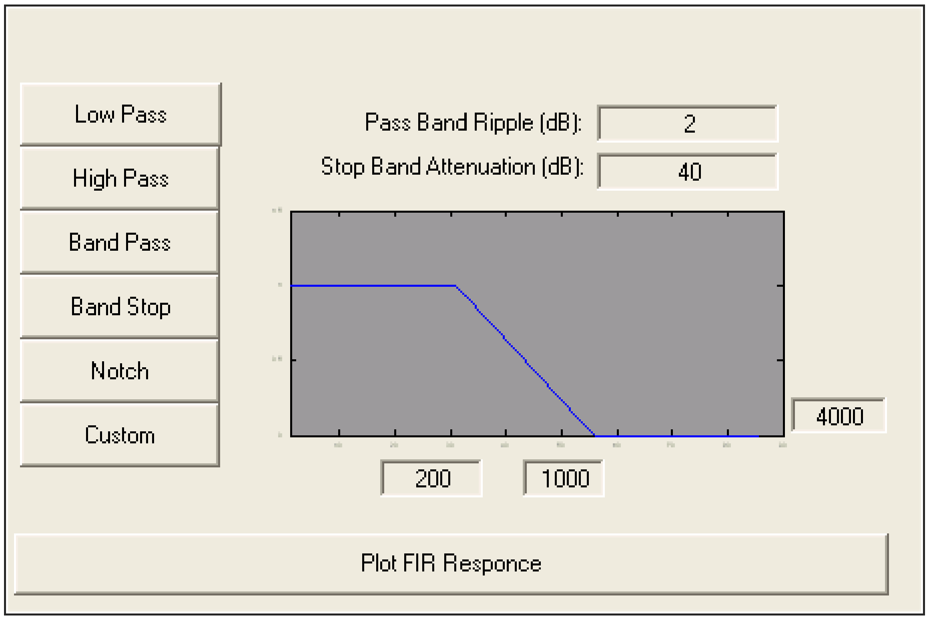


Figure 3.3.2 FIR filter creation stages

### 3.3.1 User specification

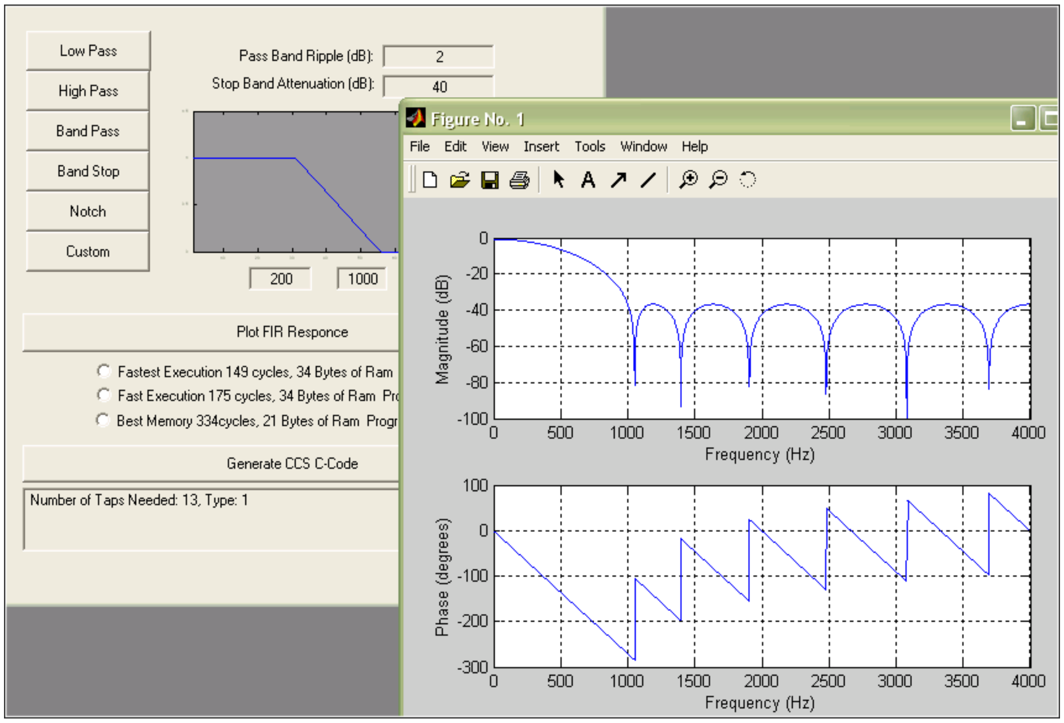
The very first logical step to making a filter is to specify filter parameters such as band edges, attenuations and ripples. To this end, the following interface was developed to allow a user to specify the type and exact parameters of the filter to be designed. Figure 3.3.2 is a snapshot of the developed filter making software.



Pass Band Ripple (dB): 2
Stop Band Attenuation (dB): 40
200
1000
4000

Figure 3.3.2 Digital Signal Processing overview

By making use of the menus the user can select from the range of filters that can be generated for real-time implementation. The available options are low-pass, high-pass, band-pass, band-stop, notch and custom. Once the type of filter is decided, the user can specify parameters such as band edges and attenuations by filling in the appropriate boxes. Before the user is allowed to generate code, the frequency and phase response for the desired filter circuit must be reviewed. The software automatically calculates the exact filter order required to achieve filtering requirements. The filter coefficients are calculated using the Remez exchange [8] method for optimal tap generation for low-pass and high-pass configurations. Figure 3.3.3 shows the frequency and phase response curves as well as the different code generation options available to the user. If satisfied the user may generate the desired filter.



Pass Band Ripple (dB): 2
Stop Band Attenuation (dB): 40

Fastest Execution 149 cycles, 34 Bytes of Ram
Fast Execution 175 cycles, 34 Bytes of Ram Pr
Best Memory 334cycles, 21 Bytes of Ram Progr

Number of Taps Needed: 13, Type: 1

Figure3.3.3 Frequency/phase review curves & code generation options

Once a satisfactory design is achieved the user is given three options for code generation. Finally C-language code, as shown in Figure 3.3.4, is generated that is ready to be compiled or edited.

```
#include <18f452.h>
#use delay(clock = 40000000)
#fuses H4,PUT,NOWDT


const int filter_length = 13;
const int taps[filter_length] = {20,35,58,83,106,121,127,121,106,83,58,35,20};


// PIC 18F452 Register MAP.....................................................

// ACCUMULATOR ADDRESS
#byte     WREG = 0xFE8            // Register Stores the Carry Bit
#byte     PRODL =0xff3            // Product Low Byte
#byte     PRODH =0xff4            // Product High Byte
#byte    ADRESL = 0xfc3           // Low Byte for ADC Sample
#byte    ADRESH = 0xfc4           // High Byte for ADC Sample
#byte    STATUS = 0xfd8           // Status Register
```

Figure 3.3.4 Section of C-Code generated

## 3.3.2 Filter Tap Generation

The Matlab filter design toolbox [13] was used to generate filter coefficients. This toolbox contains a set of functions that allow users to conveniently make and test different types of filters. If for example, a low-pass filter was desired with the characteristics given in Figure 3.3.5 and Figure 3.3.6 is the skeleton Matlab-code needed to generate it.

Sampling Frequency of 8000Hz

Pass band frequency of 500Hz

Stop Band frequency of 1000Hz

Pass band ripple of .05 dB

Stop band ripple of 55 dB

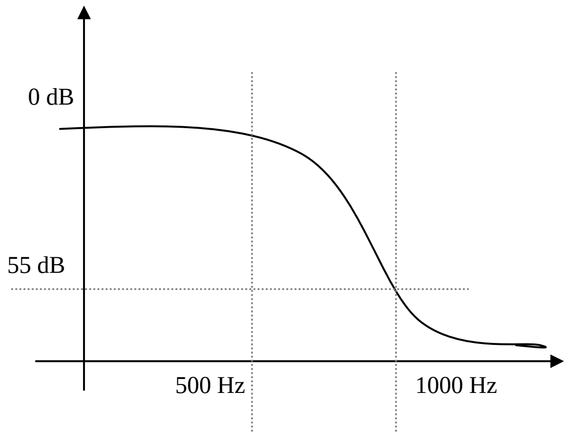
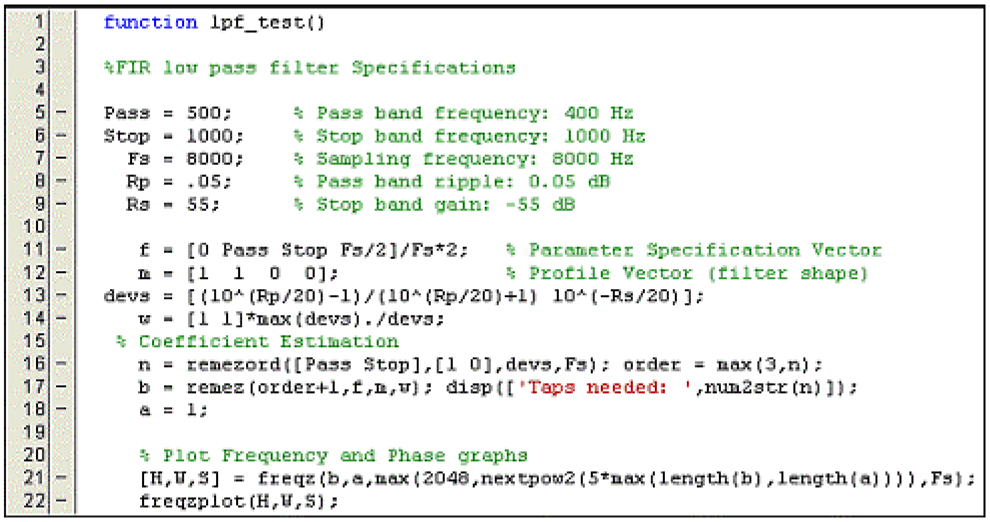


Figure 3.3.5 Intended LPF parameters



```
function lpf_test()

%FIR low pass filter Specifications

Pass = 500;      % Pass band frequency: 400 Hz
Stop = 1000;     % Stop band frequency: 1000 Hz
  Fs = 8000;     % Sampling frequency: 8000 Hz
  Rp = .05;      % Pass band ripple: 0.05 dB
  Rs = 55;       % Stop band gain: -55 dB

   f = [0 Pass Stop Fs/2]/Fs*2;   % Parameter Specification Vector
   m = [1  1  0  0];              % Profile Vector (filter shape)
devs = [(10^(Rp/20)-1)/(10^(Rp/20)+1) 10^(-Rs/20)];
   w = [1 1]*max(devs)./devs;
 % Coefficient Estimation
   n = remezord([Pass Stop],[1 0],devs,Fs); order = max(3,n);
   b = remez(order+1,f,m,w); disp(['Taps needed: ',num2str(n)]);
   a = 1;

   % Plot Frequency and Phase graphs
   [H,W,S] = freqz(b,a,max(2048,nextpow2(5*max(length(b),length(a)))),Fs);
   freqzplot(H,W,S);
```

Figure 3.3.6 Skeleton code needed for Filter

The skeleton code presented in Figure 3.3.6 upon execution will produce the graphs for both phase response and frequency response in Figure 3.3.7.

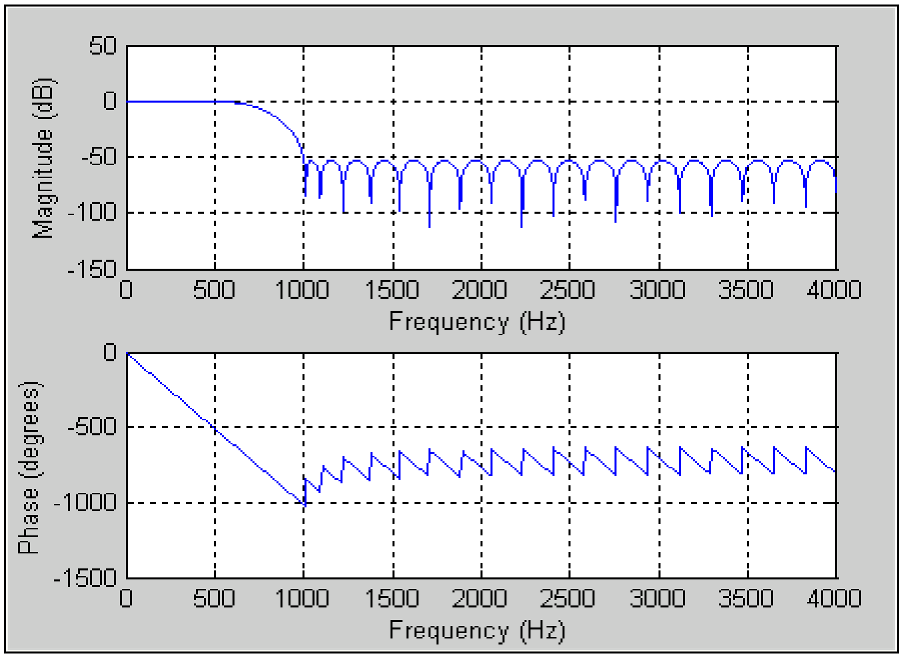


Figure 3.3.7 Frequency and phase response plot

The filter tap coefficients generated by Matlab are as plotted next:

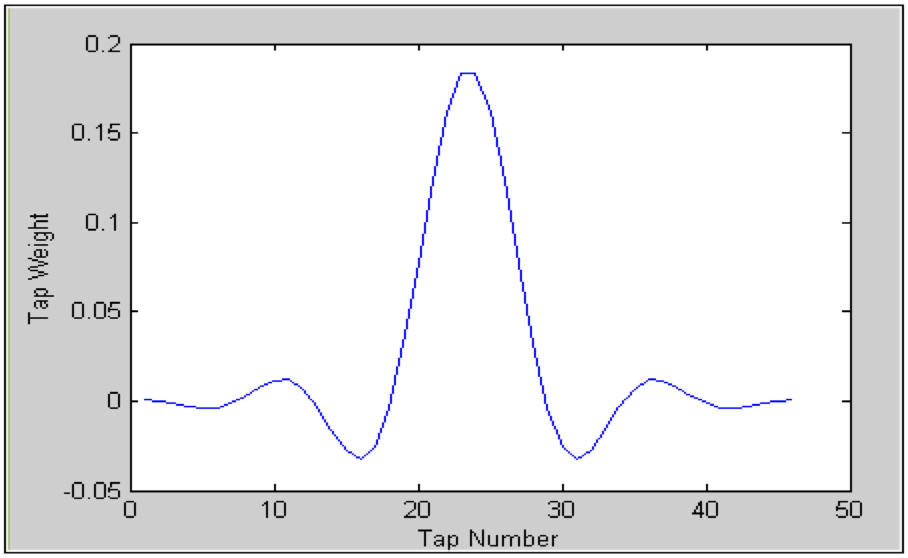


Figure 3.3.8 Frequency and phase response plot

### 3.3.3 Coefficient Scaling

The tap coefficients computed by Matlab are computed in floating point format ranging from [-1.0,1.0]. Before they can be used in the PIC chip they need to be converted into 8-bit fixed-point format and made unsigned. The following scaling function apply the to achieve this:

$$scaled\_tap_n = ceiling\left\{\frac{floating\_tap_n}{\max(floating\_tap)}*127\right\}+128 \qquad (3.1)$$

Each tap coefficient provided by Matlab is first normalized to the range [-1.00,1.00], then multiplied by 127 and rounded to the higher integer. Finally 128 is added to each tap to make it positive. After the scaling function is applied, the [-1.00,1.00] range becomes [0,255], shown in Figure 3.3.9 and now unsigned integers.

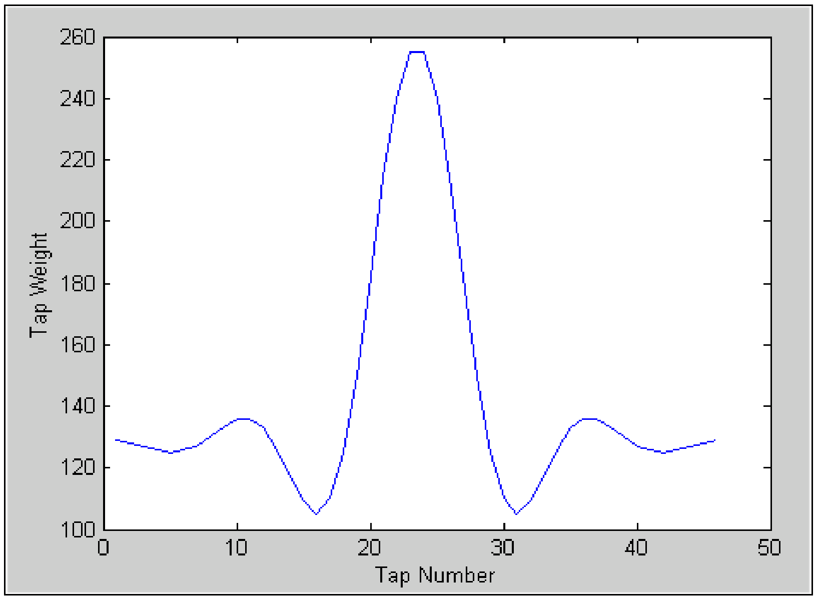


Figure 3.3.9 Eight-Bit scaled coefficients

### 3.3.4 Code generation

Once the filter parameters are established and all decisions involving implementation details are complete, compiler ready C-language source code for the CCS PIC C is generated based on a set of three different templates. Each template is a specialized implementation scheme optimized to produce a different flavor.

1. Minimum Ram: The c-code generated using this template makes minimal demand on RAM.

2. Minimum Program Size: This template minimizes the program size.

3. Minimum Execution Time: This template produces code that achieves higher execution speed.

If the PIC chip is dedicated to performing signal filtering only then either implementation scheme is suitable. However if the FIR filter is used as a supplementary application that runs along side a primary application then it competes for the same recourses as the main application. Thus it may be useful for the user to use the parametric optimizations. To facilitate the selection of which type of optimization is suitable the user interface provides exact values for RAM, program memory and execution time with each option. The optimizations in RAM, execution speed and program size were all derived using a combination different buffering techniques, loop unrolling, and inline assembly language routines for the real-time components. Each is discussed in detail in the following sections.

### 3.3.5 Buffering Data

Once Matlab has generated the filter coefficients, they need to be accommodated within the PIC memory. Additionally, the constant stream of data from the PIC ADC must be accommodated in memory with the exact chronological sequence in which it was sampled. The buffering scheme for tap coefficients is discussed first.

The tap coefficients are stored in the PIC in the form of a look-up table in its program memory. Before filtering begins, the entire table is copied to the RAM and marked with a hardware pointer. Managing coefficients is not complicated because the number of taps is finite and the list is static (needs to be initialized only once).

Buffering the ADC data is a far more interesting problem. There are several complications that have to be dealt with. The finite impulse response filter is quite simply the linear convolution between a constant set of filter taps and a discrete time capture of a signal. For example, say, the desired filter has 30 tap coefficients then we would need to capture and store not only a latest sample of the signal, but the previous 29 samples as well. To achieve this, two different buffering schemes were explored. The first one used a traditional one-dimensional circular buffer [11]. This technique uses less memory but lengthens the cycle of computations. The second technique used two adjacent one-dimensional circular buffers [12]. This technique uses more RAM than the first, but allows the speed of the filter to approach its shortest possible computation time on the PIC 18F452 chip (using 8-bit taps and 8-bit data).

**1. Circular buffer implementation on PIC 18F452:** A circular buffer is a memory allocation scheme where memory is reused (reclaimed) when an index is incremented to a multiple of the buffer size. The modulo nature of a circular buffer maintains data in a queue form (chronological order) at all times without overrunning its allocated memory or the need for re-ordering. The elegance of this type of memory allocation is that the very same pointer that is used to queue data is efficiently used to dequeue it and due to its modulo nature, the dequeueing pointer automatically terminates at the point of insertion of the next sample. On a PIC chip, the buffer that was used is illustrated in Figure 3.3.10. Oldest sample is written over the newest sample and File Select Register (FSR) is the hardware pointer used to load and unload data. The illustrated circular buffer holds four elements — EOB marks end-of-buffer, BOB marks beginning-of-buffer, the numeric values in the figure are RAM locations and the sample buffer occupies memory locations from 0x41 to 0x44.

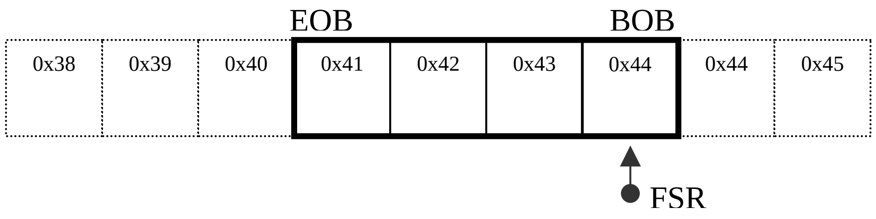
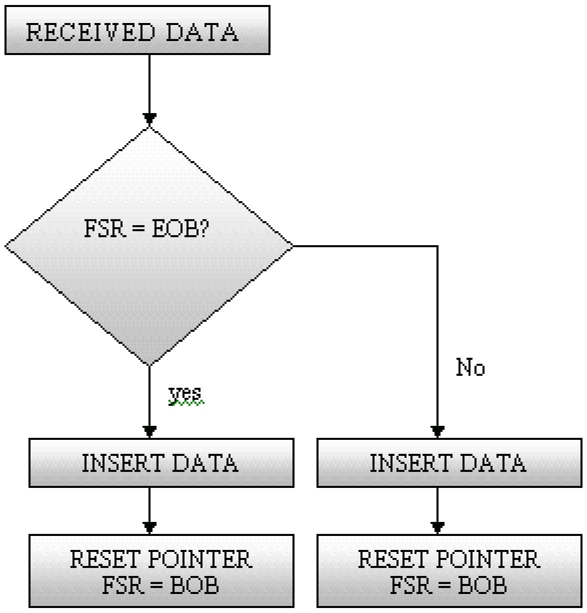
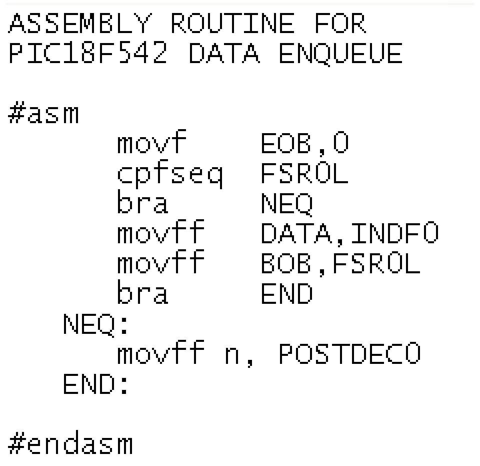


Figure 3.3.10 Circular buffer: Used to store ADC values for FIR filter

While loading the buffer the pointer FSR could be at any location within the buffer, so before the sample is stored, it is crucial to first check if the pointer has reached EOB. If it is the very first sample then the received data is placed at the *BOB* or location *0x44* and the pointer is post-decremented to location *0x43*. In PIC18 assembly, the hardware pointer FSR can load data and post-decrement in a single cycle by using the POSTDEC register. There is no post-increment feature to the hardware pointer system; hence, the BOB is at a higher memory location than the EOB. When the FSR pointer has been reached, the EOB is simply reset to BOB. Based on this concept, the newest data sample automatically replaces the oldest data sample.

Illustrated in Figure 3.3.11 is a flowchart showing the process of loading the circular buffer as well as the assembly code written to achieve it.





```
ASSEMBLY ROUTINE FOR
PIC18F542 DATA ENQUEUE

#asm
      movf    EOB,0
      cpfseq  FSR0L
      bra     NEQ
      movff   DATA,INDF0
      movff   BOB,FSR0L
      bra     END
   NEQ:
      movff n, POSTDEC0
   END:

#endasm
```

Figure 3.3.11 Algorithm developed to load circular buffer

If the ADC was capturing a ramp in the form of digital data ranging from 0 to 6 then the buffer would load in the following way: The first data point '0' will be stored at the BOB and the pointer is decremented as illustrated in Figure 3.3.12. The next sampled data point '1' is stored in the location pointed by data pointer FSR and the pointer is decremented as shown in Figure 3.3.13.

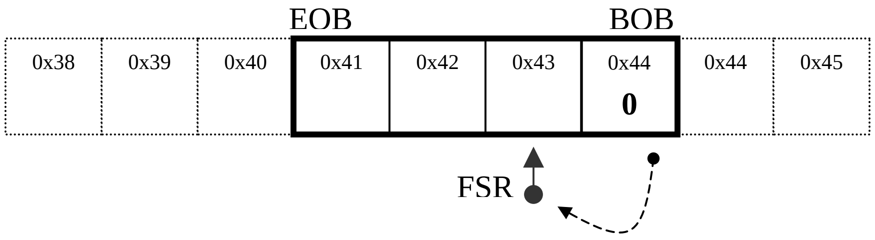


Figure 3.3.12 Step 1: Data element 0 is loaded and pointer decrements

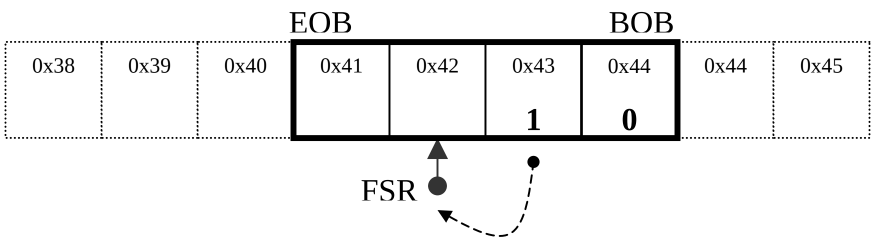


Figure 3.3.13 Step 2: Data element 1 is loaded and pointer decrements

By the time '3' is sampled the buffer is full and EOB is reached as shown by the illustration. '3' is stored at EOB and the pointer is reset to the BOB. Now notice the pointer is at the oldest element as shown in Figure 3.3.14.

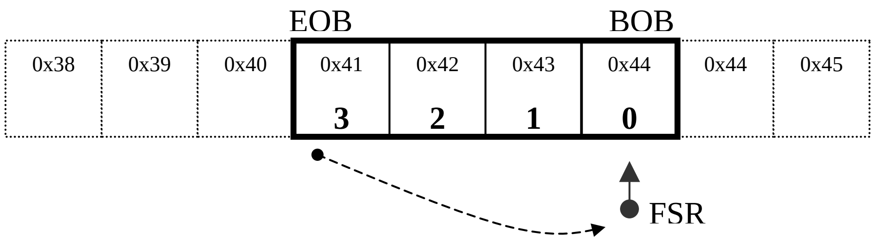


Figure 3.3.14 Step 3: Data element 3 is loaded and EOB is reached

When '4' is captured it replaces the oldest element in the buffer and the pointer FSR is incremented as normal as shown in Figure 3.3.15.

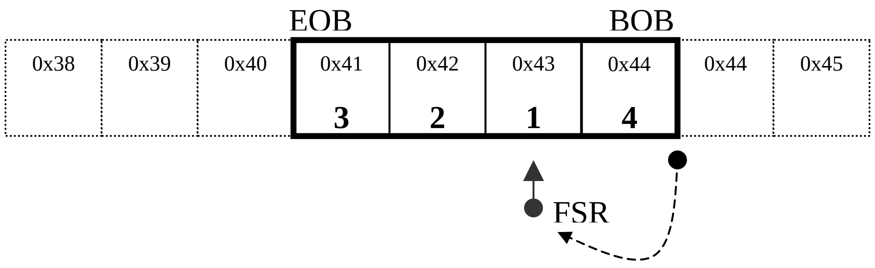


Figure 3.3.15 Step 4: Element 4 is loaded and pointer is pre-decremented

In order to pull data from the buffer, the pointer FSR would simply travel in the opposite direction and data will be obtained in the exact opposite order to which it had entered. Before each the pointer is advanced it must first check for the BOB or it will travel beyond the buffer. If BOB is reached the pointer is relocated to EOB. In order to extract data the pre-increment function of the pointer is used so data is pulled and pointer is advanced in a single-cycle. To illustrate the process the pointer is pre-incremented to 0x44 and '4' is pulled as illustrated in Figure 3.3.16.

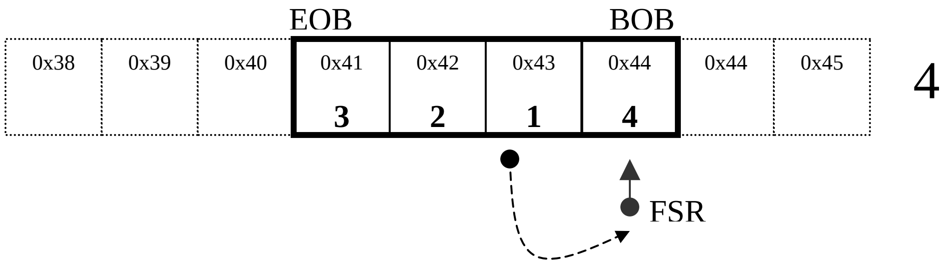


Figure 3.3.16 Element 4 is unloaded from buffer and BOB is reached

Note the pointer FSR is at the beginning-of-buffer so it is first relocated to the EOB and then the data pulled is '3' as illustrated in Figure 3.3.17.

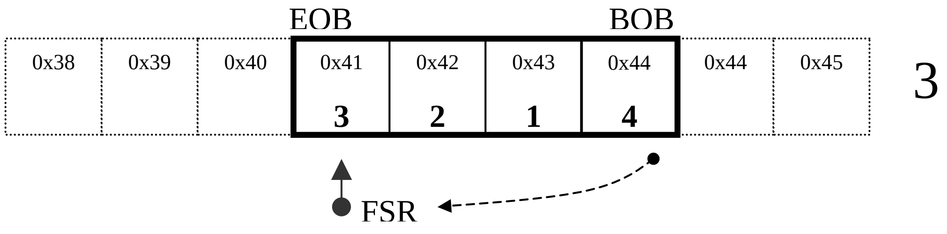


Figure 3.3.17 Pointer is relocated to EOB and 3 is pulled

Pointer is pre-incremented to 0x42 and '2' is pulled followed by '1' and after four iterations the pointer FSR has automatically terminated at the entry point where the next incoming data sample is to be placed as shown in Figure 3.3.18.

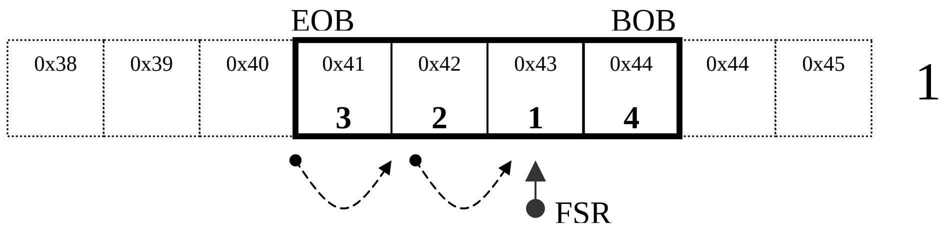
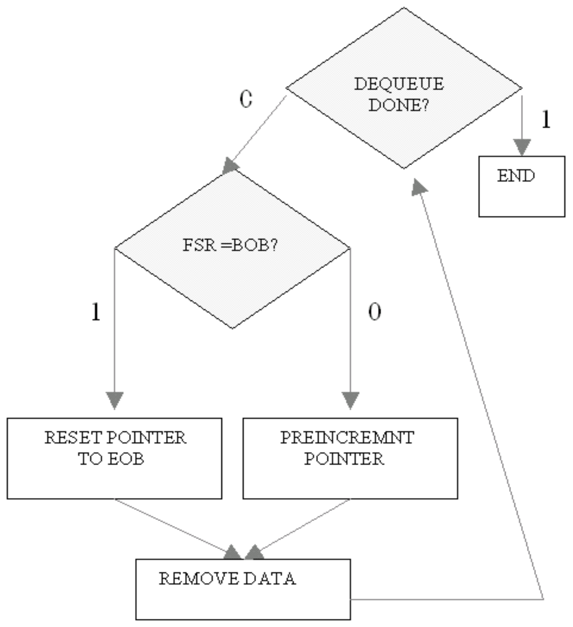


Figure 3.3.18 Data element 1 is pulled

Data went into the buffer in the order {1,2,3,4} and came back out {4,3,2,1}. The formal algorithm and assembly code is in Figure 3.3.19.



```
ASSEMBLY ROUTINE FOR
PIC18F452 DATA DEQUEUE

#asm

        movf   BOB,W
        cpfseq FSRCL
        bra    NEQ
        movff  EOB, FSR0L
        movff  INDF0, OUT
        bra    END
NEQ:
        Movff  n,PREINCC
END:

#endasm
```

Figure 3.3.19 Algorithm used to pull data from the circular buffer

**2. Double circular buffer implementation on PIC 18F452**: In the second buffering technique two adjacent circular buffers are used in such a way that the second one begins exactly where the first one ends. Every time a fresh sample is made, it is placed in both buffers in place of the oldest sample respectively. Each buffer will have it's own pointer and both buffers will contain the exact data at any given time. [8]

This buffering scheme has a very useful advantage over the previous one because the unloading pointer does no longer need to check for the end of buffer (EOB). Figure 3.3.20 of the buffering scheme might explain the process more clearly.

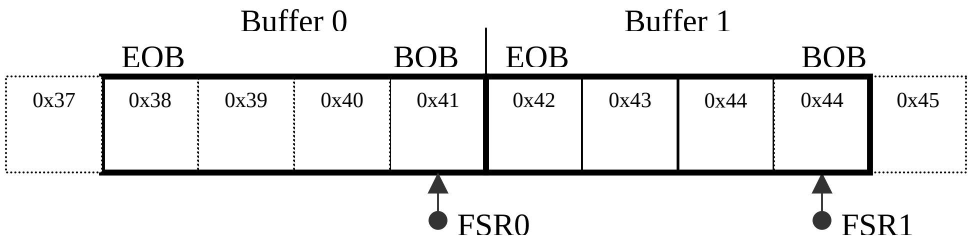


Figure 3.3.20 Topology of the Double Circular Buffers

Once again the same data is being stored in the buffer, each data element is stored in the same respective place in both buffers. If the first sample element is '1' then both buffers will store the data and post-decrement in the same manner as if each was an independent buffer.

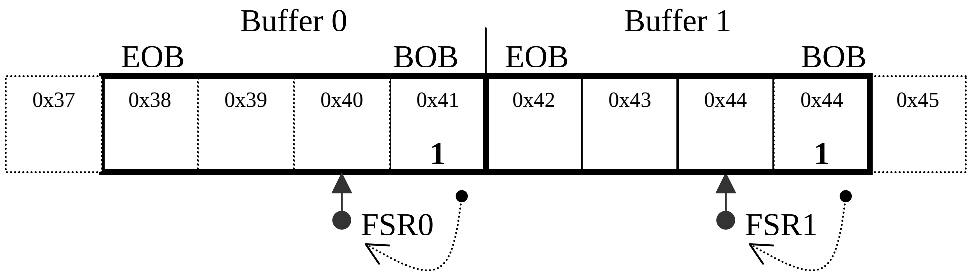


Figure 3.3.21 Data Element 1 is loaded to both buffers

Since both pointers move in tandem, only one needs to be checked for EOB and although this technique takes a little more time to load, it saves a lot more time during the unload. Since FIR filtering involves only a single load and N number of unloads (N being the number of coefficients), over all this technique produces a tremendous savings in computation time for each FIR output calculation.

After '1' the next data sample is '2' then '3' then '4' followed by '5' and the buffer will fill in the manner illustrated in Figure 3.3.22.

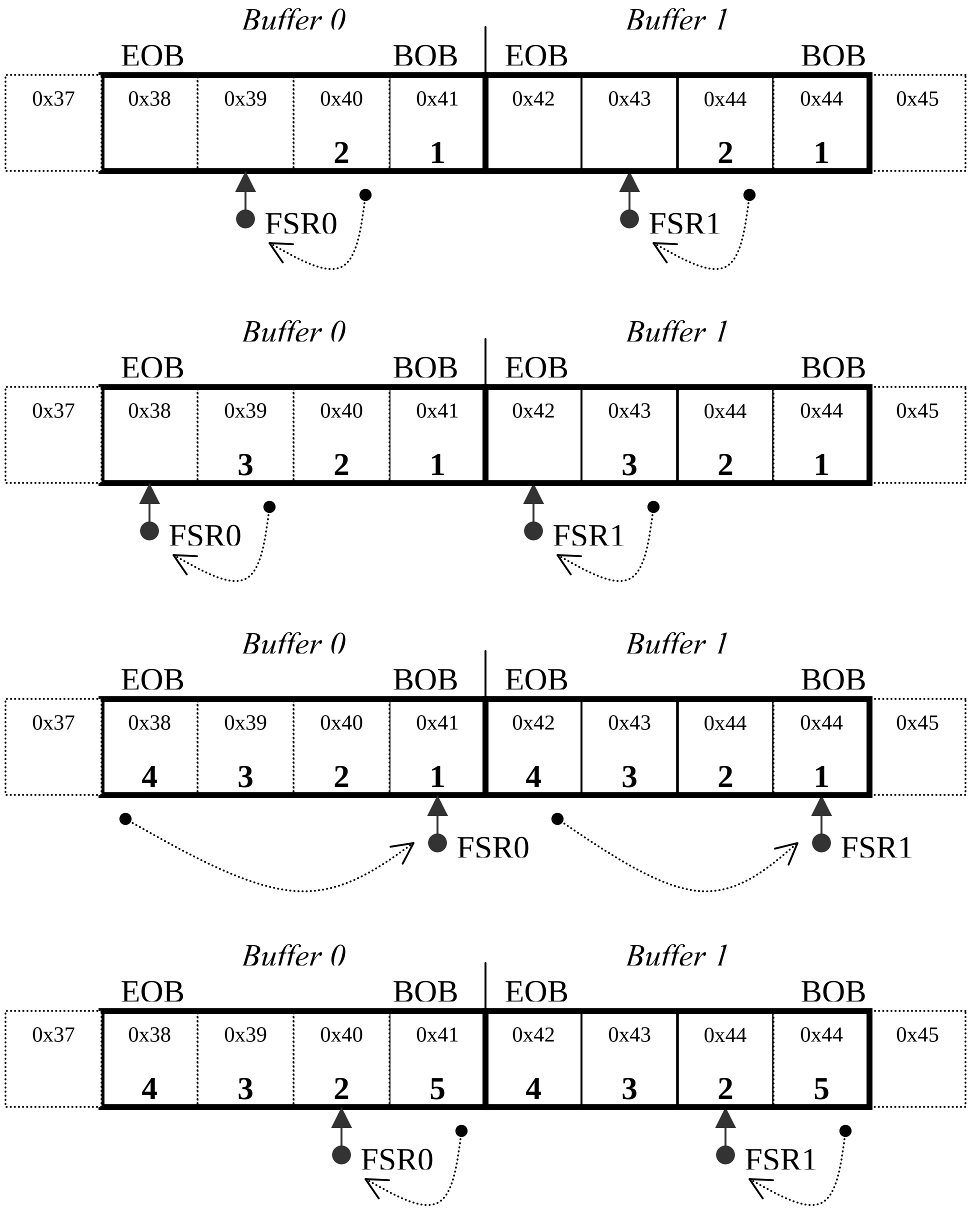


Figure 3.3.22 Data Elements 1-5 are loaded to buffer

Unloading data from the buffer involves a slightly different technique than what is used for a single buffer. Since the FIR filtering algorithm involves a convolution operation, after every fresh sample is stored the filter needs to unload each data point in reverse chronological order to perform computation. Since the size of the buffer is known, say *N*, there is no need to test of end-of-buffer or beginning-of-buffer while pulling the data because pointer FSR1 can now simply cross over from its own buffer into the adjacent one and always find the chronologically correct sample, sitting beyond the barrier of the adjoining buffer. To illustrate the point say we wanted to pull data from the current buffer. The last data sample stored was '5' as shown in Figure 3.3.23.

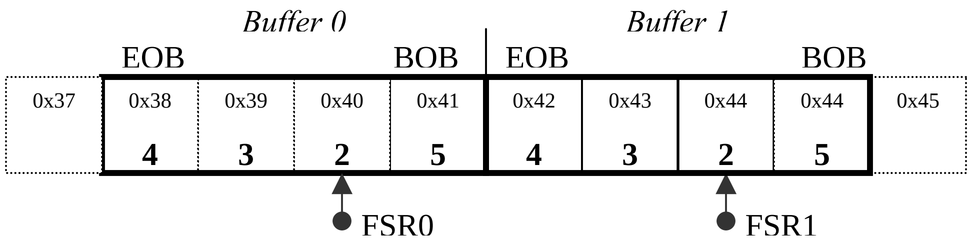


Figure 3.3.23 Buffer Data Ready to be unloaded

Data can be pulled in ascending order or descending order depending on which of the two pointers are used. For FIR filter calculations the order of the sample is not important because the coefficients are symmetric. If descending order were desired we would first relocate FSR1 to the same location as FSR0 as shown in Figure 3.3.24.

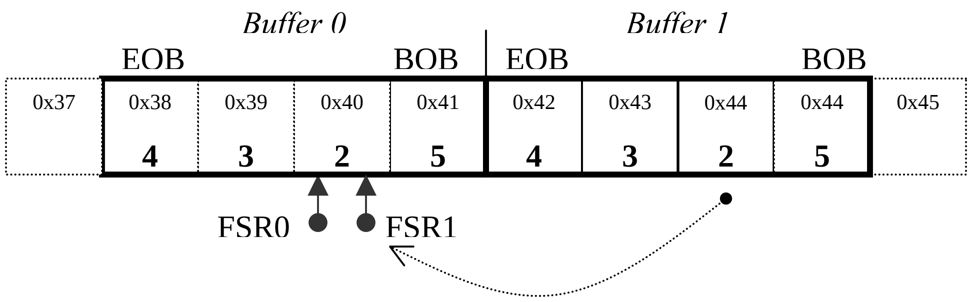


Figure 3.3.24 Pointer FSR1 is relocated to same location as FSR0

All that remains now is to pre-increment the pointer and pull the respective data sample from each location the pointer passes till it returns to the position it started (location 0x44).

The first data sample to be removed is ‘5’ as the pointer FSR1 pre-increments from location 0x40 to 0x41, both operation in one cycle as shown in Figure 3.3.25.

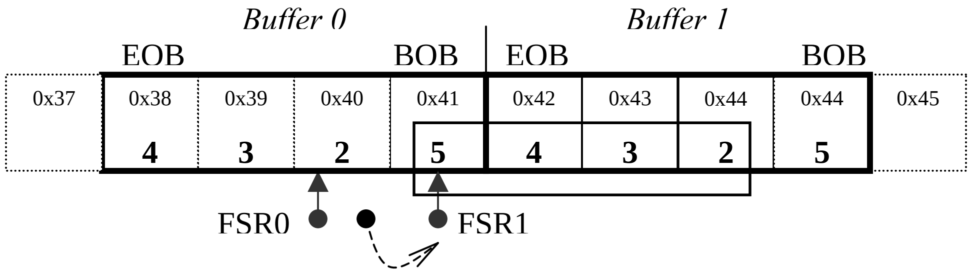


Figure 3.3.25 FSR1 pre-increments and unloads data element 5

Notice that data samples ‘5’,’4’,’3’ and ’2’ are in chronological order across both buffers. Since the buffer size is a constant, 4, then four blind pre-increment operations will unload the buffer and the pointer will automatically be returned to the point of insertion of the next sample. Using this method neither EOB nor BOB needs to be checked while unloading the buffer.

The Figure 3.3.26 shows the flow-chart describing the algorithm for loading the adjacent circular buffers and Figure 3.3.27 shows the assembly language code written to implemented it.

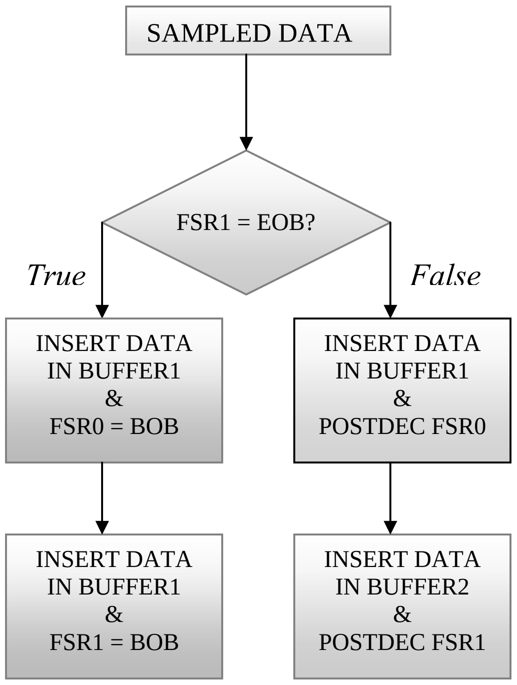

SAMPLED DATA
FSR1 = EOB?
True
False
INSERT DATA IN BUFFER1 & FSR0 = BOB
INSERT DATA IN BUFFER1 & POSTDEC FSR0
INSERT DATA IN BUFFER1 & FSR1 = BOB
INSERT DATA IN BUFFER2 & POSTDEC FSR1

Figure 3.3.26 Algorithm for loading the adjacent circular buffers

```
// Assembly code to load data into adjecent circular buffers

#asm

        mov     EOB,W           // Check for End of Buffer
        cpfseq  FSR0L
        bra     NEQ             // If end of buffer is not reached goto NEQ:
                                // else
        movff   DATA,INDF0      // move data into last pointed location
        movff   DATA,INDF1      // move data into last pointed location
        movff   BOB0,FSR0L      // Reset FSR0 to Begining of buffer0
        movff   BOB1,FSR1L      // Reset FSR1 to Begining of buffer1
        bra     end
NEQ:
        movff   DATA,POSTDEC0   // Load data to Buffer0 and post-decrement
        movff   DATA,POSTDEC1   // Load data to Buffer1 and post-decrement
END:

#endasm
```

Figure 3.3.27 Assembly routine written to load the buffers

The flowchart outlines the algorithm for pulling data from the buffers is shown in Figure 3.3.28:

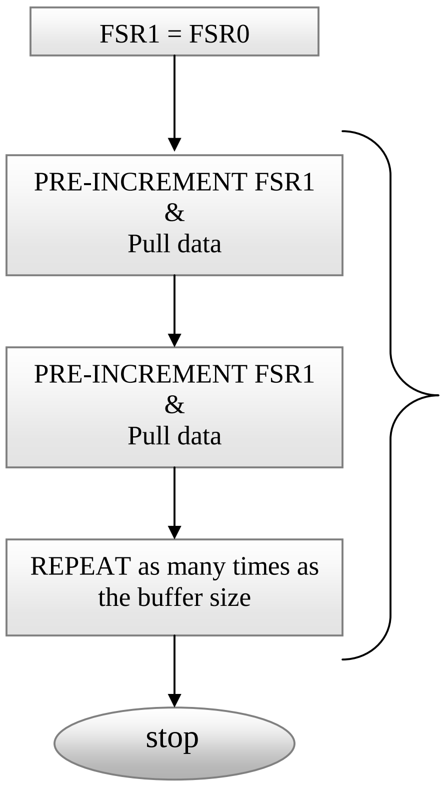

FSR1 = FSR0
PRE-INCREMENT FSR1 & Pull data
PRE-INCREMENT FSR1 & Pull data
REPEAT as many times as the buffer size
stop

Figure 3.3.28 Algorithm used to unload from adjacent circular buffers

## 3.3.6 Sampling: Analog to Digital Conversion on 18F452

The most convenient option for analog to digital conversion is by using the integrated ADC module. The built in analog to digital converter uses a successive approximation algorithm and is capable of converting an analog voltage into a proportional 10-bit number. The ADC is capable of a maximum sampling rate of 52KHz for 10Bit conversions. For 8-Bit conversions, the maximum sampling rate is 62.5KHz at same temperature and impendence [10].

The value sampled by the ADC is stored in the register pair ADRESH/ADRESL. Each is 8 bits, ADRESH contains the high-byte and ADRESL holds the low-byte. In order to configure the ADC module the ADC control register pair ADCON0 and ADCON1 must be set with appropriate values.

The analog-to-digital converter module has eight input channels for the PIC 18F452. Each input is a separate channel multiplexed with a common converter. This allows sampling of several different sources in any specified order. Since there is only one analog-to-digital converter simultaneous sampling is not possible using the internal *ADC* module. The minimum wait time between the sampling of any two channels is called acquisition time. The acquisition time is a function of the ambient temperature and the source impedance. The maximum recommended source impedance or input impedance for analog sources is 2.5K. For the FIR filter only one channel is needed but for the adaptive filter two channels must be sampled nearly simultaneously.

The first task to setting up the ADC is to setup the control register pair ADCON0 and ADCON1. Both registers are eight bits wide and allow unrestricted read/write operations. The ADCON0 register controls ADC clock options, channel selection, and the bit GO/DONE in the ADCON0 register can be polled in order to check if analog to digital conversion is complete. The ADCON1 register controls the remaining clock options, shared with ADCON0 and selects which pins are configured as digital and which are analog. In order to setup the internal ADC both registers must be loaded with the appropriate values.

Configuring ADCON0 involves setting five bits on the register. The following is the contents of the ADCON0 Register shown in Figure 3.3.29.

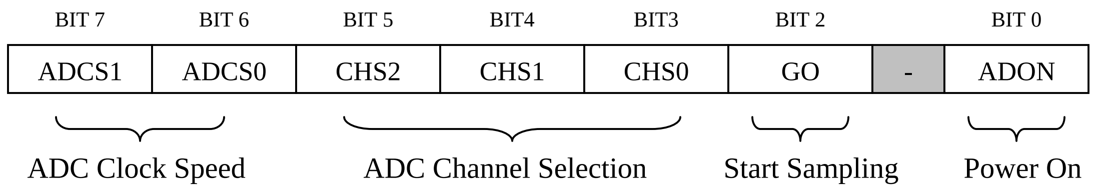


Figure 3.3.29 Description of the ADCON0 Register

The ADC clock is derived from the main external oscillator. The PIC chip can run up to 40Mhz but the ADC clock cannot exceed 625KHz. Therefore the PIC must use a clock divide to scale the 40MHz external frequency to 625KHz, a factor of 64. Hence the ADCS1 and ADCS0 are 1, 0 to make the clock divider equal to 64. The channel for FIR filter is channel-0 hence the CHS2, CHS1, CHS0 are 0,0,0 and ADON is 1.

**ADCON0 = <1 0 0 0 0 1 0 1> or 0x85**

The ADCON1, shown in Figure 3.3.30, is set in a similar manner as ADCON0 and it contains:

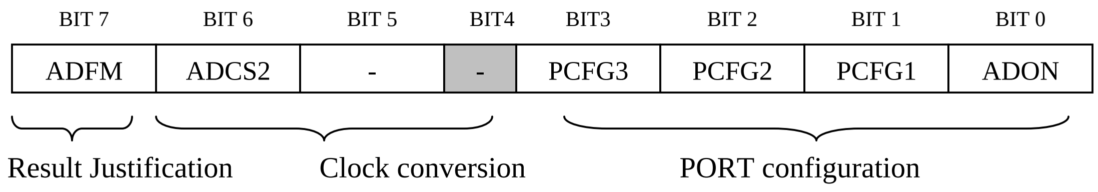


Figure 3.3.30 Description of the ADCON1 Register

The ADC stores a 10-bit result in two 8-bit registers. The ADFM bit selects if the result is left justified or right justified. Since the FIR filter is going to use 8-bit samples instead of 10-bit samples, ADFM will be set to 0 to make the result left justified. A simple way to get a fast 8-bit approximation of the 10-bit sample is to only read the *ADRESH* register as illustrated in Figure 3.3.31.

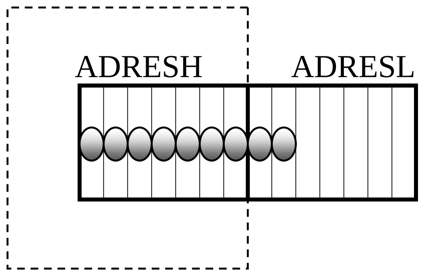


Figure 3.3.31 Reading Only ADRESH will scale down to 8-bit

The ADCS2 bit is set to 1 to make the clock divide equal to 64 as discussed before. Bits <PCFG3, PCFG2, PCFG1, PCFG0> are set to 1,1,1,0 respectively. This allows pins A0 to be analog while all other pins are made digital. Since technically only a single analog pin is required to make a FIR filter. If more analog pins are needed then this register needs to be changed. The ADCON1 register is loaded with the following:

**ADCON1 = <0 1 0 0 1 1 1 0> or 0x4E**

$V_{DD}$ and $V_{SS}$ are used as voltage references with this configuration.

The FIR filters performance depends not only on sampling signals accurately but also on a chips ability to maintain a constant sampling rate. To this end, one of the PIC chips three hardware timers; timer1 is dedicated to performing analog to digital conversion at a periodic rate. This is a 16-bit timer that derives its timing from the external clock source and interrupts the PIC chip when it overflows. Once the timer is engaged it counts from 0 till 65535 at the what ever speed it been clocked and at the end of its count generates an interrupt.

In order to make a constant sampling rate the timer is not allowed to start from 0 but instead made to start from some offset value from which it will pass 65535 at a predictable interval since the clock speed to the timer is known. This offset value is calculated using the following way:

$$timer1\_offset = 65535 - \frac{external\_osc}{4 * prescaler * sampling\_rate} \qquad (3.2)$$

In this case, the external oscillator *(external-osc)* is 40 MHz and the user determines the sampling rate in the design stage.

The interrupt service routine for timer1 will also perform all the calculations required of the filter and before exiting the Interrupt Service Routine (ISR) the result of the filter is generated.

### 3.3.7 Filter Calculations

Three different strategies are used to perform the necessary filter calculations.

1. Optimized for maximum Speed
2. Optimized to use minimal Ram
3. Optimized to generate smallest program size.

All three strategies make use of the same general idea but are different in the way the data is buffered and computation is performed. In general the realization of FIR filters is obtained by the direct computation of the Equation 3.3 [12].

$$y_n = \sum_{0}^{N} x\left[n-N+1\right]*[K_{N-1}+128] - \sum_{0}^{N} x[n-N+1]*128 \qquad (3.3)$$

The equation presented above is a variation of the classical FIR filter equation that is presented in most books:

$$y_n = \sum_{0}^{N} x[n - N + 1] * k_{N-1} \tag{3.4}$$

In both Equations 3.3 and 3.4, the term $y_n$ is the output of the filter and is computed by the linear convolution of the coefficient matrix $K_N$ and the discrete sample vector $x_n$. Both equations perform exactly the same computation and produce the same results however Equation 3.3 is far more PIC18F452 architecture-friendly because the signed multiplication operation in Equation 3.4 has been removed. This will allow PIC to maximize the use of the unsigned hardware-multiplier in the PIC hardware.

The only difference between Equation 3.4 and Equation 3.3 is that in Equation 3.3 the tap co-efficient vector $K_N$, which contains signed numbers ranging from –128 to +128 are made unsigned by adding to them the integer 128. In order to balance the result from the offset coefficients it becomes necessary to subtract $128*\Sigma\ x_n$ from $y_n$. To illustrate this point, consider the following analogy. If we wanted to calculate the $A$, which is a product between 8-Bit signed integer $B$ and 8-Bit unsigned integer $C$, it will be given by:

$$A = B * C \tag{3.5}$$

The above computation will require a signed multiplication however if we modified the above equation in the following manner:

$$E = B*(C+128) - B*128 \tag{3.6}$$

then we ultimately achieve the very same result as *A* and avoid the signed multiplication altogether.

$$A = E \tag{3.7}$$

### 3.3.8 Implementation for Shortest Execution Time

As stated before, three different implementations are possible using the filter design system. The first has the shortest possible execution time and possibly the most attractive implementation of all. The short execution time is achieved at the expense of higher RAM usage, since two adjacent circular buffers are used to store ADC samples instead of one. This doubles RAM use and also produces a much larger program, because to fully make use of the double buffer, the main multiply-accumulate loop is unrolled allowing for program to approach its theoretical minimal computation time, given the data word length constraints that is used by the program [12].

The implementation is split into two routines. The first one is the initialization Routine and the second is the Computation Routine. The initialization routine runs just once when the program begins and it serves only to initialize the buffers and other variables that are required for FIR filter calculations. The computation routine performs all calculation mandated by the filter and runs inside the interrupt service routine of timer1. With the confinement of all filter calculations inside the ISR, we achieve a degree of isolation making it possible for any main application to use the filter and not interfere with its operation or timing.

Memory Usage: The PIC 18F452 chip contains 1536 bytes of RAM and two addressing modes. There is direct-addressing and indirect-addressing. Indirect-addressing uses three pointers — FSR0, FSR1, FSR2 and each pointer is 12-Bit wide, with a 4-Bit select bank and 8-Bit select location within a bank. All memory use for the FIR filter is restricted to a single bank, thereby limiting the available memory for ADC samples and filter coefficients to a total of 256 Bytes. The obvious advantage of limiting all pointers use to a single bank is the speed that is gained because the pointers can be used faster if the bank does not need to be set before every call. Three buffers are used — a static buffer for tap coefficients that is loaded and initialized at start up and two identical adjacent circular buffers for the incoming ADC samples. All three buffers are of the same size and each buffer is given its own hardware pointer. Consider Figure 3.3.32 showing memory footprint

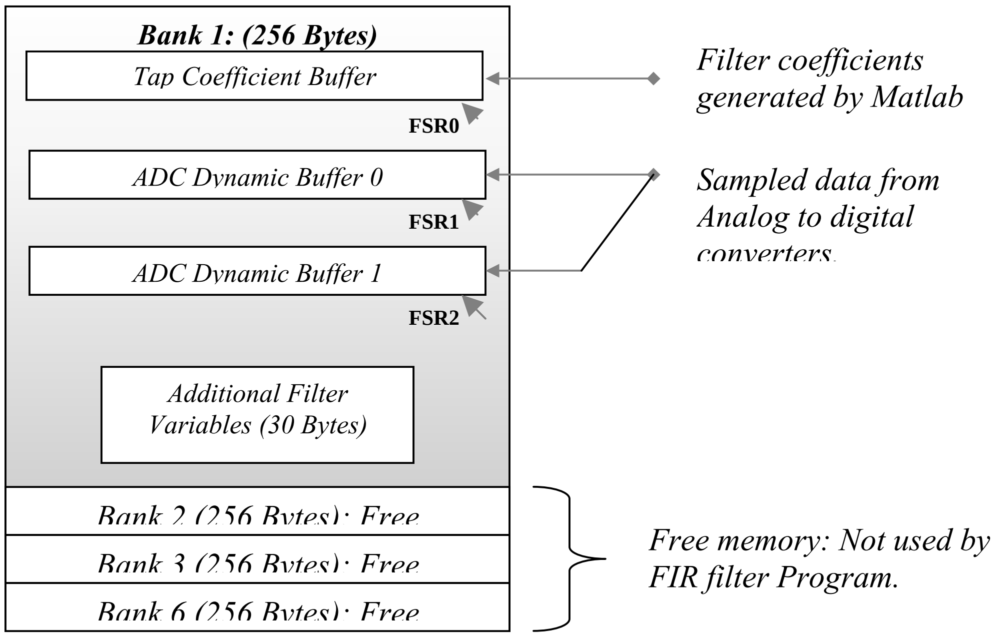

Bank 1: (256 Bytes)
Tap Coefficient Buffer
FSR0
ADC Dynamic Buffer 0
FSR1
ADC Dynamic Buffer 1
FSR2
Additional Filter Variables (30 Bytes)
Bank 2 (256 Bytes): Free
Bank 3 (256 Bytes): Free
Bank 6 (256 Bytes): Free
Filter coefficients generated by Matlab
Sampled data from Analog to digital converters.
Free memory: Not used by FIR filter Program.

Figure 3.3.32 RAM used by the FIR filtering scheme

The buffers for the tap coefficients and ADC values can be of variable size, since the number of FIR filter coefficients is not fixed. The total memory allocated to the buffers cannot exceed 226 bytes. No buffer can be larger than 75 elements. Hence this design does not allow for FIR filters larger than 75 taps. The MATLAB user interface will generate a warning if the user specifies a filter that generates more than 75 coefficients and the user is prompted to either accommodate fewer taps or select a different implementation strategy.

**Filter Processing: (Initialization Routine)**

If the filter is to produce an output, then an output port needs to be assigned, the first step of the initialization routine being the setup of an I/O port. Analog to digital converter is set to 8-Bit mode by simply left-shifting the results and reading the ADRESH register *(See 3.3.6)*.

The timer1 interrupt must be set to sample at the user specified sampling rate. All filter processing is conducted in the timer interrupt. There are three registers associated with Timer1. These are T1CON*,* TMR1H*,* and TMR2L. The first is the control register the other two are offset registers that are used to initialize the timer. Timer1 can be setup as an 8-bit as well as 16-bit as shown in Figure 3.3.33.

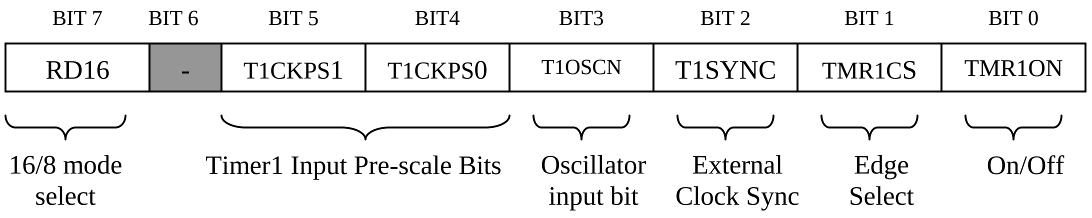


Figure 3.3.33 Description of the T1CON Register

In order to setup the timer bit RD16 = 1, T1CKPSX is calculated in Matlab and set based on user specified sampling rate. T1OSCEN, T1SYNC are not used, TMR1CS = 0 to specify internal clock and TMR1ON is 1 to power on the timer.

The timer offset is calculated using equation discussed in section 3.3.6. The 16-Bit offset is loaded into register pair TMR1L, TMR1H. All the registers are automatically loaded by code generated by the MATLAB program based on the sampling frequency selected by the user.

Before the timer is engaged the initialization routine loads the coefficients into the coefficient buffers and both the ADC buffers are set up, thereby initializing the pointers. Pointer FSR0 is used to load the co-efficient buffer and pointers FSR1, FSR2 are used to address the ADC buffers. ADC buffering method is discussed in detail in section 3.3.5. Once all the buffers are initialized the timer1 is started.

**Filter Processing: (Computation Routine)**

The computation routine involves the real-time implementation of the FIR filtering algorithm given in Equation 3.8.

$$y_n = \underbrace{\sum_0^N x\left[n-N+1\right]*[K_{N-1}+128]}_{\textit{Multiply \& Accumulate}(Y_1)} - \underbrace{\sum_0^N x[n-N+1]*128}_{\textit{Accumulate } (\Sigma \mathrm{x})} \tag{3.8}$$

In order to compute filter output Yn or Equation 3.8 is broken into three different Equations 3.9, 3.10 and 3.11.

$$y_1 = \sum_0^N \underbrace{x\left[n-N+1\right]}_{\textit{Sample vector}} * \underbrace{[K_{N-1}+128]}_{\textit{Tap Coefficients}} \tag{3.9}$$

$$\sum x = \underbrace{\sum_0^N x[n-N+1]*128}_{\textit{Sample Sum}} \tag{3.10}$$

$$Y_n = Y_1 - \sum x \tag{3.11}$$

Equation 3.9 is implemented using a fixed-point multiply-accumulate operation block. The MAC block is repeated for N times till $y_1$ is computed. A 24-Bit register comprised of three 8-bit registers is assigned to hold the MAC result shown in Figure 3.3.34.

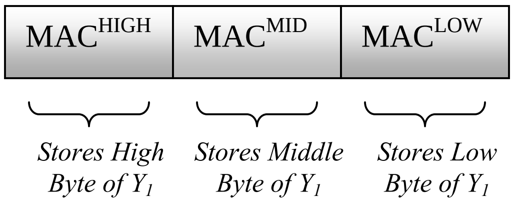


Figure 3.3.34 Word space dedicated to storing MAC result

Equation 3.10 computes the sample-sum of all the samples held in the ADC buffers. Σx is given a 16-Bit unsigned variable comprising to register pair $\Sigma x^{LOW}$ and $\Sigma x^{HIGH}$, shown in Figure 3.3.35. It is calculated with minimal computational effort by simply subtracting from the total, the oldest ADC sample and adding the newest one every time a new sample is made. Thus a running total of all the samples in the buffer is constantly maintained without having to add up every value in the buffer each time a new sample is added to it.

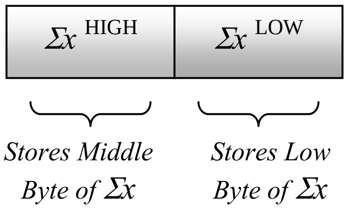

$\Sigma x^{HIGH}$
$\Sigma x^{LOW}$
Stores Middle Byte of $\Sigma x$
Stores Low Byte of $\Sigma x$

Figure 3.3.35 Word space dedicated to storing Sample Sum

Σx has to be multiplied by 128 and subtracted from $Y_1$ to obtain final output Yn. An efficient way to multiply by 128 is to copy the sum into another 24-bit variable i.e. moving $\Sigma x^{LOW}$ to $255\Sigma x^{MID}$ and $\Sigma x^{HIGH}$ to

$255\Sigma x^{HIGH}$ and clearing the $255\Sigma x^{LOW}$ will do an implicit multiply by 255 as shown in Figure 3.3.36.

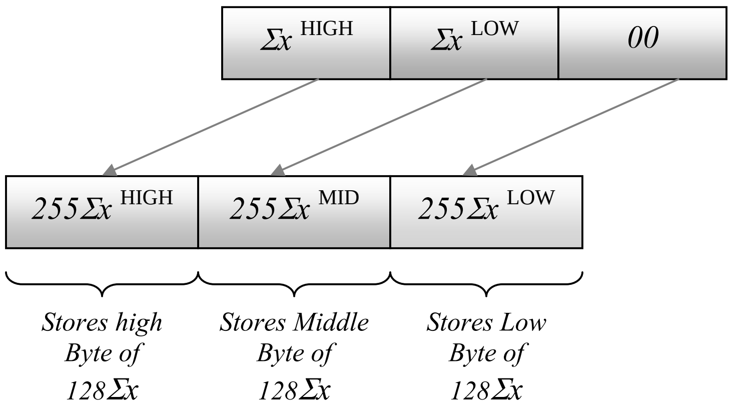


Figure 3.3.36 Multiply by 256 algorithm

Once moved a single right-shift with carry on all three registers produces the required multiply by 128 shown in Figure 3.3.37.

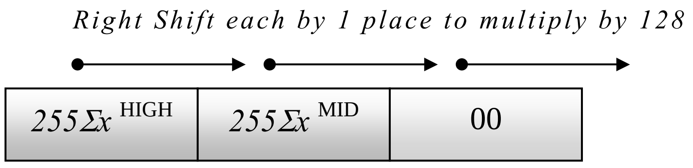


Figure 3.3.37 Multiply by 128: Single Right shift of each byte

The MAC block for computing $Y_1$ uses three buffers that are located in RAM. The Buffers shown here are 4 elements long but can extend up to 75 elements depending on filter requirements. All buffers are dynamically scaled depending on filter requirements. A double buffering scheme is used for storing ADC samples and a single static buffer is used to store filter tap coefficients. FSR0, FSR1 and FSR2 pointers dedicated to each buffer as shown in Figure 3.3.38 and Figure 3.3.39.

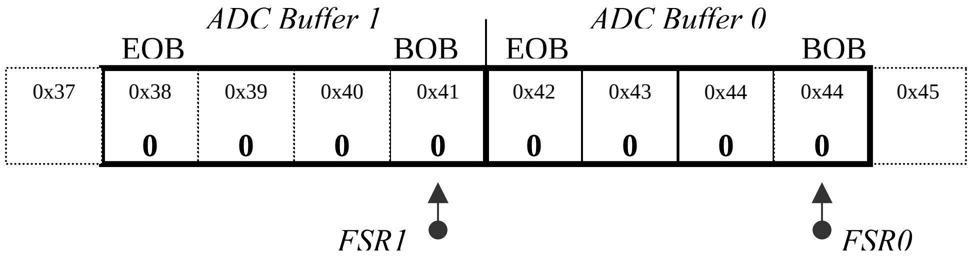


Figure 3.3.38 Buffer setup for storing ADC samples

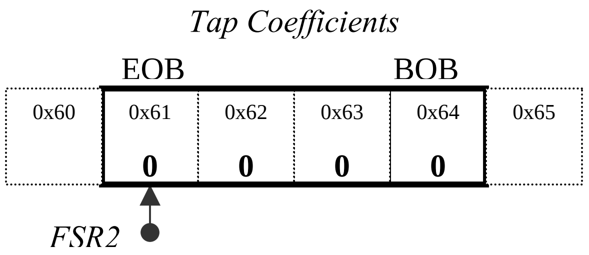


Figure 3.3.39 Buffer setup for storing Coefficients

The MAC cycle accumulator occupies three bytes of memory to store a 24-bit number as shown in Figure 3.3.40.

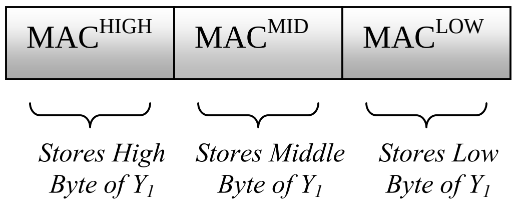
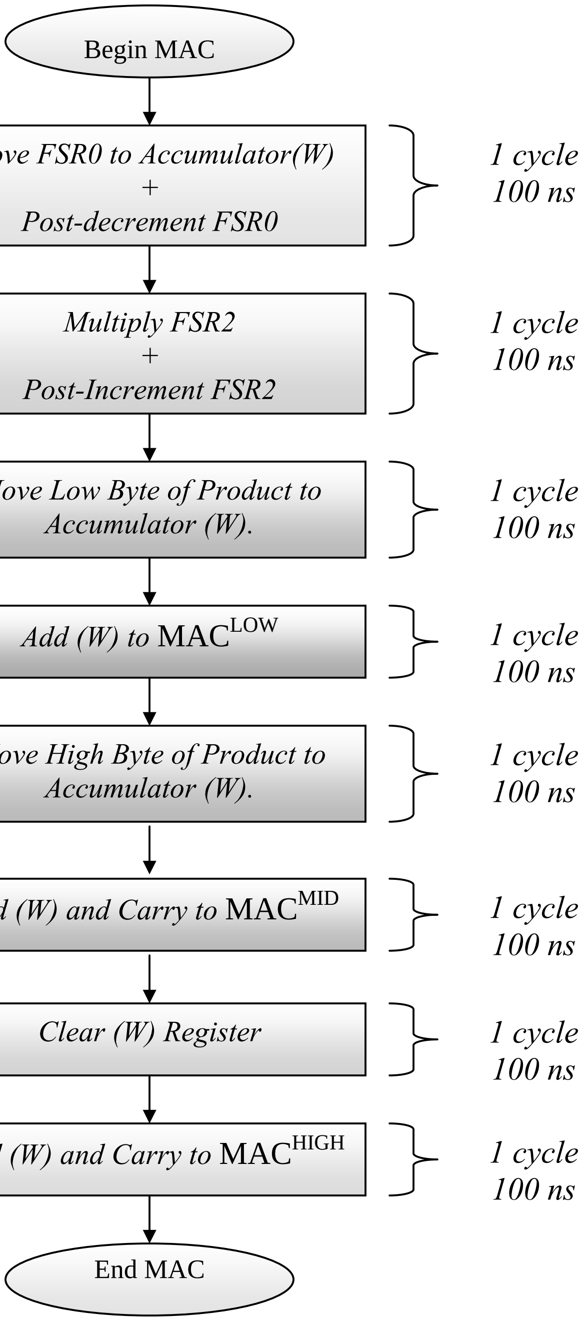


Figure 3.3.40 24-Bit Result store for MAC operation

Given the above constraints of 8-bit unsigned coefficients, 8-bit ADC samples and 24-Bit accumulator the quickest possible MAC on the PIC 18F452 is shown in Figure 3.3.41.



Begin MAC
Move FSR0 to Accumulator(W) + Post-decrement FSR0
1 cycle 100 ns
Multiply FSR2 + Post-Increment FSR2
1 cycle 100 ns
Move Low Byte of Product to Accumulator (W).
1 cycle 100 ns
Add (W) to MAC^LOW
1 cycle 100 ns
Move High Byte of Product to Accumulator (W).
1 cycle 100 ns
Add (W) and Carry to MAC^MID
1 cycle 100 ns
Clear (W) Register
1 cycle 100 ns
Add (W) and Carry to MAC^HIGH
1 cycle 100 ns
End MAC

Figure 3.3.41 Multiply-Accumulate Algorithm

The entire MAC cycle lasts 800ns and the assembly code generated for it is as follows:

```
movf     POSTDEC0,W       // Move element pointed by FSR0 to (W)
mulwf    POSTINC2         // Multiply FSR2 and Post-increment
movf     PRODL,W          // Move Product Low-byte to (W)
addwf    output_least     // Add (W) to the MAC^LOW
movf     PRODH,W          // Move Product High-Byte to (W)
addwfc   output_middle    // Add carry + (W) +  MAC^MID
clrf     WREG             // Clear (W)
addwfc   output most        // Add carry + (W) +  MAC^HIGH
```

In order to complete the implementation of Equation 3.10 the pointer FSR2 is first moved to the same location as FSR1 then the MAC block is repeated as many times as the filter order. This way there is no need to check for end-of-buffer or the beginning-of-buffer and final MAC block terminates with the pointer automatically returned to the exact point of insertion of the next incoming sample. Figure 3.3.42 is a flow diagram for the initialization routine for a 4-tap FIR filter.

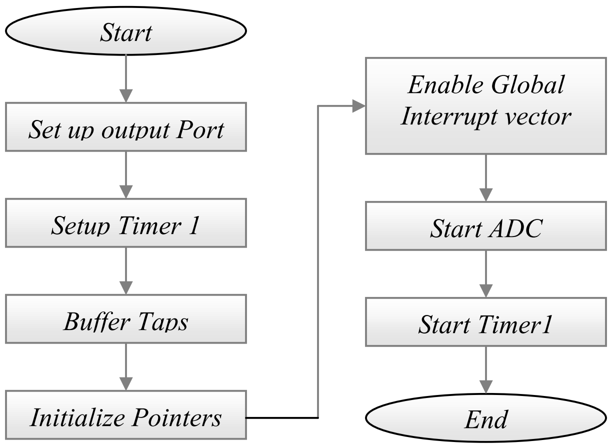


Figure 3.3.42 Initialization Routine for Fastest Execution

Figure 3.3.43 is a flow diagram for the computation routine the fastest execution time version of a 4 tap FIR filter.

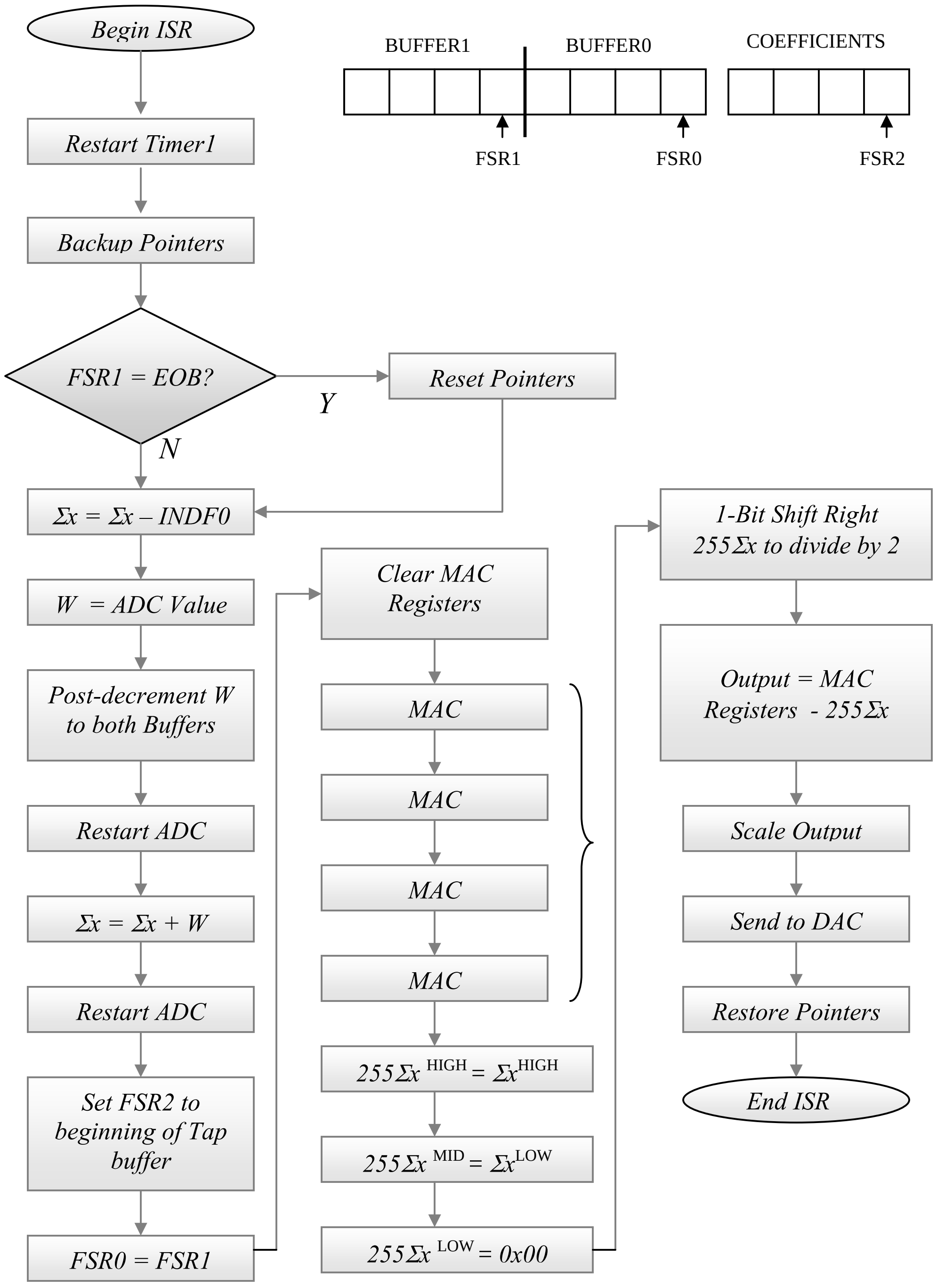


Figure 4.3.43 Fastest Execution Implementation for PIC 18f452

## 3.3.9 Implementation for Efficient RAM utilization

The use of two circular buffers for storing ADC values is at times not acceptable due to its extensive RAM overhead. Since the FIR filter is typically used as a supplementary application, it must therefore share the available RAM with a main application. It is for this reason a less memory greedy implementation scheme is developed.

This scheme uses most of the same ideas as the previous method. The coefficients are stored in memory in the same manner as before but the MAC cycle is computed differently because since there is only one buffer and both the end-of-buffer and beginning-of-buffer needs to be checked. The details of the circular buffer are presented in Section 3.3.5. Figure 3.3.44 is a flow diagram for the initialization routine for a 4 tap FIR filter.

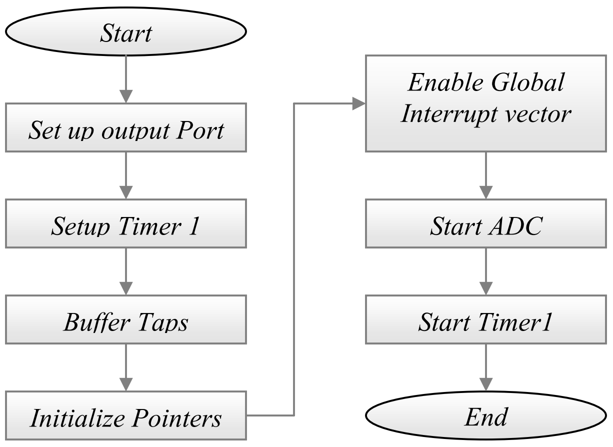
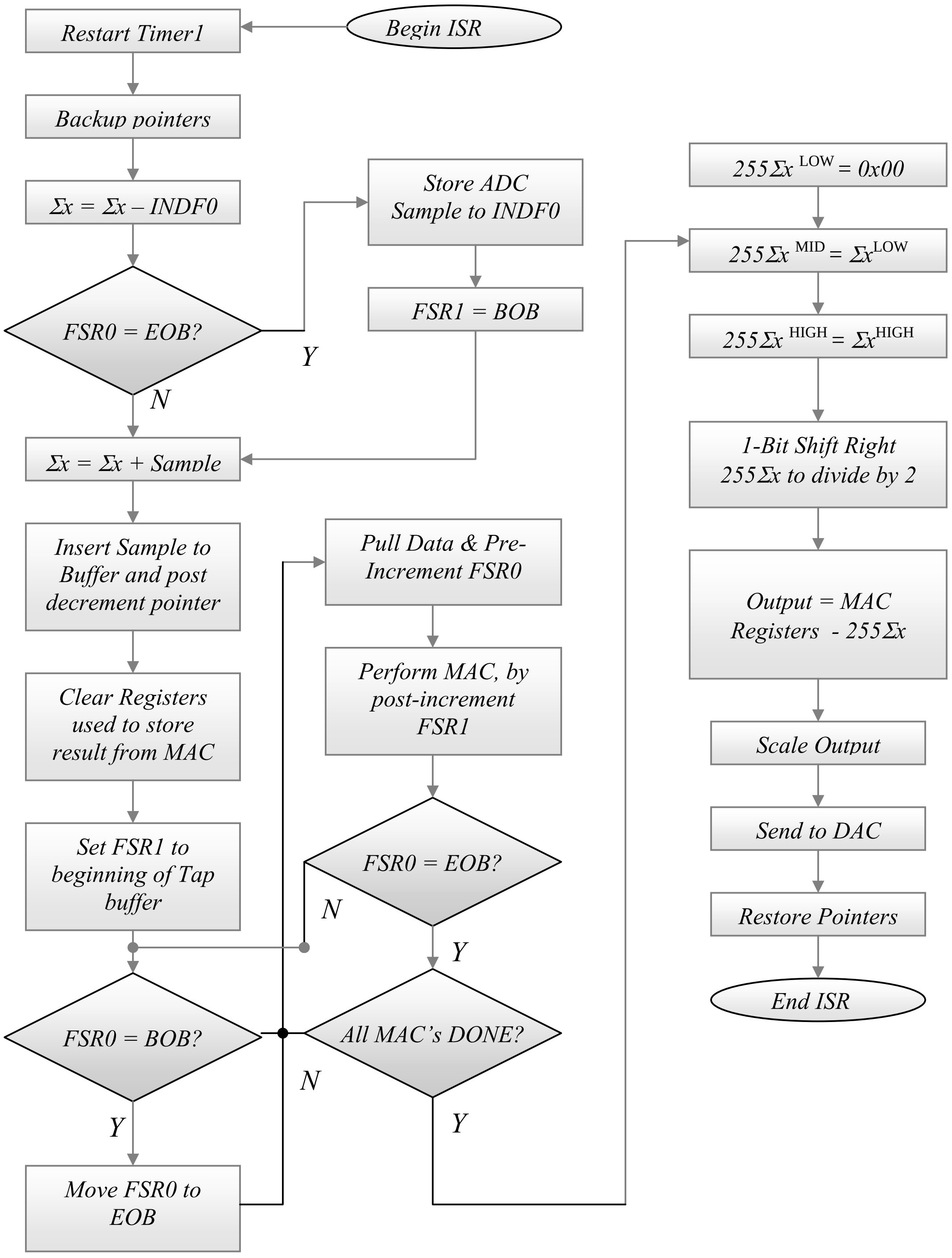


Figure 4.3.44 RAM efficient Implementation for PIC 18f452

Figure 3.3.45 is a flow diagram for the computation routine the fastest execution time version of a 4 tap FIR filter.



Begin ISR
Restart Timer1
Backup pointers
Σx = Σx – INDF0
FSR0 = EOB?
Y
N
Store ADC Sample to INDF0
FSR1 = BOB
Σx = Σx + Sample
Insert Sample to Buffer and post decrement pointer
Clear Registers used to store result from MAC
Set FSR1 to beginning of Tap buffer
FSR0 = BOB?
Y
Move FSR0 to EOB
N
Pull Data & Pre-Increment FSR0
Perform MAC, by post-increment FSR1
FSR0 = EOB?
N
Y
All MAC's DONE?
Y
255Σx LOW = 0x00
255Σx MID = Σx LOW
255Σx HIGH = Σx HIGH
1-Bit Shift Right 255Σx to divide by 2
Output = MAC Registers - 255Σx
Scale Output
Send to DAC
Restore Pointers
End ISR

Figure 4.3.45 RAM Efficient Implementation for PIC 18f452

## 3.3.10 Implementation for Minimum Program Memory Use

The implementation strategy is exactly like the first one where maximum execution speed was attained. In order to reduce program size, the main loop for the MAC cycle is not unrolled. Instead three more instructions are added into the MAC cycle. The computation cycle is shown in Figure 3.3.46.

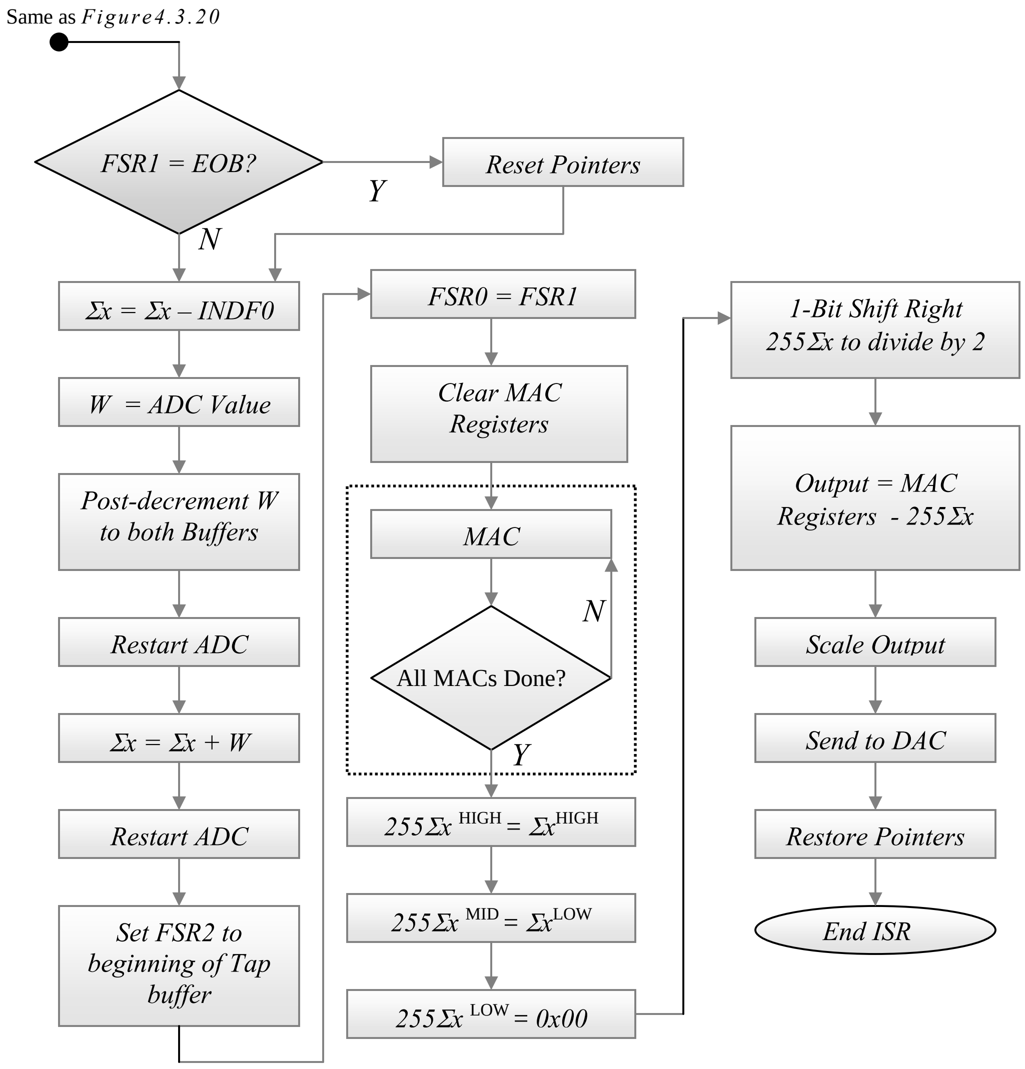


Figure 4.3.46 Minimum Program Size Implementation

## 3.4 Implementation of the floating-point LMS filter

Unlike the FIR filters that have predetermined coefficients, implemented as constant data, the coefficients of the Least-Mean Square (LMS) filter are adaptive and continuously change as a response to input. Due to this reason, several complications must be dealt with while designing and implementing them in hardware. Since the coefficients or filter weights change with input, they may grow so large they overflow the word-space assigned to them during design time.

Stability of the LMS filter is not as easily guaranteed as it is for FIR filters. The constantly adapting coefficients are controlled by a fixed value called the learning-rate. Determining an optimal value for the learning rate requires experience gained from simulations and as the order of the adaptive filter increases, thus choice for an appropriate learning rate becomes even less intuitive. Rigorous simulations were conducted before attempting to perform real-time implementation.

The choice of the floating-point system was used to perform the implementation because the floating-point system provides both convenience and degree of immunity against both roll-off errors as well as allowing for wider latitude in the selection of learning-rate.

## 3.5 The compilers floating point system

The compilers built in math abilities were evaluated to perform the necessary filter computations but later found to be inadequate because they were extremely slow. The lack of speed is attributed to several factors. Firstly, the compiler used generic routines that are designed to work on the entire PIC family rather than applying hardware specific optimizations for the PIC18452 chip. Secondly the generic algorithms are optimized to be compact and not for speed. This decision is certainly well warranted as floating point algorithms written for chips that do not contain floating point hardware can use a lot of code and the lower members in the PIC family have modest sized program memories.

In order to realize the LMS filter on the PIC 18 chip it became necessary to first develop a set of floating-point routines that are optimized for the hardware at hand. New math routines were optimized for speed and designed to perform floating-point calculations much faster than the compilers generic algorithms. A standard fixed-point realization might have been more efficient however in the long run a highly optimized floating point library is far more useful as it is a reusable resource and easily applied to many other projects in the future.

## 3.6 Floating-Point Word lengths

The word lengths used to define the stored values were selected from information gathered from simulations. Figure 3.6.1 shows the word lengths that were assigned to the floating-point format numbers were used in the implementation of the LMS filter.

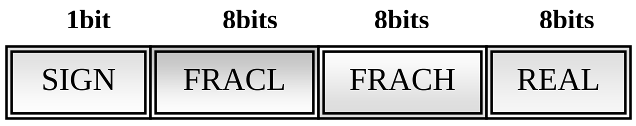


Figure 3.6.1 Assigned Word Length for Floating Point Format

The allocated word space is 1-bit for sign, 16-Bits for the fractional part of the number and 8-bits for the real part of the number. This allowed for the possible range of *[255.000000 to –255.000000]* with the smallest possible magnitude of *0.000015*. This was determined to be sufficient resolution to be able to handle the computation requirements of the LMS filter. The next step was to develop functions that would conveniently perform type conversions from the standard IEEE floating point to this modified floating point. Additionally other functions were developed to perform hardware-optimized operations such as signed multiplication and signed addition and a high-speed re-scaling algorithm was added to convert a number between [0 255] to [-0.5 0.5].

The following ideas were used to accelerate floating-point mathematics using the PIC hardware. Parameter passing was found to be the first obvious over-head because each math operation required the passing of variables into temporary ones that were then used to compute results. The computed result needed to be passed to the output variable. It takes 2 cycles to move a single byte from one register to another and considering large numbers occupy up to 4 bytes a total of 24 cycles were spent simply in the parameter passing. This overhead is easily avoided if hardware pointers are used to directly reference data. Since The PIC chip has 3 hardware pointers, 2 are used to reference the two input parameters and the last one is used to reference the output parameter. This allows efficient movement of data through memory and since the pointers auto increment or decrement, additional cycles are not lost to pointer overhead.

Table 3.6.1: Function list developed for floating point math on PIC

| FUNCTION | DESCRIPTION | TIME@40Mhz |
|---|---|---|
| void fixIeee(* float, *mfloat) | IEEE float -> modified float | Worst case (40us) |
| void fix8x16(float, *mfloat) | modified float -> IEEE float | Worst case(40us) |
| void add(void) | adds 2 modified signed floats | 5 us |
| void mul(void) | multiply 2 modified signed floats | 3 us |
| 255/integer -> mfloat | Normalize [0  255] -> [0.00  0.99] | 400ns |

## 3.6.1 Algorithm developed for floating point multiplication

The multiplication algorithm operates on two floating-point variables each stored in RAM in the format described Figure 3.6.2.

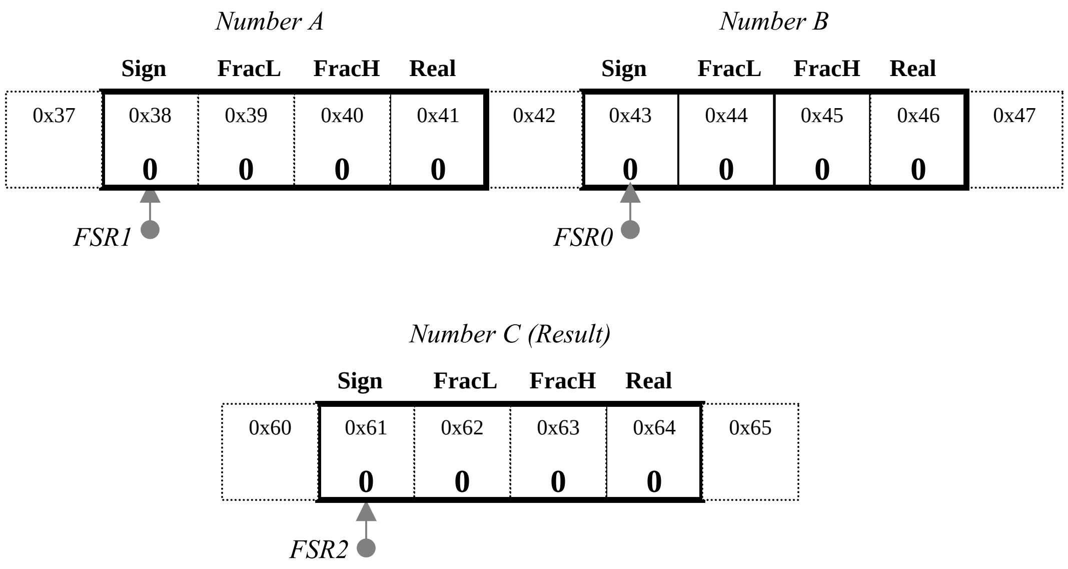


Figure 3.6.2 Memory footprint of floating point numbers

Numbers A, B and C are stored in RAM and each is given 4 bytes of memory. *FracL* and *FracH* make up the fractional portion given 16-bits of storage the real part or the integer part is given 8-Bits of storage each. In order to save time from parameter passing, pointers *FSR0*, *FSR1* and FSR2 are used to manipulate the floating-point variables. The multiplication algorithm is designed to take advantage of both the hardware-multiplier to compute the products of integer and fractional portions and *PIC18* memory addressing features that provide single cycle pointer operation and advance.

The formal algorithm for multiplication is described next. The three numbers are setup in memory as shows in Figure 3.6.2.

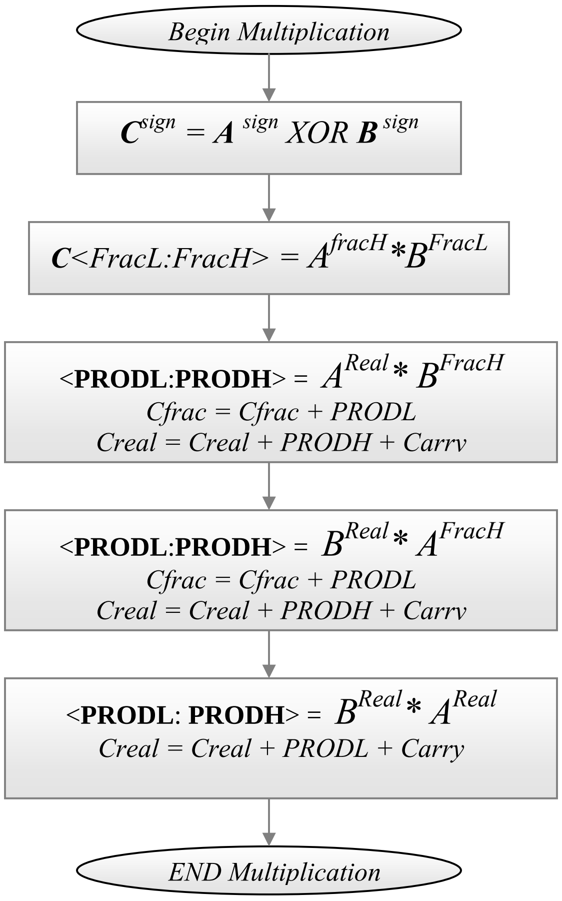
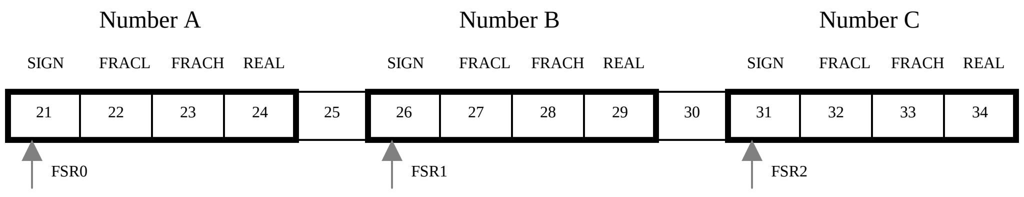

Begin Multiplication
$C^{sign} = A^{sign}$ XOR $B^{sign}$
C<FracL:FracH> = $A^{fracH} * B^{FracL}$
<PRODL:PRODH> = $A^{Real} * B^{FracH}$
Cfrac = Cfrac + PRODL
Creal = Creal + PRODH + Carrv
<PRODL:PRODH> = $B^{Real} * A^{FracH}$
Cfrac = Cfrac + PRODL
Creal = Creal + PRODH + Carrv
<PRODL: PRODH> = $B^{Real} * A^{Real}$
Creal = Creal + PRODL + Carry
END Multiplication

Figure 3.6.3 Developed Algorithm for Multiplication

The step-by-step illustration of the above algorithm is presented next along with the assembly code that was written to implement it. The purpose of such a detailed presentation is to clarify pointer use in the PIC18F452 chip and to show the functionality of the algorithm.

Step 1 shown in Figure 3.6.4, the numbers to be multiplied are stored in *RAM* and each is given it's own dedicated pointer as shown below. The first step is to determine the sign of the computed product.

Evaluating the *XOR* of the signs of the two numbers being multiplied results in the sign of *C*.



```
movf     POSTINC0,W     // Post Inc FSR0, Load Sign A into WREG
xorwf    POSTINC1,W     // Post Inc FSR1, XOR with Sign B
movwf    POSTINC2       // Post Inc FSR2, Move result to Sign C
```

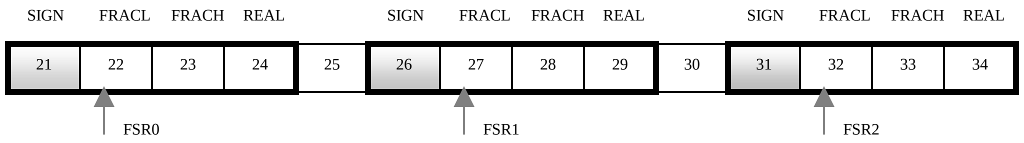


Figure 3.6.4 Multiplication Step 1: $\mathbf{C}^{sign}$ = $\mathbf{A}^{sign}$ XOR $\mathbf{B}^{sign}$

Step 2 shown in Figure 3.6.5, the fractional portion of the result *C* is evaluated next by computing the product: **C**<FracL: FracH> = $A^{fracH}*B^{fracH}$.

```
movf     PREINC0,W          // Pre Inc FSR0, Move AFracH to WREG
mulwf    PREINC1            // Pre Inc FSR1, Multiply WREG with BFracH
movff    PRODL, POSTINC2    // Post Inc FSR2, Move Product Low to CfracL
movff    PRODH, INDF2       // Move Product High to CfracH
```

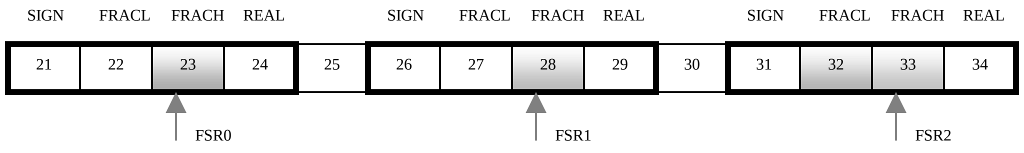


Figure 3.6.5 Multiplication Step 2: **C**<FracL: FracH> = $A^{fracH}*B^{fracH}$

Step 3 shown in Figure 3.6.6, The real part of result *C* is evaluated next by computing the product of $A^{Real} * B^{FracH}$ and adding the low-byte of the product to *Cfrac* and adding the high-byte of the product to Creal with the carry from the previous addition.

```
movf      PREINC0,W        // Pre Inc FSR0, Move AFracH to WREG
mulwf     POSTINC1         // Post Inc FSR1, Multiply WREG with BReaL
movff     PRODL,WREG       // Move Product Low to WREG
addwf     POSTINC2,F       // Post Inc FSR2, Add with W store in CfracH
movff     PRODH,WREG       // Move Product high to WREG
addwfc    POSTDEC2,F       // Post Dec FSR2, add (W+Carry+CfracL)
```

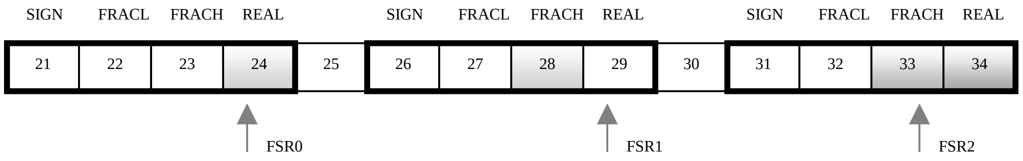


Figure 3.6.6 Multiplication Step 3: $A^{Real} * B^{FracH}$

Step 4 shown in Figure 3.6.7, The real part of result *C* is evaluated next by computing the product of $A^{FracH} * B^{Real}$ and adding the low-byte of the product to *Cfrac* and adding the high-byte of the product to Creal with the carry from the previous addition.

```
decf      FSR0L,F          // Dec FSR0,  Point to  AReaL
movf      POSTINC0,W       // Post Inc FSR0, Move AReaL To WREG
mulwf     INDF1            // Multiply BfracH with WREG
movff     PRODL,WREG       // Move Product Low to WREG
addwf     POSTINC2,F       // Post Inc FSR2, add WREG to CReaL
movff     PRODH,WREG       // Move Product High to WREG
addwfc    INDF2,F          //  Add (carry+WREG+CfracH)->CFeacH
```

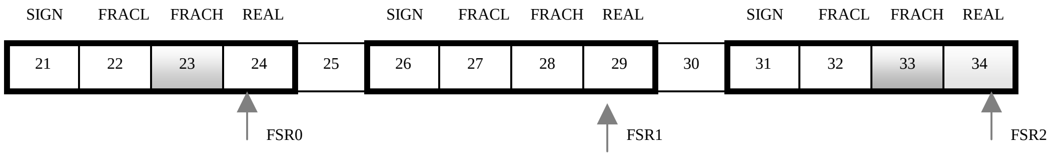


Figure 3.6.7 Multiplication Step 4: $A^{FracH} * B^{Real}$

Step 5 shown in Figure 3.6.8, The real part of number *A* and the real part of number *B* are multiplied next and the low byte of the result is added to the real part of number *C*. This concludes the multiplication operation.

```
movff      INDF0,WREG      // Move AfracL to WREG
mulwf      INDF1           // Multiply WREG with BfracL
movf       PRODL,W         // Move Product low to WREG
addwfc     INDF2,F         // Add with Carry with CFracL
```

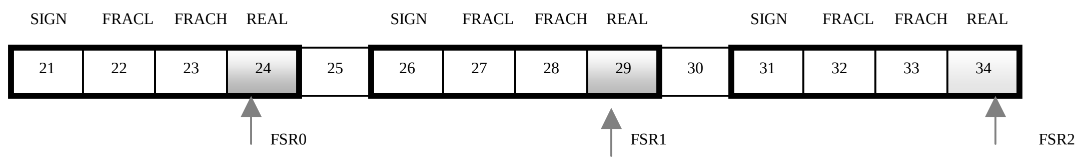


Figure 3.6.8 Multiplication Step 5: $A^{Real} * B^{Real}$

There are no conditions to be checked in the algorithm hence the two-stage pipeline of the PIC chip is constantly maintained. All instructions are single cycle (100ns) with the exception of the register-to-register move instruction (*movff*), which is two-cycle (200ns). The total time used by this algorithm is 3us. An additional advantage is that this algorithm always takes the same amount of time to execute. The compilers worst case floating point multiplication algorithm is 45us according to their published manual [3].

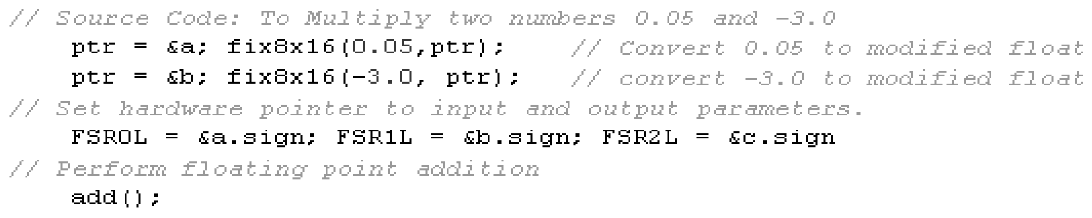
The use of the multiplication algorithm in C-language is demonstrated next. No condition checking is available to determine and warn users about over and underflows in the interest of efficiency. Figure 3.6.9 is the C-code needed to use the multiply function.

```
//  Source Code: To Multiply two numbers 0.05 and -3.0
    ptr = &a; fix8x16(0.05,ptr);     // Convert 0.05 to modified float
    ptr = &b; fix8x16(-3.0, ptr);    // convert -3.0 to modified float
// Set hardware pointer to input and output parameters.
    FSR0L = &a.sign; FSR1L = &b.sign; FSR2L = &c.sign;
// Call the multiply algorithm: completes in 3us
    mul();
```

Figure 3.6.9 C-Code for floating point multiplication

## 3.6.2 Calling The Floating-Point Add in C

The algorithm developed for performing floating-point addition was implemented along the same lines as the multiplication algorithm. Both operate on the same type of data format and both use hardware pointers to reference data.



```
// Source Code: To Multiply two numbers 0.05 and -3.0
    ptr = &a; fix8x16(0.05,ptr);     // Convert 0.05 to modified float
    ptr = &b; fix8x16(-3.0, ptr);    // convert -3.0 to modified float
// Set hardware pointer to input and output parameters.
    FSR0L = &a.sign; FSR1L = &b.sign; FSR2L = &c.sign
// Perform floating point addition
    add();
```

Figure 3.6.10 C-Code for floating point addition

## 3.6.3 Algorithm developed for floating-point addition

The floating-point addition algorithm was developed keeping in mind the fact that nether of the input parameters are corrupted during the addition process. To clarify the point assumes that two numbers A and B are being added to calculate C. After the addition is completed neither A or B will change in value. The algorithm would have been slightly shorter if this constraint were removed, however we would loose the ability to perform MAC operations where parameters are added to themselves. The detailed algorithm is illustrated in Figure 3.6.11.

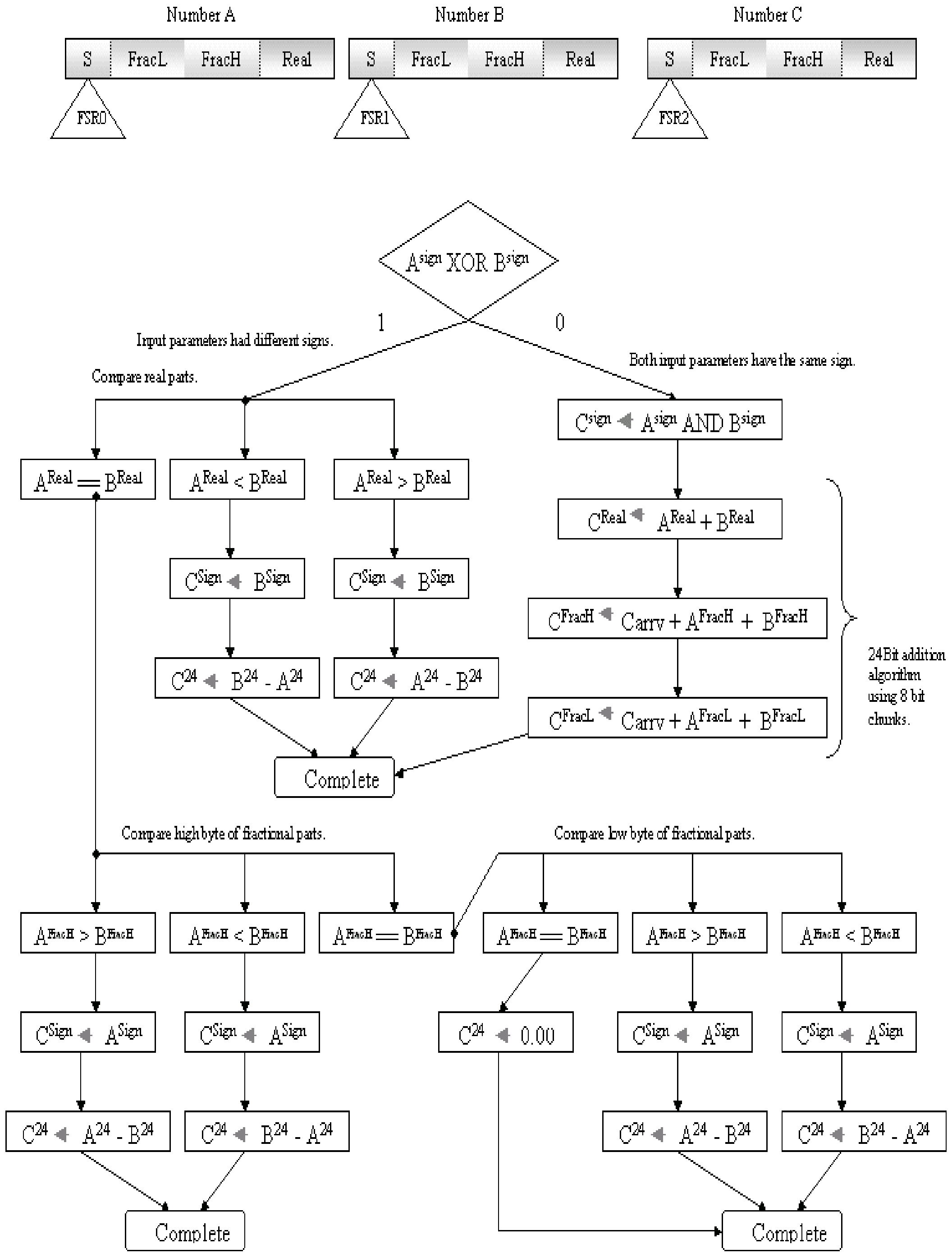


Figure 3.6.11 Developed Algorithm for Addition

## 3.6.4 Converting integer to floating-point format

A quick way to convert integer values to floating-point needs to be implemented because data read in by the analog-to-digital ranging from [0 255] needs to be converted to float range between [-0.5, 0.5].

$$Nfloat = (float)\frac{(unsigned\,\mathrm{int})N - 128}{256} \tag{3.12}$$

A division operation is out of the question because it’s computationally prohibitive if performed in real time. Installing a look up table was the first option however a more elegant approximation is given in Figure 3.6.12.

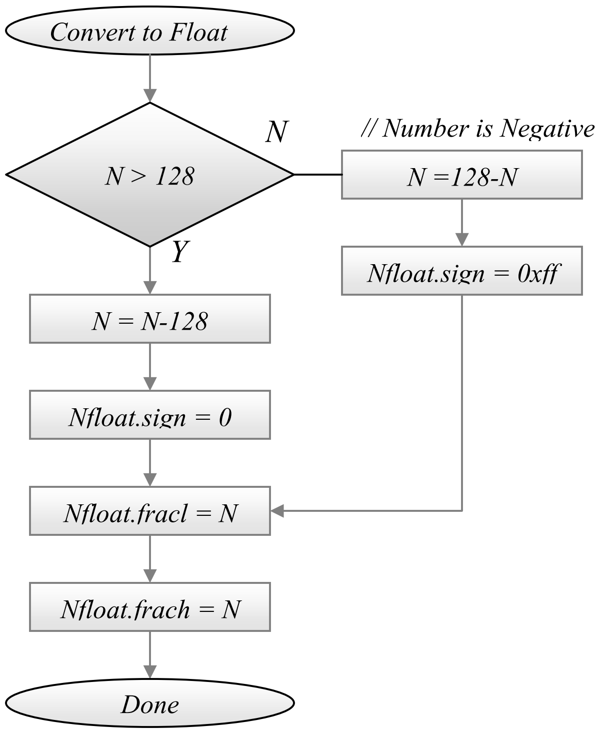


Figure 3.6.12 Developed Algorithm for Multiplication

## 3.7 Implementation of a 4th order real-time LMS algorithm

The topology of the LMS algorithm used is illustrated first. The filter samples two channels where $Y_k$ is the signal that needs to be filtered and $X_k$ is the reference. $W_0$, $W_1$, $W_2$ and $W_3$ are all weights or filter coefficients of the LMS filter. These are initialized as 0 however as the filter trains the weights converges to a solution value. The variable $e_k$ is called the error signal and it is both the output of the filter and the feedback signal that trains the filter weights.

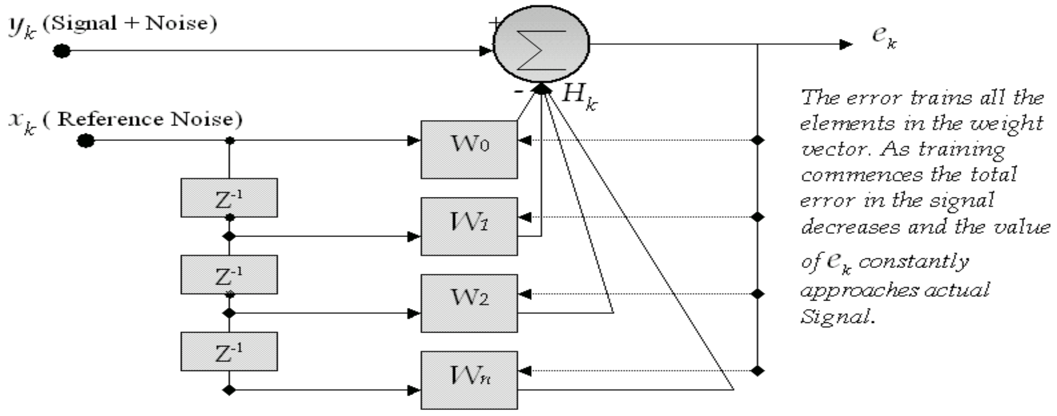

$y_k$ (Signal + Noise)
+
-
$H_k$
$e_k$
$x_k$ ( Reference Noise)
$Z^{-1}$
$Z^{-1}$
$Z^{-1}$
W0
W1
W2
Wn
The error trains all the elements in the weight vector. As training commences the total error in the signal decreases and the value of $e_k$ constantly approaches actual Signal.

Figure 3.7.1 Fourth Order LMS Filter

The filter equations are to be implemented and computed in real-time in the PIC 18F452 chip are presented next. The error signal $e_k$ is evaluated by a dot product of the weight vector $W_k$ and the reference signal vector $X_k$ and is calculated using the Equation 3.13.

$$e_k = y_k - \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix} * [X_0^{newest}, X_1, X_2, X_n^{oldest}] \qquad (3.13)$$

After each iteration the filter weights or sometimes known as filter coefficients must be updated using the feedback error value $e_k$ and update values for each of the weights are calculated separately using the following Equations 3.15a,b,c and d.

$$W_1' = W_1 + \varepsilon * e_k * X_1 \qquad (3.15a)$$
$$W_2' = W_2 + \varepsilon * e_k * X_2 \qquad (b)$$
$$W_3' = W_3 + \varepsilon * e_k * X_2 \qquad (c)$$
$$W_n' = W_n + \varepsilon * e_k * X_n \qquad (d)$$

Where,

$e_k$**:** Filter Output (used to train the filter weights)

Wn: Weight Vector also known as Filter Taps

$\varepsilon$ : Learning Rate (controls rate of descent)

n: Filter Order

All variables used in here are in the floating-point format and the floating-point math algorithms described in the previous section are used to handle the computational load of the filter.

## 3.7.1 Sampling noise and reference for LMS filter

Two channels of the PIC18F452 ADC are used to sample for the LMS filter. The first channel samples the noise Yk and the second channel samples the reference Xk. Shown in Figure 3.7.2.

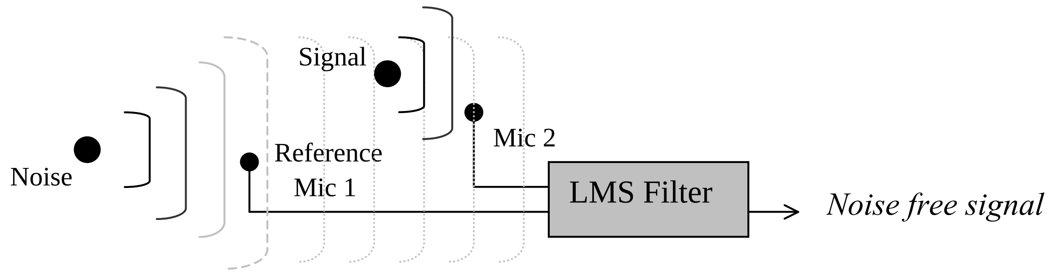


Figure 3.7.2 Sampling for LMS

The LMS filter doesn't have to necessarily be used with audio as the illustration above suggests. It can be used with any two signals that are correlated. The Analog to digital converters are used with 8-bits of precision and are configured exactly like the FIR filters except 2 channels are used for this filter instead of one. The sampling used to implement a 4th order LMS filter was 8000 Hz.

Configuring the ADC involved writing the appropriate registers as shown in Figure 3.7.3.

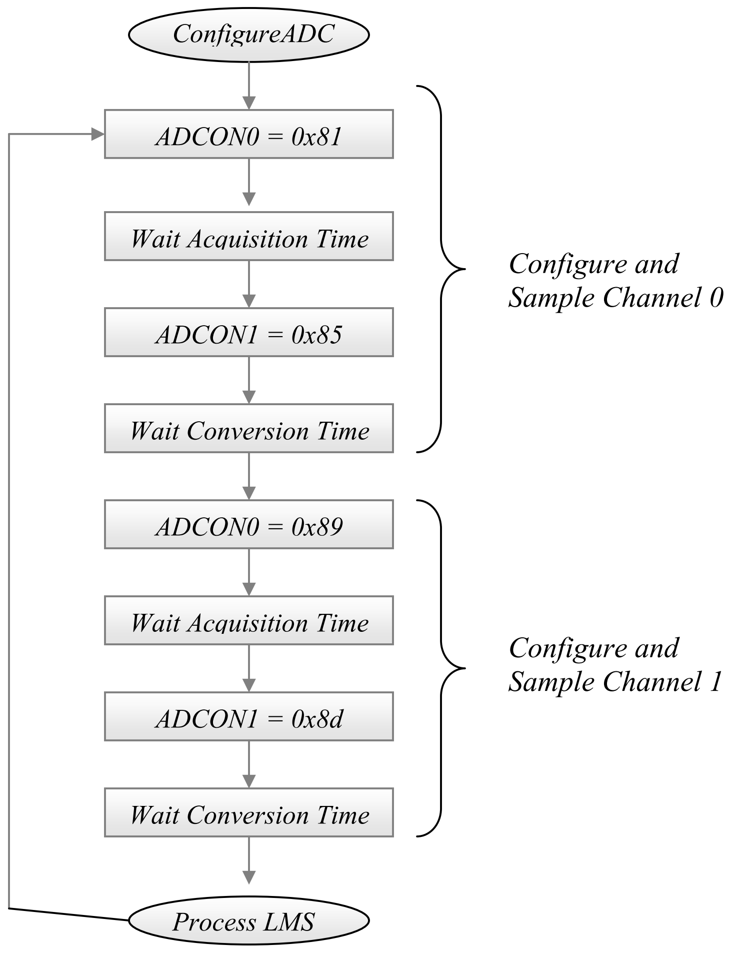


Figure 3.7.3 Configuring ADC for Sampling Two Channels

## 3.7.2 Program Outline for 4th order LMS filter

Like the FIR filter, the Implementation scheme for the LMS filter is also presented as two routines. Firstly the initialization routine, where the variables and buffers are initialized and all the hardware that plays a part in LMS are initialized for use and secondly the computation routine in which the LMS algorithm is computed. The computation routine, like the FIR filter, runs entirely in the interrupt service routine of a timer, in this case timer 2 was used.

The floating-point variables for $W_k$ , $\Delta W_k$ , $Y_k$ , $X_k$ , $H_k$ , $e_k$ are all declared as structures with four members, Sign, FracH, FracL, and Real. Data collected in real-time by the ADC's include one 8-bit sample value for the signal and four buffered 8-bit sample values for reference. Both signal and reference need to be converted into floating-point format before they can be processed. A single 4-point circular buffer is used to store four values of reference. Figure 3.7.4 presents the initialization routine for the 4th order LMS algorithm.

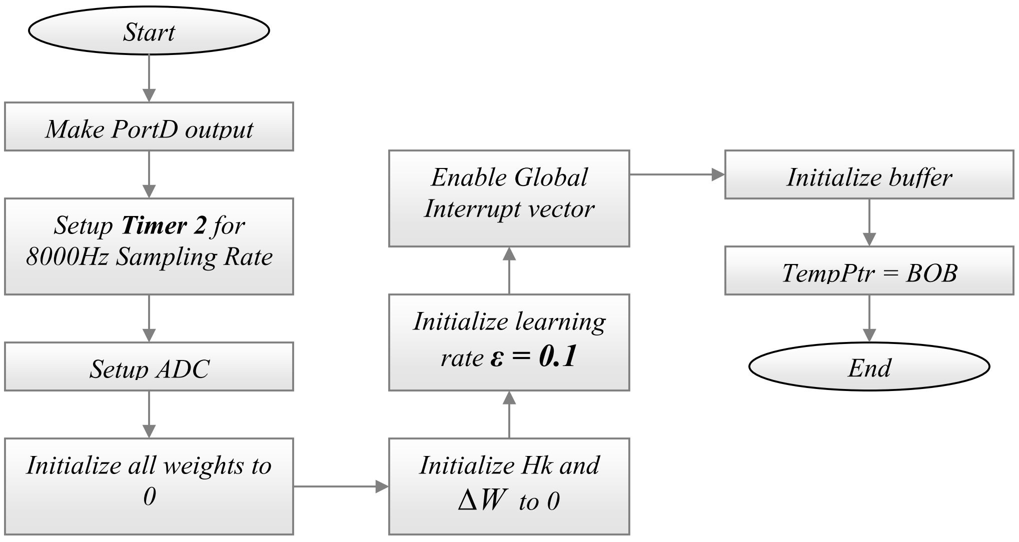

Start
Make PortD output
Setup Timer 2 for 8000Hz Sampling Rate
Setup ADC
Initialize all weights to 0
Initialize Hk and $\Delta W$ to 0
Initialize learning rate $\varepsilon = 0.1$
Enable Global Interrupt vector
Initialize buffer
TempPtr = BOB
End

Figure 3.7.4 Initialization Routine for LMS

**Computation Routine for LMS: Buffering schemes**

Most variables used in this filter are declared as global structs and are not dynamically written or read. However, the reference signal sampled by the ADC needs to be saved in chronological order for LMS calculations. A four element circular buffer was used to store and maintain the ADC samples of the reference signal as 8-bit unsigned bytes per sample as shown in Figure 3.7.5.

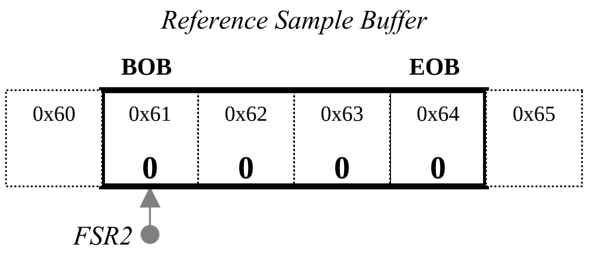


Figure 3.7.5 Four element ADC sample bvffer for LMS

The ADC samples have to be converted into floating-point format and stored in the appropriate structs efficiently before the can be used for LMS. To achieve this the structs are declared in chronological order for storing reference samples in floating-point format and placed sequentially in RAM and a single pointer is used to load all the structs with data pulled from the circular sample buffer. Once all structs are loaded they can be addressed as ordinary variables during computation time. Essentially the structs are loaded dynamically and read statically as if the were the union of four structs and a 16 byte array as shown in Figure 3.7.6.

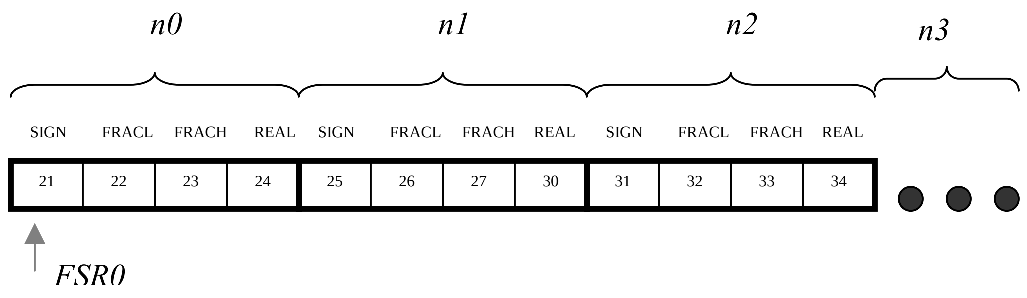


Figure 3.7.6 Arangements of Structs in Memory

## Computation Routine for LMS: Level 1

The Implementation of LMS algorithm in the PIC18F452 chip follows the following basic steps as shown in Figure 3.7.5. The level 1 flow diagram shows an over view of the installation. The entire algorithm is timing sensitive and there fore it runs in the ISR of timer 2.

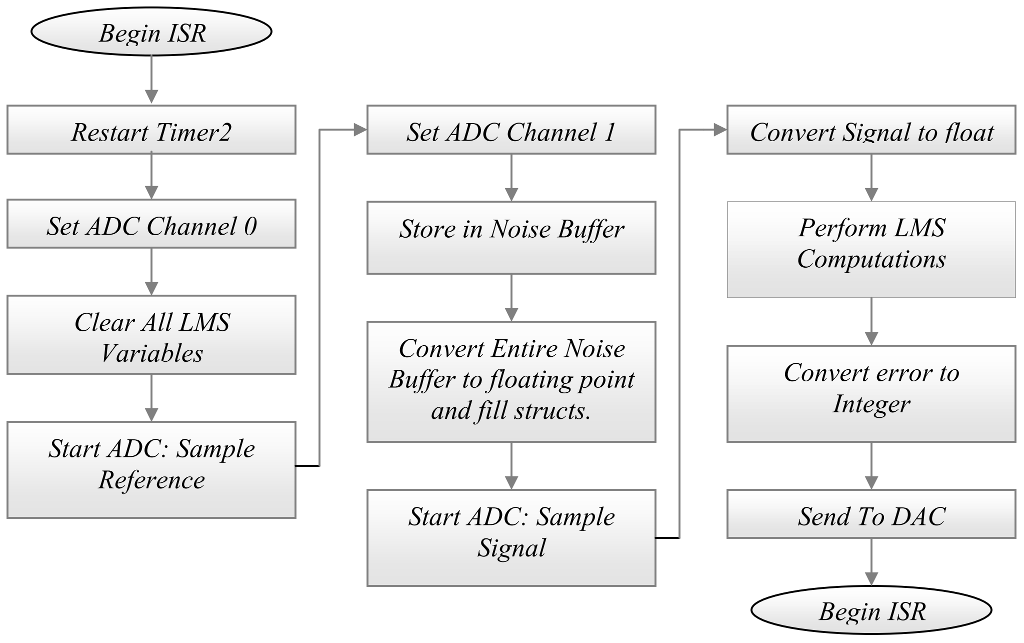


Figure 3.7.7 Level 1 Flow Diagram for LMS

Figures 3.7.8 show the contents of the ISR in detail and Figure 3.7.9 shows in expanded form the detailed computational section of the LMS algorithm.

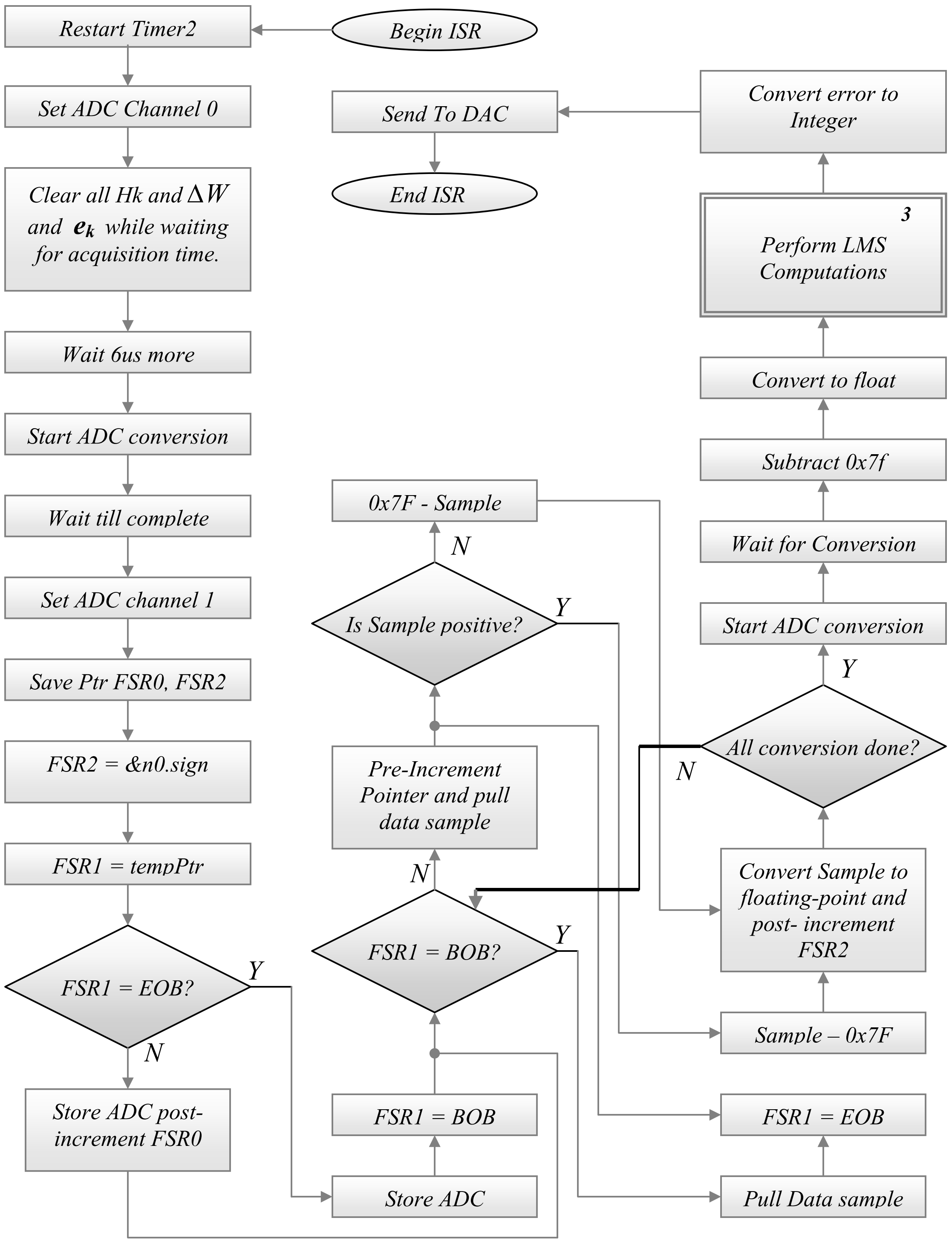


Figure 3.7.8 Level 2 Flow Diagram for LMS

The detailed computational flow diagram of the LMS algorithm is given in Figure 3.7.9.

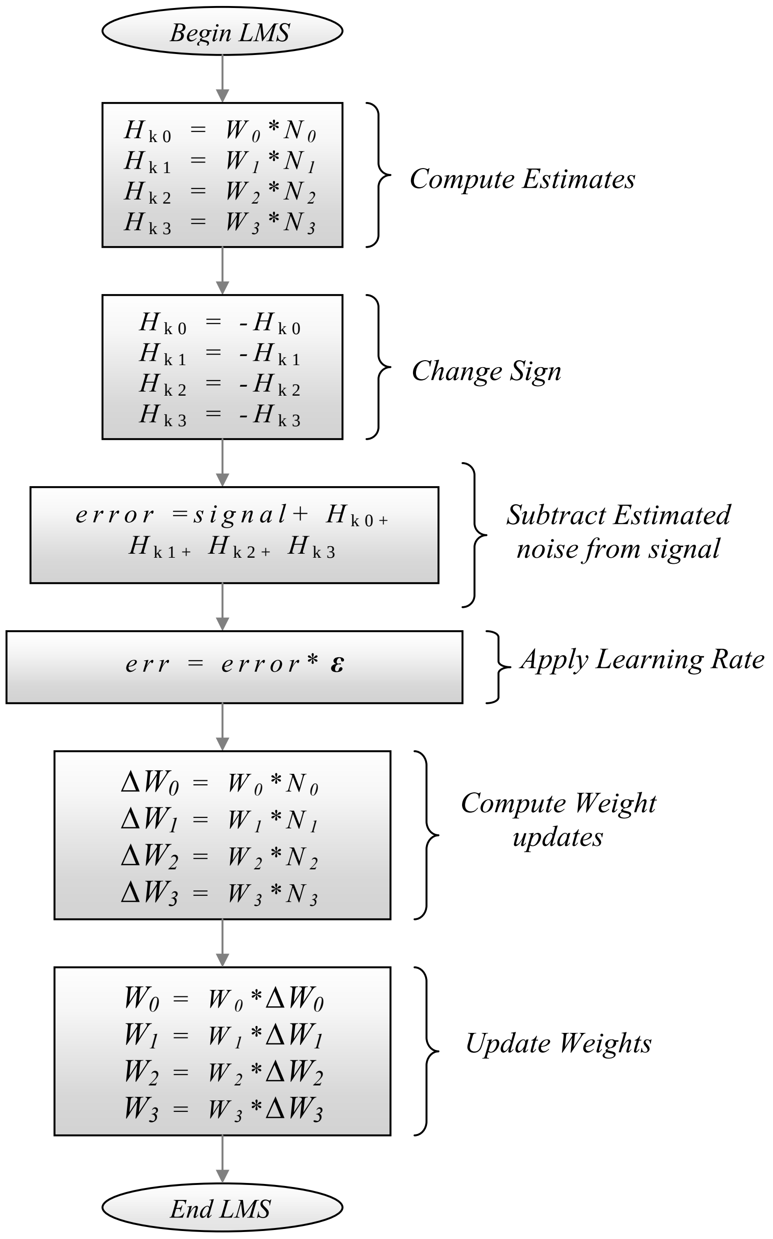


Figure 3.7.9 Level 3 Flow Diagram for LMS

## 3.8 Hardware Test Circuit

The same test circuit was used for both FIR and LMS filter. Figure 3.8.1 is a block diagram of the circuits that are used.

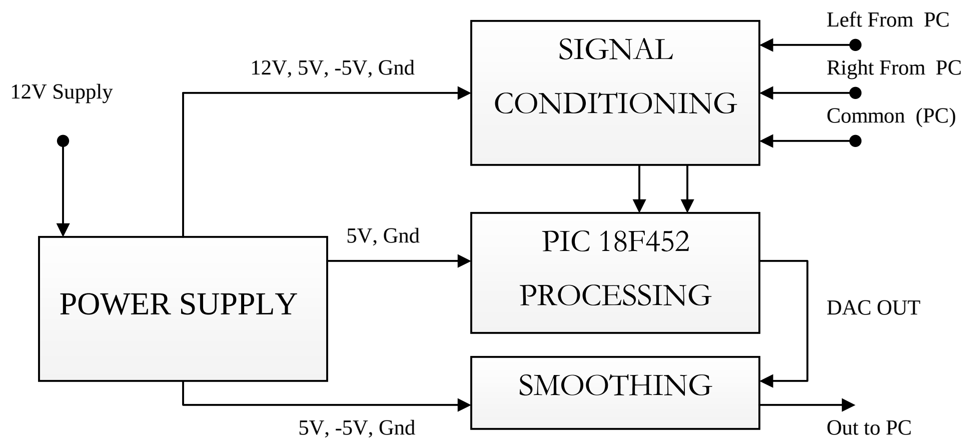


Figure 3.8.1 Block Overview of Circuit

The power supply board was developed to provide the following voltages from a unregulated 12V+ power supply. This board is labeled optional because it was developed purely to make convenient voltage supplies and references. The board provides the following voltages.

a. Regulated 5V / 1A (Power for PIC 18F452 and other IC's)
b. Regulated 2.5V (Offset Voltage for Amplifiers)
c. –5V/100mA Unregulated (DC/DC)

## 3.9 Detailed Schematic of the Power Supply

The power supply board uses the TC1121 DC/DC converter to generate the –5 supply. The 5V supply is regulated by the 7805 and the 2.5 Volt reference level is generated with a zener. All the voltages are derived from a 12 V external supply.

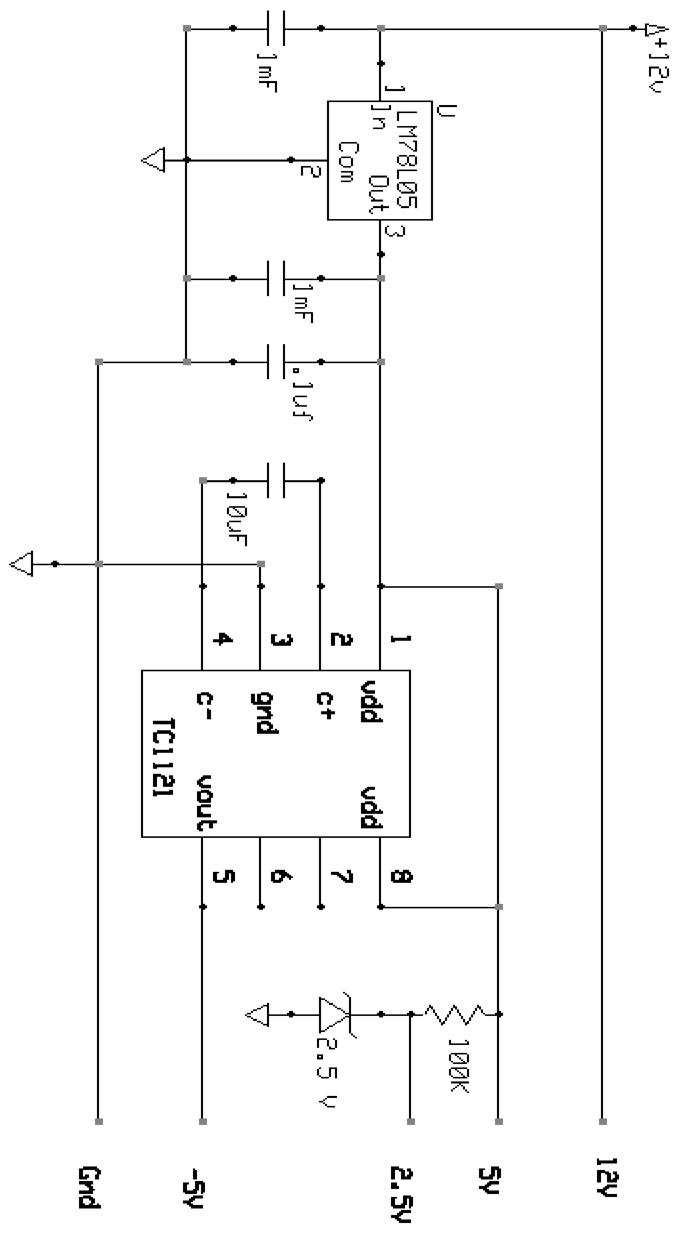


Figure 3.8.2 Power Supply Board

## 3.10 Detailed Schematic: Smoothing Filter

The output signal from the R-2R filter must be buffered and smoothed before it can be sent for data analysis. To that end the Maxim 291 switched capacitor filter is used. The filter is clocked with a small PIC chip PIC 12F629. The firmware for the switch capacitor filter clock signal is in Appendix D.

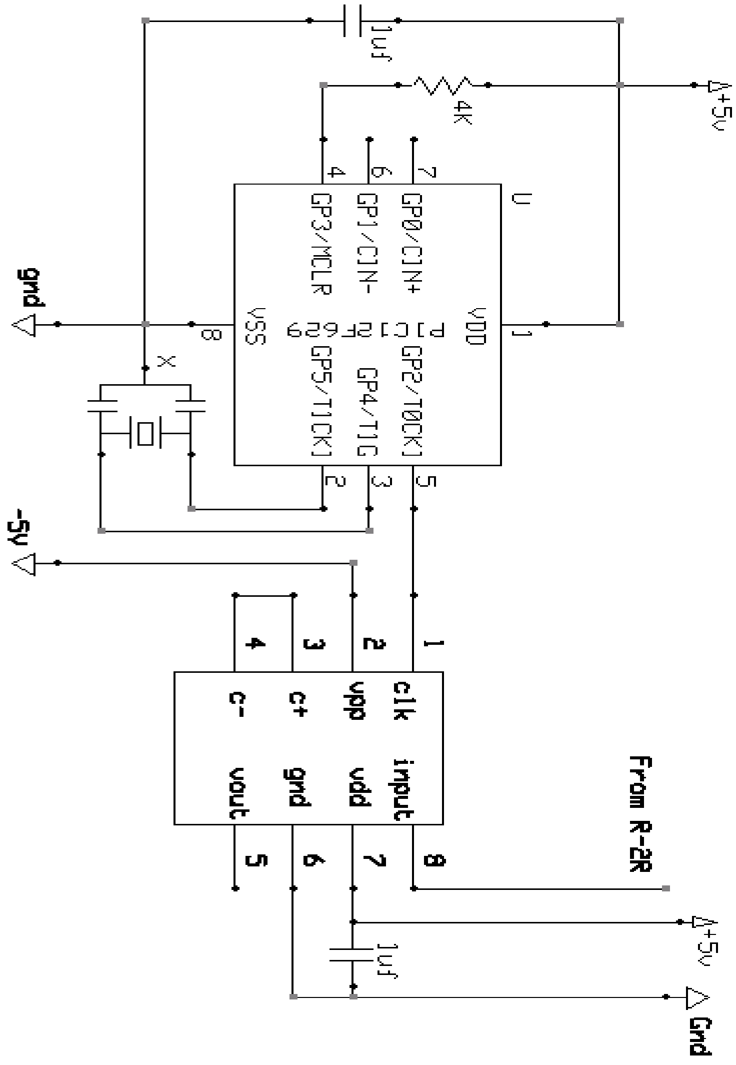


Figure 3.8.3 Smoothing Filter for R-2R Ladder

## 3.11 Detailed Schematic: Signal Conditioning Board

The signal conditioning board contains amplifiers to scale two signals from the line out of the PC sound card or external microphones to the 0 to 5 volt range so they can be sampled by the analog two digital converters as shown in Figure 3.8.4.

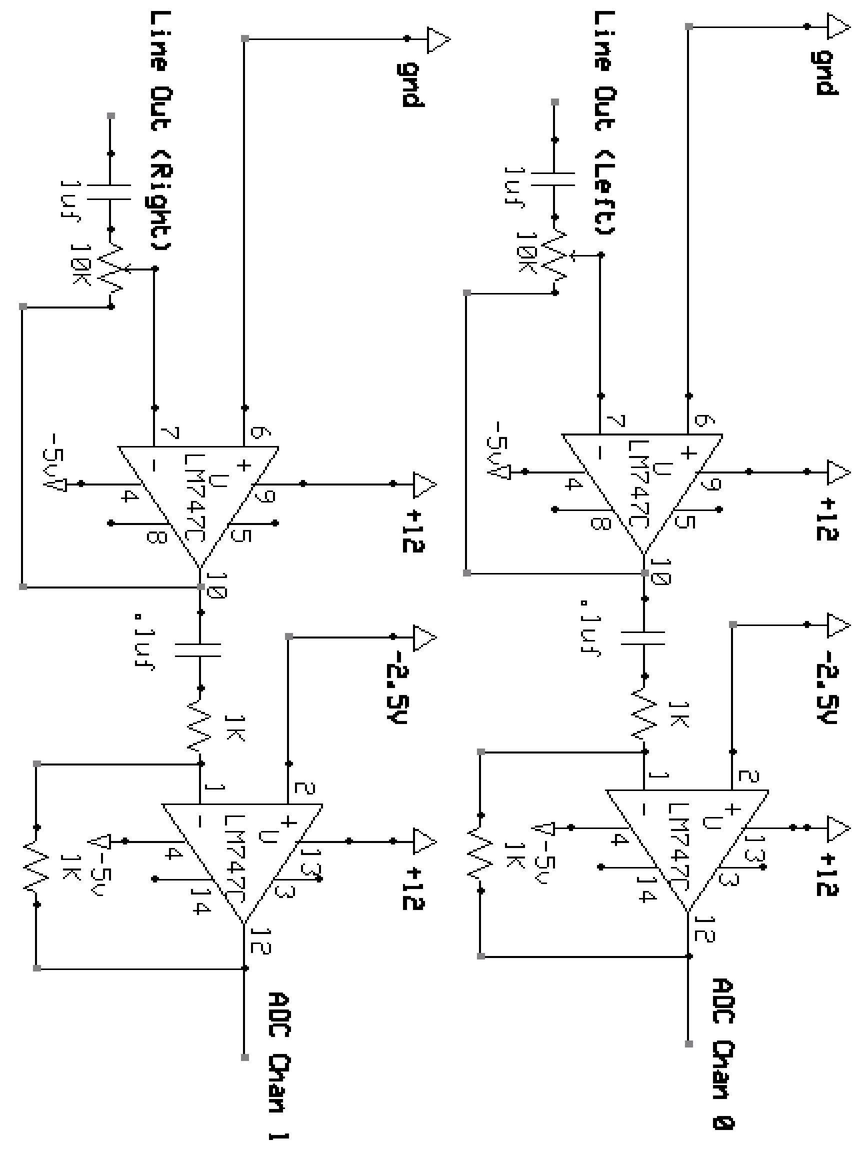


Figure 3.8.4 Signal Conditioning Block

## 3.12 Detailed Schematic: Signal Processing Board

The signal processing board is built around the PIC18F452 chip. It contains the bare minimum circuitry that is required by the filter. The DAC uses a R-2R ladder to generate an output for convenience.

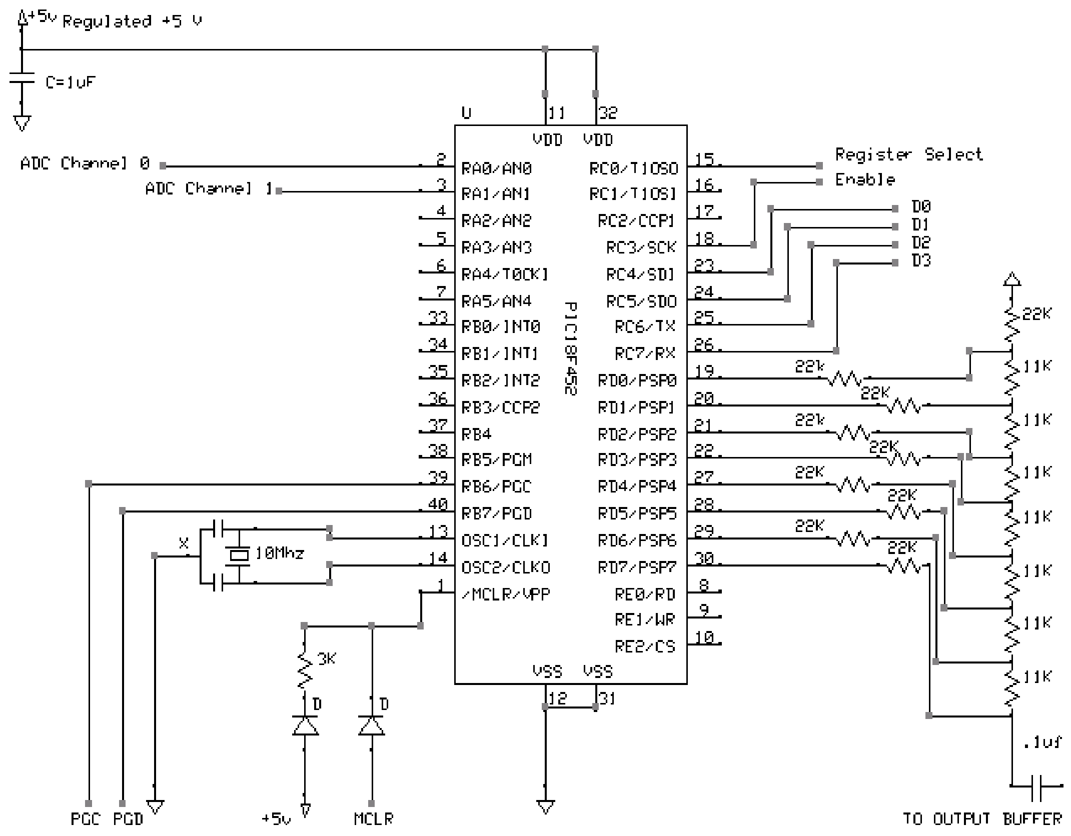


Figure 3.8.4 Block Overview of Circuit

## 3.13 Photograph of PIC18F452 based Filter

This photograph of the test device, shown in Figure 3.8.5, was built to validate the filters designed and built during this project.

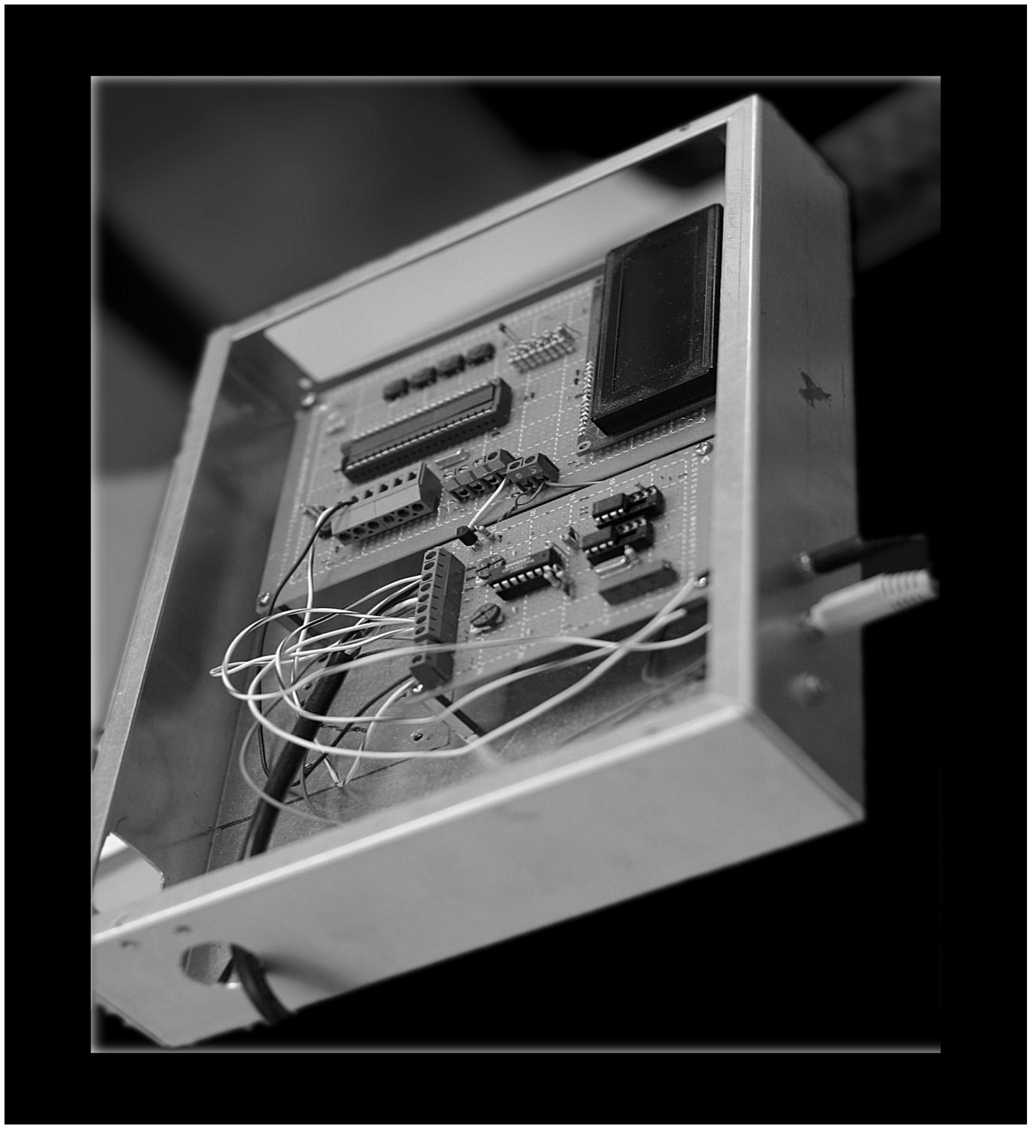


Figure 3.8.5 Photograph of Test Board

# CHAPTER 4: RESULTS FROM REAL-TIME

All data acquisition was performed using a PC based audio processing program called Wavlab™ Pro by Steinberg [14]. The program used the PC microphone input for data acquisition and contained a powerful set of visualization and analysis functions. This technique for data acquisition proved to be both elegant and efficient. The same software was used to generate various frequency sweeps that were used as input data to the PIC chip. LMS filters were also tested using the same apparatus.

## 4.1 FIR FILTER: Data Acquisition Setup

The apparatus setup for FIR filter testing is illustrated in Figure 4.1.1. In order to test the circuit with controlled waveforms, they were generated on a PC and send to the processing board via the line out of the sound card. The PIC board sampled the signals on the line out and after applying the processing LMS, send the DAC output back to the PC sound card, where it is sampled and stored as a wave file. This file is analyzed in WavLab™ [14] and results are presented.

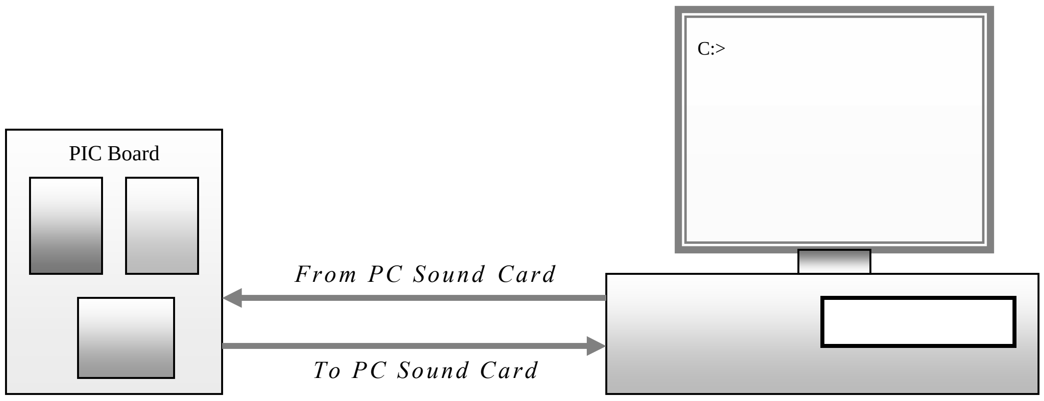


Figure 4.4.1 Basic setup for low-cost data acquisition

## 4.2 FIR FILTER: Real-Time Testing Results

Several filter configurations were tested to validate and verify the operation of the FIR filter in the PIC chip. Matlab was used to generate filter coefficients (Taps) that were then transferred to the PIC chip.

### LOW PASS FILTER: Testing and Analysis

The first test was a basic low pass filter with the following parameters. The filter illustrated in Figure 4.2.1 was generated with the PIC filter design software developed for MATLAB during the course of this research. The low-pass filter in Figure 4.2.1 was made with the desired attenuation of 50dB in the stop-band [0.6K-1K] with a 1KHz cutoff.

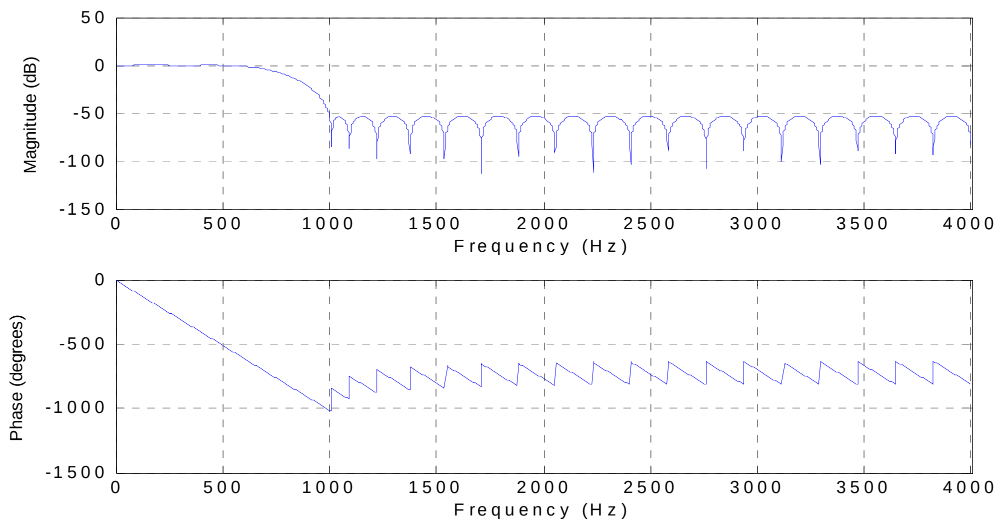


Figure 4.2.1 Response Curves of Intended Filter

The sampling frequency is 8000Hz allowing the sampling of up to 4000Hz. Once the PIC was loaded with the firmware it was then tested using the test-signal shown in Figure 4.2.2. The test-signal is a constant-power frequency sweep 10 seconds long from [200 Hz to 4000 Hz]. The sweep has constant amplitude in time domain and the frequency steadily increase from 200Hz up to 4000Hz. Since the filter is designed to begin attenuation at 600Hz and reach 50dB at 1000Hz, the analysis of the test signal after running through the PIC filter should show how well the filter worked. Since this is a constant-power sweep the amplitude of the sweep attenuates with increasing frequency in as the FFT chart.

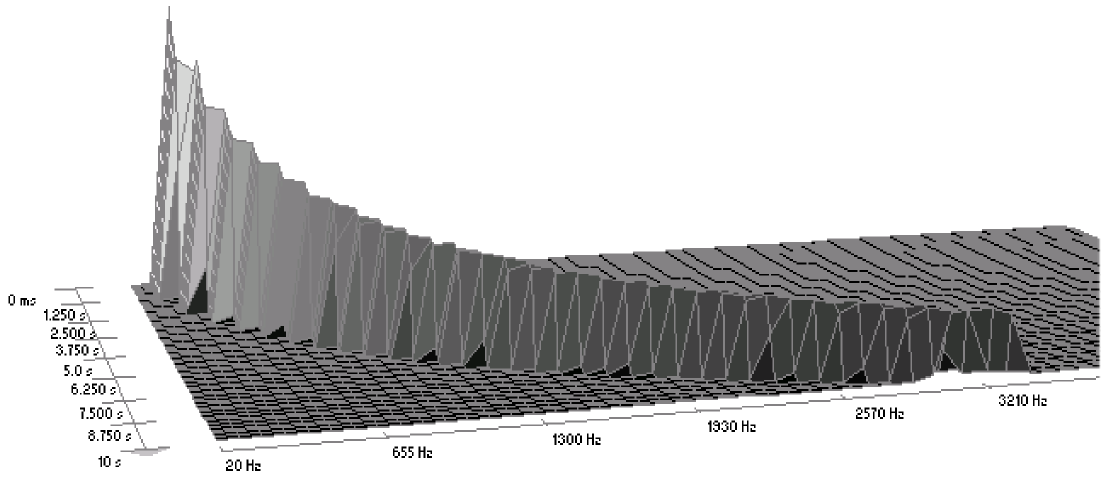


Figure 4.2.2 Test Signal: Constant Power Sweep [200Hz-4000Hz]

The frequency response of the output from the filter captured by a data acquisition system is shown in the Figure 4.2.3.

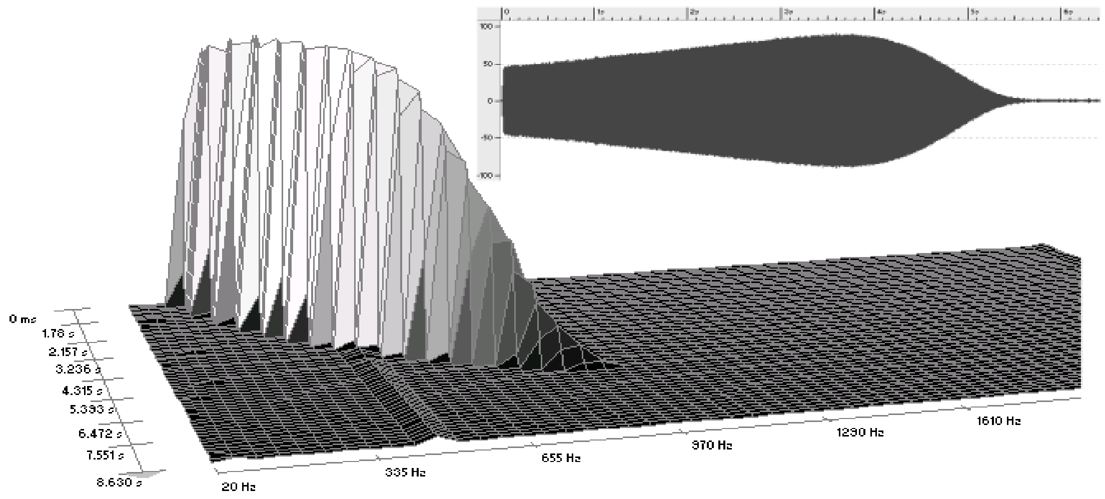


Figure 4.2.3 Filter Performance on PIC18F452 Chip

The fast Fourier transform (FFT) of the DAC output shown in Figure 4.2.4 verifies the performance of the filter. The sweep does indeed begin attenuation from 600Hz as desired and is almost entirely decimated at 1Kz. In the time domain graph (also Figure 4.2.4) the amplitude of the sweep does indeed show attenuation in the high frequency side of the sweep. Notice the small attenuation in the low-frequency side of the sweep. This attenuation is not the result of the filtering in the PIC. This attenuation is due to a DC blocking capacitor in the PC sound card. Sound cards by design cannot sample DC and this is one of the drawbacks of not using an expensive PC based data acquisition systems.

## BAND STOP FILTER: Testing and Analysis

The second filter generated by the filter design software was a band pass filter with the characteristics illustrated in Figure 4.2.4. Once a attenuation of 50dB is desired in the stop-band [600Hz-1800Hz]. The sampling frequency remains 8000Hz.

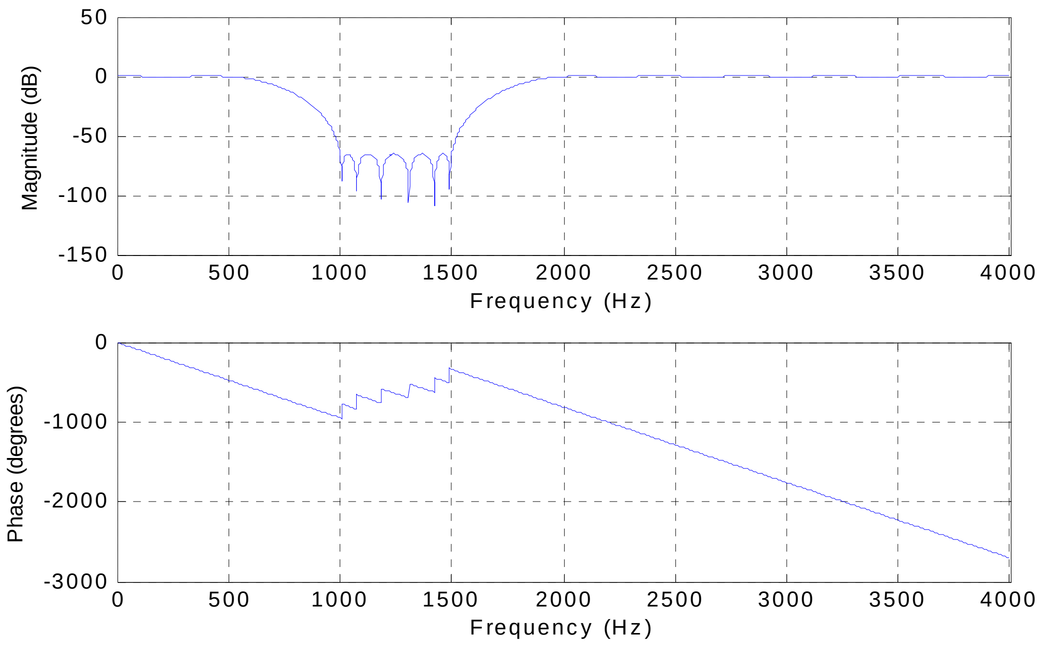


**Figure 4.2.4** BSF Filter Specifications for Band Pass filter

The FFT of the sampled data from the PIC chip illustrated in Figure 4.2.5 shows attenuation in the desired band [600Hz – 1800Hz].

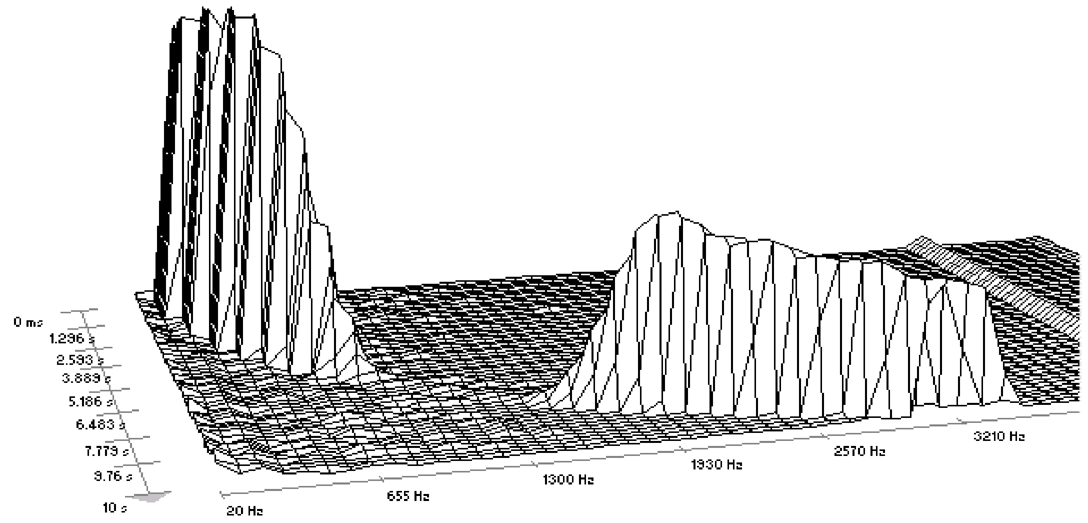


Figure 4.2.5 Measured BSF Frequency and Amplitude Response

## MULTI-BAND FILTER 1 (FIR): Testing and Analysis

The second filter that was implemented was a band stop filter. In Figure 4.2.6 a more complex multi-band FIR filter is shown. The filter has two stop-bands at [600Hz – 1800Hz] and [2700Hz-4000Hz]. Figure 4.2.7 shows the FFT of the PIC filter performance. Both stops bands have been attenuated as intended.

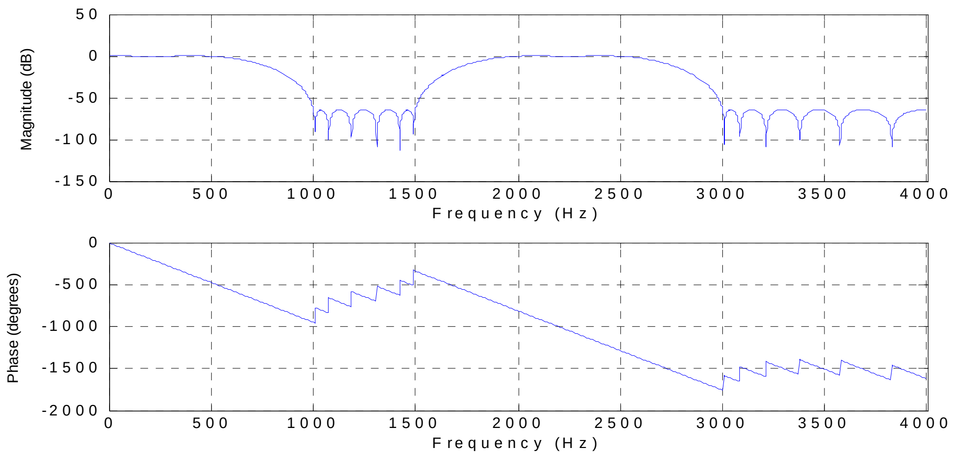


Figure 4.2.6 MBF 1 Filter Specifications for Band Pass filter

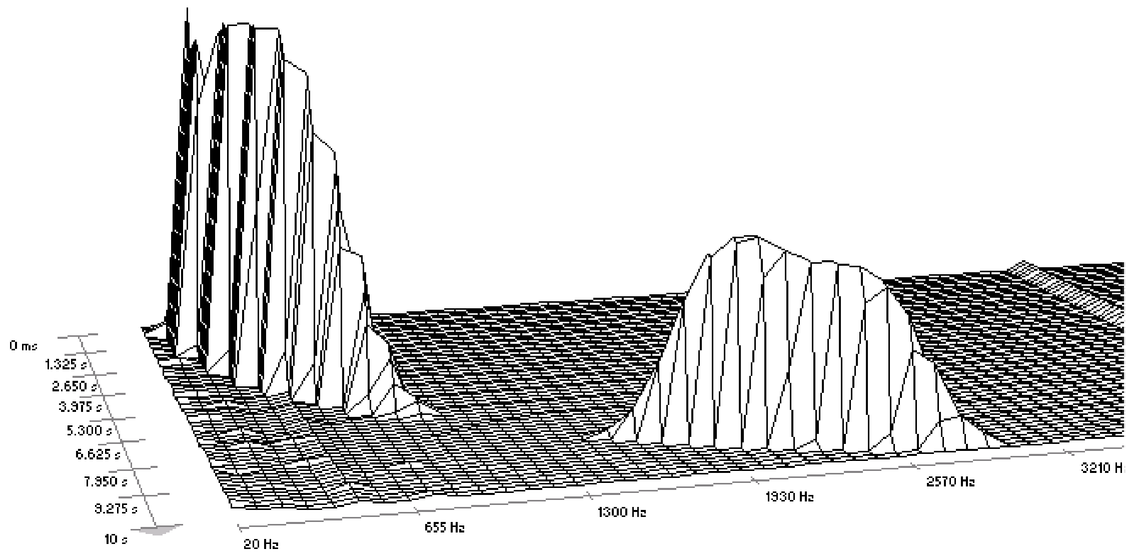


Figure 4.2.7 Frequency and Amplitude response from PIC18F452

## MULTI-BAND FILTER 2 (FIR): Testing and Analysis

Figure 4.2.8 is a filter with three pass-bands; Figure 4.2.9 shows the performance of the PIC chip. The figure shows that all three bands have been compensates as specified in Figure 4.2.8.

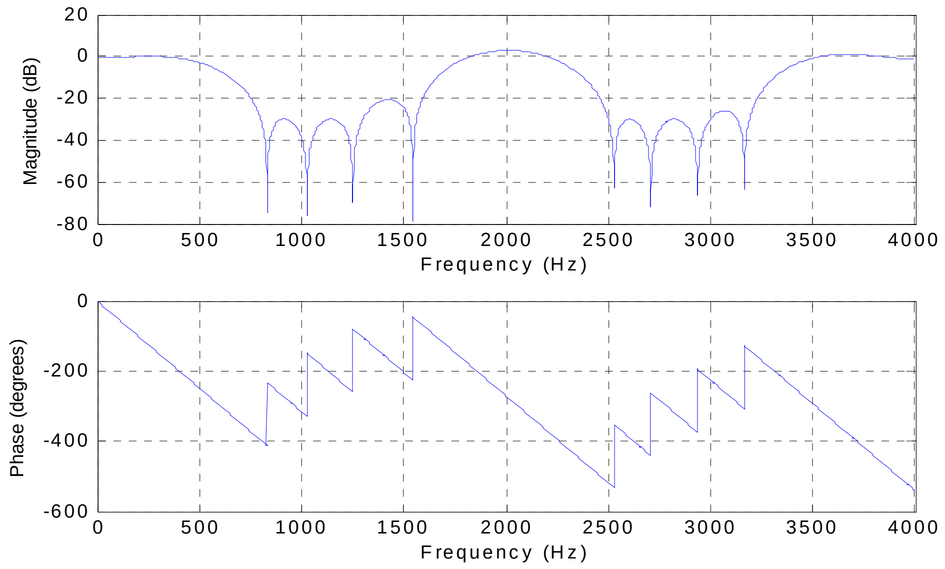


Figure 4.2.8 MBF 2 Filter Specifications for Band Pass filter

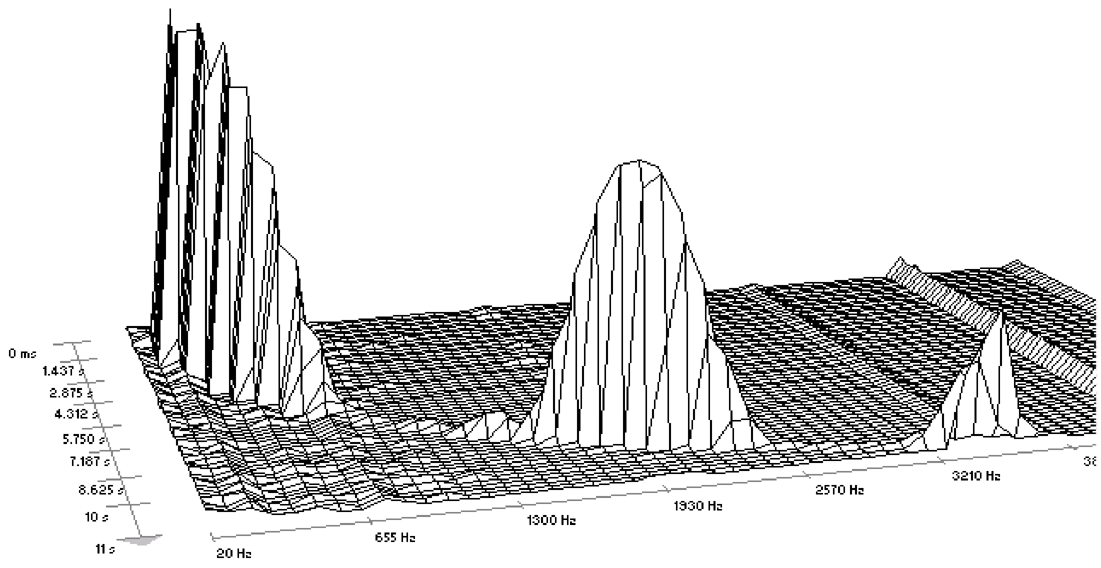


Figure 4.2.9 Frequency and Amplitude response from PIC18F452

## 4.3 LMS FILTER: Test Scenario 1

In order to test the real-time adaptive filter a signal and noise vectors are carefully prepared. The LMS adaptive filter essentially applies the phenomenon of destructive interference to perform noise cancellation. Two waves can be successfully cancelled by destructive interference if they are both correlated in phase and amplitude. To illustrate the point made in the previous statement consider the scenario presented in Figure 4.3.1, the source *A* produces a signal that is sampled at two points in space, p1 and p2. Even though at both points the signal is very similar however they cannot be directly subtracted because by the time the signal is sampled at p2 it is different in both phase as well as amplitude and cannot be simply destroyed by simple subtracting p1.

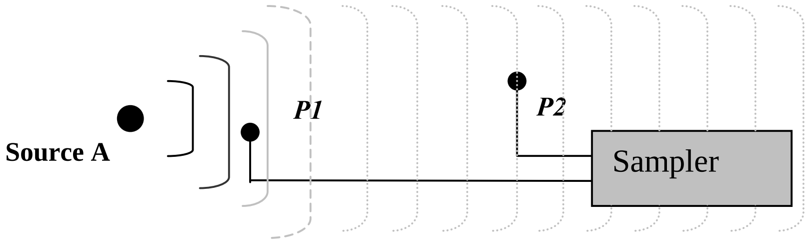


Figure 4.3.1 Sampling Source A at P1 and P2

To make the problem even more interesting a second source is added to the scenario presented in Figure 4.3.1 where a source B is introduced:

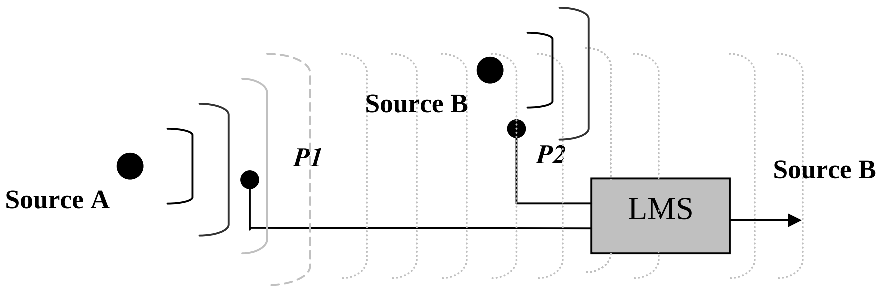


Figure 4.3.2 Source B is added to the scene

Figure 4.3.2 shows that p2 will sample not only the signal from source B but also signal from Source A. The LMS adaptive filter has the ability to intelligently subtract Source A from source B by predicting the degree of contamination from Source A in the sample of source B and recursively improving its' predictions until source A has been successfully eliminated from the sample made at p2*.* The test scenario uses two monotonic sin waves one for source A (220Hz*)* and another for Source B (340Hz*).* To simulate the effects of Source A traveling through space till sampling point p2 source A is given a phase-shift of 375us (micro seconds) and a gain added to source B and the sum is normalized to approximate the signal sampled at p2*.* See Figure 4.3.3.

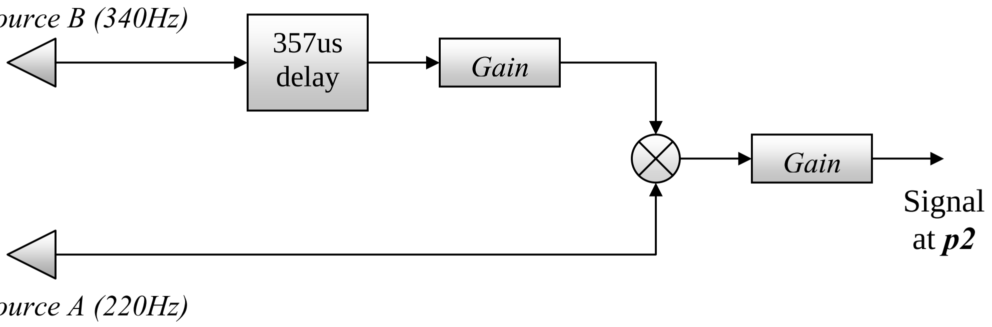


Figure 4.3.3 Approximation of Signal at p2

The signal used at p2 for the simulation was compiled in Matlab and a time domain graph is presented in Figure 4.3.4. This waveform is the superposition of source A and source B sampled at point p2.

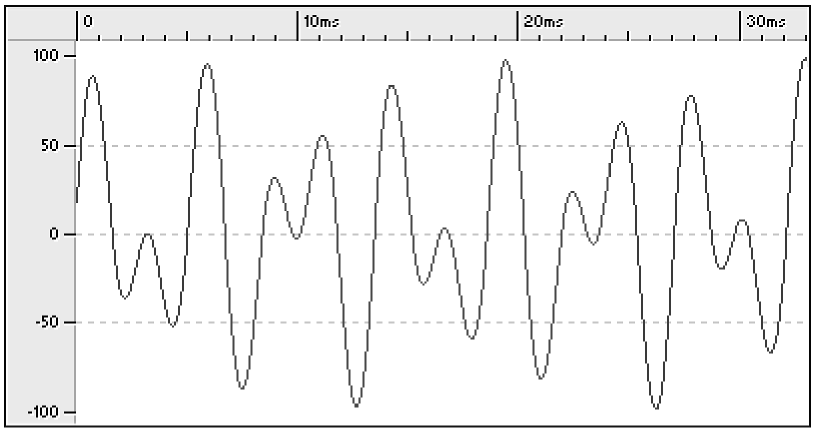


Figure 4.3.4 Time domain graph of signal at p2

Figure 4.3.5 shows the frequency domain representation of the signal sampled at p2. The two spikes are the two monotonic signatures of source A (220Hz) and source B (340Hz).

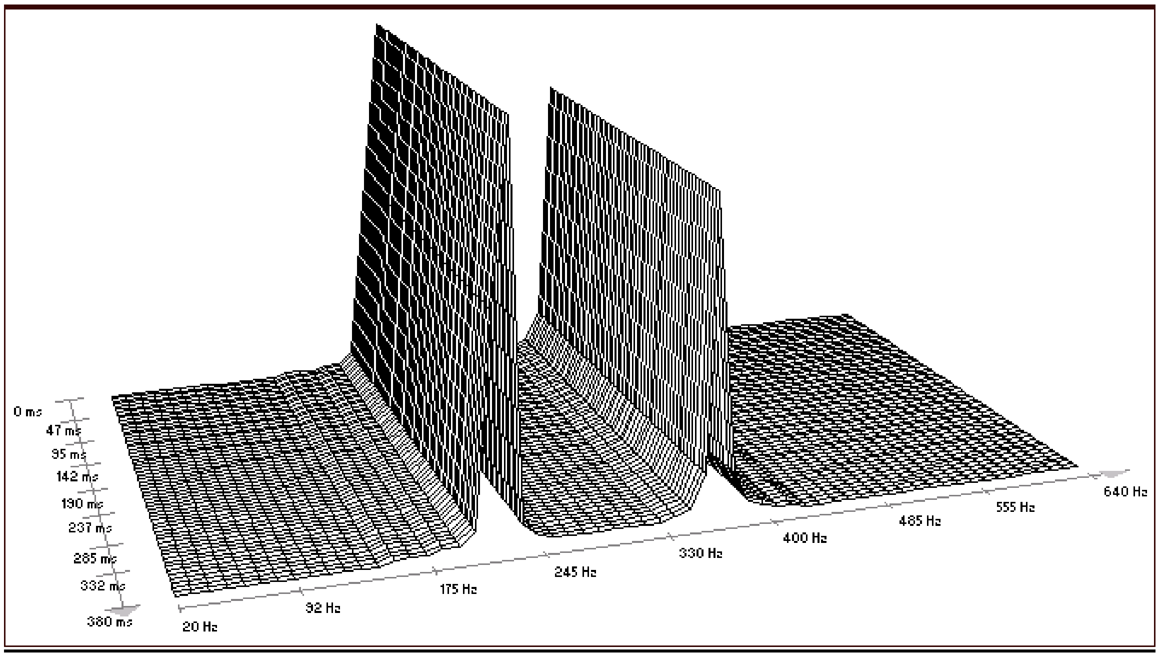


Figure 4.3.5 Frequency domain graph of signal at p2

The real-time test topology is presented in Figure 4.3.6. The signal sampled at point p1 is approximated as source A (220Hz) with a gain.

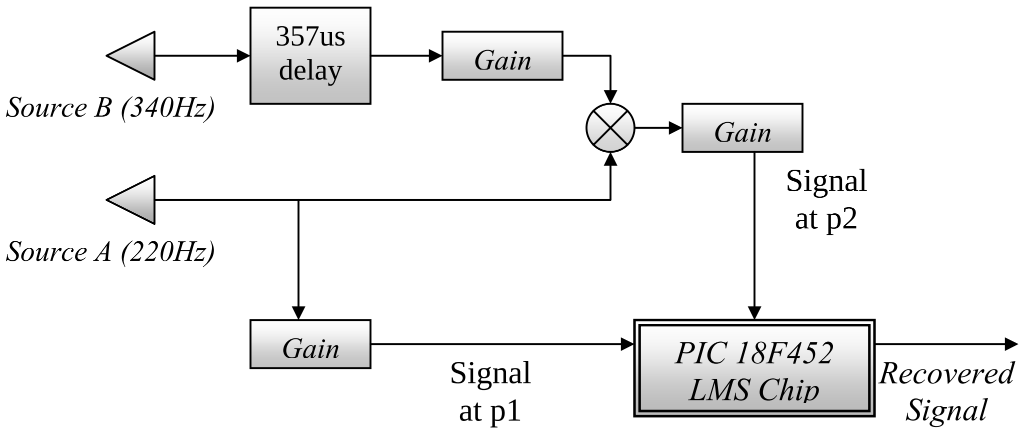


Figure 4.3.6 Real-time test topology

The signal sampled at point p2 and the signal at p1 (reference) going into the PIC chip are shown next in Figure 4.3.6 in time domain.

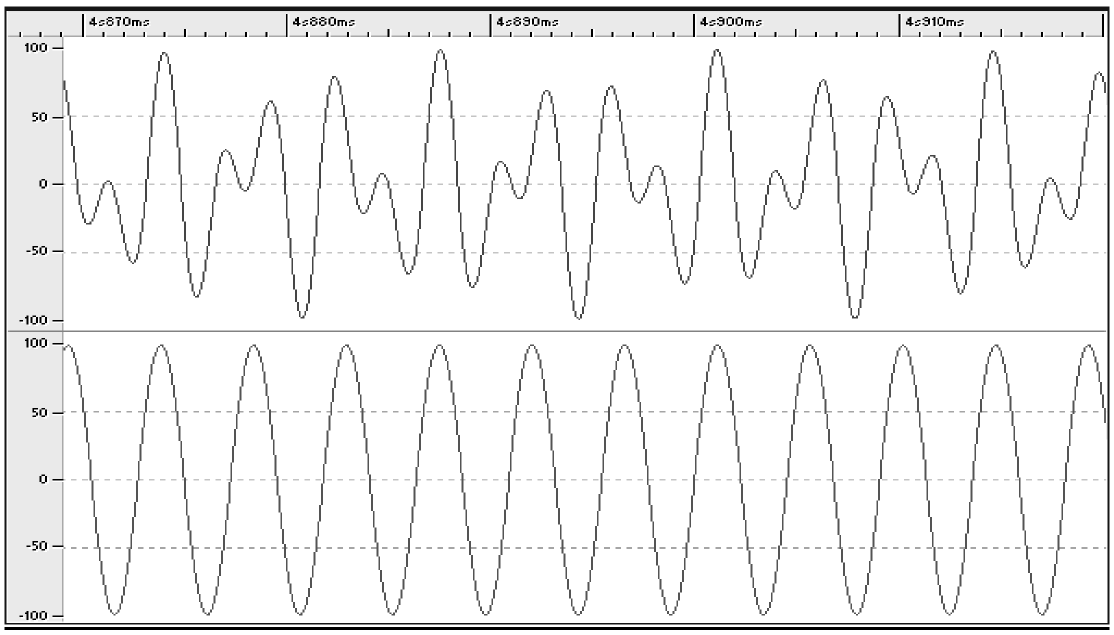


Figure 4.3.6 Signal p2 (top) and Reference p1 sampled by ADC

The signal p1 and reference p2 are sampled by the PIC 18F452. The algorithm implemented in the PIC chip is a fourth order floating-point LMS with a sampling frequency of 8000Hz and learning rate of 0.1. Both signal and reference are simultaneously presented to the PIC chip as shown in Figure 4.3.5, the hardware setup for the experiment is the same as Figure 4.4.1. The output of the chip or the recovered signal is recorded and graphed in Figure 4.3.7 in time domain.

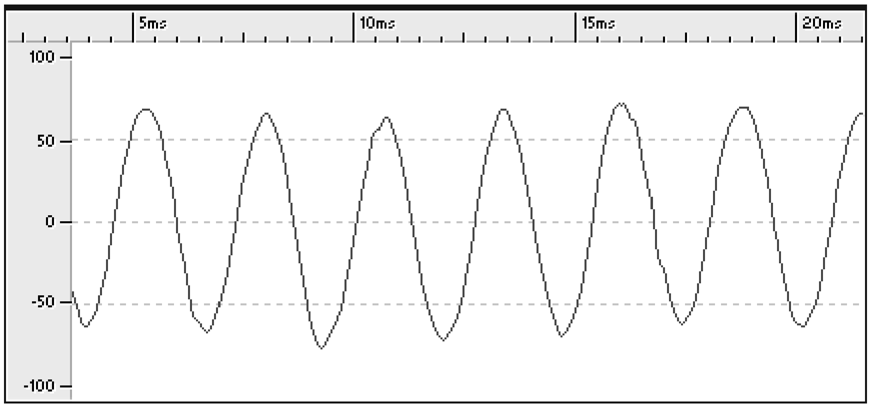


Figure 4.3.7 Signal Recovered by the PIC chop (source B: 340 Hz)

Figure 4.3.7 show that the LMS algorithm running in the PIC chip was indeed able to recover the Source B and the experiment was successful however, the lack of smoothness in the recovered signal suggests high-frequency noise. The smoothing-filter was given a cutoff of 4000Hz however shifting that cutoff to a lower value will improve the signal to noise ratio. The frequency domain graph of the recovered signal is presented next in Figure 4.3.8.

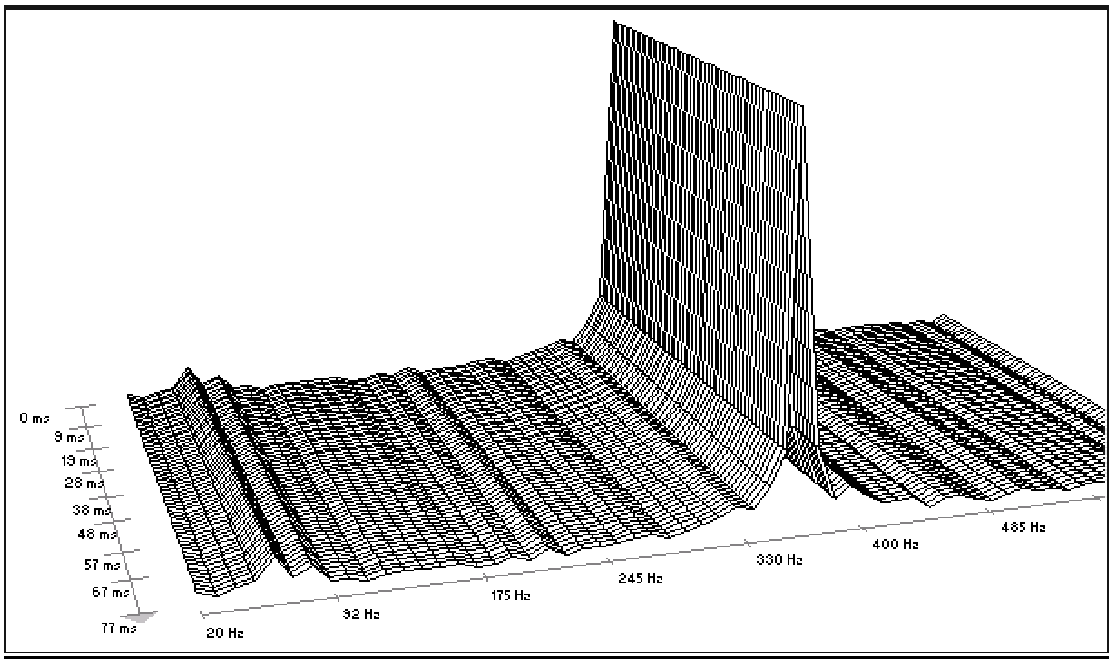


Figure 4.3.8 Signal Recovered in frequency domain (source B: 340 Hz)

# CHAPTER 5: ANALYSIS & CONCLUSIONS

Despite the hardware limitations of the PIC chip, both the FIR and LMS filters gave a strong performance with consistent, measurable and repeatable results.

## 5.1 FIR filter performance summery

The FIR filter attenuation requested in the filter presented in Section 4.2 is 50dB in Matlab. Although the PIC chip faithfully reproduces the frequency response designed by Matlab, the attenuation of 50dB could not be achieved. The best attenuation possible was 36dB.

The difference of 14dB is attributed to the combined effect produced from two main factors. Firstly, the coefficients generated by Matlab are in double precision floating-point format, which were re-scaled into 8-bit fixed-point format numbers. This rescaling process is the major factor that contributes to the observed precision gap. The other factor is that the samples of the signal are made at 8-bit precision. In Matlab the test were made with the signal data sampled at 16 bits. The low bit depth in the sampled signal is also a factor that affects precision.

The best computation speed achieved is the theoretical minimum of 800ns per MAC cycle by selecting the fastest execution-speed option. Equation 5.2 calculates the number of CPU cycles required to implement a FIR filter of a specified sized.

$$Num\_cycles(taps) = 8 * taps + 45 \tag{5.1}$$

By using the smallest program size option, two additional instructions are added to the MAC loop thus increasing the size of the MAC cycle to 1000ns. The Total cycles can is calculated using Equation 5.2.

$$Num\_cycles(taps) = 10 * taps + 45 \qquad (5.2)$$

By using the smallest RAM size option, the size of each MAC loop is extended further to 22 cycles per MAC. Thus the total number of cycles used by the PIC chip is calculated using Equation 5.3.

$$Num\_cycles(taps) = 22 * taps + 45 \qquad (5.3)$$

Equation 5.4 determines RAM usage for a given filter order for fastest-execution speed option.

$$Ram\_needed(taps) = 2 * taps + 8 \qquad (5.4)$$

Equation 5.5 determines RAM usage for a given filter-order in the minimum RAM implementation case:

$$Ram\_needed(taps) = taps + 8 \qquad (5.5)$$

The execution speeds from all three available implementation models are plotted and shown in Figure 5.1.

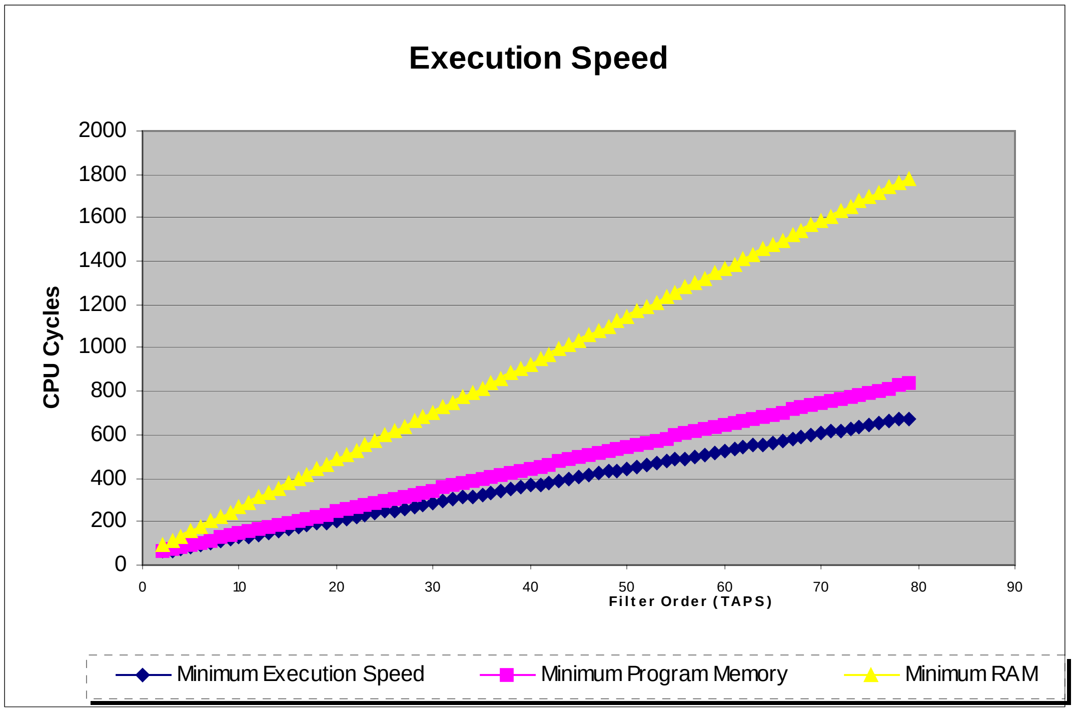


Figure 5.1 Comparison of execution speed for different implementations

## 5.2 LMS filter performance summary

One of the intentions of this research was to obtain a reasonable approximation of how many orders can be achieved on the PIC 18F452 chip given it's many limitations. Using the floating-point library and best speed achieved for the adaptive filter is 267 cycles per tap.

At 10 million instructions per second (MIPS), achievable with a 10MHz external crystal oscillator, the PIC chip can execute 267 cycles in 26.7 μs. Equation 5.6 estimates the highest achievable order for a specified sampling rate and Equation 5.7 estimates the max sampling frequency for the specified number of taps.

$$Max\_LMSOrder(sampling\_freq) = floor\left\{\frac{ext\_osc}{4*sampling\_freq*267}\right\} \quad (5.6)$$

$$Max\_LMS\_Sampling\_freq = \frac{ext\_osc}{4*LMSOrder*267} \quad (5.7)$$

Table 5.1 shows the relationship between Sampling frequency and the maximum filter order achievable using the implementation strategy developed during this research. The number of Taps cannot exceed 375 because the PIC will run out of RAM.

Table 5.1: LMS Sampling Rate vs. Taps

| Sampling Freq (Hz) | Taps |
|---|---|
| 100 | 375 |
| 200 | 187 |
| 400 | 94 |
| 800 | 47 |
| 1600 | 23 |
| 3200 | 12 |
| 6400 | 6 |
| 12800 | 3 |
| 25600 | 1 |
| *External Clock* | *40000000* |

## 5.3 Conclusions and future work

The following conclusions were draws regarding the various filter implementations that were explored during the course of this research.

1. The PIC 18F452 chip is an excellent candidate for fixed-point FIR filter implementation. At 800ns per MAC cycle, there is no obvious disadvantage to diverting part of the PIC CPU cycles.

2. No more than a fourth, order LMS filter is possible using the floating point system using the PIC 18 family. Higher orders are possible at lower sampling rates, however the lack of the normalization operation in the floating-point variables makes it prone to loss of precision from roll-off errors. The newer chips in the same class and price bracket, such as the dsPIC family, contains specific hardware such as a 40 bit barrel shifter, 16-bit signed multiplier and 16 bit ALU with speeds up to 30-40MIPS. These chips overcome many of the hardware limitations of the PIC 18 family, making them highly suited for building practical applications of adaptive filters, neural networks etc without having to resort to using the cost prohibitive DSP boards. Future work can include developing programming tools and software libraries for this new family of chips.

3. In the LMS filter developed during this research is not used in any particular applications. It was merely evaluated and tested on simulated data and meant as to be a resource that can be applied to a specific application. There is scope for finding a suitable application for this filter such as adaptive noise cancellation headphones, standing wave decimation, line echo cancellations etc.

## BIBLIOGRAPHY

[1] Ananda Mohan P.V, Ramachandran V., Swamy M.N.S, *Switch capacitor filters: Theorm, Analysis and Design*, Prentice-Hall PTR, June 1995.

[2] CCS compilers, www.ccsinfo.com/picc.shtml, 2004

[3] CCS Compilers, *C Compiler Reference Manual,* Custom Computer Services Incorporated, Brookfield WI, 2003.

[4] Digikey, www.digikey.com, 2003

[5] Emmanuel C. Ifeachor, Barrie W. Jervis. *Digital Signal Processing: A practical Approach*, Addison-Wesley Publishing Company, 1993.

[6] Hall V. Douglas, *Microprocessors and Interfacing programming and hardware, Glencoe* McGraw-Hill, New York, New York, 1997.

[7] Hamming, R. W. *Digital Filters third edition*, Dover Publications, INC. Minneola, New York, 1989.

[8] Karam L. J, McCellah, J.H., *Design of optimal digital filters with arbitrary magnitude and phase responses.* IEEE International Symposium on Circuit and Systems. Circuits and Systems connecting the World, 1996.

[9] Lathi B. P. *Signal Processing and Linear Systems*, Berkeley Cambridge Press, Carmichael, California, 1998.

[10] Microchip, *PIC18FXX2 Data Sheet: High Performance, Enhanced FLASH Microcontrollers with 10-Bit A/D, 2002.*

[11] Predko Mike, *Programming and customizing PICmicro® Microcontrollers,* McGraw-Hill, New York, New York, 2002.

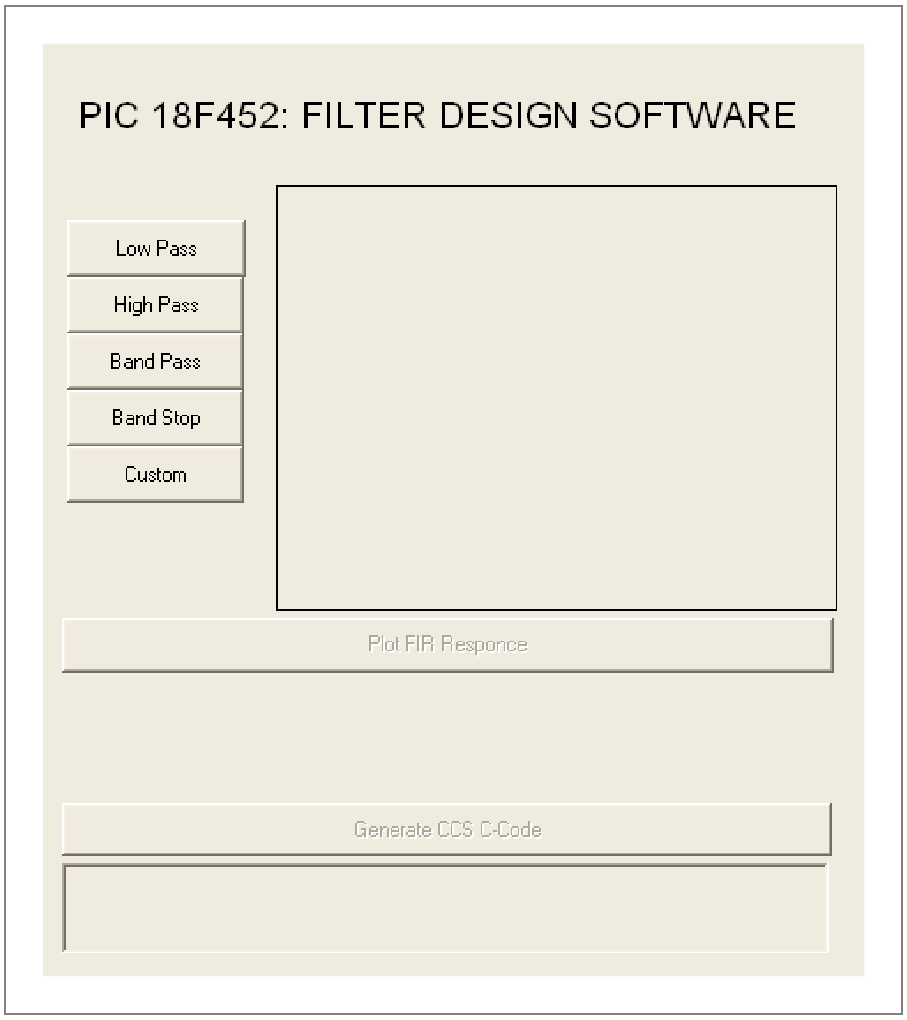
[12] Ramu Anantha B. K. *"Implementation of FIR and IIR Digital Filters Using PIC18 Microcontrollers"*, Microchip Application Note: AN853, Appendix A, 2002.

[13] The Math works, *Filter Design Toolbox 2 for designing and analyzing advanced floating-point and fixed-point filter.* www.mathworks.com/products/filterdesign, 2003.

[14] Widrow B. and *Winter R. Neural nets for adaptive filtering and adaptive pattern recognition*. IEEE Computer, 1998.

# APPENDIX A

## *USERS MANUAL FOR FILTER DESIGN SOFTWARE*

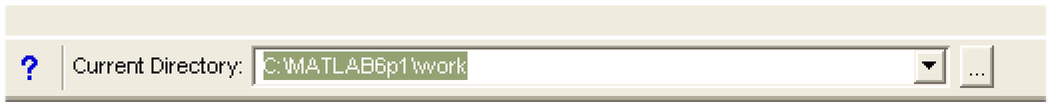


## INTRODUCTION

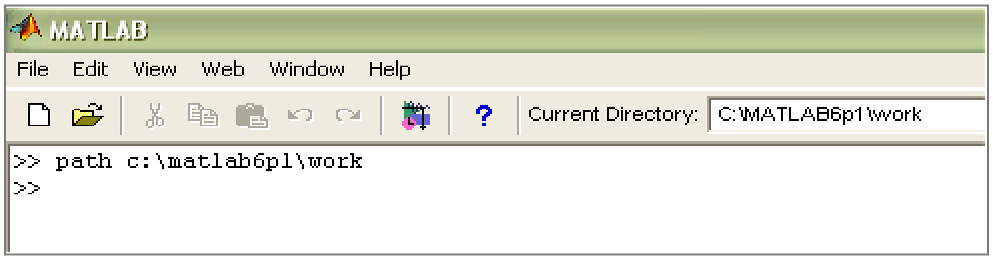
The *PIC18F452: FILTER DESIGN SOFTWARE* was built on the Matlab environment and will only operate on MATLAB Version 6.1.0.450 (R12.1) and up. Filter Design Toolbox Version 2.1 must also be installed within the MATLAB environment.

In order to begin the filter design system a path must be set to the directory in which the source files are held. There are two ways of setting the path to the correct directory.

Method1: The path can be set directly by entering it on the provided space on the main tool bar or by clicking on the ... button.

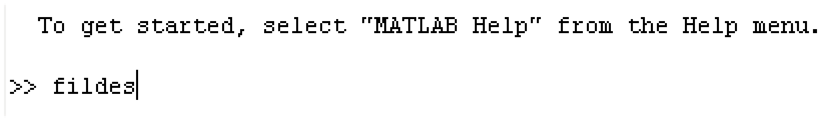


Method2: An alternate way to set the directory path is to use the command line option in the main window of Matlab.


MATLAB
File Edit View Web Window Help
?
Current Directory: C:\MATLAB6p1\work

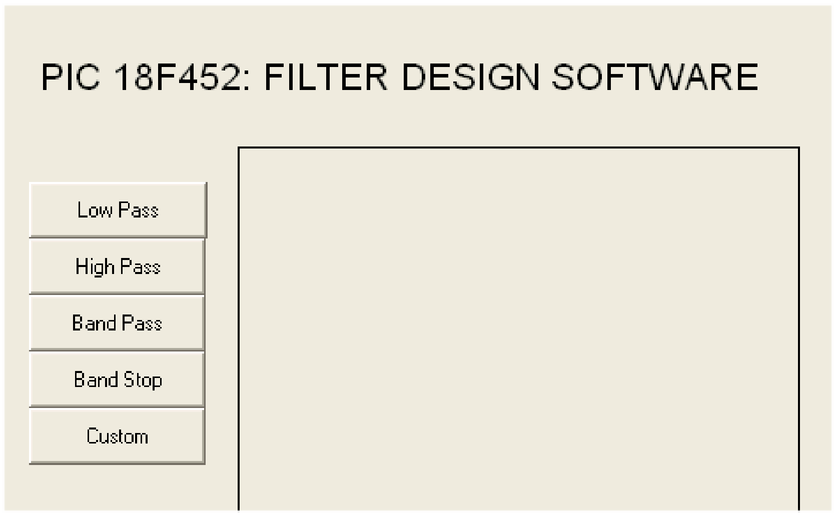
```
>> path c:\matlab6p1\work
>>
```

Once the path has been set the filter design system can be launched by typing in ***'fildes'*** at the command prompt.



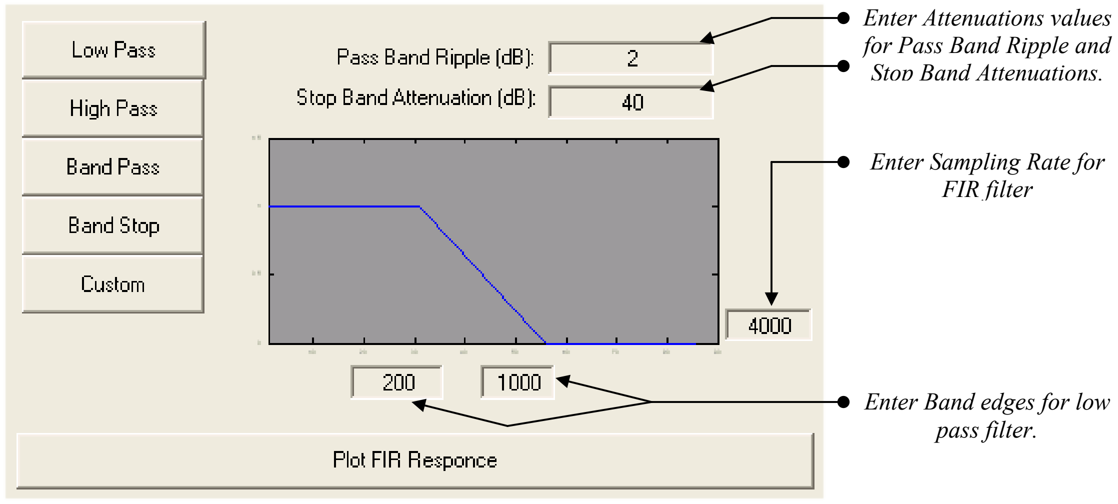
```
 To get started, select "MATLAB Help" from the Help menu.

>> fildes
```

The filter design system main window offers the following functions, low-pass filter design, high-pass filter design, band-pass filter design, band-stop filter design and custom filter design.



## LOW PASS FILTER DESIGN

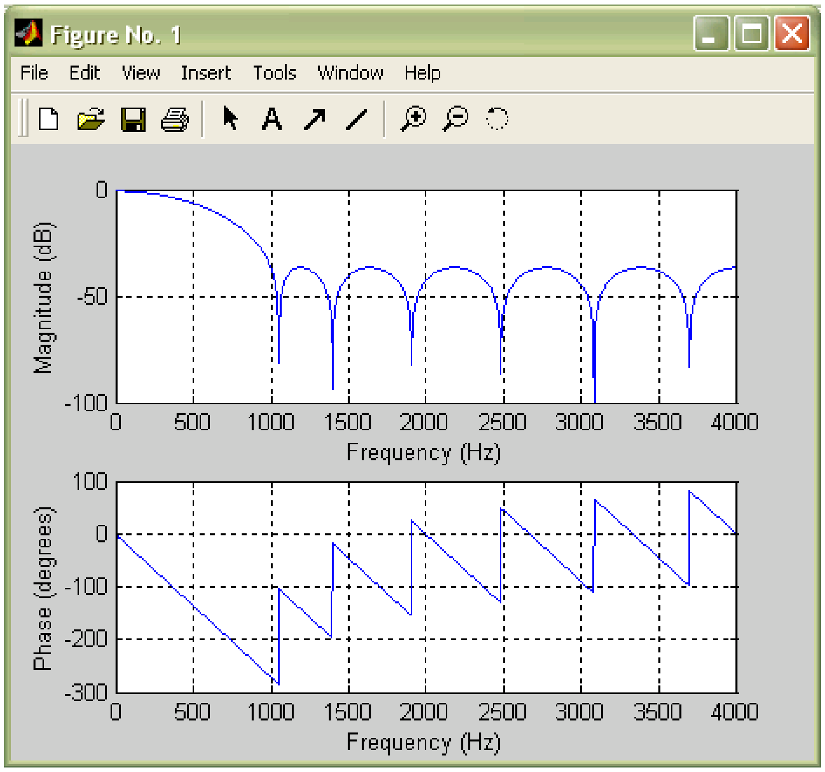
Clicking on the Low Pass button in the main window and enables the low-pass filter design interface where the parameters for the intended low-pass FIR filter can be entered.


Low Pass
High Pass
Band Pass
Band Stop
Custom
Plot FIR Responce
Enter Attenuations values for Pass Band Ripple and Stop Band Attenuations.
Enter Sampling Rate for FIR filter
Enter Band edges for low pass filter.

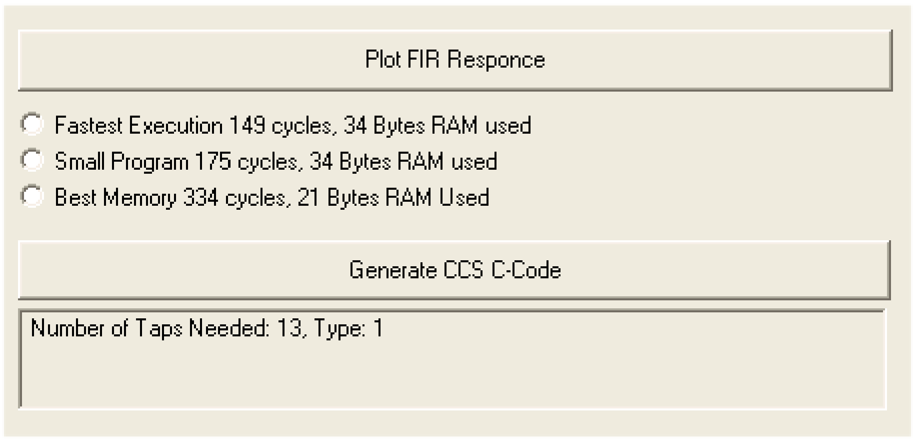
Pass Band Ripple (dB): 2
Stop Band Attenuation (dB): 40
4000
200
1000

Once the desired band-edges and attenuations have been entered, by pressing the

button the simulated filter response is plotted to the screen.



The frequency response and the phase response curves are graphed for user inspection. Additionally new options appear in the main window of the LPF design interface.

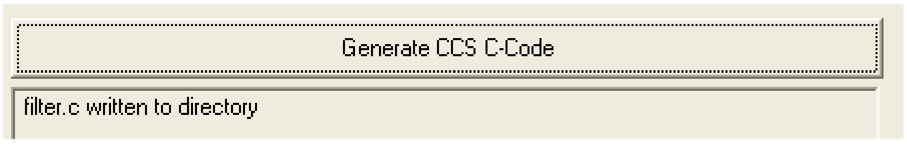


Fastest Execution 149 cycles, 34 Bytes RAM used
Small Program 175 cycles, 34 Bytes RAM used
Best Memory 334 cycles, 21 Bytes RAM Used

Number of Taps Needed: 13, Type: 1

The new items that appear in the interface present three implementation options. Each option lists the bytes of RAM used by the program on the PIC Chip as well as the number of cycles used by the program. Running at 40,000,000 each cycle lasts 100ns. Finally the

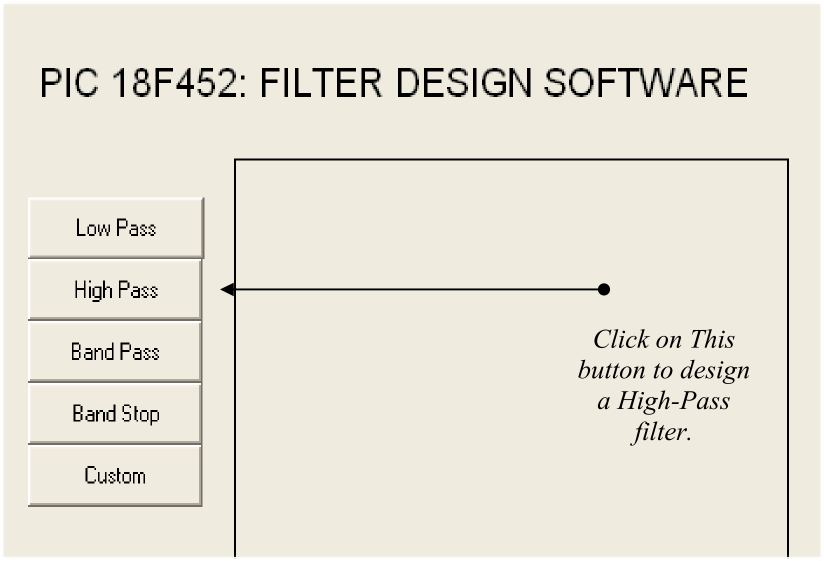
button will generate the c-language file that can be installed into the PIC chip.



filter.c written to directory

## HIGH PASS FILTER DESIGN

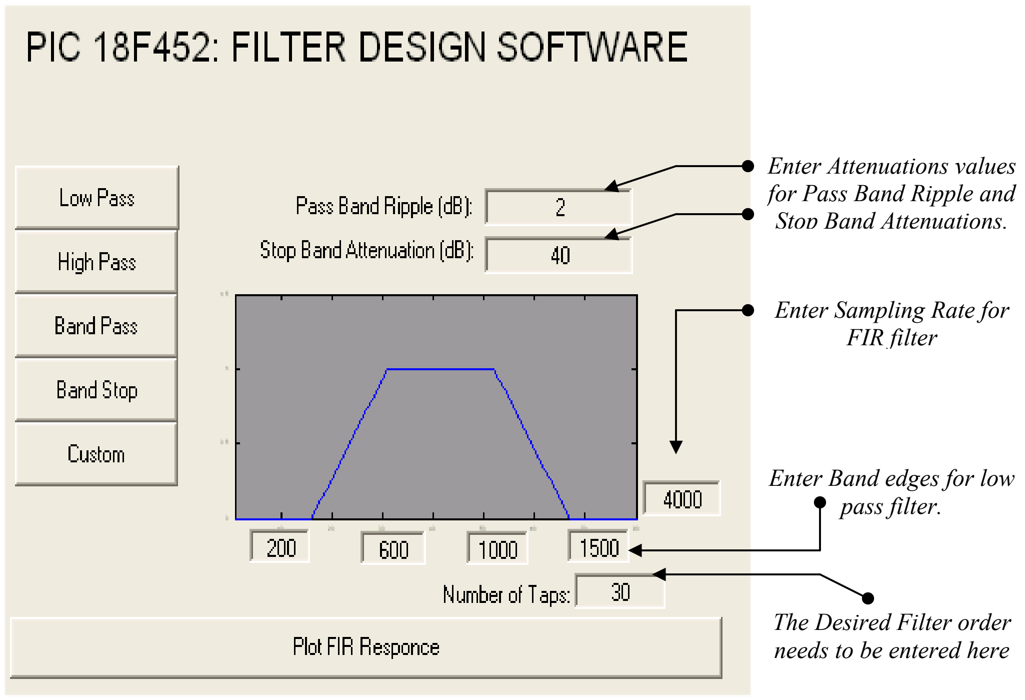
The high-pass FIR filter is designed in the same way as the low-pass filter by selecting the 'High Pass' button on the main menu.


PIC 18F452: FILTER DESIGN SOFTWARE
Low Pass
High Pass
Band Pass
Band Stop
Custom
Click on This button to design a High-Pass filter.

The rest of the design follows exactly the same set of steps as the low pass filter design.

## BAND PASS FILTER DESIGN

Designing the band-pass filter starts as the previous ones by clicking on the band-pass button in the main window.



Pass Band Ripple (dB): 2
Stop Band Attenuation (dB): 40
4000
200
600
1000
1500
Number of Taps: 30

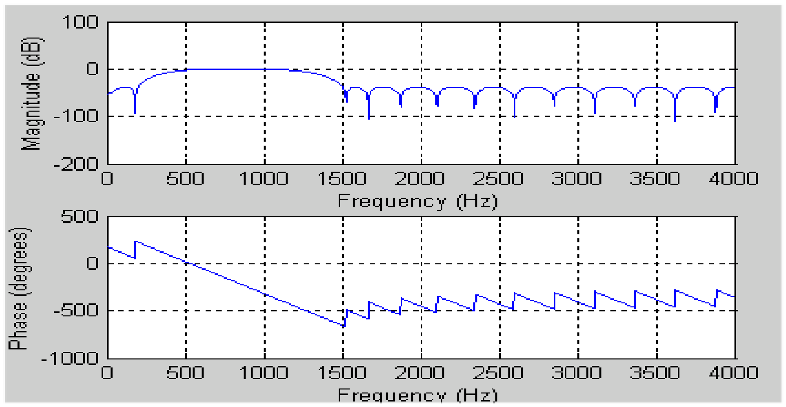
*Enter Attenuations values for Pass Band Ripple and Stop Band Attenuations.*

*Enter Sampling Rate for FIR filter*

*Enter Band edges for low pass filter.*

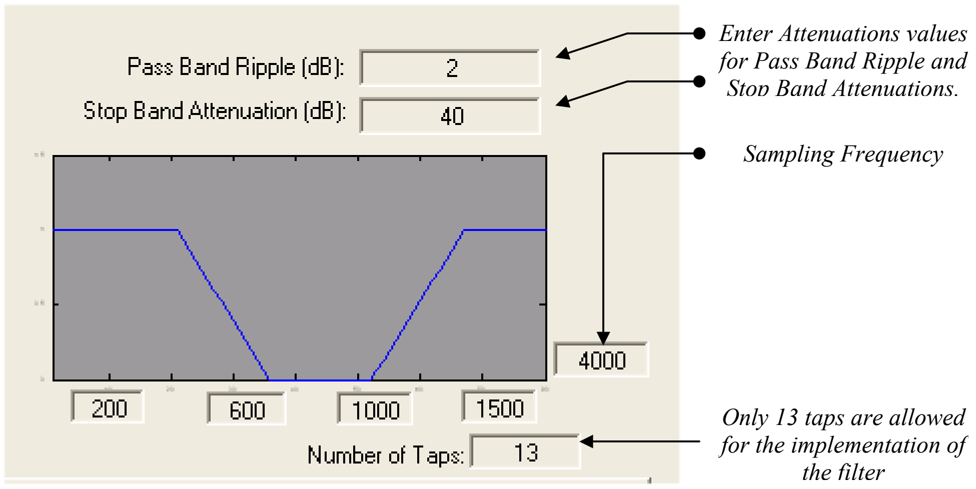
*The Desired Filter order needs to be entered here*

Unlike the LPF and the HPF the optimal number of coefficients are not automatically determined for the FIR band-pass filter configuration. As the Filter order is increased the quality of the filter improves as well.



## BAND STOP FILTER DESIGN

Band stop filter is design follows the same set of steps as the Band pass filter. If the filter order is under estimated then the filter response curves indicate the deficiency. Say the user specifies a band stop filter with the given specifications. Notice that only 13 orders are allowed to obtain a 40db drop in the stop band.


Pass Band Ripple (dB): 2

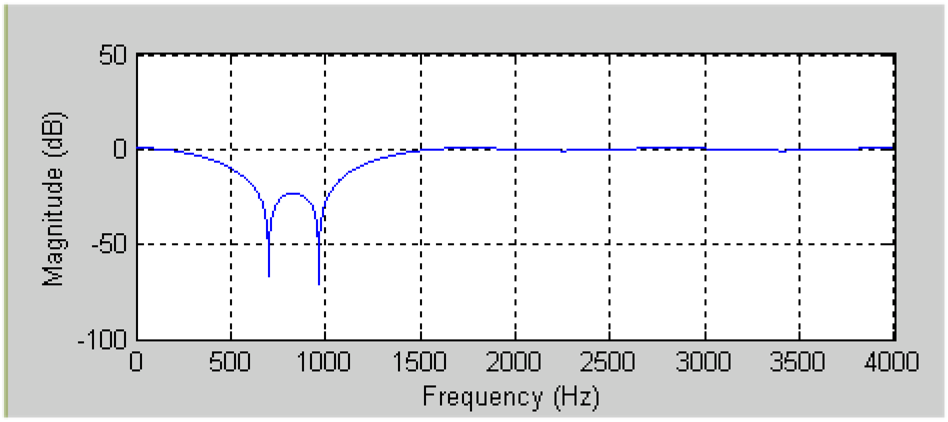
Stop Band Attenuation (dB): 40

4000

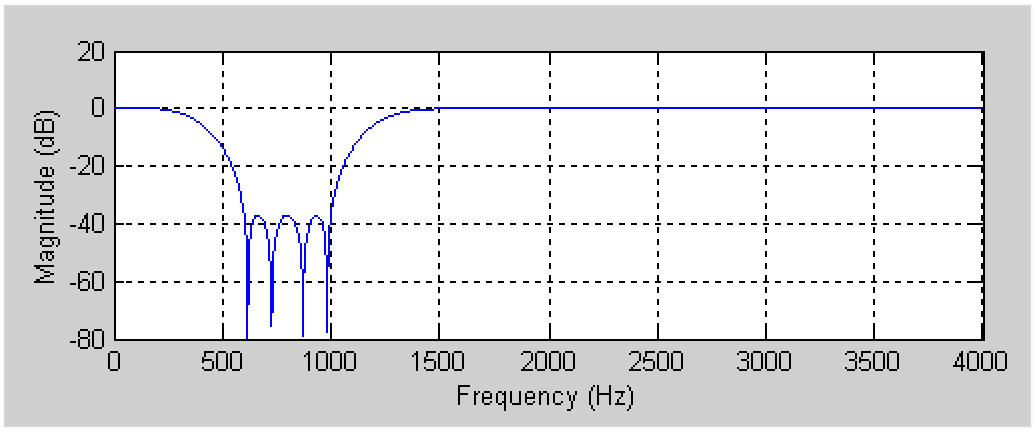
200 600 1000 1500

Number of Taps: 13

By plotting the frequency and phase curves it is possible to check whether 13 filter taps are enough to attain 40db in the stop band.



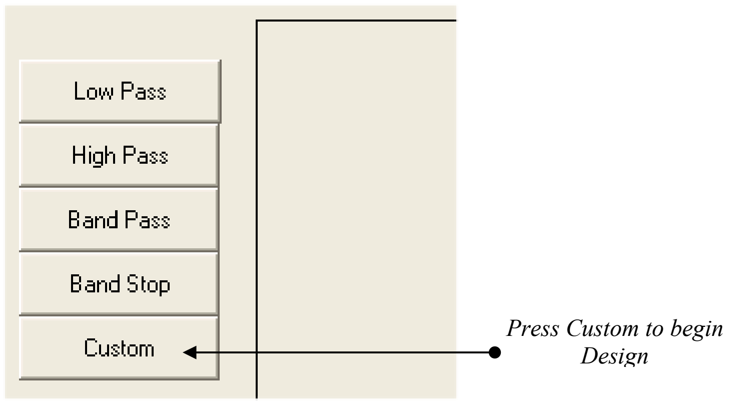
The graph reveals that a 13-tap filter is not sufficient to attain the desired attenuations. The order must be revised to a higher value, say 40 taps, and the filter curves are plotted again.



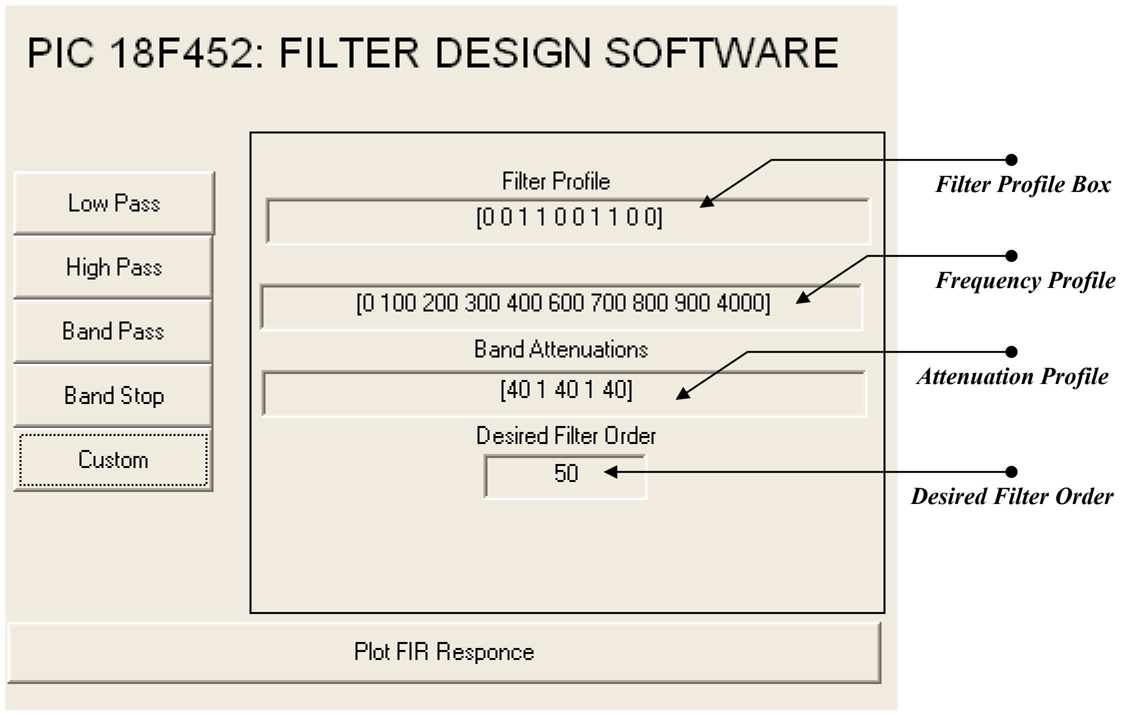
The updated response curve reveals that 40db drop has been achieved and the code can now be generated in the same manner as before.

## CUSTOM FILTER DESIGN

This is probably the most flexible aspect of FIR filter design because it allows the development of complex filters, which can compensate several different bands at once. The specification of Custom filters is a slightly different than the previous filters. In order to design them the custom button must be selected first in the main window.



The custom filter design interface is different from the basic filter design interface. In order to create a custom filter four boxes must be filled with the appropriate information. ***The filter profile***, the ***frequency profile***, the ***band attenuations*** and the ***desired filter order***.


PIC 18F452: FILTER DESIGN SOFTWARE

Filter Profile: [0 0 1 1 0 0 1 1 0 0]

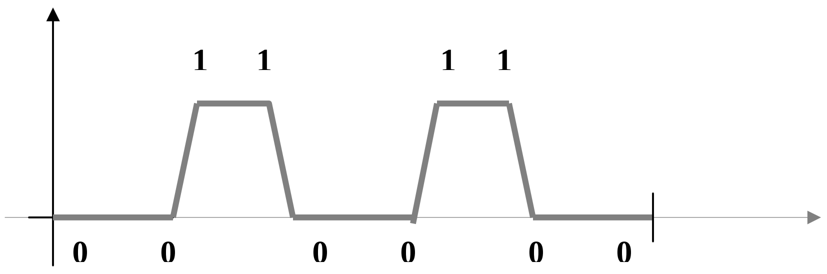
[0 100 200 300 400 600 700 800 900 4000]

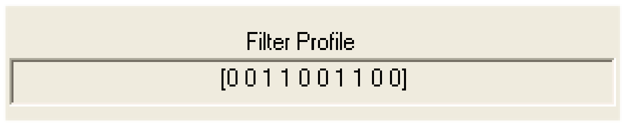
Band Attenuations: [40 1 40 1 40]

Desired Filter Order: 50

***Filter Profile***

Filter profile determines the band edges of the filter. A stop-band is designated by [0,0] and a pass-band is designated by [1,1].

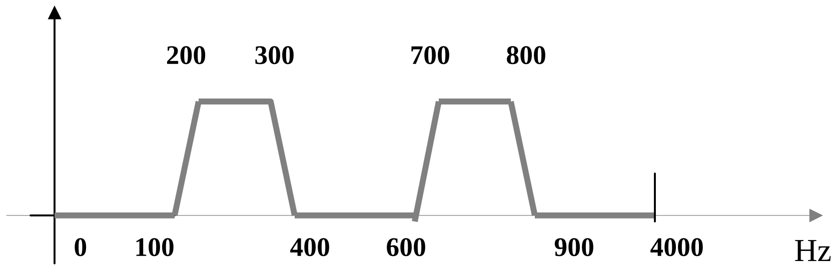


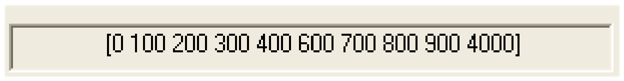

Filter Profile

[0 0 1 1 0 0 1 1 0 0]

### *Frequency Profile*

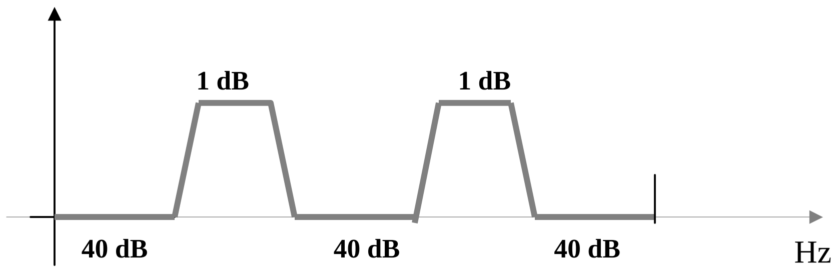
The frequency profile is simply the corner frequencies for each band edge and is supplied to the program in the appropriate box.

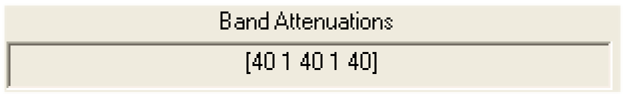



[0 100 200 300 400 600 700 800 900 4000]

### *Band Attenuation Profile*

Attenuation for each stop-band must be provided as well as the pass band ripple for every pass-band.



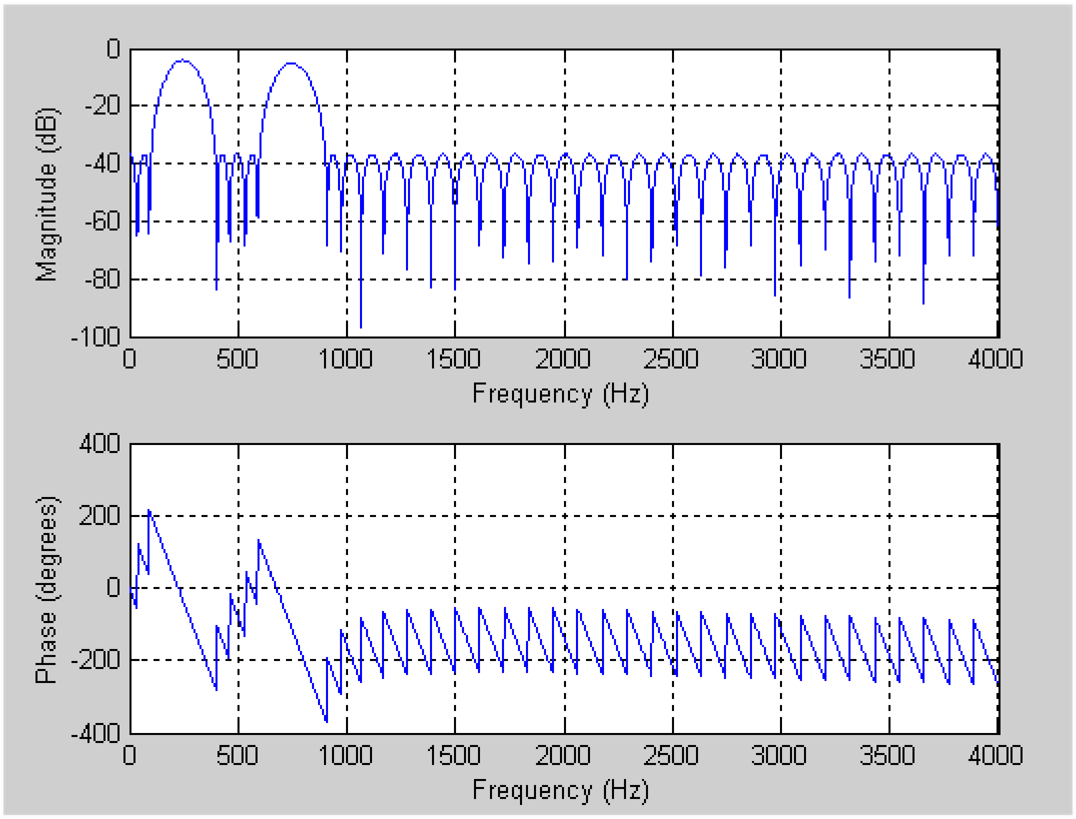

Band Attenuations

[40 1 40 1 40]

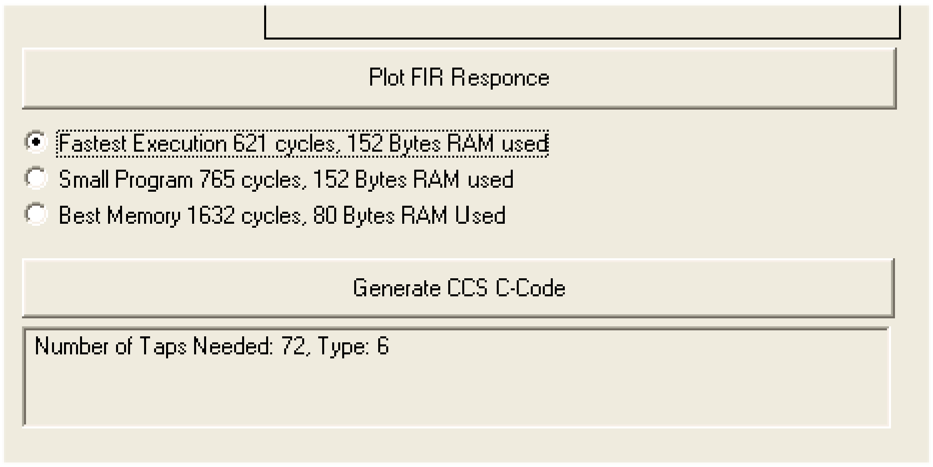
### *Desired Filter Order*

This box is filled with the number of coefficients desired by the designer. As before it is important to check the response curve to make sure the specified attenuations are being correctly met.

Plotting the frequency response button shows the curves and after increasing the filter order from 50 to 70 the target attenuations are met.



Filter code can be generated as before after selecting an appropriate implementation strategy desired by the user.



[✓ Fastest Execution 621 cycles, 152 Bytes RAM used]
Small Program 765 cycles, 152 Bytes RAM used
Best Memory 1632 cycles, 80 Bytes RAM Used

Number of Taps Needed: 72, Type: 6

# APPENDIX B

## *MATLAB CODE FOR FILTER DESIGN SOFTWARE*

```
function varargout = fildes(varargin)
% FILDES Application M-file for fildes.fig
% FIG = FILDES launch fildes GUI.
% FILDES('callback_name', ...) invoke the named callback.
% Last Modified by GUIDE v2.0 11-Apr-2004 21:33:20

global gdata;
global handles;

if nargin == 0  % LAUNCH GUI
    fig = openfig(mfilename,'reuse');

    % Use system color scheme for figure:
    set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'));

    % Generate a structure of handles to pass to callbacks, and store it.
    handles = guihandles(fig);
    guidata(fig, handles);
    if nargout > 0
            varargout{1} = fig;
    end

  % INVOKE NAMED SUBFUNCTION OR CALLBACK
  elseif ischar(varargin{1})
    try
            if (nargout)
                    [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
            else
                    feval(varargin{:}); % FEVAL switchyard
            end
    catch
            disp(lasterr);
    end
  end


% --------------------------------------------------------------------
function varargout = FIRpush(h, eventdata, handles, varargin)
set(handles.lms,'enable','off');

% --------------------------------------------------------------------
function varargout = LMSpush(h, eventdata, handles, varargin)
set(handles.fir,'enable','off');

% --------------------------------------------------------------------
function varargout = lpf_Callback(h, eventdata, handles, varargin)
global gdata;

set(handles.cover,'visible','off');
custom_off;
axes(handles.box);
x=ones(1,30);
y=1:-1/25:0;
z=zeros(1,30);
plot([x y z]);
set(handles.box,'xlim',[1 90]);
set(handles.box,'ylim',[0 1.5]);
set(handles.box,'color',[0.6 0.6 0.6]);
set(handles.pfr,'enable','on');
set(handles.bpa,'visible','off');
set(handles.bpb,'visible','off');
set(handles.bpc,'visible','off');
set(handles.bpd,'visible','off');
set(handles.pbco,'visible','on');
set(handles.sbc,'visible','on');
set(handles.tapn,'visible','off');
set(handles.taptext,'visible','off');
gdata.type = 1;

% --------------------------------------------------------------------
function varargout = hpf_Callback(h, eventdata, handles, varargin)
global gdata;
set(handles.cover,'visible','off');
custom_off;
axes(handles.box);
x=ones(1,37);
y=0:1/25:1;
z=zeros(1,30);
plot([z y x]);
set(handles.box,'xlim',[1 90]);
set(handles.box,'ylim',[0 1.5]);
set(handles.box,'color',[0.6 0.6 0.6]);
set(handles.pfr,'enable','on');
set(handles.bpa,'visible','off');
set(handles.bpb,'visible','off');
set(handles.bpc,'visible','off');
set(handles.bpd,'visible','off');
set(handles.pbco,'visible','on');
set(handles.sbc,'visible','on');
set(handles.tapn,'visible','off');
set(handles.taptext,'visible','off');
gdata.type = 2;

% --------------------------------------------------------------------
function varargout = bpf_Callback(h, eventdata, handles, varargin)
global gdata;
set(handles.cover,'visible','off');
custom_off;
axes(handles.box);
z = zeros(1,15);
x = ones(1,20);
r = 0:1/15:1;
f = 1:-1/15:0;
plot([z r x f z]);
set(handles.box,'xlim',[1 80]);
set(handles.box,'ylim',[0 1.5]);
set(handles.box,'color',[0.6 0.6 0.6]);
set(handles.pbco,'visible','off');
set(handles.sbc,'visible','off');
set(handles.bpa,'visible','on');
set(handles.bpb,'visible','on');
set(handles.bpc,'visible','on');
set(handles.bpd,'visible','on');
set(handles.pfr,'enable','on');
set(handles.tapn,'visible','on');
set(handles.taptext,'visible','on');
gdata.type = 3;

% --------------------------------------------------------------------
function varargout = bsf_Callback(h, eventdata, handles, varargin)
global gdata;
set(handles.cover,'visible','off');
custom_off;
axes(handles.box);
```

```
z = zeros(1,15);
x = ones(1,20);
r = 0:1/15:1;
f = 1:-1/15:0;
plot([x f z r x]);
set(handles.box,'xlim',[1 80]);
set(handles.box,'ylim',[0 1.5]);
set(handles.box,'color',[0.6 0.6 0.6]);
set(handles.pbco,'visible','off');
set(handles.sbc,'visible','off');
set(handles.bpa,'visible','on');
set(handles.bpb,'visible','on');
set(handles.bpc,'visible','on');
set(handles.bpd,'visible','on');
set(handles.pfr,'enable','on');
set(handles.tapn,'visible','on');
set(handles.taptext,'visible','on');
gdata.type = 4;

% --------------------------------------------------------------------
function varargout = m2o_Callback(h, eventdata, handles, varargin)
global gdata;
set(handles.cover,'visible','on');
set(handles.tprofile,'visible','on');
set(handles.profile,'visible','on');
set(handles.bedges,'visible','on');
set(handles.text12,'visible','on');
set(handles.customtaps,'visible','on');
set(handles.attnt,'visible','on');
set(handles.atten,'visible','on');
set(handles.pfr,'enable','on');
gdata.type = 6;

% --------------------------------------------------------------------
function varargout = gccc_Callback(h, eventdata, handles, varargin)
global g;
global gdata;
global imptype;

fid = fopen('filter.c','w');
time = clock;
% Come to here..........

type = imptype ;    % 1. DBUR, 2. DBNUR, 3. SBNUR

% Double Buffer + UNROLLED LOOPS..........................................................................................
if (type==1)

fprintf(fid,'%s \n',['// PIC 18F452 CODE FOR FIR FILTER GENERATION']);
fprintf(fid,'%s \n',['// Date: ' Date ' , Time (Hr:Min:Sec)-> ' num2str(time(4)) ':' num2str(time(5)) ':'
num2str(time(6))]);
fprintf(fid,'%s \n',['// FIR Filter Type: ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n',['#include <18f452.h>']);
fprintf(fid,'%s \n',['#use delay(clock = 40000000)']);
fprintf(fid,'%s \n',['#fuses H4,PUT,NOWDT']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n', ['const int filter_length = ' num2str(length(g)) ';']);

fprintf(fid,'%s', ['const int taps[filter_length] = {']);
for n=1:1:length(g)
        fprintf(fid,'%i',g(n));
```

```
    if n<length(g)
        fprintf(fid,',');
    end
end
fprintf(fid,'%s \n',['};']);


fprintf(fid,'\n \n');

fprintf(fid,'%s \n',['// PIC 18F452 Register
MAP.................................................................................................................................................. //']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['// ACCUMULATOR ADDRESS ']);
fprintf(fid,'%s \n',['#byte    WREG = 0xFE8              // Register Stores the Carry Bit                    ']);
fprintf(fid,'%s \n',['#byte    PRODL =0xff3              // Product Low Byte                                 ']);
fprintf(fid,'%s \n',['#byte    PRODH =0xff4              // Product High Byte                                ']);
fprintf(fid,'%s \n',['#byte   ADRESL = 0xfc3             // Low Byte for ADC Sample                          ']);
fprintf(fid,'%s \n',['#byte   ADRESH = 0xfc4             // High Byte for ADC Sample                         ']);
fprintf(fid,'%s \n',['#byte   STATUS = 0xfd8             // Status Register                                  ']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['// DC CONTROL REGISTERS ']);
fprintf(fid,'%s \n',['#byte   ADCON0 = 0xfc2    // ADC Control Register (High)                               ']);
fprintf(fid,'%s \n',['#byte   ADCON1 = 0xfc1    // ADC Control Register (Low)                                ']);
fprintf(fid,'%s \n',['#byte   ADRESL = 0xfc3    // Low Byte for ADC Sample                                   ']);
fprintf(fid,'%s \n',['#byte   ADRESH = 0xfc4                // High Byte for ADC Sample                      ']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['// DIGITAL IO PORT ADDRESSES                                                           ']);
fprintf(fid,'%s \n',['#byte   PORTA = 0xf80         // Port A Address                                        ']);
fprintf(fid,'%s \n',['#byte   PORTB = 0xf81         // Port B Address                                        ']);
fprintf(fid,'%s \n',['#byte   PORTC = 0xf82         // Port C Address                                        ']);
fprintf(fid,'%s \n',['#byte   PORTD = 0xf83         // Port D Address                                        ']);
fprintf(fid,'%s \n',['#byte   PORTE = 0xf84         // Port E Address                                        ']);
fprintf(fid,'%s \n',['#byte    LATA = 0xf89         // Set Driection for PORTA                               ']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['//  INDIRECT ADDRESSING ']);
fprintf(fid,'%s \n',['#byte   FSR0H = 0xfeA         // Hardware File Pointer0 (High)                         ']);
fprintf(fid,'%s \n',['#byte   FSR0L = 0xfe9         // Hardware File Pointer0 (Low)                          ']);
fprintf(fid,'%s \n',['#byte   FSR1H = 0xfe2         // Hardware File Pointer1 (High)                         ']);
fprintf(fid,'%s \n',['#byte   FSR1L = 0xfe1         // Hardware File Pointer1 (Low)                          ']);
fprintf(fid,'%s \n',['#byte   FSR2H = 0xfda         // Hardware File Pointer2 (High)                         ']);
fprintf(fid,'%s \n',['#byte   FSR2L = 0xfd9         // Hardware File Pointer2 (Low)                          ']);
fprintf(fid,'%s \n',['#byte   INDF0 = 0xfef         // Read Data Pointed by FSR0                             ']);
fprintf(fid,'%s \n',['#byte   INDF1 = 0xfe7         // Read Data Pointed by FSR1                             ']);
fprintf(fid,'%s \n',['#byte   INDF2 = 0xfdf         // Read Data Pointed by FSR2                             ']);
fprintf(fid,'%s \n',['#byte   PLUSW0 = 0xfeb        // Add Pointed data to WREG                              ']);
fprintf(fid,'%s \n',['#byte   PLUSW1 = 0xfe3        // Add Pointed data to WREG                              ']);
fprintf(fid,'%s \n',['#byte   PLUSW2 = 0xfdb        // Add Pointed data to WREG                              ']);
fprintf(fid,'%s \n',['#byte   PREINC0 = 0xfec        // Pre-increment pointer0                               ']);
fprintf(fid,'%s \n',['#byte   PREINC1 = 0xfe4        // Pre-increment pointer1                               ']);
fprintf(fid,'%s \n',['#byte   PREINC2 = 0xfdc        // Pre-increment pointer2                               ']);
fprintf(fid,'%s \n',['#byte   POSTINC0 = 0xfee        // Post-Incerement Pointer0                            ']);
fprintf(fid,'%s \n',['#byte   POSTDEC0 = 0xfed        // Post-Decrement Pointer0                             ']);
fprintf(fid,'%s \n',['#byte   POSTINC1 = 0xfe6        // Post-Increment Pointer1                             ']);
fprintf(fid,'%s \n',['#byte   POSTDEC1 = 0xfe5        // Post-Decrement Pointer1                             ']);
fprintf(fid,'%s \n',['#byte   POSTINC2 = 0xfde        // Post-Increment Pointer2                             ']);
fprintf(fid,'%s \n',['#byte   POSTDEC2 = 0xfdd        // Post-Decrement Pointer2                             ']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['// TIMER REGISTERS ']);
fprintf(fid,'%s \n',['#byte PR2 = 0xfcb  ']);
fprintf(fid,'%s \n',['#byte TMR2 = 0xfcc  ']);
fprintf(fid,'%s \n',['#byte T2CON = 0xfca  ']);

fprintf(fid,'\n \n');

fprintf(fid,'%s \n',['//  GLOBAL VARIABLES                                                   ']);
fprintf(fid,'%s \n',['int buf0[filter_length] = {0};        // Store ADC Values                  ']);
fprintf(fid,'%s \n',['int buf1[filter_length] = {0};        // Store ADC Values                  ']);
fprintf(fid,'%s \n',['int coef[filter_length] = {0};        // Store offset Coefficients         ']);
fprintf(fid,'%s \n',['int output_most = 0;                    // Most Significant Byte of Output   ']);
fprintf(fid,'%s \n',['int output_middle = 0;      // Middle Significant Byte of Output .         ']);
fprintf(fid,'%s \n',['int output_least = 0;       // Least Significant Byte of Output '         ]);
fprintf(fid,'%s \n',['int Xn_high_256=0;                 // Most Significant Byte of Xn Summation * 255   ']);
fprintf(fid,'%s \n',['int Xn_mid_256=0;                  // Mid Significant Byte of Xn Summation * 255    ']);
fprintf(fid,'%s \n',['int Xn_low_256=0;                  // Least Significant Byte of Xn Summation * 255  ']);
fprintf(fid,'%s \n',['int Xn_high_128=0;                 // Most Significant Byte of Xn Summation * 128   ']);
fprintf(fid,'%s \n',['int Xn_mid_128=0;                  // Mid Significant Byte of Xn Summation * 128    ']);
fprintf(fid,'%s \n',['int Xn_low_128=0;                  // Least Significant Byte of Xn Summation * 128  ']);
fprintf(fid,'%s \n',['int EOB, MAC_count;           // Counters for MAC and END of Buffer.           ']);
fprintf(fid,'%s \n',['int n,c, tptr0, tptr1;        // Temporary Variabes                            ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n',['// GLOBAL PROTOTYPES                                                   ']);
fprintf(fid,'%s \n',['void offset_and_buffer_tap_coefficients(void);                         ']);
fprintf(fid,'%s \n',['void initialize_pointers(void);']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n',['// INTERRUPT SERVICE ROUTINE'                                          ]);
fprintf(fid,'%s \n',['#INT_TIMER2                                                            ']);
fprintf(fid,'%s \n',['isr() {'                                                               ]);
fprintf(fid,'%s \n',['        T2CON = 0x06; // Restart Timer'                                ]);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['       FSR0L = tptr0;                                                  ']);
fprintf(fid,'%s \n',['       FSR1L = tptr1;'                                                 ]);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['        if (EOB == 0) {                                                ']);
fprintf(fid,'%s \n',['        FSR0L = &buf0[0];                                              ']);
fprintf(fid,'%s \n',['        FSR1L = &buf1[0];'                                             ]);
fprintf(fid,'%s \n',['        FSR2L = &coef[0];'                                             ]);
fprintf(fid,'%s \n',['        EOB   = filter_length;'                                        ]);
fprintf(fid,'%s \n',['        }'                                                             ]);
fprintf(fid,'%s \n',[''                                                                      ]);
fprintf(fid,'%s \n',['        // Subtract The oldest Xn Value from Total'                    ]);
fprintf(fid,'%s \n',['        #asm'                                                          ]);
fprintf(fid,'%s \n',['                movf           INDF0,W'                                ]);
fprintf(fid,'%s \n',['                    subwf      Xn_mid_256,F'                           ]);
fprintf(fid,'%s \n',['                    clrf       WREG'                                   ]);
fprintf(fid,'%s \n',['                    subwfb    Xn_high_256,F'                           ]);
fprintf(fid,'%s \n',['        #endasm'                                                       ]);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['        // Get the latest ADC value;'                                  ]);
fprintf(fid,'%s \n',['                    WREG = ADRESH;'                                    ]);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['        // Restart ADC;                                                ']);
fprintf(fid,'%s \n',['                       bit_set(ADCON0,2);'                             ]);
fprintf(fid,'%s \n',[''                                                                      ]);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['        // Add Latest ADC value to Y1(n)'                              ]);
fprintf(fid,'%s \n',['              #asm']);
fprintf(fid,'%s \n',['                    movwf      INDF0']);
fprintf(fid,'%s \n',['                    movwf     POSTINC1                               ']);
fprintf(fid,'%s \n',['                    addwf     Xn_mid_256,F                           ']);
fprintf(fid,'%s \n',['                    clrf          WREG'                              ]);
fprintf(fid,'%s \n',['                  addwfc     Xn_high_256,F                                    ']);
fprintf(fid,'%s \n',['              #endasm                                                             ']);
fprintf(fid,'%s \n',['                                                                                  ']);
fprintf(fid,'%s \n',['         // Prepare for MAC Cycles                                                ']);
fprintf(fid,'%s \n',['              FSR2L = &coef[0];'                                                  ]);
fprintf(fid,'%s \n',['              FSR0L = FSR0L+filter_length;'                                       ]);
fprintf(fid,'%s \n',['              EOB = EOB - 1;'                                                     ]);
fprintf(fid,'%s \n',['              MAC_count = filter_length;'                                         ]);
fprintf(fid,'%s \n',['                                                                                  ']);
fprintf(fid,'%s \n',['         //  Begin MAC Cycle repeat till done then computer Output'               ]);
fprintf(fid,'%s \n',['              #asm'                                                               ]);
fprintf(fid,'%s \n',['                 clrf                      output_least'                          ]);
fprintf(fid,'%s \n',['                 clrf          output_middle'                                     ]);
fprintf(fid,'%s \n',['                 clrf          output_most'                                       ]);
for n=1:1:length(g)
fprintf(fid,'%s \n',['// MAC CYCLE NUMBER: ', num2str(n) ' ----------------------------//             ']);
fprintf(fid,'%s \n',['                 movf          POSTDEC0,W'                                        ]);
fprintf(fid,'%s \n',['                 mulwf         POSTINC2'                                          ]);
fprintf(fid,'%s \n',['                 movf          PRODL,W'                                           ]);
fprintf(fid,'%s \n',['                 addwf         output_least'                                      ]);
fprintf(fid,'%s \n',['                 movf          PRODH,W'                                           ]);
fprintf(fid,'%s \n',['                 addwfc        output_middle'                                     ]);
fprintf(fid,'%s \n',['                 clrf          WREG'                                              ]);
fprintf(fid,'%s \n',['                 addwfc        output_most'                                       ]);
end


fprintf(fid,'%s \n',['              COMPUTE_OUTPUT:'                                                    ]);
fprintf(fid,'%s \n',['                 bcf            STATUS, 0'                                        ]);
fprintf(fid,'%s \n',['                 incf          FSR0L'                                             ]);
fprintf(fid,'%s \n',['                 rrcf          Xn_high_256,W                                      ']);
fprintf(fid,'%s \n',['                 movwf         Xn_high_128                                        ']);
fprintf(fid,'%s \n',['                 rrcf          Xn_mid_256,W                                       ']);
fprintf(fid,'%s \n',['                 movwf         Xn_mid_128                                         ']);
fprintf(fid,'%s \n',['                 rrcf          Xn_low_256,W                                       ']);
fprintf(fid,'%s \n',['                 movwf         Xn_low_128                                         ']);
fprintf(fid,'%s \n',['                 subwf         output_least,F                                     ']);
fprintf(fid,'%s \n',['                 movf          Xn_mid_128,W'                                       ]);
fprintf(fid,'%s \n',['                 subwfb        output_middle,F'                                   ]);
fprintf(fid,'%s \n',['                 movf          Xn_high_128,W'                                     ]);
fprintf(fid,'%s \n',['                 subwfb        output_most,F'                                     ]);
fprintf(fid,'%s \n',['                                                                                  ']);
fprintf(fid,'%s \n',['              #endasm                                                             ']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['                   tptr0 = FSR0L;']);
fprintf(fid,'%s \n',['                   tptr1 = FSR1L;']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['        // Scale output...........']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['         #asm']);
fprintf(fid,'%s \n',['                   rrcf  output_most,F']);
fprintf(fid,'%s \n',['                   rrcf  output_middle,F']);
fprintf(fid,'%s \n',['                     rrcf output_most,F']);
fprintf(fid,'%s \n',['                   rrcf output_middle,F              ']);
fprintf(fid,'%s \n',['        #endasm']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['           PORTD = output_middle;    ']);
fprintf(fid,'%s \n',['      } // End Interrupt']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n',['void main() {                                                    ']);
fprintf(fid,'%s \n',['                                                                 ']);
fprintf(fid,'%s \n',['                         set_tris_d(0);                          ']);
fprintf(fid,'%s \n',['                                                                 ']);
fprintf(fid,'%s \n',['                         // Setup ADC in interrupt mode          ']);
fprintf(fid,'%s \n',['                         setup_adc_ports(ALL_ANALOG);            ']);
fprintf(fid,'%s \n',['                         setup_adc(ADC_CLOCK_DIV_64);            ']);
fprintf(fid,'%s \n',['                         set_adc_channel(0);                     ‘]);
fprintf(fid,'%s \n',['                                                                 ']);
fprintf(fid,'%s \n',['                         // Setup Timer0 in interrupt Mode       ']);
fprintf(fid,'%s \n',['                         T2CON = 0x06;                           ']);
fprintf(fid,'%s \n',['                              PR2 = 76;                          ']);
fprintf(fid,'%s \n',['                         enable_interrupts(INT_TIMER2);          ']);
fprintf(fid,'%s \n',['                         enable_interrupts(GLOBAL);              ']);
fprintf(fid,'%s \n',['                                                                 ']);
fprintf(fid,'%s \n',['                         // FIR filter Code                      ']);
fprintf(fid,'%s \n',['                         offset_and_buffer_tap_coefficients();   ']);
fprintf(fid,'%s \n',['                                                                 ']);
fprintf(fid,'%s \n',['                         // Initialize Pointers                  ']);
fprintf(fid,'%s \n',['                         tptr0 = &buf0[0];                       ']);
fprintf(fid,'%s \n',['                              tptr1 = &buf1[0];                  ']);
fprintf(fid,'%s \n',['                              FSR2L = &coef[0];                  ']);
fprintf(fid,'%s \n',['                              EOB = filter_length;               ']);
fprintf(fid,'%s \n',['                                                                 ']);
fprintf(fid,'%s \n',['                         // Start ADC.                           ']);
fprintf(fid,'%s \n',['                         bit_set(ADCON0,2);                      ']);
fprintf(fid,'%s \n',['                         set_rtcc(65517);                        ']);
fprintf(fid,'%s \n',['                                                                 ']);
fprintf(fid,'%s \n',['                         //  Main Loop                           ']);
fprintf(fid,'%s \n',['                              while(1)  {                        ']);
fprintf(fid,'%s \n',['                         // Main Application                     ']);
fprintf(fid,'%s \n',['                              }                                  ']);
fprintf(fid,'%s \n',['}                                                                ']);


fprintf(fid,'\n \n \n');
fprintf(fid,'%s \n',['void offset_and_buffer_tap_coefficients(void)  {                ' ]);
fprintf(fid,'%s \n',['     int n;                                                      ' ]);
fprintf(fid,'%s \n',['     for (n=0; n<filter_length; n++) {                           ' ]);
fprintf(fid,'%s \n',['                 coef[n] = taps[n]+0x80;                         ' ]);
fprintf(fid,'%s \n',['     }                                                           ' ]);
fprintf(fid,'%s \n',['}                                                                ' ]);

end

% DOUBLE BUFFERED: Non UNROLLED LOOPS..................................................

if (type == 2)

fprintf(fid,'%s \n',['// PIC 18F452 CODE FOR FIR FILTER GENERATION']);
fprintf(fid,'%s \n',['// Date: ' Date ' , Time (Hr:Min:Sec)-> ' num2str(time(4)) ':' num2str(time(5)) ':'
num2str(time(6))]);
fprintf(fid,'%s \n',['// FIR Filter Type: ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n',['#include <18f452.h>']);
fprintf(fid,'%s \n',['#use delay(clock = 40000000)']);
fprintf(fid,'%s \n',['#fuses H4,PUT,NOWDT']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n', ['const int filter_length = ' num2str(length(g)) ';']);

fprintf(fid,'%s', ['const int taps[filter_length] = {']);
for n=1:1:length(g)
```

```
            fprintf(fid,'%i',g(n));
      if n<length(g)
         fprintf(fid,',');
      end
end
fprintf(fid,'%s \n',['};']);


fprintf(fid,'\n \n');

fprintf(fid,'%s \n',['// PIC 18F452 Register
MAP.......................................................................................................................................... //']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['// ACCUMULATOR ADDRESS ']);
fprintf(fid,'%s \n',['#byte    WREG = 0xFE8               // Register Stores the Carry Bit                    ']);
fprintf(fid,'%s \n',['#byte    PRODL =0xff3               // Product Low Byte                                 ']);
fprintf(fid,'%s \n',['#byte    PRODH =0xff4               // Product High Byte                                ']);
fprintf(fid,'%s \n',['#byte   ADRESL = 0xfc3              // Low Byte for ADC Sample                          ']);
fprintf(fid,'%s \n',['#byte   ADRESH = 0xfc4              // High Byte for ADC Sample                         ']);
fprintf(fid,'%s \n',['#byte   STATUS = 0xfd8              // Status Register                                  ']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['// DC CONTROL REGISTERS                                                             ']);
fprintf(fid,'%s \n',['#byte   ADCON0 = 0xfc2    // ADC Control Register (High)                            ']);
fprintf(fid,'%s \n',['#byte   ADCON1 = 0xfc1    // ADC Control Register (Low)                             ']);
fprintf(fid,'%s \n',['#byte   ADRESL = 0xfc3    // Low Byte for ADC Sample                                ']);
fprintf(fid,'%s \n',['#byte   ADRESH = 0xfc4                  // High Byte for ADC Sample                     ']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['// DIGITAL IO PORT ADDRESSES                                                        ']);
fprintf(fid,'%s \n',['#byte   PORTA = 0xf80          // Port A Address                                    ']);
fprintf(fid,'%s \n',['#byte   PORTB = 0xf81          // Port B Address                                    ']);
fprintf(fid,'%s \n',['#byte   PORTC = 0xf82          // Port C Address                                    ']);
fprintf(fid,'%s \n',['#byte   PORTD = 0xf83          // Port D Address                                    ']);
fprintf(fid,'%s \n',['#byte   PORTE = 0xf84          // Port E Address                                    ']);
fprintf(fid,'%s \n',['#byte    LATA = 0xf89          // Set Driection for PORTA                           ']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['//  INDIRECT ADDRESSING ']);
fprintf(fid,'%s \n',['#byte   FSR0H = 0xfeA          // Hardware File Pointer0 (High)                     ']);
fprintf(fid,'%s \n',['#byte   FSR0L = 0xfe9          // Hardware File Pointer0 (Low)                      ']);
fprintf(fid,'%s \n',['#byte   FSR1H = 0xfe2          // Hardware File Pointer1 (High)                     ']);
fprintf(fid,'%s \n',['#byte   FSR1L = 0xfe1          // Hardware File Pointer1 (Low)                      ']);
fprintf(fid,'%s \n',['#byte   FSR2H = 0xfda          // Hardware File Pointer2 (High)                     ']);
fprintf(fid,'%s \n',['#byte   FSR2L = 0xfd9          // Hardware File Pointer2 (Low)                      ']);
fprintf(fid,'%s \n',['#byte   INDF0 = 0xfef          // Read Data Pointed by FSR0                         ']);
fprintf(fid,'%s \n',['#byte   INDF1 = 0xfe7          // Read Data Pointed by FSR1                         ']);
fprintf(fid,'%s \n',['#byte   INDF2 = 0xfdf          // Read Data Pointed by FSR2                         ']);
fprintf(fid,'%s \n',['#byte   PLUSW0 = 0xfeb         // Add Pointed data to WREG                          ']);
fprintf(fid,'%s \n',['#byte   PLUSW1 = 0xfe3         // Add Pointed data to WREG                          ']);
fprintf(fid,'%s \n',['#byte   PLUSW2 = 0xfdb         // Add Pointed data to WREG                          ']);
fprintf(fid,'%s \n',['#byte   PREINC0 = 0xfec         // Pre-increment pointer0                           ']);
fprintf(fid,'%s \n',['#byte   PREINC1 = 0xfe4         // Pre-increment pointer1                           ']);
fprintf(fid,'%s \n',['#byte   PREINC2 = 0xfdc         // Pre-increment pointer2                           ']);
fprintf(fid,'%s \n',['#byte   POSTINC0 = 0xfee        // Post-Incerement Pointer0                         ']);
fprintf(fid,'%s \n',['#byte   POSTDEC0 = 0xfed        // Post-Decrement Pointer0                          ']);
fprintf(fid,'%s \n',['#byte   POSTINC1 = 0xfe6        // Post-Increment Pointer1                          ']);
fprintf(fid,'%s \n',['#byte   POSTDEC1 = 0xfe5        // Post-Decrement Pointer1                          ']);
fprintf(fid,'%s \n',['#byte   POSTINC2 = 0xfde        // Post-Increment Pointer2                          ']);
fprintf(fid,'%s \n',['#byte   POSTDEC2 = 0xfdd        // Post-Decrement Pointer2                          ']);
fprintf(fid,'%s \n',['                                                                                    ']);
fprintf(fid,'%s \n',['                                                                                    ']);
fprintf(fid,'%s \n',['// TIMER REGISTERS                                                                  ']);
fprintf(fid,'%s \n',['#byte PR2 = 0xfcb                                                                    ']);
fprintf(fid,'%s \n',['#byte TMR2 = 0xfcc                                                                   ']);
fprintf(fid,'%s \n',['#byte T2CON = 0xfca                                                                  ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n',['//  GLOBAL VARIABLES                                                                 ]);
fprintf(fid,'%s \n',['int buf0[filter_length] = {0};          // Store ADC Values                          ']);
fprintf(fid,'%s \n',['int buf1[filter_length] = {0};          // Store ADC Values                          ']);
fprintf(fid,'%s \n',['int coef[filter_length] = {0};          // Store offset Coefficients                 ']);
fprintf(fid,'%s \n',['int output_most = 0;                       // Most Significant Byte of Output        ']);
fprintf(fid,'%s \n',['int output_middle = 0;        // Middle Significant Byte of Output                   ']);
fprintf(fid,'%s \n',['int output_least = 0;         // Least Significant Byte of Output                    ']);
fprintf(fid,'%s \n',['int Xn_high_256=0;                 // Most Significant Byte of Xn Summation * 255    ']);
fprintf(fid,'%s \n',['int Xn_mid_256=0;                  // Mid Significant Byte of Xn Summation * 255     ']);
fprintf(fid,'%s \n',['int Xn_low_256=0;                  // Least Significant Byte of Xn Summation * 255   ']);
fprintf(fid,'%s \n',['int Xn_high_128=0;                 // Most Significant Byte of Xn Summation * 128    ']);
fprintf(fid,'%s \n',['int Xn_mid_128=0;                  // Mid Significant Byte of Xn Summation * 128     ']);
fprintf(fid,'%s \n',['int Xn_low_128=0;                  // Least Significant Byte of Xn Summation * 128   ']);
fprintf(fid,'%s \n',['int EOB, MAC_count;           // Counters for MAC and END of Buffer.                 ']);
fprintf(fid,'%s \n',['int n,c, tptr0, tptr1;        // Temporary Variabes                                  ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n',['// GLOBAL PROTOTYPES                                                                 ']);
fprintf(fid,'%s \n',['void offset_and_buffer_tap_coefficients(void);                                      ']);
fprintf(fid,'%s \n',['void initialize_pointers(void);                                                     ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n',['// INTERRUPT SERVICE ROUTINE                                                         ']);
fprintf(fid,'%s \n',['#INT_TIMER2                                                                         ']);
fprintf(fid,'%s \n',['isr() {                                                                             ']);
fprintf(fid,'%s \n',['        T2CON = 0x06; // Restart Timer                                              ']);
fprintf(fid,'%s \n',['                                                                                    ']);
fprintf(fid,'%s \n',['                                                                                    ']);
fprintf(fid,'%s \n',['        FSR0L = tptr0;                                                              ']);
fprintf(fid,'%s \n',['        FSR1L = tptr1;                                                              ']);
fprintf(fid,'%s \n',['                                                                                    ']);
fprintf(fid,'%s \n',['   if (EOB == 0) {                                                                  ']);
fprintf(fid,'%s \n',['                                    FSR0L = &buf0[0];                               ']);
fprintf(fid,'%s \n',['                                    FSR1L = &buf1[0];                               ']);
fprintf(fid,'%s \n',['                                    FSR2L = &coef[0];                               ']);
fprintf(fid,'%s \n',['                                    EOB   = filter_length;                          ']);
fprintf(fid,'%s \n',['        }                                                                           ']);
fprintf(fid,'%s \n',['                                                                                    ']);
fprintf(fid,'%s \n',['        // Subtract The oldest Xn Value from Total                                  ']);
fprintf(fid,'%s \n',['        #asm                                                                        ']);
fprintf(fid,'%s \n',['                movf      INDF0,W                                                   ']);
fprintf(fid,'%s \n',['                subwf     Xn_mid_256,F                                              ']);
fprintf(fid,'%s \n',['                clrf      WREG                                                      ']);
fprintf(fid,'%s \n',['                subwfb    Xn_high_256,F                                             ']);
fprintf(fid,'%s \n',['        #endasm                                                                     ']);
fprintf(fid,'%s \n',['                                                                                    ']);
fprintf(fid,'%s \n',['        // Get the latest ADC value;                                                ']);
fprintf(fid,'%s \n',['                WREG = ADRESH;                                                      ']);
fprintf(fid,'%s \n',['                                                                                    ']);
fprintf(fid,'%s \n',['        // Restart ADC;                                                             ']);
fprintf(fid,'%s \n',['                  bit_set(ADCON0,2);                                                ']);
fprintf(fid,'%s \n',['                                                                                    ']);
fprintf(fid,'%s \n',['                                                                                    ']);
fprintf(fid,'%s \n',['        // Add Latest ADC value to Y1(n)                                            ']);
fprintf(fid,'%s \n',['            #asm                                                                    ']);
fprintf(fid,'%s \n',['                movwf     INDF0                                                     ']);
fprintf(fid,'%s \n',['                movwf     POSTINC1                                                  ']);
fprintf(fid,'%s \n',['                addwf     Xn_mid_256,F                                              ']);
fprintf(fid,'%s \n',['                clrf      WREG                                                      ']);
fprintf(fid,'%s \n',['                addwfc    Xn_high_256,F      ']);
fprintf(fid,'%s \n',['            #endasm                                                    ']);
fprintf(fid,'%s \n',['                                                                        ']);
fprintf(fid,'%s \n',['        // Prepare for MAC Cycles                                       ']);
fprintf(fid,'%s \n',['            FSR2L = &coef[0];                                           ']);
fprintf(fid,'%s \n',['            FSR0L = FSR0L+filter_length;                                ']);
fprintf(fid,'%s \n',['            EOB = EOB - 1;                                              ']);
fprintf(fid,'%s \n',['            MAC_count = filter_length;                                  ']);
fprintf(fid,'%s \n',['                                                                        ']);
fprintf(fid,'%s \n',['        //  Begin MAC Cycle repeat till done then computer Output       ']);
fprintf(fid,'%s \n',['            #asm                                                        ']);
fprintf(fid,'%s \n',['              clrf        output_least                                  ']);
fprintf(fid,'%s \n',['              clrf        output_middle                                 ']);
fprintf(fid,'%s \n',['              clrf        output_most                                   ']);
fprintf(fid,'%s \n',['            MAC:                                                        ']);
fprintf(fid,'%s \n',['              movf        POSTDEC0,W                                    ']);
fprintf(fid,'%s \n',['              mulwf       POSTINC2                                      ']);
fprintf(fid,'%s \n',['              movf        PRODL,W                                       ']);
fprintf(fid,'%s \n',['              addwf       output_least                                  ']);
fprintf(fid,'%s \n',['              movf        PRODH,W                                       ']);
fprintf(fid,'%s \n',['              addwfc      output_middle                                 ']);
fprintf(fid,'%s \n',['              clrf        WREG                                          ']);
fprintf(fid,'%s \n',['              addwfc      output_most                                   ']);
fprintf(fid,'%s \n',['              decfsz      MAC_count                                     ']);
fprintf(fid,'%s \n',['              bra         MAC                                           ']);
fprintf(fid,'%s \n',['            COMPUTE_OUTPUT:                                             ']);
fprintf(fid,'%s \n',['              bcf         STATUS, 0                                     ']);
fprintf(fid,'%s \n',['              incf        FSR0L                                         ']);
fprintf(fid,'%s \n',['              rrcf        Xn_high_256,W                                 ']);
fprintf(fid,'%s \n',['              movwf       Xn_high_128                                   ']);
fprintf(fid,'%s \n',['              rrcf        Xn_mid_256,W                                  ']);
fprintf(fid,'%s \n',['              movwf       Xn_mid_128                                    ']);
fprintf(fid,'%s \n',['              rrcf        Xn_low_256,W                                  ']);
fprintf(fid,'%s \n',['              movwf       Xn_low_128                                    ']);
fprintf(fid,'%s \n',['              subwf       output_least,F                                ']);
fprintf(fid,'%s \n',['              movf        Xn_mid_128,W                                  ']);
fprintf(fid,'%s \n',['              subwfb      output_middle,F                               ']);
fprintf(fid,'%s \n',['              movf        Xn_high_128,W                                 ']);
fprintf(fid,'%s \n',['              subwfb      output_most,F                                 ']);
fprintf(fid,'%s \n',['                                                                        ']);
fprintf(fid,'%s \n',['            #endasm                                                     ']);
fprintf(fid,'%s \n',['                                                                        ']);
fprintf(fid,'%s \n',['                tptr0 = FSR0L;                                          ']);
fprintf(fid,'%s \n',['                tptr1 = FSR1L;                                          ']);
fprintf(fid,'%s \n',['                                                                        ']);
fprintf(fid,'%s \n',['       // Scale output...........                                       ']);
fprintf(fid,'%s \n',['                                                                        ']);
fprintf(fid,'%s \n',['        #asm                                                            ']);
fprintf(fid,'%s \n',['                rrcf  output_most,F                                     ']);
fprintf(fid,'%s \n',['                rrcf  output_middle,F                                   ']);
fprintf(fid,'%s \n',['                rrcf output_most,F                                      ']);
fprintf(fid,'%s \n',['                rrcf output_middle,F                                    ']);
fprintf(fid,'%s \n',['       #endasm                                                          ']);
fprintf(fid,'%s \n',['']);
fprintf(fid,'%s \n',['          PORTD = output_middle;                                        ']);
fprintf(fid,'%s \n',['     } // End Interrupt                                                 ']);

fprintf(fid,'\n \n');

fprintf(fid,'%s \n',['void main() {                                                          ']);
fprintf(fid,'%s \n',['                                                                        ']);
fprintf(fid,'%s \n',['       set_tris_d(0);                                                   ']);
fprintf(fid,'%s \n',['                                                                        ']);
fprintf(fid,'%s \n',['                              // Setup ADC in interrupt mode                         ']);
fprintf(fid,'%s \n',['                              setup_adc_ports(ALL_ANALOG);                           ']);
fprintf(fid,'%s \n',['                              setup_adc(ADC_CLOCK_DIV_64);                           ']);
fprintf(fid,'%s \n',['                              set_adc_channel(0);                                    ']);
fprintf(fid,'%s \n',['                                                                                     ']);
fprintf(fid,'%s \n',['                              // Setup Timer0 in interrupt Mode                      ']);
fprintf(fid,'%s \n',['                              T2CON = 0x06;                                          ']);
fprintf(fid,'%s \n',['                                        PR2 = 76;                                    ']);
fprintf(fid,'%s \n',['                              enable_interrupts(INT_TIMER2);                         ']);
fprintf(fid,'%s \n',['                              enable_interrupts(GLOBAL);                             ']);
fprintf(fid,'%s \n',['                                                                                     ']);
fprintf(fid,'%s \n',['                              // FIR filter Code                                     ']);
fprintf(fid,'%s \n',['                              offset_and_buffer_tap_coefficients();                  ']);
fprintf(fid,'%s \n',['                                                                                     ']);
fprintf(fid,'%s \n',['                              // Initialize Pointers                                 ']);
fprintf(fid,'%s \n',['                              tptr0 = &buf0[0];                                      ']);
fprintf(fid,'%s \n',['                                        tptr1 = &buf1[0];                            ']);
fprintf(fid,'%s \n',['                                        FSR2L = &coef[0];                            ']);
fprintf(fid,'%s \n',['                                        EOB = filter_length;                         ']);
fprintf(fid,'%s \n',['                                                                                     ']);
fprintf(fid,'%s \n',['                              // Start ADC.                                          ']);
fprintf(fid,'%s \n',['                              bit_set(ADCON0,2);                                     ']);
fprintf(fid,'%s \n',['                              set_rtcc(65517);                                       ']);
fprintf(fid,'%s \n',['                                                                                     ']);
fprintf(fid,'%s \n',['                              //  Main Loop                                          ']);
fprintf(fid,'%s \n',['                                        while(1)  {                                  ']);
fprintf(fid,'%s \n',['                              // Main Application                                    ']);
fprintf(fid,'%s \n',['                              }                                                      ']);
fprintf(fid,'%s \n',['}                                                                                    ']);
fprintf(fid,'\n \n \n');
fprintf(fid,'%s \n',['void offset_and_buffer_tap_coefficients(void)  {                                      ' ]);
fprintf(fid,'%s \n',['     int n;                                                                            ' ]);
fprintf(fid,'%s \n',['     for (n=0; n<filter_length; n++) {                                                ' ]);
fprintf(fid,'%s \n',['                    coef[n] = taps[n]+0x80;                                            ' ]);
fprintf(fid,'%s \n',['     }                                                                                 ' ]);
fprintf(fid,'%s \n',['}                                                                                     ' ]);
end


% SINGLE BUFFERD: Non UNROLLED LOOPS.....................................................
if (type==3)

fprintf(fid,'%s \n',['// PIC 18F452 CODE FOR FIR FILTER GENERATION                                         ']);
fprintf(fid,'%s \n',['// Date: ' Date ' , Time (Hr:Min:Sec)-> ' num2str(time(4)) ':' num2str(time(5)) ':'
num2str(time(6))                                                                                      ]);
fprintf(fid,'%s \n',['// FIR Filter Type:                                                                  ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n',['#include <18f452.h>                                                                  ']);
fprintf(fid,'%s \n',['#use delay(clock = 40000000)                                                         ']);
fprintf(fid,'%s \n',['#fuses H4,PUT,NOWDT                                                                  ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n', ['const int filter_length = ' num2str(length(g)) ';                                   ']);
fprintf(fid,'%s', ['const int taps[filter_length] = {                                                     ']);
for n=1:1:length(g)
             fprintf(fid,'%i',g(n));
   if n<length(g)
      fprintf(fid,',');
   end
end
fprintf(fid,'%s \n',['};                                                                                   ']);
fprintf(fid,'%s \n',['// PIC 18F452 Register MAP...................................................................//   ']);
fprintf(fid,'%s \n',['                                                                                     ']);
fprintf(fid,'%s \n',['// ACCUMULATOR ADDRESS                                                                    ']);
fprintf(fid,'%s \n',['#byte    WREG = 0xFE8              // Register Stores the Carry Bit                     ']);
fprintf(fid,'%s \n',['#byte    PRODL =0xff3              // Product Low Byte                                  ']);
fprintf(fid,'%s \n',['#byte    PRODH =0xff4              // Product High Byte                                 ']);
fprintf(fid,'%s \n',['#byte   ADRESL = 0xfc3             // Low Byte for ADC Sample                           ']);
fprintf(fid,'%s \n',['#byte   ADRESH = 0xfc4             // High Byte for ADC Sample                          ']);
fprintf(fid,'%s \n',['#byte   STATUS = 0xfd8             // Status Register                                   ']);
fprintf(fid,'%s \n',['                                                                                          ']);
fprintf(fid,'%s \n',['// DC CONTROL REGISTERS                                                                   ']);
fprintf(fid,'%s \n',['#byte   ADCON0 = 0xfc2             // ADC Control Register (High)                       ']);
fprintf(fid,'%s \n',['#byte   ADCON1 = 0xfc1             // ADC Control Register (Low)                        ']);
fprintf(fid,'%s \n',['#byte   ADRESL = 0xfc3             // Low Byte for ADC Sample                           ']);
fprintf(fid,'%s \n',['#byte   ADRESH = 0xfc4             // High Byte for ADC Sample                          ']);
fprintf(fid,'%s \n',['                                                                                          ']);
fprintf(fid,'%s \n',['                                                                                          ']);
fprintf(fid,'%s \n',['// DIGITAL IO PORT ADDRESSES                                                              ']);
fprintf(fid,'%s \n',['#byte   PORTA = 0xf80          // Port A Address                                        ']);
fprintf(fid,'%s \n',['#byte   PORTB = 0xf81          // Port B Address                                        ']);
fprintf(fid,'%s \n',['#byte   PORTC = 0xf82          // Port C Address                                        ']);
fprintf(fid,'%s \n',['#byte   PORTD = 0xf83          // Port D Address                                        ']);
fprintf(fid,'%s \n',['#byte   PORTE = 0xf84          // Port E Address                                        ']);
fprintf(fid,'%s \n',['#byte    LATA = 0xf89          // Set Driection for PORTA                               ']);
fprintf(fid,'%s \n' [‘                                                                                          ']);
fprintf(fid,'%s \n',['                                                                                          ']);
fprintf(fid,'%s \n',['//  INDIRECT ADDRESSING                                                                   ']);
fprintf(fid,'%s \n',['#byte   FSR0H = 0xfeA          // Hardware File Pointer0 (High)                         ']);
fprintf(fid,'%s \n',['#byte   FSR0L = 0xfe9          // Hardware File Pointer0 (Low)                          ']);
fprintf(fid,'%s \n',['#byte   FSR1H = 0xfe2          // Hardware File Pointer1 (High)                         ']);
fprintf(fid,'%s \n',['#byte   FSR1L = 0xfe1          // Hardware File Pointer1 (Low)                          ']);
fprintf(fid,'%s \n',['#byte   FSR2H = 0xfda          // Hardware File Pointer2 (High)                         ']);
fprintf(fid,'%s \n',['#byte   FSR2L = 0xfd9          // Hardware File Pointer2 (Low)                          ']);
fprintf(fid,'%s \n',['#byte   INDF0 = 0xfef          // Read Data Pointed by FSR0                             ']);
fprintf(fid,'%s \n',['#byte   INDF1 = 0xfe7          // Read Data Pointed by FSR1                             ']);
fprintf(fid,'%s \n',['#byte   INDF2 = 0xfdf          // Read Data Pointed by FSR2                             ']);
fprintf(fid,'%s \n',['#byte   PLUSW0 = 0xfeb         // Add Pointed data to WREG                              ']);
fprintf(fid,'%s \n',['#byte   PLUSW1 = 0xfe3         // Add Pointed data to WREG                              ']);
fprintf(fid,'%s \n',['#byte   PLUSW2 = 0xfdb         // Add Pointed data to WREG                              ']);
fprintf(fid,'%s \n',['#byte   PREINC0 = 0xfec        // Pre-increment pointer0                                ']);
fprintf(fid,'%s \n',['#byte   PREINC1 = 0xfe4        // Pre-increment pointer1                                ']);
fprintf(fid,'%s \n',['#byte   PREINC2 = 0xfdc        // Pre-increment pointer2                                ']);
fprintf(fid,'%s \n',['#byte   POSTINC0 = 0xfee       // Post-Incerement Pointer0                              ‘]);
fprintf(fid,'%s \n',['#byte   POSTDEC0 = 0xfed       // Post-Decrement Pointer0                               ']);
fprintf(fid,'%s \n',['#byte   POSTINC1 = 0xfe6       // Post-Increment Pointer1                               ']);
fprintf(fid,'%s \n',['#byte   POSTDEC1 = 0xfe5       // Post-Decrement Pointer1                               ']);
fprintf(fid,'%s \n',['#byte   POSTINC2 = 0xfde       // Post-Increment Pointer2                               ']);
fprintf(fid,'%s \n',['#byte   POSTDEC2 = 0xfdd       // Post-Decrement Pointer2                               ']);
fprintf(fid,'%s \n',['                                                                                          ']);
fprintf(fid,'%s \n',['                                                                                          ']);
fprintf(fid,'%s \n',['// TIMER REGISTERS                                                                        ']);
fprintf(fid,'%s \n',['#byte PR2 = 0xfcb                                                                         ']);
fprintf(fid,'%s \n',['#byte TMR2 = 0xfcc                                                                        ']);
fprintf(fid,'%s \n',['#byte T2CON = 0xfca                                                                       ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n', ['int buf[filter_length] = {0};          // Store ADC Values ....................\\         ']);
fprintf(fid,'%s \n', ['int coef[filter_length] = {0};         // Store offset Coefficients ...........\\         ']);
fprintf(fid,'%s \n', ['int output_most = 0;     // Most Significant Byte of Output .............\\              ']);
fprintf(fid,'%s \n', ['int output_middle = 0;   // Middle Significant Byte of Output ...........\\              ']);
fprintf(fid,'%s \n', ['int output_least = 0;    // Least Significant Byte of Output ............\\              ']);
fprintf(fid,'%s \n', ['int Xn_high_256=0;    // Most Significant Byte of Xn Summation * 255.....\\              ']);
fprintf(fid,'%s \n', ['int Xn_mid_256=0;     // Mid Significant Byte of Xn Summation * 255......\\              ']);
fprintf(fid,'%s \n', ['int Xn_low_256=0;     // Least Significant Byte of Xn Summation * 255....\\              ']);
fprintf(fid,'%s \n', ['int Xn_high_128=0;    // Most Significant Byte of Xn Summation * 128.....\\              ']);
fprintf(fid,'%s \n', ['int Xn_mid_128=0;     // Mid Significant Byte of Xn Summation * 128......\\             ']);
fprintf(fid,'%s \n', ['int Xn_low_128=0;     // Least Significant Byte of Xn Summation * 128....\\             ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n\n', [' // General Globals                                                                     ']);
fprintf(fid,'%s \n\n', ['int b,EOB,BOB,x,tptr, out, mac_count;                                                  ']);
fprintf(fid,'%s \n\n', [' // FIR Filter Prototypes                                                               ']);
fprintf(fid,'%s \n\n', ['void offset_and_buffer_tap_coefficients(void);                                         ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n', ['#INT_TIMER2                                                                              ']);
fprintf(fid,'%s \n', ['void t2_isr() {                                                                          ']);
fprintf(fid,'%s \n', ['                    T2CON = 0x06;                // Restart Timer                        ']);
fprintf(fid,'%s \n', ['                    ADCON0 = 0x8d;               // Start ADC Conversion                 ']);
fprintf(fid,'%s \n', ['                    while(bit_test(ADCON0,2));   // Wait for Conversion to Complete      ']);
fprintf(fid,'%s \n', ['                                                                                         ']);
fprintf(fid,'%s \n', ['                    b = ADRESH;                  // Read ADC Value                       ']);
fprintf(fid,'%s \n', ['']);
fprintf(fid,'%s \n', ['                    FSR0L = tptr;                                                        ']);
fprintf(fid,'%s \n', ['                    // Subtract the oldest ADC value in buffer from total                ']);
fprintf(fid,'%s \n', ['                                                                                         ']);
fprintf(fid,'%s \n', ['                    #asm                                                                 ']);
fprintf(fid,'%s \n', ['                           movf     INDF0,W                                              ']);
fprintf(fid,'%s \n', ['                           subwf    Xn_mid_256,F                                         ']);
fprintf(fid,'%s \n', ['                           clrf     WREG                                                 ']);
fprintf(fid,'%s \n', ['                           subwfb  Xn_high_256,F                                         ']);
fprintf(fid,'%s \n', ['                                                                                         ']);
fprintf(fid,'%s \n', ['                          // Add the latest ADC value to the buffer                      ']);
fprintf(fid,'%s \n', ['                                                                                         ']);
fprintf(fid,'%s \n', ['                          movf      EOB,0      // Move to W Register                     ']);
fprintf(fid,'%s \n', ['                          cpfseq  FSR0L      // Check if ptr is at EOB                   ']);
fprintf(fid,'%s \n', ['                          bra        neq                                                  ']);
fprintf(fid,'%s \n', ['                          movff   b,INDF0     // ptr has reached EOB: insert value        ']);
fprintf(fid,'%s \n', ['                          movff    BOB,FSR0L  // Reset pointer to begining of Buffer']);
fprintf(fid,'%s \n', ['                          bra         end']);
fprintf(fid,'%s \n', ['                       neq:                                                              ']);
fprintf(fid,'%s \n', ['                          movff   b,POSTDEC0  // Put data in Buffer and advance ptr']);
fprintf(fid,'%s \n', ['                       end:                                                              ']);
fprintf(fid,'%s \n', ['']);
fprintf(fid,'%s \n', ['                          // Add the latest value to ADC to total                        ']);
fprintf(fid,'%s \n', ['                                                                                         ']);
fprintf(fid,'%s \n', ['                          movf    b,0                                                    ']);
fprintf(fid,'%s \n', ['                          addwf   Xn_mid_256,F                                           ']);
fprintf(fid,'%s \n', ['                          clrf    WREG                                                   ']);
fprintf(fid,'%s \n', ['                          addwfc  Xn_high_256,F                                          ']);
fprintf(fid,'%s \n', ['                                                                                         ']);
fprintf(fid,'%s \n', ['                          #endasm                                                        ']);
fprintf(fid,'%s \n', ['                                                                                         ']);
fprintf(fid,'%s \n', ['                                                                                         ']);
fprintf(fid,'%s \n', ['                               // Prepare for MAC cycles.                                ']);
fprintf(fid,'%s \n', ['                               // Set pointer to begining of coeff buffer.               ']);
fprintf(fid,'%s \n', ['                                                                                         ']);
fprintf(fid,'%s \n', ['                               FSR1L = &coef[0];                                         ']);
fprintf(fid,'%s \n', ['                               mac_count = filter_length;                                ']);
fprintf(fid,'%s \n', ['                               output_least = 0;                                         ']);
fprintf(fid,'%s \n', ['                               output_middle = 0;                                        ']);
fprintf(fid,'%s \n', ['                               output_most = 0;                                          ']);
fprintf(fid,'%s \n', ['']);
fprintf(fid,'%s \n', ['                           #asm                                                          ']);
fprintf(fid,'%s \n', ['']);
fprintf(fid,'%s \n', ['                               // (1) Unload data from Buffer: Newest First.             ']);
fprintf(fid,'%s \n', ['                                                                                         ']);
fprintf(fid,'%s \n', ['        mac:                                                                        ']);
fprintf(fid,'%s \n', ['                    movf    BOB,0                    // Move to W Register           ']);
fprintf(fid,'%s \n', ['                    cpfseq  FSR0L                    // Check if ptr is at BOB       ']);
fprintf(fid,'%s \n', ['                    bra     aneq                                                    ']);
fprintf(fid,'%s \n', ['                    movff   EOB,FSR0L   // Pointer is at BOB.. Warp Pointer to EOB   ']);
fprintf(fid,'%s \n', ['                    movff   INDF0,out  // Extract Data                              ']);
fprintf(fid,'%s \n', ['                    bra     aend                                                    ']);
fprintf(fid,'%s \n', ['                    aneq:                                                           ']);
fprintf(fid,'%s \n', ['                    movff   PREINC0,out // Extract Data from Buffer                 ']);
fprintf(fid,'%s \n', ['                    aend:                                                           ']);
fprintf(fid,'%s \n', ['                                                                                    ']);
fprintf(fid,'%s \n', ['                    // (2) Perform MAC cycle.                                       ']);
fprintf(fid,'%s \n', ['                    movf      out,W                                                 ']);
fprintf(fid,'%s \n', ['                    mulwf     POSTINC1                                              ']);
fprintf(fid,'%s \n', ['                    movf      PRODL,W                                               ']);
fprintf(fid,'%s \n', ['                    addwf     output_least                                          ']);
fprintf(fid,'%s \n', ['                    movf      PRODH,W                                               ']);
fprintf(fid,'%s \n', ['                    addwfc    output_middle                                         ']);
fprintf(fid,'%s \n', ['                    clrf      WREG                                                  ']);
fprintf(fid,'%s \n', ['                    addwfc     output_most                                          ']);
fprintf(fid,'%s \n', ['                    decfsz     mac_count                                            ']);
fprintf(fid,'%s \n', ['                    bra        mac                                                  ']);
fprintf(fid,'%s \n', ['                                                                                    ']);
fprintf(fid,'%s \n', ['                    // (3) Compute output.                                          ']);
fprintf(fid,'%s \n', ['                                                                                    ']);
fprintf(fid,'%s \n', ['                    bcf        STATUS,0']);
fprintf(fid,'%s \n', ['                    rrcf       Xn_high_256,W                                        ']);
fprintf(fid,'%s \n', ['                    movwf      Xn_high_128                                          ']);
fprintf(fid,'%s \n', ['                    rrcf       Xn_mid_256,W                                         ']);
fprintf(fid,'%s \n', ['                    movwf      Xn_mid_128                                           ']);
fprintf(fid,'%s \n', ['                    rrcf       Xn_low_256,W                                         ']);
fprintf(fid,'%s \n', ['                    movwf     Xn_low_128                                            ']);
fprintf(fid,'%s \n', ['                    subwf      output_least,F                                       ']);
fprintf(fid,'%s \n', ['                    movf      Xn_mid_128,W                                          ']);
fprintf(fid,'%s \n', ['                    subwfb     output_middle,F                                      ']);
fprintf(fid,'%s \n', ['                    movf      Xn_high_128,W                                         ']);
fprintf(fid,'%s \n', ['                    subwfb     output_most,F                                        ']);
fprintf(fid,'%s \n', ['                                                                                    ']);
fprintf(fid,'%s \n', ['                    #endasm                                                         ']);
fprintf(fid,'%s \n', ['                                                                                    ']);
fprintf(fid,'%s \n', ['                    tptr = FSR0L;                                                   ']);
fprintf(fid,'%s \n', ['                                                                                    ‘]);
fprintf(fid,'%s \n', ['                    // Scale output...........                                      ']);
fprintf(fid,'%s \n', ['']);
fprintf(fid,'%s \n', ['                    #asm                                                            ']);
fprintf(fid,'%s \n', ['                    bcf        STATUS,0                                             ']);
fprintf(fid,'%s \n', ['                    rrcf       output_most,F                                        ']);
fprintf(fid,'%s \n', ['                    rrcf      output_middle,F                                       ']);
fprintf(fid,'%s \n', ['                    rrcf      output_most,F                                         ']);
fprintf(fid,'%s \n', ['                    #endasm                                                         ']);
fprintf(fid,'%s \n', ['                                                                                    ']);
fprintf(fid,'%s \n', ['                    PORTD = output_middle;                                          ']);
fprintf(fid,'%s \n', ['                                                                                    ']);
fprintf(fid,'%s \n', ['                    }                                                               ']);
fprintf(fid,'\n \n');
fprintf(fid,'%s \n', ['        void main() {                                                               ']);
fprintf(fid,'%s \n', ['                                                                                    ']);
fprintf(fid,'%s \n', ['        set_tris_d(0);                                                              ']);
fprintf(fid,'%s \n', ['        x = 0;                                                                      ']);
fprintf(fid,'%s \n', ['                                                                                    ']);
fprintf(fid,'%s \n', ['                                                                                    ']);
fprintf(fid,'%s \n', ['        T2CON = 0x06;                                                               ']);
fprintf(fid,'%s \n', ['           PR2 = 76;                                                            ']);
fprintf(fid,'%s \n', ['                                                                            ']);
fprintf(fid,'%s \n', ['       // Setup ADC for conversion                                         ']);
fprintf(fid,'%s \n', ['          ADCON0 = 0x85;   // Start ADC:                                   ']);
fprintf(fid,'%s \n', ['          ADCON1 = 0x02;   // Right Justified Result, All Analog           ']);
fprintf(fid,'%s \n', ['                                                                            ']);
fprintf(fid,'%s \n', ['                                                                            ']);
fprintf(fid,'%s \n', ['      enable_interrupts(INT_TIMER2);                                       ']);
fprintf(fid,'%s \n', ['          enable_interrupts(GLOBAL);                                       ']);
fprintf(fid,'%s \n', ['                                                                            ']);
fprintf(fid,'%s \n', ['       // Setup ADC Channel 1                                              ']);
fprintf(fid,'%s \n', ['           ADCON0 = 0x89;              // Set ADC Channel 1                ']);
fprintf(fid,'%s \n', ['           delay_us(10);                                                   ']);
fprintf(fid,'%s \n', ['                                                                            ']);
fprintf(fid,'%s \n', ['       // FIR Filter Initializations                                       ']);
fprintf(fid,'%s \n', ['            offset_and_buffer_tap_coefficients();                          ']);
fprintf(fid,'%s \n', [                                                                            '']);
fprintf(fid,'%s \n', ['       // Buffer Stuff                                                     ']);
fprintf(fid,'%s \n', ['                        EOB = &buf[0];                                      ']);
fprintf(fid,'%s \n', ['                        BOB = &buf[filter_length-1];                        ']);
fprintf(fid,'%s \n', ['                        tptr = BOB;                                         ']);
fprintf(fid,'%s \n', ['']);
fprintf(fid,'%s \n', ['   while(1) {                                                              ']);
fprintf(fid,'%s \n', ['         }                                                                 ']);
fprintf(fid,'%s \n', ['                                                                            ']);
fprintf(fid,'%s \n', ['}                                                                           ']);
fprintf(fid,'\n \n \n');
fprintf(fid,'%s \n',['void offset_and_buffer_tap_coefficients(void)  {                           ' ]);
fprintf(fid,'%s \n',['     int n;                                                                ' ]);
fprintf(fid,'%s \n',['     for (n=0; n<filter_length; n++) {                                     ' ]);
fprintf(fid,'%s \n',['                     coef[n] = taps[n]+0x80;                               ' ]);
fprintf(fid,'%s \n',['     }                                                                     ' ]);
fprintf(fid,'%s \n',['}                                                                          ' ]);
end
set(handles.messages, 'String', 'filter.c written to directory');
fclose(fid);

% ------------------------------------------------------------------
function varargout = pfr_Callback(h, eventdata, handles, varargin)
  global han;
  global gdata;
  global g;
  hand = 0;

  % Read Sampling Rate:
  sf = str2num(get(handles.sf,'string'));
  sf = 2*sf;
  set(handles.gccc,'enable','on');

  % Low Pass Filter..........................................................................
  if (gdata.type == 1)
      pbc = str2num(get(handles.pbco,'string'));
      pba = str2num(get(handles.pba,'string'));
      sbc = str2num(get(handles.sbc,'string'));
      sba = str2num(get(handles.sba,'string'));

      % Generate Filter:
      Pass = pbc;
      Stop = sbc;
      Fs = sf;
      Rp = pba;
      Rs = sba;
```

```
    f = [0 Pass Stop Fs/2]/Fs*2;
    m = [1  1  0  0];
    devs = [(10^(Rp/20)-1)/(10^(Rp/20)+1) 10^(-Rs/20)];
    w = [1 1]*max(devs)./devs;
    n = remezord([Pass Stop],[1 0],devs,Fs); order = max(3,n);
    b = remez(order+1,f,m,w); disp(['Taps needed: ',num2str(n)]);
    a = 1;

    % scaled taps
    for n=1:1:length(b)
        g(n) = round(b(n)/max(b)*127);
    end

    msg = ['Number of Taps Needed: ',num2str(length(g))];
    set(handles.messages,'string', [msg, ', Type: ', num2str(gdata.type)]);

    figure;
    [H,W,S] = freqz(b,a,max(2048,nextpow2(5*max(length(b),length(a)))),Fs);


    if ishandle(han)
            delete (han);
    end

    han = freqzplot(H,W,S);

    end



% High Pass Filter..........................................................................
if (gdata.type == 2)

    pbc = str2num(get(handles.pbco,'string'));
    pba = str2num(get(handles.pba,'string'));
    sbc = str2num(get(handles.sbc,'string'));
    sba = str2num(get(handles.sba,'string'));


    Pass = sbc;
    Stop = pbc;
    Fs = sf;
    Rp = pba;
    Rs = sba;

    f = [0 Stop Pass Fs/2]/Fs*2;
    m = [0  0  1  1];
    devs = [(10^(Rp/20)-1)/(10^(Rp/20)+1) 10^(-Rs/20)];
    w = [1 1]*max(devs)./devs;
    n = remezord([Pass Stop],[1 0],devs,Fs); order = max(3,n);

    if isodd(order)
       order = order+1;
    end

    b = remez(order,f,m,w); disp(['Taps needed: ',num2str(n)]);
    a = 1;


    % scale taps
    for n=1:1:length(b)
        g(n) = round(b(n)/max(b)*127);
    end

    msg = ['Number of Taps Needed: ',num2str(length(g))];
    set(handles.messages,'string', [msg, ', Type: ', num2str(gdata.type)]);

    figure;
    figure;
    [H,W,S] = freqz(b,a,max(2048,nextpow2(5*max(length(b),length(a)))),Fs);

     if ishandle(han)
         delete (han);
     end

     han = freqzplot(H,W,S);

end



% band pass filter...................................................................
 if (gdata.type == 3)
    num_of_taps = str2num(get(handles.tapn,'string'));
             a = str2num(get(handles.bpa,'string'));
             b = str2num(get(handles.bpb,'string'));
             c = str2num(get(handles.bpc,'string'));
             d = str2num(get(handles.bpd,'string'));
           pba = str2num(get(handles.pba,'string'));
           sba = str2num(get(handles.sba,'string'));

   % Code to Generate Filter
     Rp = pba;
     Rs = sba;
     Fs = sf;

  f = [0 a b c d Fs/2]/Fs*2;
  m = [0 0 1 1 0 0];
 devs = [(10^(Rp/20)-1)/(10^(Rp/20)+1) 10^(-Rs/20) (10^(Rp/20)-1)/(10^(Rp/20)+1)];
  w = [Rs Rp Rs]*max(devs)./devs;

  n = num_of_taps;
  order = n;

  if isodd(order)
        order = order+1;
  end

  b = remez(order,f,m,w);
  a = 1;

  % scaled taps
  g = round(b/max(b)*127);


  for n=1:1:order
         g(n) = round(b(n)/max(b)*127);
  end

  msg = ['Number of Taps Needed: ',num2str(length(g))];
  set(handles.messages,'string', [msg, ', Type: ', num2str(gdata.type)]);

  figure;
  figure;
```

```
    [H,W,S] = freqz(b,a,max(2048,nextpow2(5*max(length(b),length(a)))),Fs);

    if ishandle(han)
              delete (han);
    end
    han = freqzplot(H,W,S);
    end

  % band stop filter....................................................................

   if (gdata.type == 4)
          num_of_taps = str2num(get(handles.tapn,'string'));
              a = str2num(get(handles.bpa,'string'));
              b = str2num(get(handles.bpb,'string'));
              c = str2num(get(handles.bpc,'string'));
              d = str2num(get(handles.bpd,'string'));
             pba = str2num(get(handles.pba,'string'));
             sba = str2num(get(handles.sba,'string'));

     % Code to Generate Filter
     % bpf....................................................................

       Rp = pba;
       Rs = sba;
       Fs = sf;

      if isodd(num_of_taps)
         num_of_taps = num_of_taps+2;
      end

      f = [0 a b c d Fs/2]/Fs*2;
      m = [1 1 0 0 1 1];
      devs = [(10^(Rp/20)-1)/(10^(Rp/20)+1) 10^(-Rs/20) (10^(Rp/20)-1)/(10^(Rp/20)+1)];
      w = [Rs Rp Rs]*max(devs)./devs;

      n = num_of_taps;
      order = num_of_taps;

      if isodd(order)
         order = order+1;
      end

      b = remez(order,f,m,w);
      a = 1;

      figure;
      figure;
      [H,W,S] = freqz(b,a,max(2048,nextpow2(5*max(length(b),length(a)))),Fs);

      % scale taps
      g = round(b/max(b)*127);

      msg = ['Number of Taps Needed: ',num2str(length(g))];
      set(handles.messages,'string', [msg, ', Type: ', num2str(gdata.type)]);

      if ishandle(han)
        delete (han);
      end

      han = freqzplot(H,W,S);
      end
```

```
 % Custom Filter

if (gdata.type == 6)

        num_of_taps = str2num(get(handles.customtaps,'string'));
             edges = str2num(get(handles.bedges,'string'));
           profile = str2num(get(handles.profile,'string'));
              attn = str2num(get(handles.atten,'string'));
                Fs = sf;
                f = edges;
                m = profile;
                f = f/Fs*2;
                w = attn;
                n = num_of_taps;
                order = num_of_taps;

              if isodd(order)
                  order = order+1;
              end

              b = remez(order+1,f,m,w);
              a = 1;


              figure;
              figure;
             [H,W,S] = freqz(b,a,max(2048,nextpow2(5*max(length(b),length(a)))),Fs);


             g = round(b/max(b)*127);

             msg = ['Number of Taps Needed: ',num2str(length(g))];
             set(handles.messages,'string', [msg, ', Type: ', num2str(gdata.type)]);

             if ishandle(han)
                   delete (han);
             end

             han = freqzplot(H,W,S);


         end

set(handles.ra,'visible','on');
set(handles.rb,'visible','on');
set(handles.rc,'visible','on');

tra = ['Fastest Execution ', num2str(8*length(g)+45), ' cycles, ', num2str(length(g)*2+8), ' Bytes RAM used'  ];
trb = ['Small Program ', num2str(10*length(g)+45), ' cycles, ', num2str(length(g)*2+8), ' Bytes RAM used '];
trc = ['Best Memory ', num2str(22*length(g)+48), ' cycles, ',num2str(length(g)+8), ' Bytes RAM Used '];

set(handles.ra,'string',tra);
set(handles.rb,'string',trb);
set(handles.rc,'string',trc);

function y = isodd(x)

g = x - floor(x);
if (g > 0)
       y = 1;
    else
       y = 0;
```

```
  end

function custom_off()

global handles;

set(handles.cover,'visible','off');
set(handles.tprofile,'visible','off');
set(handles.profile,'visible','off');
set(handles.bedges,'visible','off');
set(handles.text12,'visible','off');
set(handles.customtaps,'visible','off');
set(handles.attnt,'visible','off');
set(handles.atten,'visible','off');

% --------------------------------------------------------------------
function varargout = ra_Callback(h, eventdata, handles, varargin)
global imptype;
set(handles.rb, 'value', 0);
set(handles.rc, 'value', 0);
imptype = 1;

% --------------------------------------------------------------------
function varargout = rb_Callback(h, eventdata, handles, varargin)
global imptype;
set(handles.ra, 'value', 0);
set(handles.rc, 'value', 0);
imptype = 2;

% --------------------------------------------------------------------
function varargout = rc_Callback(h, eventdata, handles, varargin)
global imptype;
set(handles.ra, 'value', 0);
set(handles.rb, 'value', 0);
imptype = 3;

function varargout = sba_Callback(h, eventdata,handles,varargin)
```

# APPENDIX C
## *CODE FOR 4th ORDER Floating-Point LMS Filter*

```
#include <18f452.h>
#use delay(clock = 40000000)
#fuses H4,PUT,NOWDT
#include <lmslib.h>
#include <lmslib.c>
#include <clcd.c>

// Globals..............................................................................\\

const int filter_length = 4;
int buf[filter_length] = {0};      // Store ADC Values ...................\\
int signal,noise,EOB,BOB,tptr,buf_count;
int out,i,outs;

// LMS variables.......................................................................\\
   split_float fout, *fptr;
   split  w0,w1,w2,w3;
   split  n0,n1,n2,n3;
   split  es0,es1,es2,es3;
   split  up0,up1,up2,up3;
   split  s, eta, err, error, * ptr;

#INT_TIMER2
void t2_isr() {
                T2CON = 0x06;                     // Restart Timer
              // Sample channel 0 for noise
                ADCON0 = 0x81;                  // Set ADC Channel 0
                es0.real = 0; es0.frach = 0; es0.fracl = 0; es0.sign = 0;
                es1.real = 0; es1.frach = 0; es1.fracl = 0; es1.sign = 0;
                es2.real = 0; es2.frach = 0; es2.fracl = 0; es2.sign = 0;
                es3.real = 0; es3.frach = 0; es3.fracl = 0; es3.sign = 0;
                up0.real = 0; up0.frach = 0; up0.fracl = 0; up0.sign = 0;
                up1.real = 0; up1.frach = 0; up1.fracl = 0; up1.sign = 0;
                up2.real = 0; up2.frach = 0; up2.fracl = 0; up2.sign = 0;
                up3.real = 0; up3.frach = 0; up3.fracl = 0; up3.sign = 0;
                err.real = 0; err.frach = 0; err.fracl = 0; err.sign = 0;

                delay_us(6);
                ADCON0 = 0x85;                // Start ADC Conversion
                while(bit_test(ADCON0,2));
                noise = ADRESH;               // Read ADC Value
                ADCON0 = 0x89;                // Set ADC Channel 1

             // Buffer Noise Values and convert to floats
                buf_count = filter_length;
                FSR1L = tptr;
                FSR2L = &n0.sign;

                #asm
             // Add the latest ADC value to the buffer
                        movf    EOB,0          // Move to W Register
                        cpfseq  FSR1L          // Check if ptr is at EOB
                        bra     neq
                        movff   noise,INDF1      //ptr has reached EOB: insert value
                        movff   BOB,FSR1L        //Reset pointer to begining of Buffer
                        bra     end
                neq:
                        movff  noise,POSTDEC1 //Put data in Buffer and advance ptr
                end:
                        // Unload ADC Value from Buffer and poppulate n0..nN
                unl:
                        movf    BOB,0               // Move to W Register
                        cpfseq  FSR1L               // Check if ptr is at BOB
                        bra     aneq
```

```
        movff   EOB,FSR1L         // Pointer is at BOB.. Warp Pointer to EOB
        movff   INDF1,out         // Extract Data
        bra     aend
        aneq:
        movff    PREINC1,out     // Extract Data from Buffer
 aend:

// Store popped value into n0
        movlw    0x80
        cpfslt   out             // Skip next inst if (f) < (W)
        bra      n0pos
        movlw    0x7f
        bsf      STATUS,0
        subfwb    out,W          // W-f-B -> W
        clrf      INDF2
        incf      POSTINC2
        movff     WREG,POSTINC2
        movff     WREG,POSTINC2
        clrf      POSTINC2
        bra       dne
n0pos:
        movlw     0x7f
        subwf     out,W                              // f - W -> W
        clrf      POSTINC2
        movff     WREG,POSTINC2
        movff     WREG,POSTINC2
        clrf      POSTINC2
        dne:
                decfsz      buf_count
                bra          unl
#endasm

         tptr = FSR1L;
        // Sample Channel 1 for Signal

        ADCON0 = 0x8d;               // Start ADC Conversion
        while(bit_test(ADCON0,2));
        signal = ADRESH;                // Read ADC Value

        if (signal>=127) {
                                signal = signal-127;
                                s.frach = signal;
                                s.fracl = signal;
                                s.real = 0;
                                s.sign = 0;
                        }
                       else {
                                signal = 128-signal;
                                s.frach = signal;
                                s.fracl = signal;
                                s.real = 0;
                                s.sign = 1;
                        }



           // Calculate estimate using...... //' esN = wN * nN;'

        FSR0L = &w0.sign; FSR1L = &n0.sign; FSR2L = &es0.sign;  mul();
        FSR0L = &w1.sign; FSR1L = &n1.sign; FSR2L = &es1.sign;  mul();
        FSR0L = &w2.sign; FSR1L = &n2.sign; FSR2L = &es2.sign;  mul();
        FSR0L = &w3.sign; FSR1L = &n3.sign; FSR2L = &es3.sign;  mul();
```

```
// Change Sign of Estimates......//'es0 = -es0;'
                        es0.sign ^= 1;
                        es1.sign ^= 1;
                        es2.sign ^= 1;
                        es3.sign ^= 1;


// Calculate Error..............//' error = s + es0..esN;'
        FSR0L = &s.sign;   FSR1L = &es0.sign; FSR2L = &error.sign;  add();
        FSR0L = &error.sign; FSR1L = &es1.sign; FSR2L = &error.sign;  add();
        FSR0L = &error.sign; FSR1L = &es2.sign; FSR2L = &error.sign;  add();
        FSR0L = &error.sign; FSR1L = &es3.sign; FSR2L = &error.sign;  add();

// Modulate Error using learning constant.......//' err = error*eta;'
        FSR0L = &error.sign; FSR1L = &eta.sign; FSR2L = &err.sign;  mul();

 // Calculate Weight Updates.......//' up0 = err * n0..nN;'
        FSR0L = &err.sign; FSR1L = &n0.sign; FSR2L = &up0.sign;  mul();
        FSR0L = &err.sign; FSR1L = &n1.sign; FSR2L = &up1.sign;  mul();
        FSR0L = &err.sign; FSR1L = &n2.sign; FSR2L = &up2.sign;  mul();
        FSR0L = &err.sign; FSR1L = &n3.sign; FSR2L = &up3.sign;  mul();


// Apply updates to weights.......//' wN = wN + upN;'
        FSR0L = &w0.sign; FSR1L = &up0.sign; FSR2L = &w0.sign;  add();
        FSR0L = &w1.sign; FSR1L = &up1.sign; FSR2L = &w1.sign;  add();
        FSR0L = &w2.sign; FSR1L = &up2.sign; FSR2L = &w2.sign;  add();
        FSR0L = &w3.sign; FSR1L = &up3.sign; FSR2L = &w3.sign;  add();

// Change Sign of Estimates......//'es0 = -es0;'

        if(error.frach > 80) error.frach = 80;

                        if(error.sign)
                                  outs = 127 - error.frach;
                        else
                                  outs = error.frach + 127;

                        PORTD = outs;
}



void main() {

// Setup Ports and Peripherals

        set_tris_d(0);
        lcd_init();

        // Set sampling rate of 8000 Hz

          T2CON = 0x06;
          PR2 = 76;

        // Setup ADC for conversion

           ADCON0 = 0x85;   // Start ADC:
           ADCON1 = 0x02;   // Right Justified Result, All Analog

        // Enable Timer interrupts for sampling.

          enable_interrupts(INT_TIMER2);
          enable_interrupts(GLOBAL);
```

```
                // Initialize LMS variables

                 ptr = &w0; fix8x16(0.0,ptr);
                 ptr = &w1; fix8x16(0.0,ptr);
                 ptr = &w2; fix8x16(0.0,ptr);
                 ptr = &w3; fix8x16(0.0,ptr);

                 ptr = &es0; fix8x16(0.0,ptr);
                 ptr = &es1; fix8x16(0.0,ptr);
                 ptr = &es2; fix8x16(0.0,ptr);
                 ptr = &es3; fix8x16(0.0,ptr);

                 ptr = &up0; fix8x16(0.0,ptr);
                 ptr = &up1; fix8x16(0.0,ptr);
                 ptr = &up2; fix8x16(0.0,ptr);
                 ptr = &up3; fix8x16(0.0,ptr);

                 ptr = &err; fix8x16(0.0,ptr);
                 ptr = &eta; fix8x16(0.1,ptr);
                 ptr = &error; fix8x16(0.0,ptr);

        // Initialize buffer pointers for LMS

               EOB = &buf[0];
                 BOB = &buf[filter_length-1];
                 tptr = BOB;

   while(1) {

                   }

}
```

**FileName: lmslib.h**

```
// ACCUMULATOR ADDRESS.............................................         //
#byte    WREG = 0xFE8           // Register Stores the Carry Bit            //
#byte    PRODL =0xff3    // Product Low Byte                                //
#byte    PRODH =0xff4           // Product High Byte                        //
#byte   ADRESL = 0xfc3          // Low Byte for ADC Sample                  //
#byte   ADRESH = 0xfc4          // High Byte for ADC Sample                 //
#byte   STATUS = 0xfd8          // Status Register                          //

// ADC CONTROL REGISTERS..........................................          //
#byte   ADCON0 = 0xfc2          // ADC Control Register (High)              //
#byte   ADCON1 = 0xfc1          // ADC Control Register (Low)               //
#byte   ADRESL = 0xfc3          // Low Byte for ADC Sample                  //
#byte   ADRESH = 0xfc4          // High Byte for ADC Sample                  //
#byte  INTCON  = 0xff2          // Interrupt control register               //
#byte  INTCON2 = 0xff1          // Interrupt control register               //
#byte  INTCON3 = 0xff0          // Interrupt control register               //

// DIGITAL IO PORT ADDRESSES ......................................         //
#byte   PORTA = 0xf80     // Port A Address                                  //
#byte   PORTB = 0xf81     // Port B Address                                  //
#byte   PORTC = 0xf82     // Port C Address                                  //
```

```
#byte   PORTD = 0xf83     // Port D Address                              //
#byte   PORTE = 0xf84     // Port E Address                              //
#byte    LATA = 0xf89      // Set Driection for PORTA                    //
#byte    LATB = 0xf8a     // Set Driection for PORTB                     //
#byte    LATC = 0xf8b     // Set Driection for PORTC                     //
#byte    LATD = 0xf8c      // Set Driection for PORTD                    //
#byte    LATE = 0xf8d      // Set Driection for PORTE                    //


// INDIRECT ADDRESSING............................................      //
#byte   FSR0H = 0xfeA     // Hardware File Pointer0 (High)              //
#byte   FSR0L = 0xfe9     // Hardware File Pointer0 (Low)               //
#byte   FSR1H = 0xfe2     // Hardware File Pointer1 (High)              //
#byte   FSR1L = 0xfe1      // Hardware File Pointer1 (Low)              //
#byte   FSR2H = 0xfda     // Hardware File Pointer2 (High)              //
#byte   FSR2L = 0xfd9     // Hardware File Pointer2 (Low)               //
#byte   INDF0 = 0xfef     // Read Data Pointed by FSR0                  //
#byte   INDF1 = 0xfe7     // Read Data Pointed by FSR1                  //
#byte   INDF2 = 0xfdf      // Read Data Pointed by FSR2                 //
#byte   PLUSW0 = 0xfeb    // Add Pointed data to WREG                   //
#byte   PLUSW1 = 0xfe3    // Add Pointed data to WREG                    //
#byte   PLUSW2 = 0xfdb    // Add Pointed data to WREG                    //
#byte   PREINC0 = 0xfec   // Pre-increment pointer0                      //
#byte   PREINC1 = 0xfe4   // Pre-increment pointer1                      //
#byte   PREINC2 = 0xfdc    // Pre-increment pointer2                     //
#byte   POSTINC0 = 0xfee    // Post-Incerement Pointer0                  //
#byte   POSTDEC0 = 0xfed    // Post-Decrement Pointer0                   //
#byte   POSTINC1 = 0xfe6    // Post-Increment Pointer1                   //
#byte   POSTDEC1 = 0xfe5    // Post-Decrement Pointer1                   //
#byte   POSTINC2 = 0xfde    // Post-Increment Pointer2                   //
#byte   POSTDEC2 = 0xfdd    // Post-Decrement Pointer2                   //

// INTERRUPT REGISTERS............................................       //
#byte   INTCON = 0xff2            // Interrupt Register0                 //
#byte   INTCON2 = 0xff1     // Interrupt Register2                       //
#byte   INTCON3 = 0xff0     // Interrupt Register3                       //

// STACK ADDRESSES................................................       //
#byte   STKPTR = 0xffc            //  Stack Pointer                      //
#byte     TOSU = 0xfff     //  Top of Stack                              //
#byte     TOSH = 0xffe      //  Top of Stack High                        //
#byte     TOSL = 0xffd      //  Top of Stack Low                         //

// EEPROM ADDRESSES...............................................      //
#byte    EEADR = 0xfA9      // EEPROM Register                          //
#byte   EEDATA = 0xfa8      // EEPROM Register                          //
#byte   EECON2 = 0xfa7            // EEPROM Register                    //
#byte   EECON1 = 0xfa6      // EEPROM Register                          //

// TIMER REGISTERS................................................      //
#byte PR2 = 0xfcb
#byte TMR2 = 0xfcc
#byte T2CON = 0xfca
```

**FileName: lmslib.c**

```
typedef struct gtype {
                 int sign;
                       int fracl;
                       int frach;
                       int real;
                   } split;
```

```
typedef union ftype { float op;
                  struct { int exp;
                           int mana;
                           int manb;
                           int manc; } s; } split_float;



void int2float(int adc) {

        #asm
           movlw   0x7f
           subwf   adc,W
           btfsc   WREG,7
           bra     ng
           clrf    POSTINC0
           movwf   POSTINC0
           movwf   INDF0
           bra     over
         ng:
           negf    WREG
           clrf    INDF0
           incf    POSTINC0
           movwf   POSTINC0
           movwf   INDF0
      over:
        #endasm
}


void fixIeee(split_float * fptr, split * ptr) {

int sign,real,frach,fracl,expo;

  sign = ptr->sign;
  real = ptr->real;
 frach = ptr->frach;
 fracl = ptr->fracl;
  expo = 0x86;


        #asm
           bsf     fracl,0
         adj:
           btfsc   real,7
           bra     done
           bcf     STATUS,0
           rlcf    fracl
           rlcf    frach
           rlcf    real
           decf    expo
           bra     adj
       done:
        #endasm

           fptr->s.mana = real;
           fptr->s.manb = frach;
           fptr->s.manc = fracl;
           fptr->s.exp = expo;

   if (sign) {
                fptr->s.mana = fptr->s.mana | 0x80;
            }
```

```
                    else
                {
                    fptr->s.mana = fptr->s.mana & 0x7f;
                }
}

void fix8x16(float num, split * ptr) {

split_float a;
int shift, left;
int32 out;


a.op = num;

ptr->real = 0;
ptr->sign = 0;
ptr->frach = 0;
ptr->fracl = 0;



if (a.s.exp >= 0x7f) {
                           shift = a.s.exp - 0x7f;
                            left = 1;
                          }
 else {
            shift = 0x7f - a.s.exp;
             left = 0;
 }

 // Get Sign and Restore high Bit.

 ptr->sign = bit_test(a.s.mana,7);
 a.s.mana = a.s.mana | 0x80;


 if (left) {
                    ptr->real = a.s.mana >> (7 - shift);
                    #asm
                       movlw  0x02
                       addwf  shift,F
                     adj:
                        decf    shift
                        bcf     STATUS,0
                        bz      fin
                        rlcf    a.s.manc
                        rlcf    a.s.manb
                        rlcf    a.s.mana
                        bcf     STATUS,0
                        bra      adj
                      fin:
                    #endasm

                    ptr->frach = a.s.mana;
                    ptr->fracl = a.s.manb;
               }

                else {
                        #asm
                            adja:
                              decf   shift
                              bcf    STATUS,0
                              bz     over
                    rrcf   a.s.mana
                    rrcf   a.s.manb
                    rrcf   a.s.manc

                    bcf    STATUS,0
                    bra    adja
                  over:
                #endasm

              ptr->frach = a.s.mana;
              ptr->fracl = a.s.manb;

              }

}



void add(void) {

        #asm
                movf    INDF0,W
                xorwf   INDF1,W
                bnz     ds

              ss:
                movf    POSTINC0,W
                andwf   POSTINC1,W
                movwf   POSTINC2
                movf    POSTINC0,W
                addwf   POSTINC1,W
                movwf    POSTINC2

                movf     POSTINC0,W
                addwfc  POSTINC1,W
                movwf   POSTINC2

                movf      INDF0,W
                addwfc   INDF1,W
                movwf    INDF2
                bra        done

             ds:
                 movlw   0x3
                 addwf    FSR0L,F
                 addwf    FSR1L,F

                 movf     INDF0,W
                 cpfseq   INDF1
                 bra        rneq
                 bra        requ

             rneq:
                 cpfsgt    INDF1
                 bra        ah

             bh:
                  movlw   0x3
                  subwf    FSR0L
                  subwf    FSR1L

                  movff    POSTINC1,POSTINC2
                  incf       FSR0L
```

```
  movf     POSTINC0,W
  subwf    POSTINC1,W
  movwf    POSTINC2

  movf     POSTINC0,W
  subwfb   POSTINC1,W
  movwf    POSTINC2

  movf     INDF0,W
  subwfb   INDF1,W
  movwf    INDF2
  bra      done

ah:
   movlw    0x3
   subwf    FSR0L
   subwf    FSR1L

   movff    POSTINC0,POSTINC2
   incf     FSR1L

   movf     POSTINC1,W
   subwf    POSTINC0,W
   movwf    POSTINC2

   movf     POSTINC1,W
   subwfb   POSTINC0,W
   movwf    POSTINC2

   movf     INDF1,W
   subwfb   INDF0,W
   movwf    INDF2
   bra      done

 requ:
   decf     FSR0L
   decf     FSR1L
   movf     INDF0,W
   cpfseq   INDF1
   bra      fhneq
   bra      fhequ

 fhneq:
   cpfsgt   INDF1
   bra      afh

 bfh:
    movlw    0x02
    subwf    FSR0L
    subwf    FSR1L

    movff    POSTINC1,POSTINC2
    incf     FSR0L

    movf     POSTINC0,W
    subwf    POSTINC1,W
    movwf    POSTINC2

    movf     POSTINC0,W
    subwfb   POSTINC1,W
    movwf    POSTINC2

    clrf     INDF2
   bra      done

afh:
   movlw    0x02
   subwf    FSR0L
   subwf    FSR1L

   movff    POSTINC0,POSTINC2
   incf     FSR1L

   movf     POSTINC1,W
   subwf    POSTINC0,W
   movwf    POSTINC2

   movf     POSTINC1,W
   subwfb   POSTINC0,W
   movwf    POSTINC2

   clrf     INDF2
   bra      done

fhequ:
   decf     FSR0L
   decf     FSR1L
   movf     INDF0,W
   cpfseq   INDF1
   bra      flneq
   bra      flequ

flneq:
   cpfsgt   INDF1
   bra      afl

bfl:
   decf     FSR0L
   decf     FSR1L

   movff    POSTINC1,POSTINC2
   incf     FSR0L

   movf     POSTINC0,W
   subwf    POSTINC1,W
   movwf    POSTINC2

   clrf     POSTINC2
   clrf     POSTINC2
   bra      done

afl:
   decf     FSR0L
   decf     FSR1L

   movff    POSTINC0,POSTINC2
   incf     FSR1L

   movf     POSTINC1,W
   subwf    POSTINC0,W
   movwf    POSTINC2
   bra      done

flequ:
   clrf     POSTINC2
   clrf     POSTINC2
   clrf     POSTINC2
```

```
            clrf    POSTINC2
           done:
        #endasm
    }




    void mul(void) {

        #asm
            movf      POSTINC0,W
            xorwf     POSTINC1,W
            movwf     POSTINC2

            movf     PREINC0,W
            mulwf    PREINC1
            movff    PRODL, POSTINC2
            movff    PRODH, INDF2

            movf     PREINC0,W
            mulwf    POSTINC1
            movff    PRODL,WREG
            addwf    POSTINC2,F
            movff    PRODH,WREG
            addwfc   POSTDEC2,F

            decf     FSR0L,F
            movf     POSTINC0,W
            mulwf    INDF1
            movff    PRODL,WREG
            addwf    POSTINC2,F
            movff    PRODH,WREG
            addwfc   INDF2,F
            movff    INDF0,WREG
            mulwf    INDF1
            movf     PRODL,W
            addwfc   INDF2,F
    #endasm
}
```

# APPENDIX D

## *C-Code for the Clock Signal to the Switched Cap filter*

```
// The following code generates a 50000Hz Clock signal of 55555 Hz
// Allowing the Switched capasitor MAX 297 to have a cutoff of
//1KHz.


#include <12f629.h>
#use delay(clock = 10000000)

#fuses HS,PUT,NOWDT
#define GP0 PIN_A0
#define GP1 PIN_A1
#define GP2 PIN_A2
#define GP3 PIN_A3
#define GP4 PIN_A4
#define GP5 PIN_A5

void main() {

while(1) {
        output_high(PIN_A2);
        delay_us(9);
        output_low(PIN_A2);
        delay_us(9);
}

}
```

## APPENDIX E

*C-Code for the PORTC HD44780 LCD DEVICE*

```
struct lcd_pin_map {                // This structure is overlayed
         boolean rs;                // on to an I/O port to gain
         boolean unused1;           // access to the LCD pins.
         boolean unused2;           //
         boolean enable;            //
         int     data : 4;          //
       } lcd;

#byte lcd = 0xf82                  // This puts the entire structure
                                   // on to port C (at address 7)

byte CONST LCD_INIT_STRING[4] = {0x28, 0xc, 1, 6};
byte CONST LCD_LINE_ADDRESSES[4] = {0x00, 0x40, 0x14, 0x54};


// Sends a single nibble to the LCD.

void lcd_send_nibble( byte n ) {
     lcd.data = n;
     delay_cycles(1);
     lcd.enable = 1;
     delay_us(2);
     lcd.enable = 0;
}

// Sends a whole byte to the LCD by making use of the Send nibble function
// The first parameter 'address' decided whether the byte is an instruction or data

void lcd_send_byte( byte address, byte n ) {
     delay_ms(3);
     lcd.rs = 0;
     delay_us(1);
     lcd.rs = address;
     delay_cycles(1);
     lcd.enable = 0;
     lcd_send_nibble(n >> 4);
     lcd_send_nibble(n & 0xf);
}


// Initializes the LCD display..

void lcd_init() {
   byte i;
   set_tris_c(0);
   lcd.rs = 0;
   lcd.enable = 0;
   delay_ms(15);
   for(i=1;i<=3;++i) {
      lcd_send_nibble(3);
      delay_ms(5);
   }
   lcd_send_nibble(2);
   for(i=0;i<=3;++i)
      lcd_send_byte(0,LCD_INIT_STRING[i]);
}
```

```
// Sets the cursor on the screen where the character is to be printed.

void lcd_gotoxy( byte x, byte y) {
   byte address;
   address=lcd_line_addresses[y]+x;
   lcd_send_byte(0,0x80|address);
}


void lcd_putc( byte c) {
   switch (c) {
     case '\f'   : lcd_send_byte(0,1);
                   delay_ms(2);
                                       break;
     case '\b'   : lcd_send_byte(0,0x10);  break;
     default     : lcd_send_byte(1,c);     break;
   }
}
```

# VITA

I was born in Lucknow, India and I spent most of my childhood in New Delhi. I received my undergraduate in Electrical Engineering from West Virginia University and continued on to finish my masters degree from here as well. During my graduate years, I have held both GTA and GRA positions and spend my time doing either research or teaching. My research interests, inherited from my teacher Dr. Klinkhachorn, are Neural Networks, Fuzzy Logic, Digital Filter design and implementation, and embedded control.