School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

Implementation of the LMS Algorithm for Noise Cancellation on Speech Using the ARM LPC2378 Processor.

Cesar Augusto Azurdia Meza
Yaqub Jon Mohamadi

September
2009

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 09053
ISSN 1650-2647
ISRN VXU/MSI/ED/E/--09053/--SE

Cesar Augusto Azurdia Meza

Yaqub Jon Mohamadi

Implementation of the LMS Algorithm For Noise Cancellation on Speech Using the ARM LPC2378 Processor

Master Thesis

School of Mathematics and Systems Engineering

2009

Växjö University

Abstract

On this thesis project, the LMS algorithm has been applied for speech noise filtering and different behaviors were tested under different circumstances by using Matlab simulations and the LPC2378 ARM Processor, which does the task of filtering in real time. The thesis project is divided into two parts: the theoretical and practical part.

In the theoretical part there is a brief description of the different aspects of signal processing systems, filter theory, and a general description of the Least-Mean-Square Adaptive Filter Algorithm.

In the practical part of the report a general description of the procedure will be summarized, the results of the tests that were conducted will be analyzed, a general discussion of the problems that were encounter during the simulations will be mention, and suggestion for the problems will be given.

Acknowledgements

The development of the project started on May 2009 and lasted until August 2009. The presentation seminar was held at the end of August 2009. We hereby are thankful with our supervisor Sven Nordebo, and also with Göran Ewing for his assistance during the project. Finally, we would like to thank our families for their supports along the process of this thesis.

Contents

I Theoretical Part

The following five chapters are a general description of the basic theory needed to understand and develop the practical task of the thesis project. The written definitions and the mathematical expressions have been compiled from a series of specialized books in the subject. The theoretical concepts that will be discussed in the report are the following:

- **I.** Signal Processing Systems
- **II.** Digitization of Analog Signals
- **III.** Filter Theory
- **IV.** Least-Mean-Square Adaptive Filters
- **V.** Noise in Speech Signals

The specialized literature implemented for compiling the basic theory is the following:

S. Haykin, *Adaptive Filter Theory.* Fourth Edition.
Upper Saddle River, NJ, USA: Prentice Hall 2002.

Douglas E. Comer, *Computer Networks and Internets.* Fifth Edition.
Upper Saddle River, NJ, USA: Prentice Hall 2009.

C. Marven and Gillian Ewers, *A Simple Approach to Digital Signal Processing.*
New York, NY, USA: Wiley-Interscience Publication 1996.

H. Baher, *Analog and Digital Signal Processing.*
Chichester, United Kindom: John Wiley and Sons 1990.

Sven Nordebo. *Signal Processing Antennas.*
September 18, 2004. http://w3.msi.vxu.se/~sno/ED4024/ED4024.pdf

Chapter 1 Signal Processing Systems

A signal is a physical number or feature that conveys information. The human brain is said to be frequently sending signals to various parts of the body, which cause it to carry out certain actions or react in a meticulous way. Consequently, the brain signal is call an *excitation* and the act performed by the specific organ is called a *response*. The external physical world has several examples of signals. Examples of man-made electrical signals include wireless transmission of television and radio, radio-astronomical signals, signals used for the communication of information by satellites orbiting the earth, sonar signals for underwater acoustics. Transmission of signals connecting two points can also be achieved using cables or optical fibers, as in the case of telephone lines. Today, signals transmission among distant points may be performed using a mixture of cable and wireless techniques.

Signals may have different non-electrical origins, such as those generated by natural, acoustic, or mechanical sources. It is still possible to model the physical amount by way of an electric signal such as a voltage or a current. For the explanation and manipulation of signals, it is important to use some techniques for the correct representation of these signals. A signal *f(x)* can be defined as a function of x which varies in a certain way so as to communicate some information. In most of the cases, the independent variable *x* is real time *t*, and one defines a signal as a function of time which carries information regarding an extent of interest. On the other hand, the variable *x* might be any type of variable (complex or real) different than time.

1.1 Analog Signals

The ordinary world we live in produces signals that are analog. It means that the signal *f(t)* is defined for all values within the continuous variable *t*, and its amplitude can be any value of a continuous range. Such a signal is referred to as an *analog signal.* Sometimes one can use the terms *continuous-time*, or simply *continuous*, to describe analog signals. Examples of analog signals that one uses in every day life include sound, temperature, pressure, voltage, current and light.

1.2 Discrete-Time and Digital Signals

In contrast to analog signals, other signals which are mostly made by humans are defined only for *discrete* values of time *t*. The independent variable *t* takes discrete values which are usually integral multiples of the same basic quantity T, therefore one can write it in the following notation:

$$f(nT) \cong f(t) \quad n = 0, \pm 1, \pm 2, \ldots \quad \textbf{(1)}$$

Such a signal is called a *discrete-time* signal, or just *discrete* signal. It might had been acquired by taking samples of an analog signal at regular intervals of time *nT(n* = 0, ± 1, ± 2,...). Therefore, this type of signal is also identified as an *analog sampled-data* signal.

If additionally to being an analog sampled-data type, the amplitudes of the signal can assume only discrete values which are integral multiples of the same quantity *q*, and each amplitude value is represented by a *code* (hex or binary number), then the resulting signal is called a digital signal. Such a signal is suitable for manipulation using a digital computer or microprocessor (DSP microcontroller). It is due to the advances in digital computers that digital signals are at the moment commonly used in many applications which were previously implemented by analog techniques.

1.3 Deterministic and Random Variables

A signal can either be considered random or deterministic. Random signals are the ones which can be described by a mathematical expression (function) or by using an arbitrary graph. On the other hand, random signals cannot be described by such methods. As an alternative, one implements statistical methods for their description. The random nature of a signal is due to undesirable additive noise. The noise might be introduced into the system when the signal is transmitted over noisy channels. At the moment, a great deal of effort in the areas of signal transmission and detection is devoted to the problem of extracting the information contained in a signal contaminated by random noise. This concept applies to both analog and digital signals.

1.4 Processing Systems

Signals of all types are transmitted from one point to another for the purpose of transmitting information. In order to perfect the method of revision, transmission and detection of the required information, a very big amount of techniques have been accumulated over the last years. These techniques need systems for the manipulation of signals. Such manipulations consist of rather simple operations such as amplification, differentiation, integration, addition and multiplication as a more complex operation. Additionally, if the signal is in analog form, it has to be digitized before it can be manipulated by a digital processor.

1.4.1 Analog Systems

An analog system is a mechanism of defined building blocks that accepts an analog *excitation signal f(t)* and produces an analog *response signal g(t)*. Both signals *f(t)* and *g(t)* are related by the system itself. The transfer function of the system determines the relationship between the *excitation signal* and the *response signal*. The most commonly used analog system is the *electrical network* made up of some or all of the conventional elements: resistors, capacitors, inductors, transformers, transistors, diodes and operational amplifiers. Depending on the structure that one implements, the *response signal g(t)* will differ from the *excitation f(t)* according to the expression of the *transfer function h(t)*.

A very important aspect to mention regarding Analog Systems consists on the concept of *linear time-invariant* systems. The expression linearity means that if an excitation $f_1(t)$ produces a response $g_1(t)$ and an excitation $f_2(t)$ produces a response $g_2(t)$, then the response g(t) due to the excitation

$$f(t)=af_1(t) + bf_2(t) \quad \textbf{(2)}$$

is given by

$$g(t)=ag_1(t) + bg_2(t) \quad \textbf{(3)}$$

Where a and b are arbitrary constants. In other words, it can be stated that if g(t) is the response of the system to an excitation f(t), in that case the response to a delayed excitation $f(t-\tau)$ is $g(t-\tau)$, therefore the response is delayed by the same time value τ.

An additional property of physical systems that holds true if we consider the signals as functions of real time t is *causality*. This means that the system is non-anticipatory (the system can not predict it), for example if

$$f(t)=0 \text{ for } t<\tau \quad \textbf{(4)}$$

then

$$g(t)=0 \text{ for } t<\tau \quad \textbf{(5)}$$

The response must be zero for all t prior to applying the excitation.

There are straightforward operations in signal processing which are performed by some idealized essential building blocks. The elementary systems performing the operations are defined as follows:

1. Multiplier (proportional system)
2. Differentiator
3. Integrator
4. Adder

From the above systems more elaborate signal processing systems can be constructed.

1.4.2 Discrete and Digital Systems

A digital system is one that accepts a discrete (or digital) excitation *f(nT)* that produces a discrete (or digital) response *g(nT)*. The two signals *f(nT)* and *g(nT)* are related in a behavior determined by the systems transfer function h(nT). A representation of discrete systems can be expressed using the following expression

$$g(nT) = f(nT)h(nT) \quad \textbf{(6)}$$

The transfer function of the digital system can be expressed using the following expression

$$h(nT) = g(nT) \div f(nT) \quad \textbf{(7)}$$

A discrete system doesn't need to be digital since there are various categories of systems in the *sampled-data mode,* although being analog in nature. These systems do not employ digital hardware or software. These systems accept analog signals directly, but the operations performed by the system are carried out on samples of signals. Therefore, these systems are of the *analog sample-data type.* As building blocks, they employ operational amplifiers, capacitors and analog switches.

On the other hand, a discrete system is known as digital if it operates on discrete-time signals whose amplitudes are quantized and encoded. The signal amplitudes can only assume values which are integral multiples of some basic magnitude, and each value is expressed by a code as a binary or hexadecimal number. The system uses digital hardware in the form of logical circuits or completely by writing a computer program.

The most important group of digital systems are the ones for which the excitation *f(nT)* and the response *g(nT)* are related by a linear differential equation with constant coefficients. It can also be stated that the definitions of time-invariance and causality are basically the same as in the analog case for a digital system.

The basic operations performed on signals by a digital system are addition, multiplication by a constant, and delay. These operations can be implemented either in hardware form or in software.

The recent extensive use of digital signal processing systems is due to many factors including high precision, temperature effects, consistency, reproducibility, reduced cost and improvements done in computational algorithms. Major improvements have been made in analog designs over the last few years. This period has also seen the huge increase in the use of DSP devices and systems. The advantages of digital signal processing are very influential and it is unlikely that there will be any great reappearance of general-purpose analog signal processing in the near future. At the present time, analog circuits are being used widely in partnership with DSP in the same design, mainly to provide some basic pre-processing function.

There is a limitation when DSP are implemented because of bandwidth limitations related to processor's cycle times and algorithm difficulty. Analog circuits can be implemented if there is a need for a higher bandwidth signal.

Chapter 2 Digitization of Analog Signals

A signal *f(t)* is called a *continuous-time* or an *analog* signal if it is defined for all values of time *t.* If f*(t)* is defined only at discrete values of *t*, it is call a *discrete-time* signal or an *analog sampled data* signal. An analog signal needs to be digitalized for processing by a digital system, such as a digital computer or microprocessor. If we initially assume to have an analog signal (continuous-time and continuous-amplitude), the signal needs to be converted into digital form.

The first step in the digitization process is to take samples of the signal *f(t)* at regular time intervals: *nT (n* = 0, ± 1, ± 2,...). This yields to converting the continuous time variable *t* into a discrete one. A signal *f(nT)* is obtained and it is defined only at discrete instants, which are integral multiples of the same quantity T. Such a signal may be thought of as a series of number with sampling period of T as

$$f(nT) \cong f(0), f(\pm T), f(\pm 2T), \ldots.. \quad \textbf{(8)}$$

The discrete-time signal is then *quantized*, as a result the amplitude is changed into a discrete level. From the sequence *f(nT)*, a new quantized sequence *$f_q(nT)$* is created by assigning to each *f(nT)* the value of a quantization level. Finally, the discrete-time quantized sequence *$f_q(nT)$* is encoded, which means that each component of the series is represented by a code: the most commonly used one is a binary code.

In summary it can be stated that the entire process of sampling, quantization and encoding is usually called analog-to-digital (A/D) conversion.

2.1 Ideal Impulse Sampling

It is known that impulses cannot be produced physically because of bandwidth limitations, but the concept will be used for explaining the sampling process. If we let *f(t)* be a continuous-time (analog) signal and one multiplies it by a periodic train of impulses:

$$\delta(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT) \quad \textit{Periodic train of impulses} \quad \textbf{\textit{(9)}}$$

To obtain the signal

$$fs(t) = f(t) \sum_{n=-\infty}^{\infty} \delta(t - nT) \quad \textbf{\textit{(10)}}$$

And we finally obtain the following expression

$$fs(t) = \sum_{n=-\infty}^{\infty} f(nT)\delta(t - nT) \quad \textbf{(11)}$$

Which is another periodic train of impulses, each of multiplied by *f(nT),* which is the value of the function *f(t)* at the *n*th instant. This impulse train is known as the sampled signal. The value of the *nth* impulse is the sample value of the function *f(nT),* and if each impulse is replaced by its amplitude (area), then one obtains a set of numbers *{f(nT)}* defining the discrete-time signal and the sampling process has been achieved.

By considering the influence of the sampling process on the Fourier spectrum of the original continuous-time signal it can be seen that

$$f(t) \leftrightarrow F(w) \quad \textbf{(12)}$$

And if we suppose that F(w) is limited to w_m, where

$$| F(w) | = 0 \qquad | w | \geq w_m \quad \textbf{(13)}$$

Using the frequency convolution relation applied to *f(t)* and δ(t), the Fourier transform of the sampled signal is given by the following expression

$$F[fs(t)] = Fs(w) = \frac{1}{T}\sum_{n=-\infty}^{\infty} F(w - nw_0) \quad \textbf{(14)}$$

With $T = \dfrac{2\pi}{w_0}$ **(15)**

The spectrum of the sampled signal consists of the periodic addition of the spectrum F(w) of the original continuous-time signal. It can seeing that the sampling frequency of the system is

$$w_0 = \frac{2\pi}{T} \quad \textbf{(16)}$$

The *F(w)* signal can be recovered by passing the spectrum $F_s(w)$ through a low-pass filter. In this case, the sampling frequency needs to be at least twice the highest frequency component of the spectrum F(w) of the original signal in theory.

It can be concluded that the minimum sampling rate required to prevent *aliasing* must exceed twice the highest frequency component in the spectrum *F(w)* of *f(t)* before sampling. This minimum sampling rate is called the Nyquist Frequency w_N.

2.2 The Sampling Theorem

Previously we saw that signals in the real world are analog, therefore we need to convert them into digital patterns that can be used by a DSP or any digital computer. The first part of the process is called *sampling.* The samples taken can be described as snapshots of the input signal at a fixed point in time. In other words, sampling an input signal is a method of registering an instantaneous value of the signal.

If we have a signal *f(t)* whose spectrum is band-limited to a frequency below w_m, *it* can be completely recovered from the samples *{f(nT)}* taken at the following rate

$$f_N = \frac{W_N}{2\pi}\left(=\frac{1}{T}\right) \text{ Where } w_N = 2w_N \quad \textbf{(17)}$$

The signal *f(t)* is determined from its sample values *{f(nT)}* by

$$f(t) = \sum_{n=-\infty}^{\infty} f(nT)\frac{\sin W_m(t-nT)}{W_m(t-nT)} \quad \textbf{(18)}$$

$$T = \frac{\pi}{W_m} = \frac{2\pi}{W_N} = \frac{1}{f_N} \quad \textbf{(19)}$$

It is straightforward to see that *F(w)* can be recovered from $F_s(w)$ by passing it through an ideal low-pass filter of amplitude T and cut-off frequency of w_m. As a consequence, and by sampling at twice the highest frequency in *F(w)*, the transfer function of the required filter is obtained from

$$h(t) = F'[H(jw)] \quad \textbf{(20)}$$

Where

$$H(jw)=T \qquad l\, w\, l \le w_m \quad \textbf{(21)}$$
$$H(jw)=0 \qquad l\, w\, l > w_m \quad \textbf{(22)}$$

So,

$$h(t) = T\frac{\sin(w_m t)}{\pi t} = \frac{\sin(w_m t)}{w_m t} \quad \textbf{(23)}$$

And the output of the filter is

$$f(t) = f_s(t) * h(t) = \int_{-\infty}^{\infty}\left(\sum_{n=-\infty}^{\infty} f(nT)\delta(t-nT)\right)\frac{\sin w_m(t-\tau)}{w_m(t-\tau)}d\tau \quad \textbf{(24)}$$

$$f(t) = \sum_{n=-\infty}^{\infty} f(nT)\frac{\sin w_m(t-nT)}{w_m(t-nT)} = \sum_{n=-\infty}^{\infty} f(nT)\frac{\sin w_m(t-nT)}{w_m(t-nT)} \quad \textbf{(25)}$$

The previously expression is fundamentally a formula for the interpolation of the signal values by its values at the sampling points. *F(w)* is recovered from $F_s(w)$ only when *F(w)* does not hold any frequency components higher than half the sampling frequency. From the previous mathematical analysis it can be stated the following important conclusions

- The selection of a sampling frequency for a signal is determined by the highest frequency component of the Fourier spectrum of the signal *f(t)*.
- If critical sampling is needed at $w_N=2w_m$, then an ideal filter is mandatory for the reconstruction of a signal having a frequency component up to w_m.
- It has been assumed a signal f(t) with spectrum that extends from $w=0$ to $l\ w\ l = w_m$, which is a low-pass signal. In this case the signal is completely determined from its sets of values at regularly spaced intervals of period $T = \frac{1}{2f_m} = \frac{\pi}{w_m}$. A band-pass signal can be represented by the following spectrum range

$$w_1 < |w| > w_2$$

It can be shown that the minimum sampling frequency in that case is given by $W_N = 2(w_2 - w_1)$. A band-pass filter is implemented for the reconstruction of the original signal.

For the case where critical sampling is needed at $w_N=2w_m$, such a filter is physically impossible to construct. Therefore, a higher sampling frequency than the theoretical Nyquist rate must be applied to be able to reconstruct the signal. *This is the reason of why speech signals are band limited to about 3.4kHz, but are sampled at a rate of 8 kHz instead of the critical rate of 6.8 kHz.*

2.3 Encoding and Binary Number Representation

Time-domain samples and amplitude quantization produces a series of numbers $\{f_q(nT)\}$ that are encoded in binary form. This is the last step in the analog-to-digital conversion process, and finally the coded sequence becomes acceptable to a digital system. Such systems or processors manipulate the binary sequence by performing a number of standard operations that include storage in memory, multiplication, shifting in time and addition. Each operation carried out is represented by a binary number. There are several ways of representing a number, but a general way of a number can be represented with a finite precision is the following

$$N = \sum_{a=-s}^{q} c_a r^a \quad \textbf{\textit{(26)}}$$

With

$$0 \le c_a \le (r-1)$$

In which c_a is the *ath* coefficient and *r* is called the radix of the representation. If distinct symbols are used for assigning values c_a, the number N can be represented by the notation

$$N = \left(c_q c_{q-1} \cdots c_0 c_{-1} c_{-2} \cdots c_{-s}\right)_r \quad \textbf{\textit{(27)}}$$

The point between the two parts of the number is called the radix point. In order to store the *L*-bit word one needs a memory register of length L. This means that one has L flip-flops and therefore each flip-flop is capable of storing one bit of the *L*-bit word. When one implements an arithmetic operation on the binary numbers using digital processors it can be represented by using a fixed-point number representation or floating-point number representation. The binary point occupies a specific physical position in the register but no specific physical location is assigned to the binary point in the latter.

2.3.1 Fixed-Point Numbers

In fixed-point arithmetic the location of the binary point is held at a fixed position for all arithmetic operations. It is assumed that this type of arithmetic represents proper fractions only. Mixed numbers are avoided because it is difficult to multiply, while integers are avoided because the number of bits in a product cannot be reduced by rounding or quantization.

The use of fractions does not represent any loss of generality in many applications such as digital filtering because signal levels may be scaled to lie within any range. Depending on the way in which negative numbers are represented, there are three types of fixed-point arithmetic. These are a) signed magnitude, b) one's complement, and c) two's complement.

In the signed magnitude representation the binary word consists of the *positive* magnitude and a sign bit as the most significant bit. The signed-magnitude representation is used for positive numbers.

2.3.2 Number Quantization

When an arithmetic operations is performed using a fixed-point representation using a digital microprocessor, the length of the register is fixed and consequently fixes the set of numbers to be represented. For this reason, certain information is lost when the quantization process takes place. If the word-length excluding the sign is n, then any number of length $c > n$ that results from arithmetic operations is approximated by a word of length n. This is obtained by either rounding or truncating the number.

2.3.3 Floating-Point Numbers

If fixed-point arithmetic numbers are used, there are two big disadvantages. The first difficulty consists on the fact that the range of numbers which can be represented is small. Secondly, the error resulting from rounding or truncation increases as the magnitude of the number decreases. The problems mentioned above can be somewhat solved by the use of *floating-point arithmetic.* In this system a number is expressed in the form $N = 2^c M$

When the form is being used, c is called *the characteristic* (or exponent) and M is the *mantissa* of the number.

Chapter 3 Filter Theory

The definition of a *filter* can be seen as a signal processing system whose response differs from the excitation in a given way. One can also defined a *filter* as a frequency selective system or network which takes an input signal that has a certain spectrum *F(jw)*, and attenuates certain frequency components and allows the rest of the spectrum to pass without any change. For this reason, the output spectrum is said to be a filtered description of the original signal *f(t)* or input.

If we consider an input signal *f(t) and* output *g(t)* and by using the original notation of the Fourier transform, the following can be written

$$f(t) \leftrightarrow F(jw) \text{ and } g(t) \leftrightarrow G(jw).$$

The transfer function or filter of the system is

$$H(jw) = \frac{G(jw)}{F(jw)} = |H(jw)| \exp\left(j\Psi(w)\right) \quad \textbf{\textit{(28)}}$$

where $|H(jw)|$ is the amplitude response and $\Psi(w)$ is the phase response. The system described previously is an ideal filter if one has a constant amplitude response inside certain frequency bands, and exactly zero outside these bands. In the bands where the amplitude is constant, the phase is a linear function of the frequency *w*. The basic ideal filter amplitude characteristics are low-pass, high-pass, band-stop, and band-pass filter.

It is very important to now that the ideal filter characteristics cannot be obtained using causal transfer functions and therefore must be approximated. Any variation from the amplitude characteristic is called amplitude distortion, while any difference from the ideal linear phase characteristic is called phase distortion. In applications on which speech is involved, filters are designed on amplitude basis only. This is because the human ear is relatively not sensitive to phase distortion. Other applications tolerate some amplitude distortion while requiring a close approximation to the ideal linear phase response.

Since the ideal filter characteristics cannot be achieved exactly by a realizable causal transfer function, it has to be approximated under certain restrictions. This means that a transfer function must be derived meeting the specification of the filter response within a given tolerance, and on the other hand it must meet certain parameters to be constructed as a physical system.

3.1 Analog Filter Performance Criteria

There are some criteria by which one can compare the relative qualities of two similar filters. The first criteria that one has to analyze is the *pass band*, which is defined as the range of frequencies over which signals pass almost untenanted through the system. The pass band extends to the point where the responds drops off by 3dB, which is known as the cut-off frequency. It is possible to design filters that have no ripple over this pass band, but usually a certain level of ripple is admitted in this region in exchange of a faster roll-off of gain with frequency in the transition region.

The transition region can be defined as the area between the pass band and stop band. The stop band is chosen by the designer depending on certain requirements. The important characteristic is that the response all over the stop band should always be under the design specification.

It is quite impractical to describe a filter exclusively by the variation of its gain with frequency. Therefore, the other important characteristic is its phase response. The phase response is given by the following expression

$$\Psi = \tan^{-1}\frac{\Im H(w)}{\Re H(w)} \quad \textbf{\textit{(29)}}$$

The phase is significant because it is directly correlated to the time delay of the different frequencies passing through the filter. Therefore, a filter with a linear phase response will delay all frequencies by the same amount of time. On the other hand, a filter that has a nonlinear phase response produces all frequencies to be delayed by different periods of time. This means that the signal emerges deformed at the output of the filter.

When a filter is being designed, a linear phase is in fact important only over the pass band and transition band of the filter because all frequencies ahead of are attenuated. It is common to come across with small variations in the phase response in order to obtain better presentation in some other characteristics.

Alternatively, a filter can be described by using its time domain response. The general measures of performance for the response of the filter in the time domain are:

- Rise Time: it is the time taken for the output to reach 90% of full range
- Settling Time: it is the time taken to settle inside 5% of the final value
- Overshoot: it is the maximum amount by which the output temporarily exceeds the needed value after level transition.

3.2 Analog Filter Types

Practical filters are an arrangement between the various measurements described in the previous section. The reason for choosing a particular filter depends on the importance of the different measurements (pass-band ripple, cut-off frequency, roll-off, stop-band attenuation, as well as the phase response). There are a number of filter types that optimize one or more of these features and are widespread to both analog and digital designs.

3.2.1 Butterworth Filter

The main attribute of the filter consists on the fact that it has a totally flat gain response over the pass band. The flat pass band is obtained at the cost of the transition region.

The transition region has a very slow roll-off, and phase response, which is nonlinear around the cut-off-frequency. The gain of a Butterworth Filter is the following

$$\left|\frac{V_{out}}{V_{in}}\right| = \frac{1}{\sqrt{1+\left(\frac{f}{f_{3dB}}\right)^{2n}}} \quad \textbf{\textit{(30)}}$$

Where n is the order of the filter. By increasing the order of the filter, the flat region of the pass band gets closer to the cut-off frequency before it drops away and one can improve the roll-off.

3.2.2 Chebyshev Filter

The Chebyshev filter permits a certain amount of ripple in the pass band, but on the other hand it has a very much steeper roll-off. The gain response of the Chebyshev filter is given by:

$$\left|\frac{V_{out}}{V_{in}}\right| = \frac{1}{\sqrt{1+\varepsilon^2 C_n^2\left(\frac{f}{f_{3dB}}\right)}} \quad \textbf{\textit{(31)}}$$

Where C_n is a special polynomial that is a function of the order of the order of the filter n and ε is a constant that establishes the amount of ripple in the pass band. The improvement in roll-off is considerable when compared with the Butterworth filter. Although the Chebyshev filter is an improvement on the Butterworth filter with respect to the roll-off, they both have poor phase responses. The number of ripples increases proportional to the order of the filter.

3.2.3 Elliptic Filter

The elliptic filter can be measured as an extension of the trade-off between ripple in the pass ban and roll-off. The transfer function of the elliptic filter is given by the following expression:

$$\left|\frac{V_{out}}{V_{in}}\right| = \frac{1}{\sqrt{1+\varepsilon^2} U_n^2\left(\frac{f}{f_{3dB}}\right)} \quad \textbf{\textit{(32)}}$$

The function $U_n \frac{f}{f_{3dB}}$ is called the Jacobian elliptic function. The main achievement of using an elliptic filter is that the maximum possible roll-off is obtained for a particular filter order. The phase response of an elliptic filter is very nonlinear, so we can only use this design in the cases where phase is not an important design parameter.

3.2.4 Bessel Filter

A poor phase response is unwanted in many applications, mainly in audio systems. A badly designed filter in such systems could be devastating. The Bessel filter has a maximally flat phase response in its pass band. When one uses a Bessel filter there is a good phase response, but one has to give up steepness in the transition region.

3.3 Digital Filters

For many years digital filters have been the most frequent application of digital signal processors and computers. By digitizing any system, one is ensured that the signal can be reproduced with exactly the same characteristics. The main advantages of why digital filters are being used instead of analog filters are the following: it is possible to reprogram the DSP and radically alter the filter's gain or phase response, and one can update the filter coefficients while the program is running, therefore *adaptive filters* can be built. If we go into detail in the mathematical description of digital filters one will have to work with functions called *Z-transforms.*

The same way as for analog filters, the design of digital filters in the frequency domain can be done using the following procedure:

- The specifications on the filter characteristics are used to get a description of the filter in terms of a stable transfer function. One can see this as the approximation problem.
- After the transfer function of the filter has been determined, the filter can be realized using either a recursive or non-recursive technique. This consists on the synthesis or realization problem, and its solution requires only the knowledge of the coefficients of the transfer function.

For a recursive filter the relationship between the input sequence *{f(n)}* and the output sequence *{g(n)}* is given by

$$g(n) \equiv function\_of\_\{g(n-1), g(n-2), \ldots; f(n), f(n-1), \ldots\} \quad \textbf{\textit{(33)}}$$

The present output sample *g(n)* is therefore a function only of past, present and future inputs.

For a non recursive realization one has

$$g(n) \equiv function\_of\{f(n), f(n-1), \ldots\} \quad \textbf{\textit{(34)}}$$

Which means that the present output sample g(n) is a function only of past and present inputs.

At this moment, there is an alternative way of classifying digital filters. If a filter has a finite duration impulse response, then such a filter is called a *finite-duration impulse response* (FIR) filter. The filter can be realized either recursively or non-recursively; but it is almost always realized in the non-recursive mode. This is due to the fact that its direct description is given by the following differential equation

$$g(n) = \sum_{r=0}^{M} a_r f(n-r) \quad \textbf{\textit{(35)}}$$

The previous expression satisfies the requirement in the non-recursive realization $g(n) \equiv function\_of\{f(n), f(n-1), \ldots\}$. A significant characteristic of the FIR filter consists on the fact that it is essentially stable since its transfer function has all its poles at *z=0*, which is inside the unit circle.

Next we need to consider the more general case of the differential equation

$$g(n) = \sum_{r=0}^{M} a_r f(n-r) - \sum_{r=1}^{N} b_r g(n-r) \quad \textbf{\textit{(36)}}$$

And the corresponding transfer function for which $Q(z^{-1}) \neq 1$, so one has a non-trivial polynomial. Such a filter has a transfer function given by

$$h(n) = F^{-1}\left(\frac{P(z^{-1})}{Q(z^{-1})}\right) \quad \textbf{\textit{(37)}}$$

Which has an infinite duration, therefore it is called an *infinite impulse response* (IIR) filter. The IIR filter can be realized either recursively or non-recursively. Nevertheless, IIR filters are always implemented recursively.

It can be seeing that the terms FIR and IIR refer to the type of filter, whereas the definitions of non-recursive and recursive refer to the method of realization of the filter. However, by common usage, the terms “non-recursive” and FIR filter are often taken as the same. The same holds for the terms “recursive” and IIR filters.

3.4 Implementation of a Digital Filter

Previously we introduced several types of analog filters such as the Butterworth and Bessel filters. Each of these filters is implemented depending on a specific feature in either the gain or phase response of the filter. These filter types are found in both analog and digital circuits and certain techniques are used to convert the analog circuit description into digital filters.

The implementation of digital filters is quite easy to implement on a processor. We can explain this if we work with a second-order FIR filter with the following output

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) \quad \textbf{\textit{(38)}}$$

Alternatively, for any order FIR filter the general expression is the following

$$y(n) = \sum_{Q=0}^{P} a_Q x(n-P) \quad \textbf{(39)}$$

Where P is the order of the filter.

The filter is constructed of a series of coefficients a_Q and a set of input values x(n). In order to compute the output at any given time one then simply has to multiply the corresponding values in each table and sum the results. The best method for handling these data tables is to load all the input values into a *circular* buffer. The start of the data, x(n), is located by a pointer and the previous data values are loaded sequentially from that point in a clockwise direction. As a new sample is received, it is placed at the position x(n) and the subroutine of the filter starts with x(n), multiplying the data values with the corresponding coefficient value. After calculating the final output, the pointer is moved counter clockwise one position of the buffer and then one waits for the next input sample. The oldest value of the buffer is dropped off the end of the delay chain after the previous calculation; therefore the circular buffer is a *FIFO* type.

The circular buffer is only ideal: it saves moving all the data each time the system receives a new input sample. All the types of microprocessors, including DSP, have linear memory spaces. The lowest address is 0 and the highest XXXX (hex), therefore if one tries to move a value past the bottom of the memory it will either cause an error or the information will just be lost. Hence, one can implement linear addressing with data moves or simulated circular buffers.

3.5 Adaptive Wiener Filter

Previously various types of digital filters were discussed that were time invariant. It means that the coefficients remain fixed throughout the life of the filter. An adaptive filter has coefficients that are designed to vary with time in a strongly controlled manner and it is usually assumed that this happens by using an adaptive algorithm.

Despite the fact that it is not required, *it is also assumed that the frequency of the variation in coefficient values is greatly lower than the bandwidth of the signal being filtered*. In general, one uses adaptive filtering techniques when it is difficult to understand the properties of the signal one wishes to filter. It might be due to genuine doubt about the incoming signal or because it will change a little with time. In this way, the adaptive filter will be able to adjust to the incoming signal's characteristics.

There is a type of linear optimal discrete-time filters known as *Wiener filters.* Wiener filter are formulated for the general case of a complex valued stochastic process with the filter specified in terms of its impulse response. The reason for using complex-valued time series consists on the fact that in many practical situations the samples are measured for a certain band of frequencies.

It is wished to process the input signal *x(n)* by using a discrete-time linear system, which produces an output $\bar{f}(n)$ as an estimate of the original signal f(n). Therefore a system capable of suppressing the effect of the noise is needed. The estimator implemented is an impulse response sequence of finite duration, therefore one is using an FIR filter. The length of the filter is equal to the number of samples within the input signal. Now one considers the FIR filter with the following transfer function

$$H(z) = \sum_{n=0}^{M-1} h(n) z^{-n} \quad \textbf{\textit{(40)}}$$

And now we consider the input random signal as x(n) with the following characteristics:

i. $E[x(n)] = n$, which is a constant

ii. $E[x(n)x(n-m] = R_{xx}(n,m)$

The constant n is known as the mean and one can subtract it so that the process has zero mean. Now we have the following output $\bar{f}(n)$ expressed in the following way

$$f'(n) = \sum_{k=0}^{M-1} h(k)x(n-k) \quad \textbf{\textit{(41)}}$$

$$f'(n) = [h(n)]^{'}[x(n)] \text{ Matrix form} \quad \textbf{\textit{(42)}}$$

Which is a linear combination of the input data x(n-k) and since the desired filter output signal is *f(n)*, then the error of the estimation is the following:

$$e(n) = f(n) - \hat{f}(n) \quad \textbf{\textit{(43)}}$$

At this moment we proceed into calculating the filter coefficients *h(n)* in a way that the error signal is relatively small. This is done by using the orthorgonality principle for stochastic processes. Therefore, the mean square of the error is given by the following relation

$$\varepsilon(n) = E[e^2(n)] = E\left[\left(f(n) - \sum_{k=0}^{M-1} h(k)x(n-k)\right)^2\right] \quad \textbf{\textit{(44)}}$$

In order to minimize the mean square error with respect to the filter coefficients *h(n)* the previous expression needs to be differentiated.

$$\frac{\partial \varepsilon(n)}{\partial h(m)} = 2E\left[\left(f(n) - \sum_{k=0}^{M-1} h(k)x(n-k)\right)x(n-m)\right] \quad \textbf{\textit{(45)}}$$

To make the above expression a minimum, we need to make it equal to zero for all of the samples. This means that the error signal is orthogonal to the data.

$$E\left[((f(n) - \sum_{k=0}^{M-1} h(k)x(n-k))x(n-m)\right] = 0 \quad m = 0,1,\ldots(M-1) \quad \textbf{\textit{(46)}}$$

Chapter 4 Least-Mean-Square Adaptive Filters

The main inconvenience of the steepest-descent (MMS) gradient algorithm consists in the detail that exact measurements of the gradient vector are required at each step in the iteration process. This is not practical and one needs an algorithm for deriving estimates of the *gradient vector* only from the available data. This is achieved by using the *least-mean-square (LMS)* error gradient algorithm. Its advantages are the following: it is done straightforward, does not require matrix inversion, and it does not require correlation measurements.

The MMS error gradient algorithm uses expectation values for calculating the gradient vector. The expression one uses is the following

$$\nabla(n) = -2E\{e(n)[x(n)]\} \quad \textbf{\textit{(47)}}$$

Instead of using expectation values, the LMS algorithm uses instantaneous estimates of the vector based on sample values of the input $[x(n)]$ and the error *e(n)*. Therefore the following expression is used for calculating the instantaneous estimate of the gradient vector

$$\bar{\nabla}(n) = -2e(n)[x(n)] \quad \textbf{\textit{(48)}}$$

Such an estimate is clearly neutral since its expected value is the same as the value in the expression from equation (47). Along the LMS algorithm, the filter coefficients are updated along the direction of the gradient vector estimate according to following expression

$$[h(n+1)] = [h(n)] + \frac{1}{2}\mu\left[-\bar{\nabla}(n)\right] = [h(n)] + \mu e(n)[x(n)] \quad \textbf{\textit{(49)}}$$

This is much simpler than the expression used in the MMS algorithm for calculating the filter coefficients:

$$[h(n+1)] = [h(n)] + \mu E[e(n)x(n)] = [h(n)] + \mu[R_{ex}(n)] \quad \textbf{\textit{(50)}}$$

The gradient vector estimate of the LMS algorithm only requires knowledge of the data $[x(n)]$ and no cross-correlation estimate. The error signal is defined by the following expression:

$$e(n) = f(n) - [x(n)]^t[h(n)] \quad \textbf{\textit{(51)}}$$

The adaptive filtering process ends up working very similar to the steepest-descent algorithm. One has to initialize the coefficient values as $[h(0)]$ at n=0, therefore a value of zero is given as an initial guess, but as the algorithm is started the coefficient values of the filter will begin to converge to the correct value. At a time *n*, the coefficients are updated in the following way:

1. From $[h(n)]$, the input vector (input signal) $[x(n)]$, and the desired output *f(n)* we calculate the error signal by the expression given by equation (51)

2. The new estimate $[h(n+1)]$ is obtained by the expression given in equation (49)

3. The index *n* is incremented by one, and the process is iterated until one reaches a steady state, hence the system has converged.

One can also summarize the steps of the Complex Least Mean Squares (LMS) algorithm as follows:

- An initial value for the filters coefficient is given by $[h(n)] = 0$, at *n=0.*
- The system is repeated for $n = 1,2,\ldots$
- The error signal is calculated $e(n) = f(n) - [x(n)]^{t}[h(n)]$
- The new estimate for the filter coefficient is calculated using the following expression $[h(n+1)] = h(n) - \mu \bar{\nabla} f[h(n)] = [h(n)] + 2\mu e(n)[x(n)]$

Where μ is a positive step size parameter.

It is important to notice that the LMS algorithm provides only an approximation to the optimum Wiener solution. But on the other hand, one is able to avoid estimating the mean-square gradient vector expressed by the following equation

$$\nabla(n) = \begin{bmatrix} \partial\varepsilon(n)/\partial h(0,n) \\ \partial\varepsilon(n)/\partial h(1,n) \\ \vdots \\ \partial\varepsilon(n)/\partial h(M-1,n) \end{bmatrix} \quad \textbf{\textit{(52)}}$$

It can be done by replacing group averages by time averages. Despite the fact that one can manipulate the algorithm in a very simple form, the convergence behavior of the LMS algorithm is very complicated to analyze. This is because the gradient estimate is data dependent and stochastic in its character.

Chapter 5 Noise in Speech Signals

The most frequent problem in speech processing is the consequence of interference noise in speech signals. Interference noise in a certain way modulates the speech signal and reduces its clearness. Interference noise can be produced from acoustical sources such as ventilation equipment, echoes, crowds, and in general with any type of signal that interferes with the speech signals.

A relationship between the strength of the speech signal and the modulation due to the noise is called the signal-to-noise ratio, which expressed in decibels. In an ideal world, the signal-to-noise ratio is greater that 0dB. This indicates that the speech is louder than the noise. The sort and spectral content of the noise determines how much the speech is understood. The most common effective mask is broadband noise. Although, narrow-band noise is less effective at modulating speech than broadband noise, the degree of its effectiveness varies with the frequency.

High-frequency noise encapsulates only the constants, and its effectiveness decreases as the noise increases its amplitude. On the other hand, low frequency noise is a much more effective modulation when the noise signal is louder than the speech signal. The noise is able to encapsulate vowels and consonants at high pressure levels.

In general, noise that affects the speech signals can be described using one of the following criteria:

1. *White Noise*: is a sound or signal consisting of all audible frequencies that have equal intensity. For each frequency, the phase of the noise is completely uncertain. The phase can be shifted up to 360^o, and its value is unrelated to the phase for a given frequency value. In the case where two or more noise signals are added together, the resultant signal has a power equal to the sum of the component powers. The white noise has a strong encapsulating property because of the broad-band spectrum.
2. *Colored Noise*: one can say that any noise that is not white can be considered as colored noise. Colored noise has a frequency spectrum that is band limited inside a range, therefore is different to white noise which extends over the entire spectrum. There are different types of colored noise (pink noise, orange noise, brown noise etc.) depending on the limited range in the Power Spectral Density of the noise. We can generate a specific color noise by filtering white noise by using a filter that has a required frequency response.
3. *Impulsive Noise*: this type of noise refers to unexpected peaks of noise with quite high amplitude. Impulsive noise is generally modeled as contaminated Gaussian noise.

5.1 Noise in Filter Designs

A system that has been digitized has errors; therefore the circuit can only be an approximation to the original analog system. In general, the result of the noise introduced into the system due to the errors produced is more pronounced in IIR filters, because of the feed-back elements cause the errors to increase over time. On the other hand, FIR filters are feed-forward circuits, therefore any error that appears in the output signal is seeing only once per sampling period.

5.1.1 Signal Quantization

It is well-known that all analog-to-digital converters introduce quantization errors to the incoming signal. For a successive approximation or dual slope ADC, the noise introduced is directly proportional to the number of bits in the output of the system. For a sigma delta ADC, the noise introduced depends on the architecture of its modulator and filters.

When the output of the digital-to-analog converter resolution is less than the internal resolution of the processor, a certain amount of noise is added to the output signal.

If one uses a 10 or 12 bit ADC/DAC arrangement with a 16-bit processor, the maximum noise possible at the input to the filter is 2^{-10} or 2^{-12}. If one assumes that the filter has a gain of less than 0dB over its whole frequency limit range, then the noise added by the different sources is insignificant.

5.1.2 Truncation Noise

If we multiply two *n*-bit numbers together, one requires 2*n* bits to store the answer. This is the reason to why in all fixed-point processors the product register and the accumulator are double the length of all the other registers.

When we work with a 32-bit product, after the product is added to the accumulator, the 32 bits are maintained during the filter subroutine. The 32-bit result is then stored in a 16-bit wide memory. One can use two instructions, but this doubles the processing time and the amount of memory required. It also increases the time required to recover the value of use in future operations. Hence, it is usual to store only the most significant 16 bits and truncate the result of the operation.

The error due truncation is quite small, it is present only in the 16th bit and represents <0.001%. In the general real-world applications this is enough precision for most applications.

5.1.3 Internal Overflow

In the majority of processors, when ever a mathematical function produces an output that is greater than the processor's accumulator, the result wraps *round.* For example, in a 4-bit processor only the four bits can be held in the accumulator, therefore the MSB is lost. In a similar way, underflow also causes the processor to wrap around in the negative direction.

The effect of internal overflow or underflow is not very important in FIR filters. This is because the error will only take place for one output. But for IIR filters an error in one output will be fed back into the filter and cause all the following outputs to be in error.

In order to avoid the previous problems, processors normally allow the designer to switch to a saturation mode. In this mode, if a result is greater than the largest possible value, the processors accumulator will saturate to all ones and if the result is less than the smallest value, the values will saturate to zeros.

5.1.4 Coefficient Quantization

The effects of coefficient quantization are most significant in fixed-point processors. Larger floating-point processors that use 32 bits or more are accurate enough; therefore the effects can be ignored. Fixed-point processors have word lengths of 16 bits, and all data is expressed as fractions in order to avoid overflow in the processors accumulator. The accuracy of the coefficients is extremely important because when one multiplies two fixed-point numbers together, it is possible to amplify any error in the values.

In general, 16 bit coefficients are enough to ensure such errors are not important. An error in the last significant bit is less than 0.002%. In the case of an IIR filter, an error in the LSB of a coefficient may produce a big problem. A single LSB error in one feed-back element may cause only a small error for the first samples, but after many samples within the filter might produce an unstable system.

II Implementation of the LMS Algorithm using the LPC2378 ARM Processor

The following chapters will discuss the basic concepts found in the Processor theory. The basic concepts found in the DSPs Processors will be discussed, as well as a brief description of the ARM Processors will be given, and we will go into detail into the LPC2378 Processor. In the last chapters the results of the different experiments that were conducted will be reported, we will discuss the problems that were encounter during the simulations, and finally suggest solutions for them.

Chapter 6 The Processors

There are several options for processors, but we need to decide which one is better for a specific task. Here, we will briefly talk about two possible options for filtering; the ARM processors and DSPs.

6.1 DSPs

The Digital Signal Processors are High Performance microprocessors and have significant advantage among the normal general purpose microprocessors: they execute all of their instructions using only one clock cycle, they have plenty of resources to do mathematical assignments (including hardware multipliers, instructions for working with tables etc). This is why Fast Fourier Transforms or Filter designs made in assembler with DSPs take very few lines of code and are able to execute hundreds of times faster than a normal general purpose processor. Another great feature is that DSP peripherals are extremely fast. The previous statement is because these powerful devices have to feed fast input data and also be able to send back fast the processed information. The TMS320 C2000 base line for instance, have microcontrollers with two build-in 12-bit ADCs with sampling speed from 3.75 to 12.5 MSPS and enhanced PWMs with 150 ps resolution. The SPI and serial ports are also at least 10 times faster than the similar peripherals on general purpose microcontrollers [3].

Texas Instruments is the recognized leader in DSP field and recently they released the world's lowest cost DSP with CAN - TMS320F28016 which cost only $3.60 in 1000 pcs quantity [3].

Well then, if the DSPs are so good and the price tag is as normal as a 8-bit controller but offer much more features, why is it that they are not so widely used as PICs or AVRs and other conventional microcontrollers [3]?

The main reason is that there were no low cost entry tools and starter kits for DSPs (until recently). To start working with DSP we needed to buy a US $500-1500 JTAG emulator, a C development IDE for $500, and a development board for another $500. Spending $2000 just to start learning the new architecture is not a problem for an established company, but totally out of range for small start-up companies, students, hobbyist, and amateurs [3].

6.2 ARM Processors

6.2.1 ARM Introduction

ARM processors have 16 or 32-bit core and they use RISC architecture. They are new in microcontrollers markets. Because of low price (close to 8-bit microcontrollers), many peripheral abilities, and faster speed than convectional 8-bit microcontrollers, they are getting extremely common [4].

Today several companies are producing ARM processors. Some of these companies are the following: Analog Devices, Atmel, Cirrus Logic, OKI, Philips Semiconductors, ST Microelectronics, Texas Instruments and many others (Intel, Freescale, Samsung, Sharp, Hynix, etc). According to many experts in the microprocessors world, in the next 5 years ARM Processors will replace the industry standard 8051 architecture in almost all applications [4].

A few years ago, when ARMs just came out to the markets, it was very expensive to work with these microcontrollers, for example for buying compiler, debugger, and development board it was needed to spend thousands of US dollars. But now there are several companies that produce cheap development boards and JTAG debuggers. An important aspect to mention is that the GCC C compiler is a free compiler [4].

6.2.2 ARM Processors Advantages

Very fast: most ARM7 cores run at 60Mhz and ARM9 cores run at 150Mhz+ providing more power than old 386 Intel processors.
Low power: ARM7 cores need only approx 0.5-1mA per MHz. Great range of peripherals: ADC, DAC, USB, SPI, UART, I2C, CAN, Ethernet and SDRAM.
Lot of internal Flash: 32KB-1MB, Lot of internal RAM: 4-256KB [4].

6.2.3 ARM Processors Disadvantages

The ARM Processors are too complex for beginners; without doubt this one should not be used by someone that does not have a good background in programming microcontrollers.

6.2.4 ARM Architecture

There are 4 main types of ARM processors: ARM7TDMI, ARM9TDMI, ARM10TDMI and ARM11 on the figure 6.1 one can see these four architectures.

6.2.5 ARM7TDMI, The 3-stage pipeline

In Figure 6.1 we can see the original 3-stage ARM pipeline. It is a conventional fetch-decode-execute pipeline, which in the lack of pipeline hazards and memory accesses, completes one instruction per cycle. In the first stage, it reads an instruction from memory and increments the value of the instruction pointer register and in the next stage it decodes the instruction and prepares the control signals to execute it. The third stage does all the actual work: it reads operands from the register file, performs ALU operations, reads or writes memory, and finally writes back the modified register value. In case the instruction being executed is a data processing instruction, the result generated by the ALU is written directly to the register file and the execution stage completes in only one cycle. If it is a load or store instruction, the memory address computed by the ALU is placed on the address bus and the actual memory access is performed during the second cycle of the execute stage [5].

Since the PC (Program Counter) is accessible as a general-purpose register, there are several places in the pipeline where the next instruction address can be issued. Under normal conditions, it is incremented on every cycle during the fetch stage. If a data processing instruction specifies R15 as its destination operand, then the result of the ALU operation is used as the next instruction address. Finally, a load to R15 has a similar effect [5].

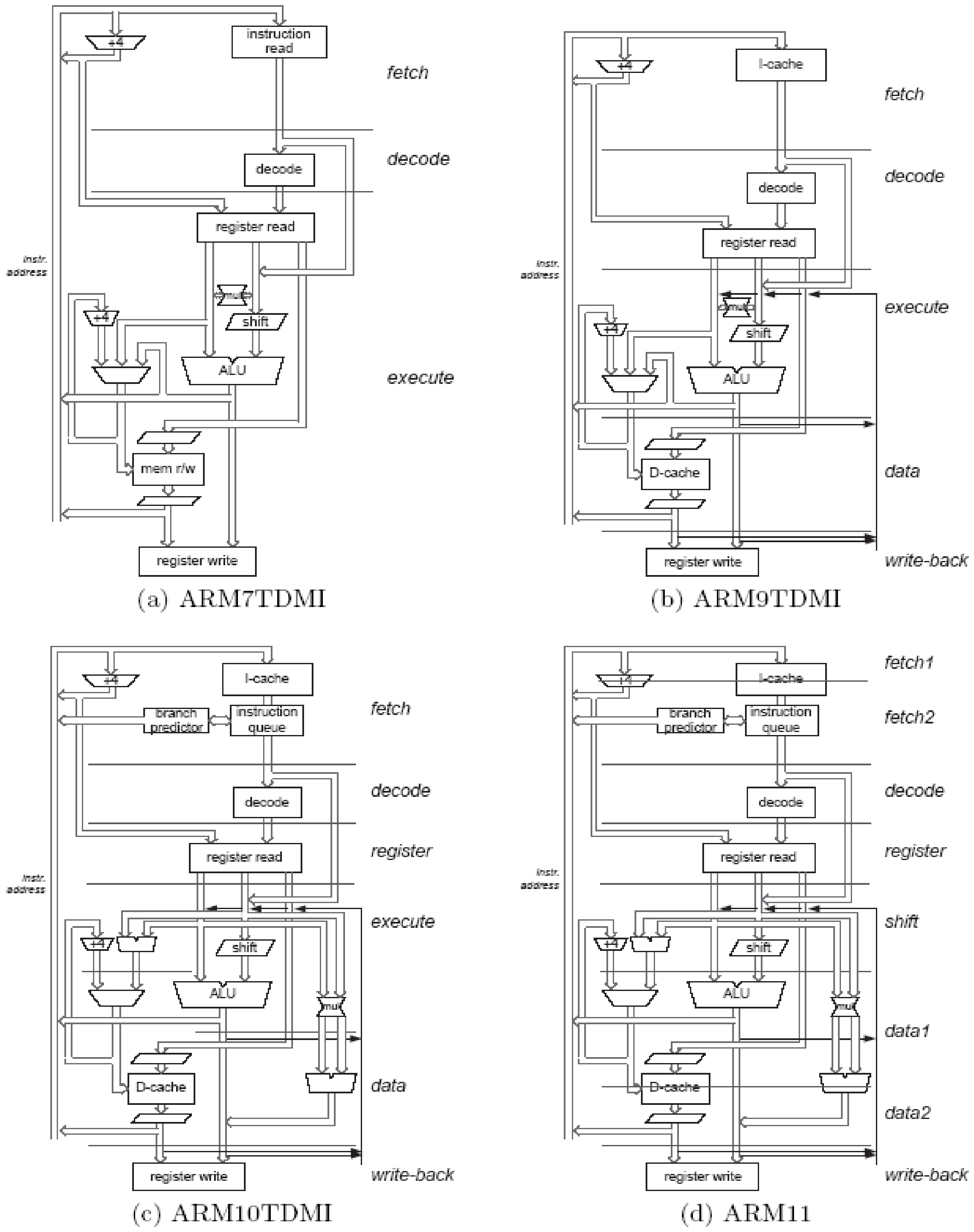


Figure 6.1: ARM architectures (Leonid Ryzhyk. *The ARM Architecture)* [5]

6.3. Working with ARM Processors

To work with an ARM processor the following accessories are needed:

1- Computer.
2- A compiler for writing the code. There are some companies that have compilers for ARM processors. Some of these companies are the following: Hitex, Keil, IAR and Eclipse. The one that we have applied is IAR EWARM 5.4, which is a Japanese company and is a compiler for C/C++ languages and also assembly itself. It is recommended to buy the Olimex Company development board to be able to work with the GNU open source C, which is a free compiler and made by Eclipse.
3- The program loader is a software that loads the output of compiler into the hardware (programmer) connected to the computers port. Some common program loaders are the following: ARMProg, Embest, and H-JTAG. We have used H-JTAG V.2, which is free software.
4- The programmer (JTAG) is the one that reads the output of the computer and sends it to the development board. For debugging purposes we have used JTAG. There are different options that can be implemented like the following: ARM-JTAG, ARM-USB-OCD, ARM-USB-TINY, ARM-JTAG-EW. We have used ARM-JTAG, which uses the parallel port of the computer to communicate with the microcontroller.
5- The development board is the one in which the ARM processor and other electronic elements are located. We have used the LPC2378-STK, which is a common board for LPC2378 ARM processors.

The processor that we used on this project is the LPC2378, an ARM based processor from PHILIPS Company. The features of this processor will be discussed in the next part of the report.

6.4 LPC2378 Processor

6.4.1 LPC2378 General Description

The LPC2377/78 microcontroller is based on a 16-bit/32-bit ARM7TDMI-S CPU with real-time emulation that combines the microcontroller with 512 kB of embedded high-speed flash memory. A 128-bit wide memory interface and unique accelerator architecture enables 32-bit code execution at the maximum clock rate. For critical performance in interrupt service, routines and DSP algorithms are implemented, this increases the performance up to 30 % over Thumb mode. For critical code size applications, the alternative 16-bit Thumb mode reduces code by more than 30 % with minimal performance penalty [7].

The LPC2377/78 is ideal for multi-purpose serial communication applications. It incorporates a 10/100 Ethernet Media Access Controller (MAC), USB full speed device with 4 kB of endpoint RAM (LPC2378 only), four UARTs, two CAN channels (LPC2378 only), an SPI interface, two Synchronous Serial Ports (SSP), three I2C-bus interfaces, an I2S-bus interface, and an External Memory Controller (EMC). This mix of serial communications interfaces combined with an on-chip 4 MHz internal oscillator, SRAM of 32 kB, 16 kB SRAM for Ethernet, 8 kB SRAM for USB and general purpose use, together with 2 kB battery powered SRAM make this device very well suited for communication gateways and protocol converters. Various 32-bit timers, an improved 10-bit ADC, 10-bit DAC, PWM unit, a CAN control unit, and up to 104 fast GPIO lines with up to 50 edge and up to four level sensitive external interrupt pins make these microcontrollers predominantly appropriate for industrial control and medical systems [7].

6.4.2 LPC2378, Features [7]

ARM7TDMI-S processor is able to run at up to 72 MHz.

- Up to 512 kB on-chip flash program memory with In-System Programming (ISP) and In-Application Programming (IAP) capabilities. Flash program memory is on the ARM local bus for high performance CPU access.
- 32 kB of SRAM on the ARM local bus for high performance CPU access.
- 16 kB SRAM for Ethernet interface. It can also be used as a general purpose SRAM.
- 8 kB SRAM for general purpose DMA use also accessible by the USB.
- Dual Advanced High-performance Bus (AHB) system that provides for simultaneous Ethernet DMA, USB DMA, and program execution from on-chip flash with no contention between those functions. A bus bridge allows the Ethernet DMA to access the other AHB subsystem.
- EMC provides support for static devices such as flash and SRAM as well as off-chip memory mapped peripherals.
- Advanced Vectored Interrupt Controller (VIC), supporting up to 32 vectored interrupts.
- General Purpose DMA controller (GPDMA) on AHB that can be used with the SSP serial interfaces, the I2S port, and the Secure Digital/Multimedia Card (SD/MMC) card port, as well as for memory-to-memory transfers.
- Serial Interfaces:

    1. Ethernet MAC with associated DMA controller. These functions exist in on an independent AHB.
    2. USB 2.0 full-speed device with on-chip PHY and associated DMA controller (LPC2378 only).
    3. Four UARTs with fractional baud rate generation, one with modem control I/O, one with IrDA support, all with FIFO.
    4. CAN controller with two channels (LPC2378 only).
    5. SPI controller.
    6. Two SSP controllers with FIFO and multi-protocol capabilities. One is an alternate for the SPI port, sharing its interrupt and pins. These can be used with the GPDMA controller.
    7. Three I2C-bus interfaces (one with open-drain and two with standard port pins).
    8. I2S (Inter-IC Sound) interface for digital audio input or output. It can be used with the GPDMA.

- Other peripherals:
    1. SD/MMC memory card interface.
    2. 104 General purpose I/O pins with configurable pull-up/down resistors.

3. 10-bit ADC with input multiplexing among 8 pins.
4. 10-bit DAC.

5. Four general purpose timers/counters with 8 capture inputs and 10 compare outputs. Each timer block has an external counter input.
6. One PWM/timer block with support for three-phase motor control. The PWM has two external count inputs.
7. Real-Time Clock (RTC) with separate power pin, clock source can be the RTC oscillator or the APB clock.
8. 2 kB SRAM powered from the RTC power pin, allowing data to be stored when the rest of the chip is powered off.
9. WatchDog Timer (WDT). The WDT can be clocked from the internal RC oscillator, the RTC oscillator, or the APB clock.

- Standard ARM test/debug interface for compatibility with existing tools.
- Emulation trace module supports real-time trace.
- Single 3.3 V power supply (3.0 V to 3.6 V).
- Three reduced power modes: idle, sleep, and power-down.
- Four external interrupt inputs configurable as edge/level sensitive. All pins on PORT0
  and PORT2 can be used as edge sensitive interrupt sources.
- Processor wake-up from Power-down mode via any interrupt able to operate during.
- Power-down mode (includes external interrupts, RTC interrupt, USB activity, Ethernet wake-up interrupt).
- Two independent power domains allow fine tuning of power consumption based on needed features.
- Each peripheral has its own clock divider for further power saving.
- Brownout detect with separate thresholds for interrupt and forced reset.
- On-chip power-on reset.
- On-chip crystal oscillator with an operating range of 1 MHz to 25 MHz.
- 4 MHz internal RC oscillator trimmed to 1 % accuracy that can optionally be used as the system clock. When used as the CPU clock, does not allow CAN and USB to run.
- On-chip PLL allows CPU operation up to the maximum CPU rate without the need for a high frequency crystal. May be run from the main oscillator, the internal RC oscillator, or the RTC oscillator.
- Boundary scans for simplified board testing.
- Versatile pin function selections allow more possibilities for using on-chip peripheral functions.

6.4.3 LPC2378, Applications [7]

The LPC2378 is implemented nowadays in the following application:

- Industrial control
- Medical Systems

- Protocol converter
- Communications

Chapter 7 Practical Part of the Thesis Project

7.1 General Descriptions

We can categorize the whole of what has been done in this project in 3 stages.

7.1.1 Analog Hardware

Analog Hardware is implemented where the signals are picked up by the microphone and noise also is added to it, then amplified, and finally sent to the Analog to Digital Converter of the processor. The same noise is also amplified and sent to another A/D channel of the processor. In figure 7.1 we can see the schematic of the analog circuit implemented in the project.

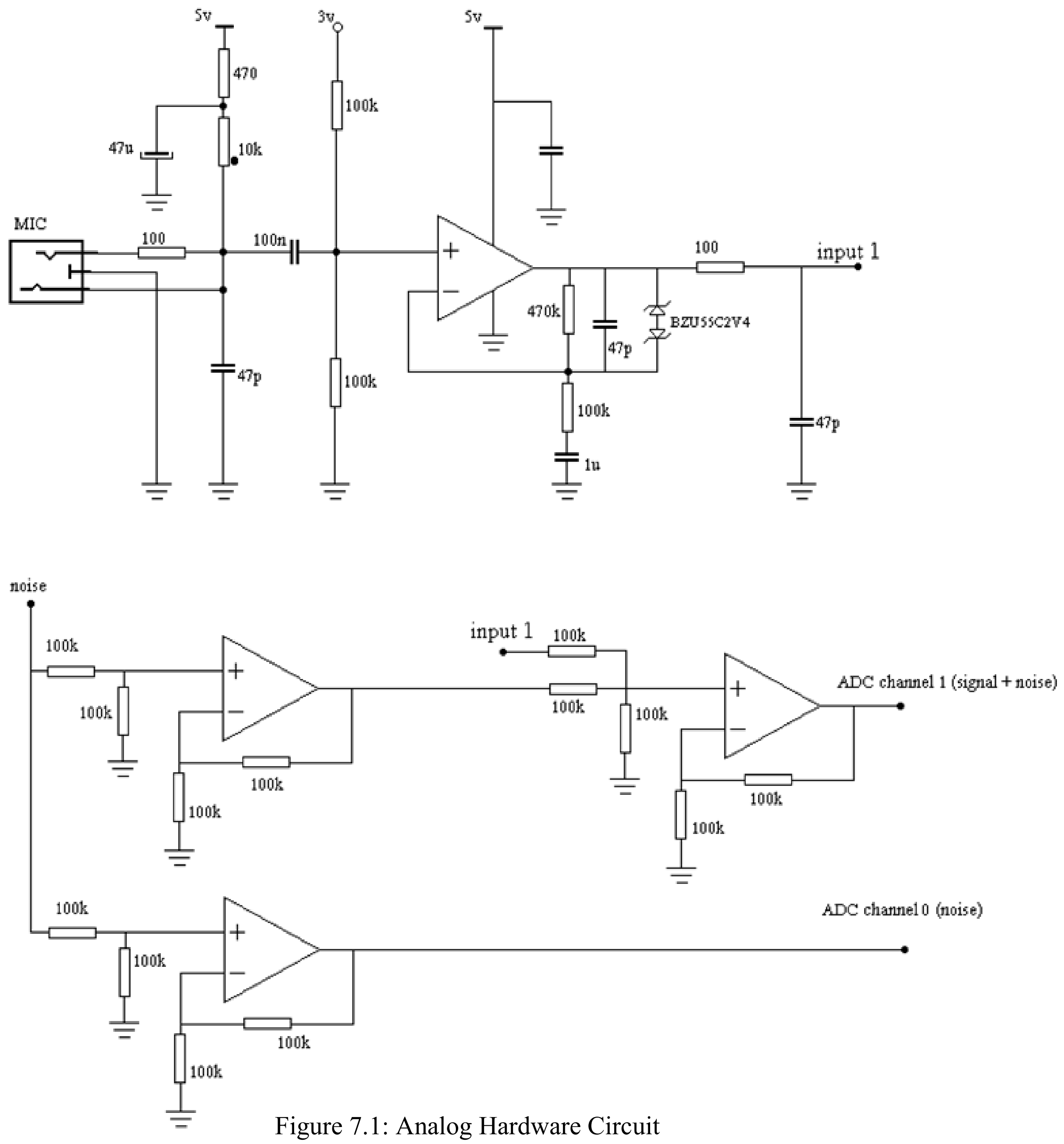


Figure 7.1: Analog Hardware Circuit

7.1.2 Processor

The processor implemented in the project is the LPC2378. The code related to the filter is written in C language, and we have used the IAR EWARM compiler for developing the code. Here we will briefly describe the *main.c* of the code where the main body of the filter is located.

```
#include <nxp/iolpc2378.h>
#include "type.h"
#include "irq.h"
#include "target.h"
#include "uart.h"
//#include "fio.h"
#include "adc.h"
#include "board.h"

extern volatile DWORD UART0Count;
extern volatile DWORD ADC0Value[];
extern volatile DWORD ADC0IntDone;
extern volatile unsigned char chan_no;
extern volatile signed int ADC_Data2;
extern volatile signed int ADC_Noise;
BOOL AD_ready_N = 1 ;
BOOL AD_ready_S = 1 ;
extern volatile BYTE UART0TxEmpty ;
volatile  signed char ADC_signal;
volatile  signed char ADC_noise[50] ;                    //declaring an array for buffering 50
data bytes
                                                         //this array is applied for updating
u(step size)
volatile  double  filter_weight[4] ;                     //order of the filter is 4
extern volatile unsigned int ADisOK  ;
volatile unsigned char ADC_Da ;
volatile double u_noise = 0 ;
volatile double u_under_new = 0 ;
BOOL filte_on_off = 1 ;
/*****************************************************************************
**   Main Function  main()
*****************************************************************************/
int main (void)                                          // start of the main code
{
  TargetResetInit();
  ADCInit( ADC_CLK );                                    // Initialize ADC
  UARTInit(115200);                                      // baud rate setting
  UART0Count = 1 ;
  FIO0DIRU &= 0xDFFBFFFF ;
  FIO4DIR3 = 0x0 ;
      u_under_new = 800;                                 //initial step size =  0.00025 ,  u0 =
0.02
      while(1)                                           //satart of main loop
      {
       if(AD_ready_N) ADC0Read( 1 );                     //noise is sampled
       if(((FIO0PIN)& 0x20000000) == 0 ) filte_on_off = 1 ;   //filter is on, button 1 is pressed
       if(((FIO0PIN)& 0x00040000) == 0 ) filte_on_off = 0 ;   //filter is off, button 1 is pressed
       if((AD_ready_S))                                  //speech is sampled
       {
        AD_ready_N = 0 ;
        AD_ready_S = 0 ;

        unsigned int index = 0 ;
        signed char  yhat  = 0 ;
        while(index != 3)                                    //this is the loop for obtaining yhat
        {
          ADC_noise[index+1] = ADC_noise[index] ;
          yhat = yhat + (ADC_noise[index+1] * filter_weight[index+1])/100;
          index = index +1 ;
        }
        ADC_signal = ADC_Data2 ;                             //latest signal is recorded
        ADC_noise[0] = ADC_Noise ;                           //latest noise is recorded
        ADC0Read( 0 );
        yhat = yhat + (ADC_noise[0] * filter_weight[0])/100;
        u_under_new  = u_under_new + ADC_noise[0]*ADC_noise[0] -
ADC_noise[49]*ADC_noise[49] ;

        while(index != 49)                                   //the 50 byte is shifted
        {
          ADC_noise[index+1] = ADC_noise[index] ;
          index = index +1 ;
        }
        u_noise = u_under_new/ 2 ;
        if(u_under_new == 0 ) u_noise = 1 ;                  //step size is updated
        signed int filterd_speech = 0 ;
        filterd_speech = ADC_signal - yhat;                  //noisy signal is filtered here
        if(filte_on_off) ADC_Da = filterd_speech;            // if the button 2 is pressed filter is
not applied
        else ADC_Da = ADC_signal ;

        index = 0 ;
        while(index != 4)                                    //this loop is for updating the filter
weights
        {
          filter_weight[index] = filter_weight[index] + (filterd_speech*ADC_noise[index])/u_noise;
          index= index +1 ;
        }
        while ( !(UART0TxEmpty & 0x01) );                    //out put of the code is sent to serial
port
        if(ADisOK == 4)
        {
          U0IER = IER_THRE | IER_RLS;
          UARTSend( (BYTE *)ADC_Data2, 1 );
          U0IER = IER_THRE | IER_RLS | IER_RBR;
        }
      }
    }
  return 0;
}
```

In figure 7.2 it is demonstrated an equivalent diagram of the source code.

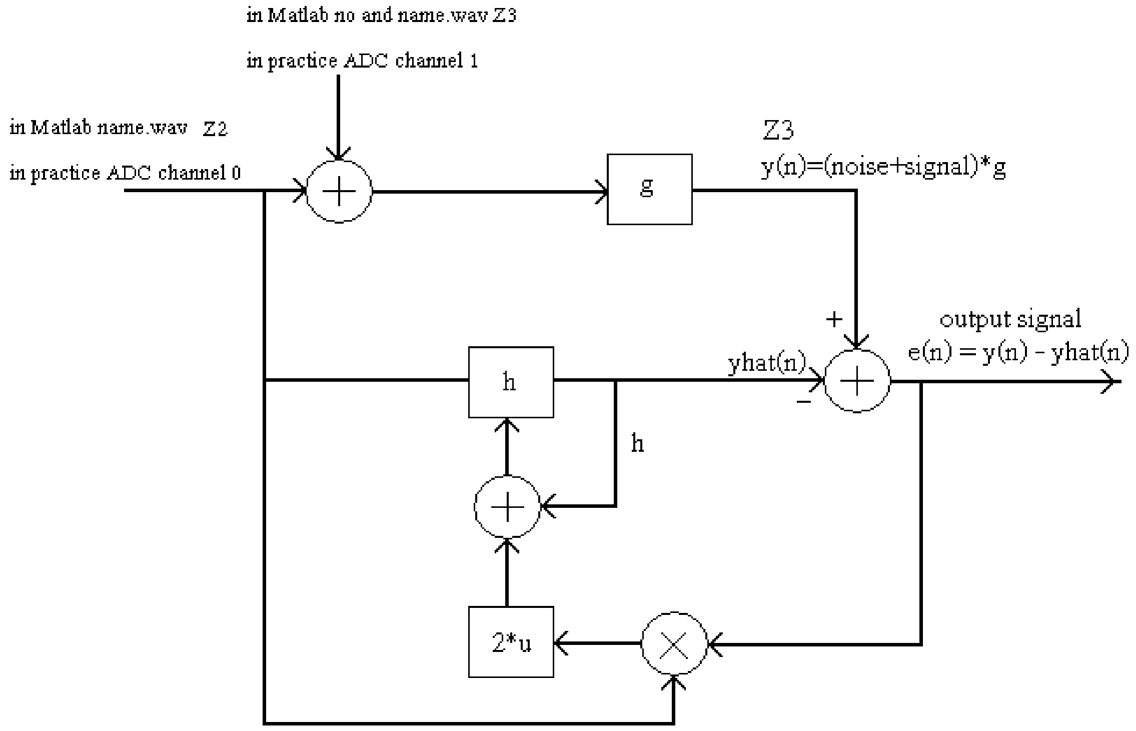


Figure 7.2: diagram of the LMS algorithm implemented in the processor.

Note: on figure 7.2 the value of ‘g’ can be 1.

7.1.3 Computer

At the computer, by the help of RS232 LOGGER software, we recorded the incoming data from the processor’s serial port and then the information is saved in a directory as a .txt file. After one has saved the filter’s output as a.txt file, then Matlab code is implemented to convert the .txt file into an audio .wav file. The RS232 LOGGER software is free. The Matlab code for converting data to .wav file is below:

```
file=fopen('test1.txt')                 %opening the saved file
h = fread(file);                        %reading the saved file as data
h3 = (h-128)/128 ;                      %converting the data to a .wav file spectra
plot(h3)
fclose(file)
wavwrite(h3,sampling_frequency,'test')   %recording the read data as .wav file
                                         %here sampling frequency is 6500 Hz
```

For comparing the practical results obtained by implementing the LMS algorithm with the ARM LPC2379 Processor, a Matlab simulation was implemented. The source code for the Matlab simulation is the following:

```
clear all
close all
u0 = 0.03 ;
[z1,Fs1,bits1] = wavread('no.wav');
[z2,Fs2,bits2] = wavread('name.wav');                        %noise
 [z3,Fs3,bits3] = wavread('no and name.wav');                %noisy signal
   Nx(1,:) = size(z2) ;
   N = 4 ;
   u = u0/(N*(z2'*z2)/Nx(1,1)) ;
   %z2 = z2 + 0.2;
   yhat  = zeros(1,Nx(1,1));
   %h=[0.9957; 0.0271 ; -0.0191 ; 0.0289 ; -0.0137 ; 0.0075 ; 0.0133 ; -0.0137 ; 0.0207 ; -0.0050];

   h     = zeros(N,1);
   signal= zeros(1,Nx(1,1));
   for k=1:N
      X(k)=z2(k,:);
   end
   for count= 1:Nx(1,:)-0 ;
      for n = N:-1:2
         X(n)=X(n-1);
      end
      X(1) = z2(count) ;
      yhat =  X*h ;
      signal(count)= z3(count+0) - yhat;
      h = h+ 2*u*signal(count)*X' ;
   end
   figure(2)
   subplot(2,2,1);plot((h))
   xlabel('z1')
   subplot(2,2,2);plot(z3)
   xlabel('z3')
   axis([0 22500 -1 1])
   subplot(2,2,4);plot(signal)
   xlabel('error')
   axis([0 22500 -1 1])
   subplot(2,2,3);plot(z2)
   xlabel('z2')
   axis([0 22500 -1 1])
soundsc(z3,Fs1)                                   %this is mixture of the two first signals
pause(3.5)
soundsc(signal,Fs1)                               %this signal is filtered and tries to retrieve the first
signal
pause(3.5)
```

7.2 The Thesis Project Functions and its Abilities

The main objective of this project was to see how in reality the LMS algorithm works, and by doing this ones has experimented several conditions that will be discussed later.

Now we can talk about the project's abilities. The processor is programmed to work in two modes; in the first mode the filter is applied and in the second mode one does not filter the noisy signal. One can change the modes by pressing a button on the circuit board. It has been done like this to see how much the filter is able to cancel the noise.

Before talking about noise conditions, we should talk about a big obstacle that happens in real experimental conditions, which is a shift phase in the noise. The LMS algorithm is capable of filtering a noisy signal if we know the behavior of the noise source. It is important to know what will happen if the noise from figure 7.2 has shift in phase like the one in figure 7.3.

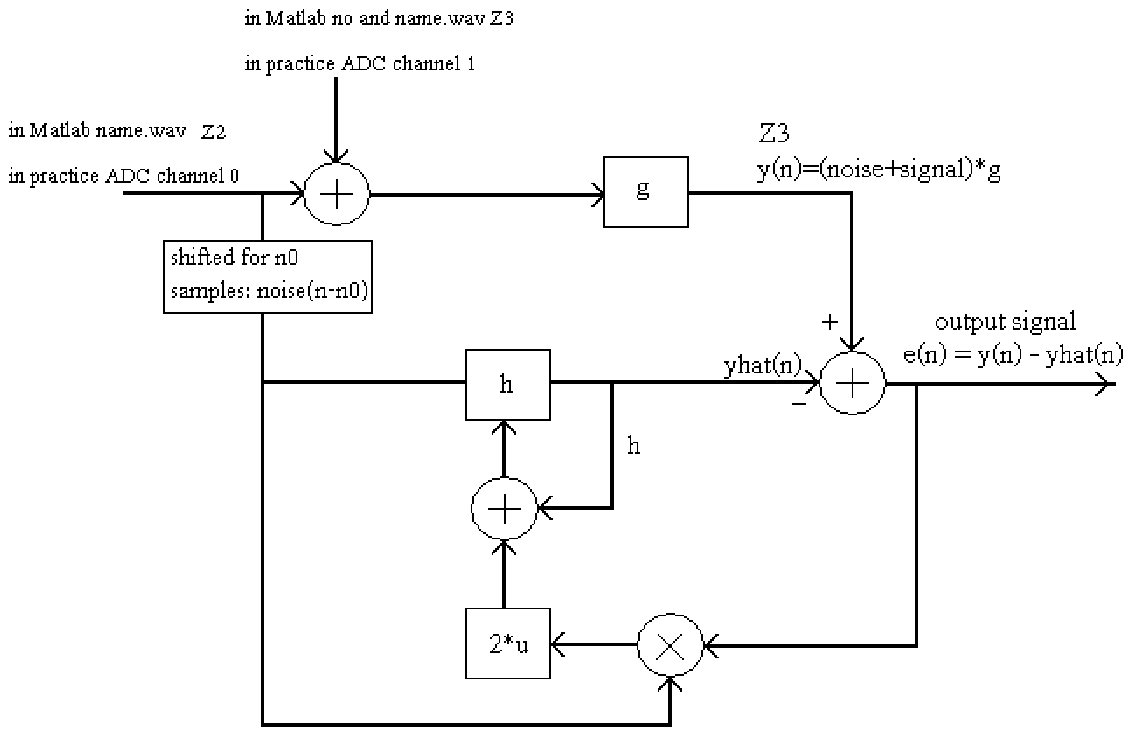


Figure 7.3 Phase shifted noise is available for filtering.

Now consider figure 7.4 as a symbolic situation in an experimental practical condition: We can consider a simple sinus wave noise signal with a fixed frequency of 500Hz, with a period of 2ms, and it is also known that the speed of sound is 10cm/ms. As a result in the second microphone we are always missing the first half cycle that one has in the first microphone. This is because of the distance (in Figure 7.4 it is position of A'). This means that if the electronic equipment takes a sample of both microphones at moment of t1 (refer to the figure 7.4), the first microphone will record point A and the second microphone will record point B, and therefore the noise signal is not equal in both microphones. A phase shift between the signals in each microphone is created, and it is a function of the distance between the two microphones. Another difference encountered is the amplitude of the two signals, which is also related to the distance of the two microphones. Therefore, the main task of the filter weights is to adjust efficiently to be able to cancel the noise as much as possible. One will see how successful the LMS algorithm is with two different kinds of noise signals that were used under experimental conditions.

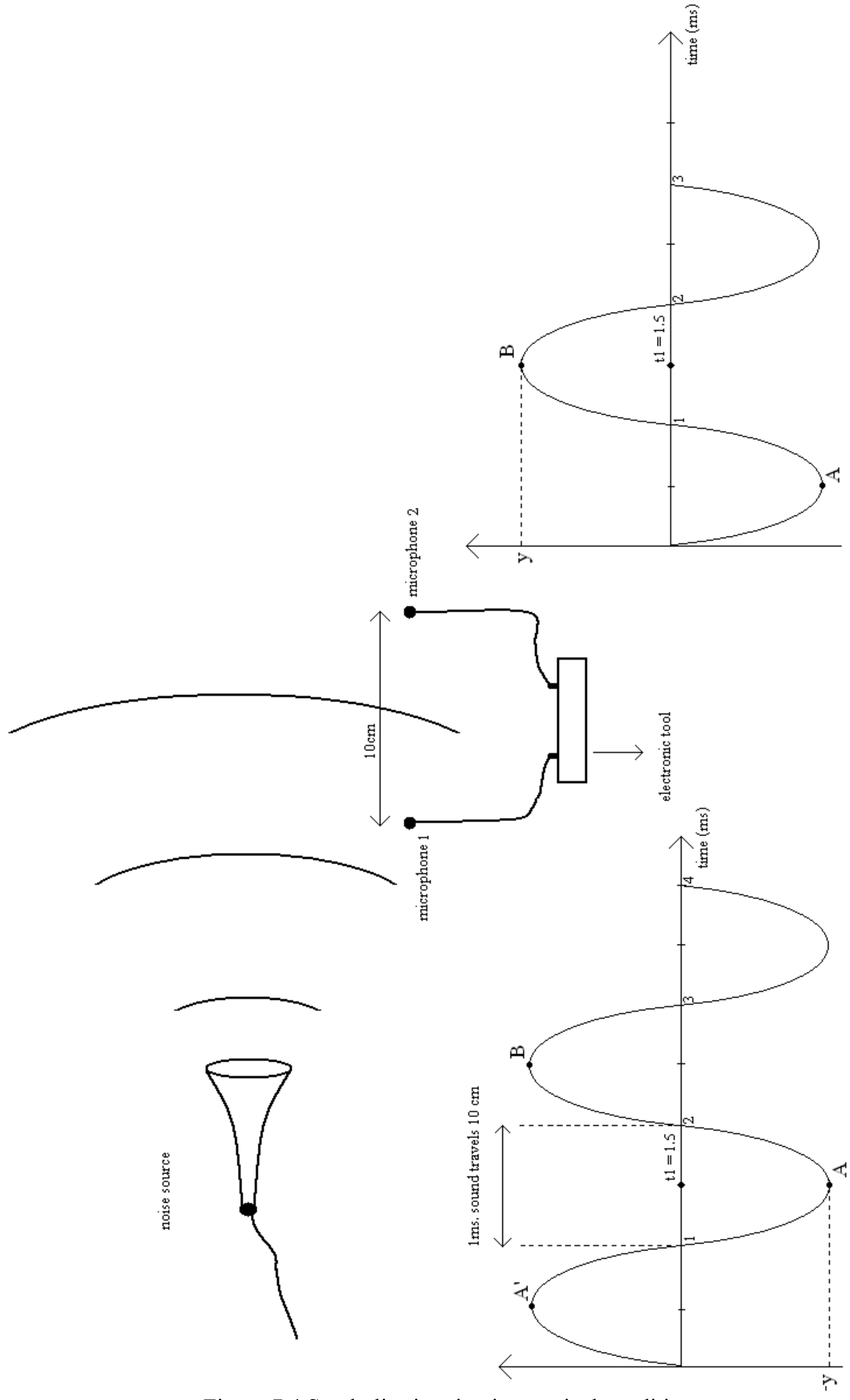


Figure 7.4 Symbolic situation in practical condition.

We should know that in real conditions, noise is not always a simple pattern wave and the shift in phase is higher.

We will consider noise in terms of two main types of signals: a simple pattern wave like a sinusoidal periodic wave, and a not simple pattern wave. Now we will go into each type of pattern wave.

7.2.1 Simple Pattern Wave Noise

In this condition the noise is a simple pattern wave like a sinusoidal periodic wave. This kind of noise can be generated by a function generator. A sinusoidal wave was used as the noise source with amplitude of 440 mV peak-to peak and a frequency range of about 200 Hz up to 1000 Hz.

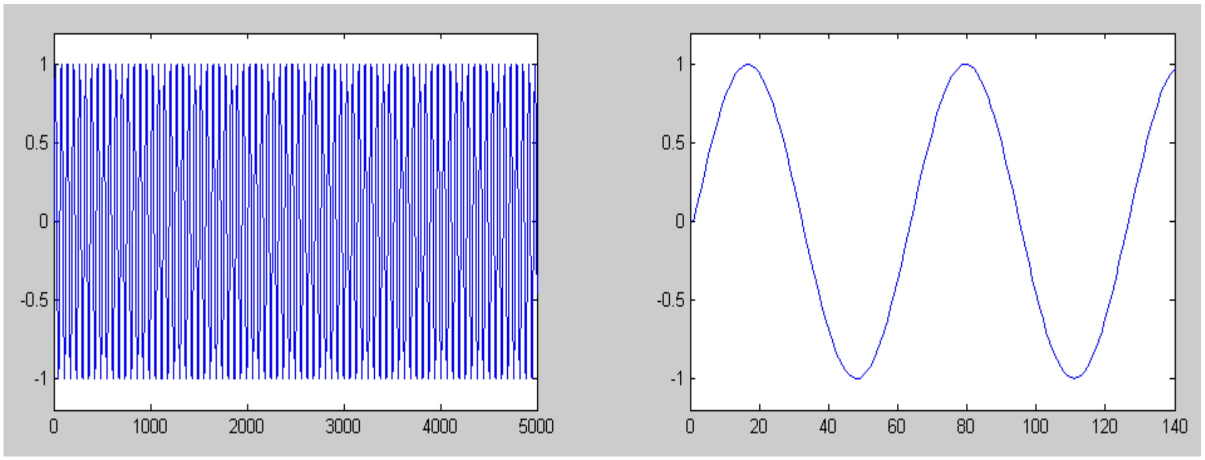


Figure 7.5: Simple pattern sinusoidal wave.

For this kind of noise the processor (LPC2378) succeeded in damping the noise by implementing a 4th order filter with sampling rate of 6.8kps. It even worked when there was a phase shifted noise. Figure 7.7 shows the practical results and Figure 7.6 shows simulation results done by Matlab. After looking at Figure 7.7, we can see clearly that phase shift in noise affects dramatically the output result of the filter.

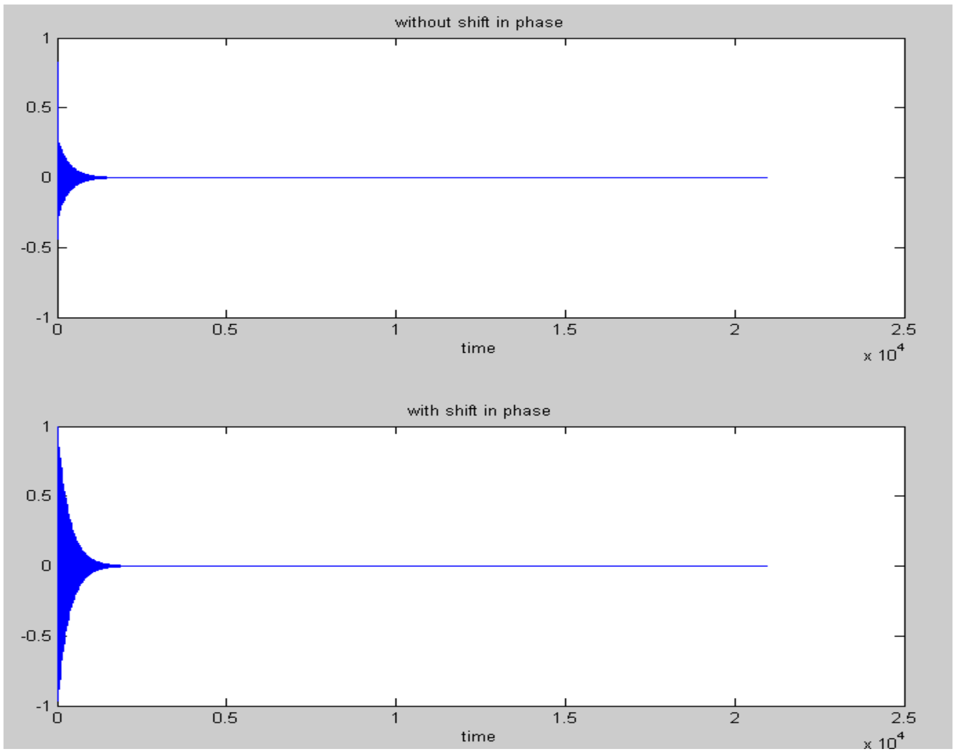


Figure 7.6 Matlab simulation. The sinusoidal noise source
Has being completely damped with and without phase shift.

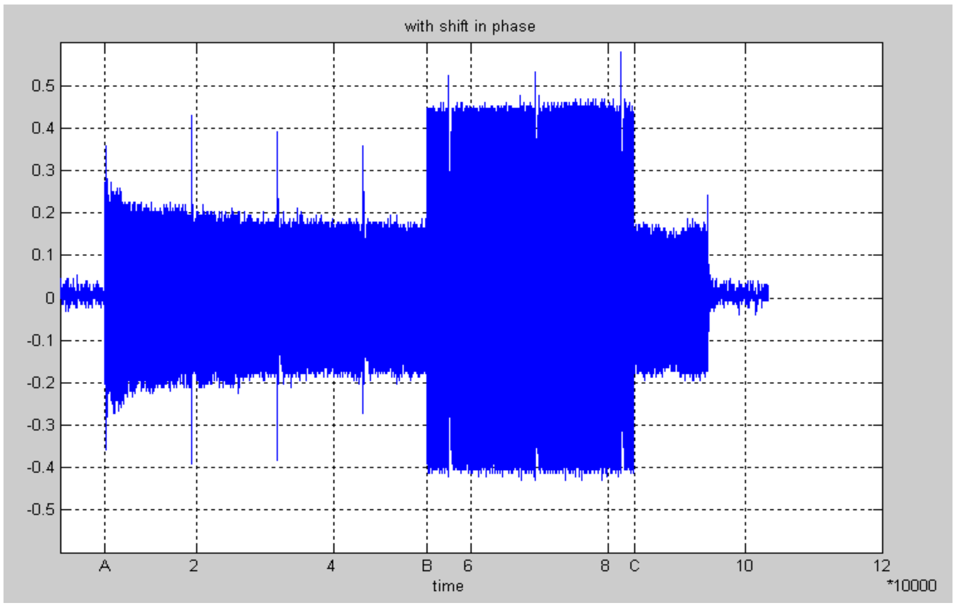


Figure 7.7 Output of the filter using the ARM LPC2378 Processor
At point 'A' the noise source is started, at point 'B' the filter is stopped to see the
Output without filtering, at point 'C' the filter is applied once again.

7.2.2 Non Simple Pattern Wave Noise

For this case the noise source is not a simple pattern wave, this kind of noise can be produced by an engine or any other non-periodic signal, figure 7.8 shows one of these types of noises.

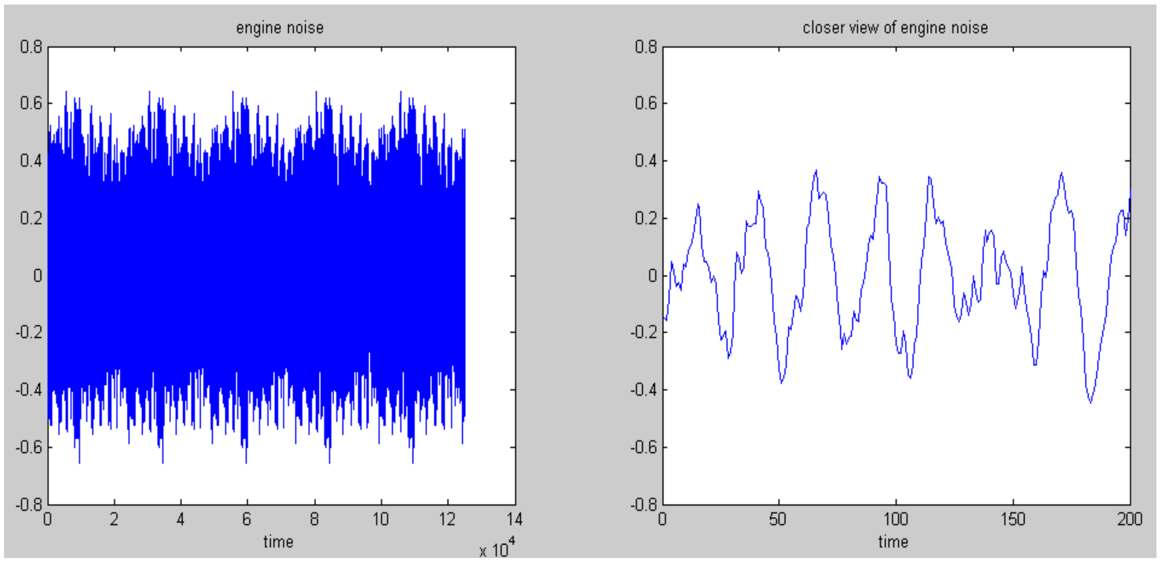


Figure 7.8: Non simple pattern wave noise.

For this kind of noise a 4th order filter was implemented to cancel the noise when there is no shift in phase, but for a phase shifted noise a 50th order filter is ideal when one has 25 samples shifted or a 200 order filter would be ideal when one has 100 shifted samples. Figures 7.9 and 7.10 show how Matlab can damp this kind of noise with and without shift in phase respectively, and figure 7.11 shows how the LPC2378 Processor can damp this noise when there in no shift in phase with a 4th order filter.

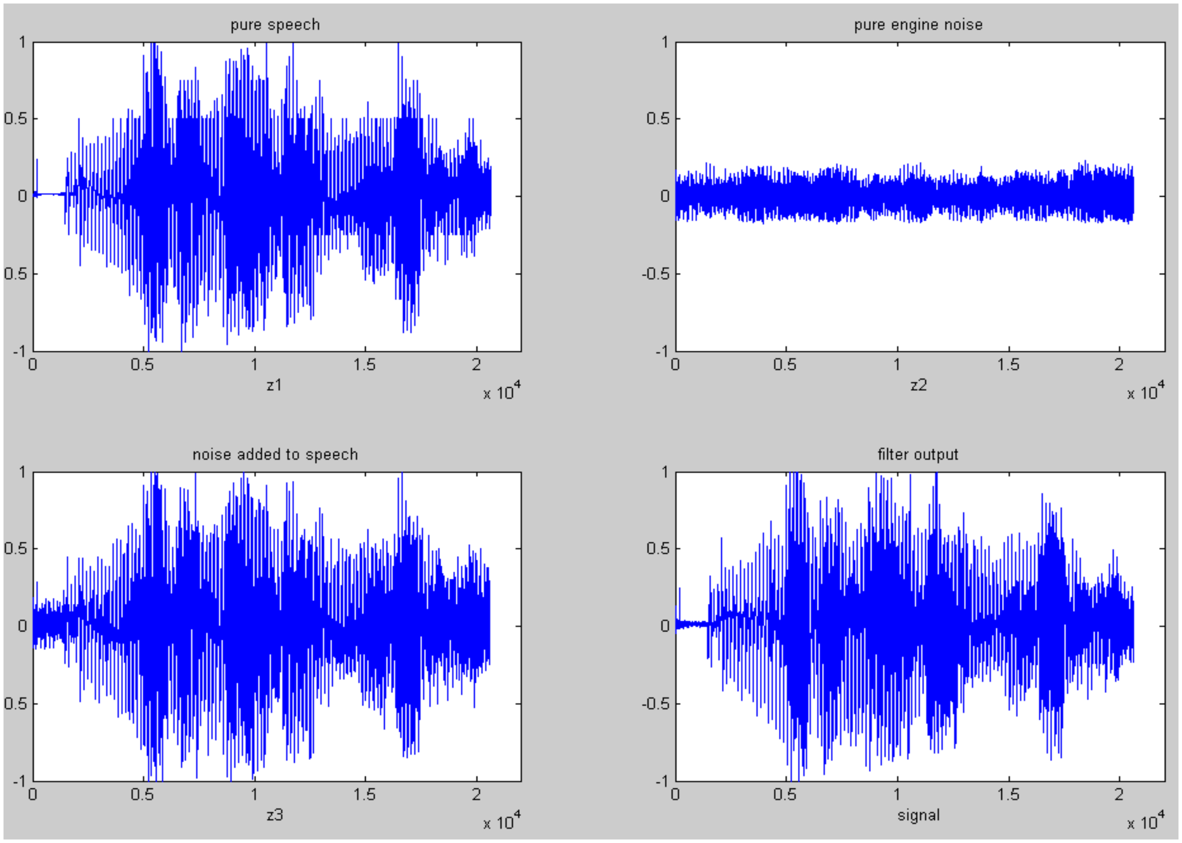
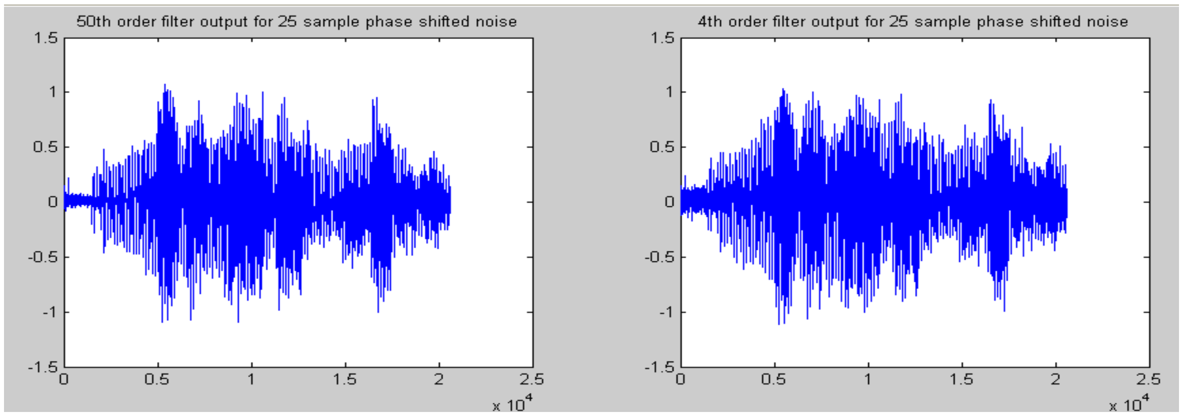


Figure 7.9 Matlab 4th filter output when the noise source Is produced by an engine without phase shift.



Figures 7.10 Matlab 4th and 50th order filter outputs for phase shifted noise.

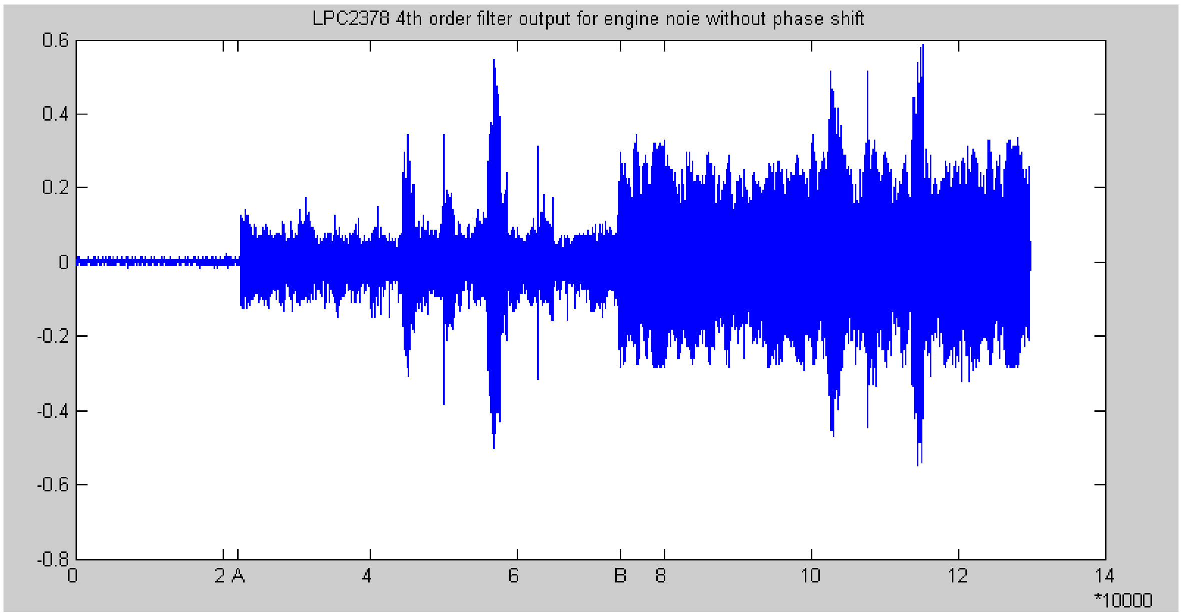


Figure 7.11 LPC2378 4th output filter when there is no phase shift in noise.
If there is a phase shift in the noise, the LPC2378 filter can not filter the noise.
At moment of A noise is started and then 4 characters (A, B, C and D) are
Read by the speaker and at moment B the filter is stopped to see how much noise
We have without the filter and again the 4 characters are read by the speaker.

It is important to know that a higher order filter is not always a better solution, according to what we experienced for a shifted phase noise, the optimum order of the filter is calculated by the number of samples shifted multiplied by two. Figure 7.12 is a proof for this statement, but it is important to know that the previous statement is from experimental results.

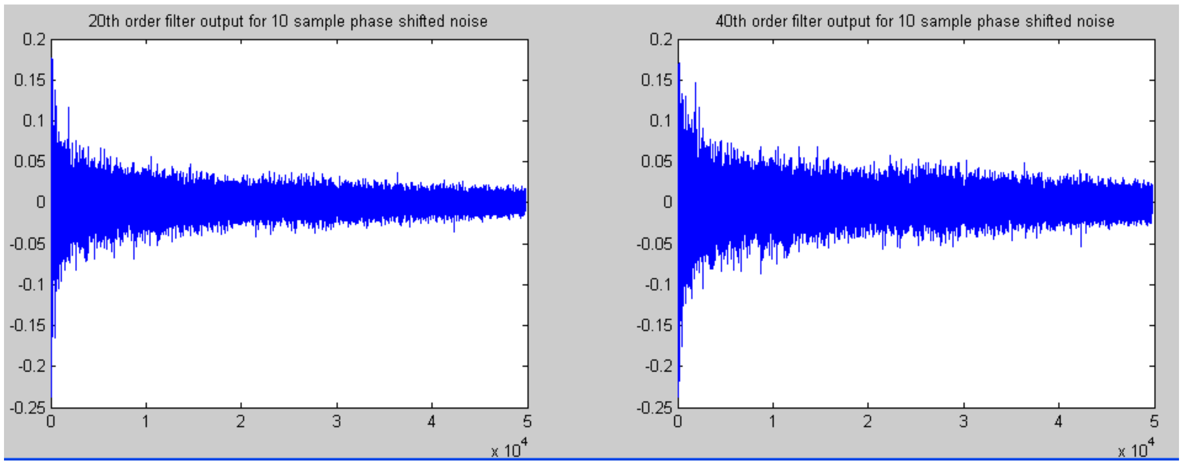


Figure 7.12: Comparison of a 40th and 20th order filter output for a 10 sample shifted noise signal.

7.2.3 **Changes in the Noise Frequency**

Another problem, besides phase shift, that we faced is the variation within the frequency of the noise. It means that for different noise frequencies, the efficiency of the filter changes. Figure 7.13 shows the practical results of the output of the 4th order filter when the frequency of the noise goes from 100Hz to 1000Hz and again returns to 100Hz using a sinus wave signal with an amplitude of 440 mV peak-to-peak.

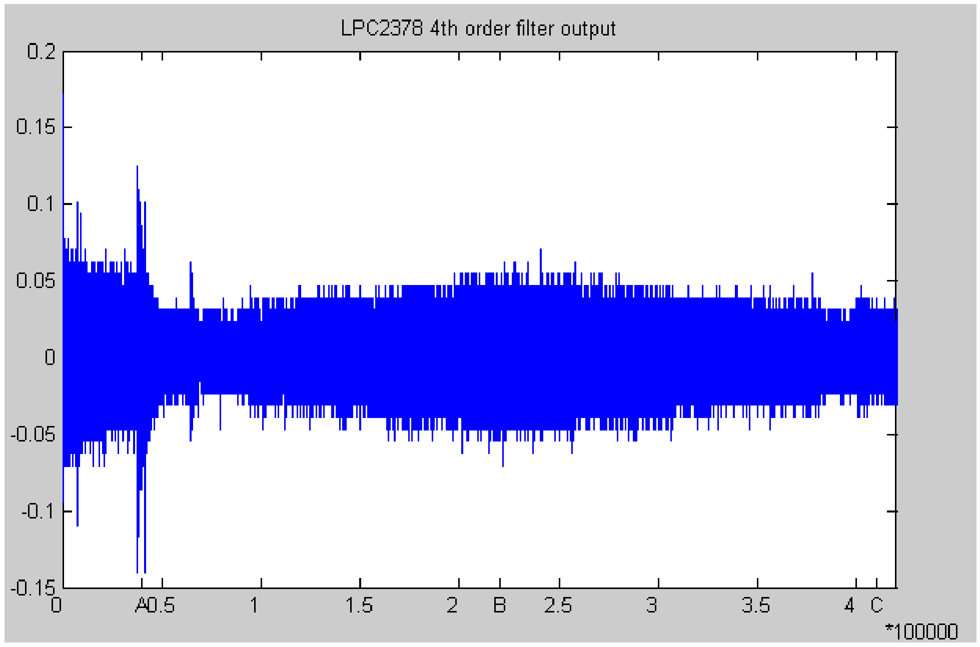


Figure 7.13 One can see that when the frequency of the noise increases, the Efficiency of the filter reduces. At point A the frequency of noise is 100Hz, at Point B frequency reaches 1000Hz, and at point C it returns to 100Hz.

Chapter 8   Conclusions

After implementing the LMS algorithm by using the ARM LPC2378 processor and testing it under several conditions, one can conclude the following aspects:

1. If the noise found in the signal channel has a phase shift in relation to the noise channel (figure 7.3), then one needs a higher order filter to be able to cancel the noise. The optimum order of the filter depends on the of phase (number of samples) shift of the signals.
2. The efficiency of the filter is a function of the noise frequency; this means that with a higher noise frequency, one will have a low efficiency filter. For reducing this problem our suggestion is to increase the sampling rate of the Analog to Digital Converter.
3. Although the LPC2378 is a 32bit microcontroller and is much more powerful than a simple 8-bit microcontroller, it is still not a good processor for filter applications. For filter applications DSPs are recommended, like the TI DSPs, which are optimized for filter and mathematical applications.

References

[1] The Olimex company website, *Schematic for LPC2378-STK.* http://olimex.com/dev/images/LPC2378-STK-sch.gi

[2] The Olimex company website, *LPC2378-STK Development Board.* *http://www.olimex.com/dev/index.html*

[3] The Olimex company website, *DSP Processors.* *http://www.olimex.com/dev/index.html*

[4] The Olimex company website, *ARM Controllers.* http://www.olimex.com/dev/index.html.

[5] Leonid Ryzhyk. *The ARM Architecture*. Chicago University, Illinois, U.S.A. http://www.cse.unsw.edu.au/~cs9244/06/seminars/08-leonidr.pdf

[6] UM10211 LPC23XX User manual http://www.nxp.com/acrobat_download/usermanuals/UM10211_2.pdf

[7] NXP LPC2378 Data Sheet http://www.keil.com/dd/docs/datashts/philips/lpc2378_ds.pdf

[8] S. Haykin, *Adaptive Filter Theory.* Fourth Edition. Upper Saddle River, NJ, USA: Prentice Hall 2002.

[9] Douglas E. Comer, *Computer Networks and Internets.* Fifth Edition. Upper Saddle River, NJ, USA: Prentice Hall 2009.

[10] C. Marven and Gillian Ewers, *A Simple Approach to Digital Signal Processing. New York, NY, USA:* Wiley-Interscience Publication 1996.

[11] H. Baher, *Analog and Digital Signal Processing.* Chichester, United Kindom: John Wiley and Sons 1990.

[12] Sven Nordebo. *Signal Processing Antennas.* September 18, 2004. http://w3.msi.vxu.se/~sno/ED4024/ED4024.pdf

[13] Sven Nordebo, *Signal Processing Antennas I.* January 19, 2004. http://w3.msi.vxu.se/~sno/ED4034/ED4034.pdf

[14] National Semiconductor. *LM741 Operational Amplifier Data Sheet.* http://www.national.com/ds/LM/LM741.pdf

[15] Microchip. *MCP6001/2/4 1 MHz Bandwidth Low Power Operation Amplifier Data Sheet. http://www.datasheetcatalog.org/datasheet/microchip/21733d.pdf*

Appendix 1 Main Routine C Source Code

```
/*****************************************************************************
 *****************************************************************************/
#include <nxp/iolpc2378.h>
#include "type.h"
#include "irq.h"
#include "target.h"
#include "uart.h"
//#include "fio.h"
#include "adc.h"
#include "board.h"

extern volatile DWORD UART0Count;
extern volatile BYTE UART0Buffer[BUFSIZE];
extern volatile DWORD ADC0Value[];
extern volatile DWORD ADC0IntDone;
extern volatile unsigned char chan_no;
extern volatile signed int ADC_Data2;
extern volatile signed int ADC_Noise;
BOOL AD_ready_N = 1 ;
BOOL AD_ready_S = 1 ;
extern volatile BYTE UART0TxEmpty ;
volatile  signed char ADC_signal;
volatile  signed char ADC_noise[50] ;
volatile  double  filter_weight[4] ;
extern volatile unsigned int ADisOK  ;
volatile unsigned char ADC_Da ;
volatile double u_noise = 0 ;
volatile double u_under_new = 0 ;
BOOL filte_on_off = 1 ;
/*****************************************************************************
**   Main Function  main()
*****************************************************************************/
int main (void)
{
  TargetResetInit();
  /* Initialize ADC  */
  ADCInit( ADC_CLK );
    //USB_LINK_LED_FIO = 0 ;
  UARTInit(115200); /* baud rate setting */
  UART0Count = 1 ;
  //GPIOInit( 0, REGULAR_PORT, DIR_IN, B2_MASK);
  FIO0DIRU &= 0xDFFBFFFF ;
  //FIO4PIN = 0x0 ;
  FIO4DIR3 = 0x0 ;
       u_under_new = 800;
       while(1)
       {
        if(AD_ready_N) ADC0Read( 1 );           //speech is sampled
        if(((FIO0PIN)& 0x20000000) == 0 ) filte_on_off = 1 ;
        if(((FIO0PIN)& 0x00040000) == 0 ) filte_on_off = 0 ;
        if((AD_ready_S))
        {
          AD_ready_N = 0 ;
          AD_ready_S = 0 ;

          unsigned int index = 0 ;
          signed char  yhat  = 0 ;
          while(index != 3)
          {
            ADC_noise[index+1] = ADC_noise[index] ;
            yhat = yhat + (ADC_noise[index+1] * filter_weight[index+1])/100;
            index = index +1 ;
          }
          ADC_signal = ADC_Data2 ;
          ADC_noise[0] = ADC_Noise ;
          ADC0Read( 0 );             // noise is sampled
          //while  (!(AD_ready_N));

          yhat = yhat + (ADC_noise[0] * filter_weight[0])/100;

          u_under_new  = u_under_new + ADC_noise[0]*ADC_noise[0] - ADC_noise[49]*ADC_noise[49] ;
          while(index != 49)
          {
            ADC_noise[index+1] = ADC_noise[index] ;
            index = index +1 ;
          }

          u_noise = u_under_new/ 2 ;
          if(u_under_new == 0 ) u_noise = 1 ;

          signed int filterd_speech = 0 ;
          filterd_speech = ADC_signal - yhat;
          //if(((FIO0PIN)& 0x00040000) == 0 ) ADC_Da =  ADC_signal;
          if(filte_on_off) ADC_Da = filterd_speech;
          else ADC_Da = ADC_signal ;

          index = 0 ;
          while(index != 4)
          {
            filter_weight[index] = filter_weight[index] + (filterd_speech*ADC_noise[index])/u_noise;  //For 7 seconds
of sampling 35000
            index= index +1 ;
          }
          while ( !(UART0TxEmpty & 0x01) );
          if(ADisOK == 4)
          {
            U0IER = IER_THRE | IER_RLS;              /* Disable RBR */
            UARTSend( (BYTE *)ADC_Data2, 1 ); //it was ADC_Data2
            U0IER = IER_THRE | IER_RLS | IER_RBR; /* Re-enable RBR */

          }

        }
       }
    //}

  return 0;
}

/******************************************************************************
**                      End Of File
******************************************************************************/
```

Appendix 2 ADC Routine C Source Code

```
/****************************************************************************
****************************************************************************/
#include <nxp/iolpc2378.h>
#include "type.h"
#include "irq.h"
#include "target.h"
#include "adc.h"
#include "board.h"
#include <intrinsics.h>

volatile DWORD ADC0Value[ADC_NUM];
volatile DWORD ADC0IntDone = 0;
volatile signed int ADC_Data2 = 9 ;
volatile signed int ADC_Noise = 9 ;
volatile unsigned char chan_no = 0 ;
#if ADC_INTERRUPT_FLAG
extern BOOL AD_ready_N  ;
extern BOOL AD_ready_S  ;
extern volatile signed char ADC_signal ;
extern volatile signed char ADC_noise[50] ;
volatile unsigned int ADisOK = 0  ;
/*****************************************************************************
** Function name:   ADC0Handler
**
** Descriptions:    ADC0 interrupt handler
**
** parameters:      None
** Returned value:    None
**
****************************************************************************/
__irq __nested __arm void ADC0Handler (void)
{
DWORD regVal;

  __enable_interrupt();       /* handles nested interrupt */

 regVal = ADSTAT;            /* Read ADC will clear the interrupt */
 if ( regVal & 0x0000FF00 )  /* check OVERRUN error first */
 {
   regVal = (regVal & 0x0000FF00) >> 0x08;
   /* if overrun, just read ADDR to clear */
   /* regVal variable has been reused. */
   switch ( regVal )
   {
   case 0x01:
     regVal = ADDR0;
     break;
   case 0x02:
     regVal = ADDR1;
     break;
   case 0x04:
     regVal = ADDR2;
     break;
   case 0x08:
     regVal = ADDR3;
     break;
   case 0x10:
     regVal = ADDR4;
     break;
   case 0x20:
     regVal = ADDR5;
     break;
   case 0x40:
     regVal = ADDR6;
```

```
    break;
  case 0x80:
    regVal = ADDR7;
    break;
  default:
    break;
  }
  AD0CR &= 0xF8FFFFFF;  /* stop ADC now */
  ADC0IntDone = 1;
  return;
 }

 if ( regVal & ADC_ADINT )
 {
  //chan_no = (AD0GDR >> 24) & 0x7 ;
  if( ADisOK != 4)
  {

    ADisOK = ADisOK + 1   ;
  }
  switch ( regVal & 0xFF )  /* check DONE bit */
  {
  case 0x01:
    ADC0Value[0] = ( ADDR0 >> 8 ) & 0x0FF;
    ADC_Noise = ADC0Value[0]- 127 ;
    AD_ready_N = 1 ;                        // noise is sampled
    break;
  case 0x02:
    ADC0Value[1] = ( ADDR1 >> 8 ) & 0x0FF;
                       // speech is sampled
    ADC_Data2 = ADC0Value[1] -127 ;
    AD_ready_S = 1 ;
    break;
  case 0x04:
    ADC0Value[2] = ( ADDR2 >> 8 ) & 0xFF;
    //ADC_Data2 = ADC0Value[2] -127 ;
    //AD_ready_S = 1 ;
    break;
  case 0x08:
    ADC0Value[3] = ( ADDR3 >> 8 ) & 0x0FF;
     ADC_Data2 = ADC0Value[3] ;
    break;
  case 0x10:
    ADC0Value[4] = ( ADDR4 >> 8 ) & 0x0FF;
    ADC_Data2 = ADC0Value[4] ;
    break;
  case 0x20:
    ADC0Value[5] = ( ADDR5 >> 8 ) & 0xFF;
    ADC_Data2 = ADC0Value[5] ;
    break;
  case 0x40:
    ADC0Value[6] = ( ADDR6 >> 6 ) & 0x3FF;
    break;
  case 0x80:
    ADC0Value[7] = ( ADDR7 >> 6 ) & 0x3FF;
    break;
  default:
    break;
  }
  AD0CR &= 0xF8FFFFFF;  /* stop ADC now */
  ADC0IntDone = 1;
 }
 VICADDRESS = 0;   /* Acknowledge Interrupt */
}
#endif

/*****************************************************************************
```

```
** Function name:   ADCInit
**
** Descriptions:    initialize ADC channel
**
** parameters:      ADC clock rate
** Returned value:    true or false
**
*****************************************************************************/
DWORD ADCInit( DWORD ADC_Clk )
{
  /* Enable CLOCK into ADC controller */
  PCONP |= (1 << 12);

  /* all the related pins are set to ADC inputs, AD0.0~7 */
  PINSEL0 |= 0x0F000000;  /* P0.12~13, A0.6~7, function 11 */
  PINSEL1 &= ~0x003FC000; /* P0.23~26, A0.0~3, function 01 */
  PINSEL1 |= 0x00154000;
  PINSEL3 |= 0xF0000000;  /* P1.30~31, A0.4~5, function 11 */

  AD0CR = ( 0x01 << 0 ) |   /* SEL=1,select channel 0~7 on ADC0 */
        ( ( Fpclk / ADC_Clk - 1 ) << 8 ) |  /* CLKDIV = Fpclk / 1000000 - 1 */
        ( 0 << 16 ) |     /* BURST = 0, no BURST, software controlled */
        ( 0 << 17 ) |     /* CLKS = 0, 11 clocks/10 bits */
        ( 1 << 21 ) |     /* PDN = 1, normal operation */
        ( 0 << 22 ) |     /* TEST1:0 = 00 */
        ( 0 << 24 ) |     /* START = 0 A/D conversion stops */
        ( 0 << 27 );      /* EDGE = 0 (CAP/MAT singal falling,trigger A/D conversion) */

  /* If POLLING, no need to do the following */
#if ADC_INTERRUPT_FLAG
  ADINTEN = 0x1FF;    /* Enable all interrupts */
  if ( install_irq( ADC0_INT, (void *)ADC0Handler, HIGHEST_PRIORITY ) == FALSE )
  {
    return (FALSE);
  }
#endif
  return (TRUE);
}

/*****************************************************************************
** Function name:   ADC0Read
**
** Descriptions:    Read ADC0 channel
**
** parameters:      Channel number
** Returned value:    Value read, if interrupt driven, return channel #
**
*****************************************************************************/
DWORD ADC0Read( BYTE channelNum )
{
#if !ADC_INTERRUPT_FLAG
DWORD regVal, ADC_Data;
#endif

  /* channel number is 0 through 7 */
  if ( channelNum >= ADC_NUM )
  {
    channelNum = 0;   /* reset channel number to 0 */
  }
  AD0CR &= 0xFFFFFF00;
  AD0CR |= (1 << 24) | (1 << channelNum);
  /* switch channel,start A/D convert */
#if !ADC_INTERRUPT_FLAG
  while ( 1 )     /* wait until end of A/D convert */
  {
    regVal = *(volatile unsigned long *)(AD0_BASE_ADDR
    + ADC_OFFSET + ADC_INDEX * channelNum);
    /* read result of A/D conversion */
    if ( regVal & ADC_DONE )
    {
      break;
    }
  }

  AD0CR &= 0xF8FFFFFF;  /* stop ADC now */
  if ( regVal & ADC_OVERRUN ) /* save data when it's not overrun, otherwise, return zero */
  {
    return ( 0 );
  }
  ADC_Data = ( regVal >> 6 ) & 0x3FF;
  return ( ADC_Data );  /* return A/D conversion value */
#else
  return ( channelNum );  /* if it's interrupt driven, the ADC reading is
                          done inside the handler. so, return channel number */
#endif
}

/******************************************************************************
**                            End Of File
******************************************************************************/
```

Appendix 3 UART Routine C Source Code

```
/*****************************************************************************
******************************************************************************/
#include <nxp/iolpc2378.h>
#include "type.h"
#include "target.h"
#include "irq.h"
#include "uart.h"
#include "board.h"
#include <intrinsics.h>

#include "timer.h"

volatile DWORD UART0Status;
volatile BYTE UART0TxEmpty = 1;
volatile BYTE UART0Buffer[BUFSIZE];
volatile DWORD UART0Count = 0;
extern volatile DWORD ADC0Value[];

extern volatile unsigned char ADC_Da ;
/*****************************************************************************
** Function name:   UART0Handler
**
** Descriptions:    UART0 interrupt handler
**
** parameters:      None
** Returned value:    None
**
*****************************************************************************/
__irq __nested __arm void UART0Handler (void)
{
BYTE IIRValue, LSRValue;
volatile BYTE Dummy;

  __enable_interrupt();   /* handles nested interrupt */

  IIRValue = U0IIR;
  IIRValue >>= 1;     /* skip pending bit in IIR */
  IIRValue &= 0x07;     /* check bit 1~3, interrupt identification */
  if ( IIRValue == IIR_RLS )    /* Receive Line Status */
  {
    LSRValue = U0LSR;
    /* Receive Line Status */
    if ( LSRValue & (LSR_OE|LSR_PE|LSR_FE|LSR_RXFE|LSR_BI) )
    {
      /* There are errors or break interrupt */
      /* Read LSR will clear the interrupt */
      UART0Status = LSRValue;
      Dummy = U0RBR;    /* Dummy read on RX to clear interrupt, then bail out */
      VICADDRESS = 0; /* Acknowledge Interrupt */
      return;
    }
    if ( LSRValue & LSR_RDR ) /* Receive Data Ready */
    {
      /* If no error on RLS, normal ready, save into the data buffer. */
      /* Note: read RBR will clear the interrupt */
      UART0Buffer[UART0Count] = U0RBR;
      UART0Count++;
      if ( UART0Count == BUFSIZE )
      {
        UART0Count = 0; /* buffer overflow */
      }
    }
  }
  else if ( IIRValue == IIR_RDA ) /* Receive Data Available */
  {
   /* Receive Data Available */
   UART0Buffer[UART0Count] = U0RBR;
   UART0Count++;
   if ( UART0Count == BUFSIZE )
   {
    UART0Count = 0; /* buffer overflow */
   }
  }
  else if ( IIRValue == IIR_CTI ) /* Character timeout indicator */
  {
   /* Character Time-out indicator */
   UART0Status |= 0x100; /* Bit 9 as the CTI error */
  }
  else if ( IIRValue == IIR_THRE )  /* THRE, transmit holding register empty */
  {
   /* THRE interrupt */
   LSRValue = U0LSR; /* Check status in the LSR to see if valid data in U0THR or not */
   if ( LSRValue & LSR_THRE )
   {
    UART0TxEmpty = 1;
   }
   else
   {
    UART0TxEmpty = 0;
   }
  }

  VICADDRESS = 0;          /* Acknowledge Interrupt */
  return;
}

/*****************************************************************************
** Function name:   UARTInit
**
** Descriptions:    Initialize UART0 port, setup pin select,
**          clock, parity, stop bits, FIFO, etc.
**
** parameters:      UART baudrate
** Returned value:    true or false, return false only if the
**          interrupt handler can't be installed to the
**          VIC table
**
*****************************************************************************/
DWORD UARTInit( DWORD baudrate )
{
PINSEL0 = 0x00000050;       /* RxD0 and TxD0 */

  U0LCR = 0x83;   /* 8 bits, no Parity, 1 Stop bit */
  //Fdiv = ( Fpclk / 16 ) / baudrate ;  /*baud rate */
  //U0DLM = Fdiv / 256;
  //U0DLL = Fdiv % 256;
  U0DLM = 0 ;
  U0DLL = 7 ;
  U0FDR = 0x45 ;
  U0LCR = 0x03;   /* DLAB = 0 */
  U0FCR = 0x07;   /* Enable and reset TX and RX FIFO. */

  if ( install_irq( UART0_INT, (void *)UART0Handler, HIGHEST_PRIORITY ) == FALSE )
  {
   return (FALSE);
  }

  U0IER = IER_RBR | IER_THRE | IER_RLS; /* Enable UART0 interrupt */
  return (TRUE);
}
```

```
/*****************************************************************************
** Function name:   UARTSend
**
** Descriptions:    Send a block of data to the UART 0 port based
**         on the data length
**
** parameters:      buffer pointer, and data length
** Returned value:    None
**
*****************************************************************************/
void UARTSend(BYTE *BufferPtr, DWORD Length )
{
  while ( Length != 0 )
  {
    /* THRE status, contain valid data */
    ADC_Da += 128 ;
    U0THR = ADC_Da ;

    //  U0THR = (BYTE)ADC_noise[9] ;
    //U0THR = *BufferPtr;
    //U0THR = 0x55 ;
      UART0TxEmpty = 0; /* not empty in the THR until it shifts out */
    //BufferPtr++;
    Length--;
  }
  return;
}

/******************************************************************************
**                        End Of File
******************************************************************************/
```

Appendix 4 Matlab Simulation Code for LMS Algorithm

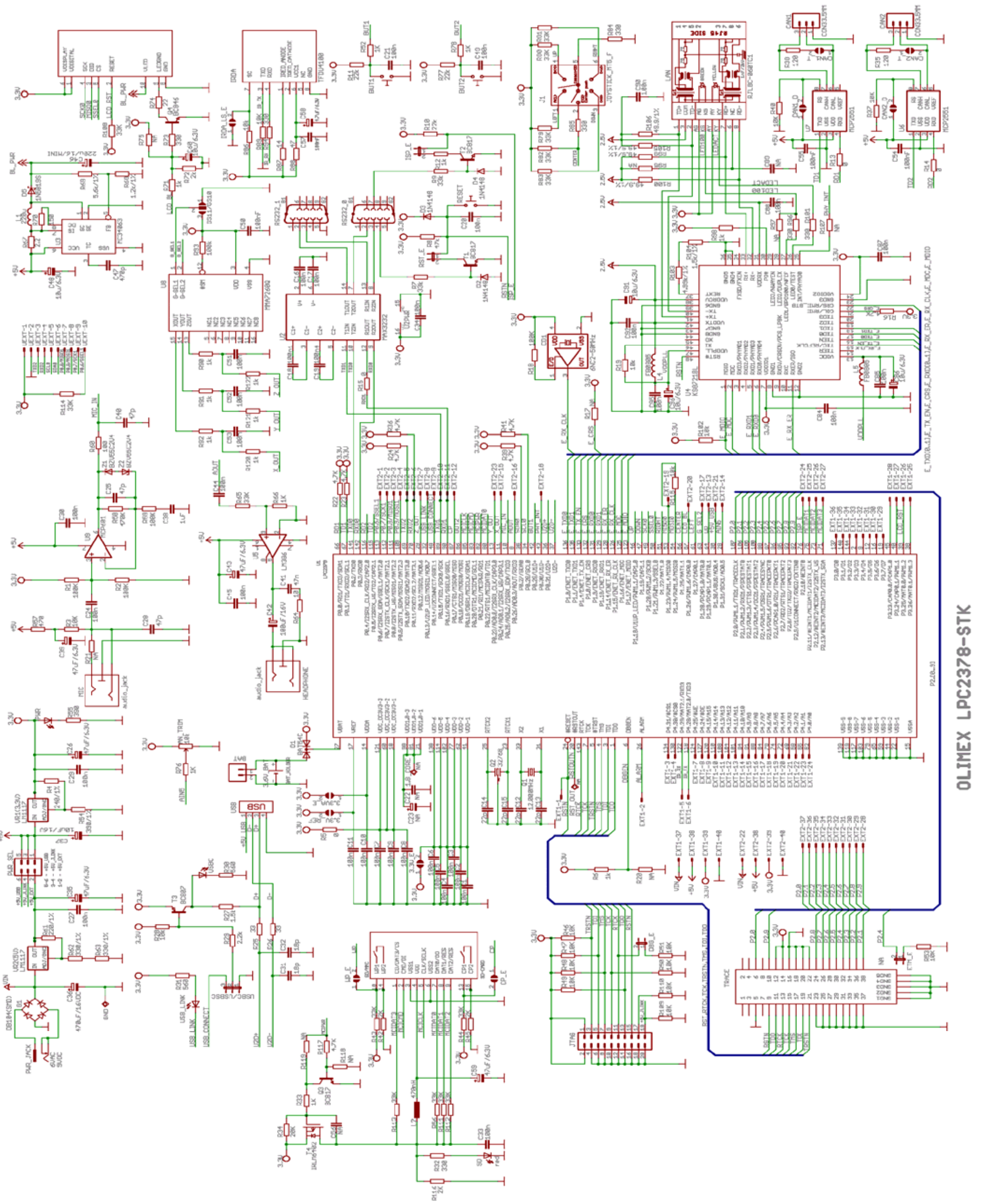
```
clear all
close all
u0 = 0.03 ;
[z1,Fs1,bits1] = wavread('no.wav');
[z2,Fs2,bits2] = wavread('sin_noise2.wav');                    %noise
 [z3,Fs3,bits3] = wavread('sin_name2.wav');                   %noisy signal
   Nx(1,:) = size(z2) ;
   N = 4 ;
   u = u0/(N*(z2'*z2)/Nx(1,1)) ;
   %z2 = z2 + 0.2;
   yhat  = zeros(1,Nx(1,1));
   %h=[0.9957; 0.0271 ; -0.0191 ; 0.0289 ; -0.0137 ; 0.0075 ; 0.0133 ; -0.0137 ; 0.0207 ; -0.0050];

   h     = zeros(N,1);
   signal= zeros(1,Nx(1,1));
   for k=1:N
      X(k)=z2(k,:);
   end
   for count= 1:Nx(1,:)-0 ;
      for n = N:-1:2
         X(n)=X(n-1);
      end
      X(1) = z2(count) ;
      yhat =  X*h ;
      signal(count)= z3(count+0) - yhat;
      h = h+ 2*u*signal(count)*X' ;
   end
   figure(2)
   subplot(2,2,1);plot((h))
   xlabel('z1')
   subplot(2,2,2);plot(z3)
   xlabel('z3')
   axis([0 22500 -1 1])
   subplot(2,2,4);plot(signal)
   xlabel('error')
   axis([0 22500 -1 1])
   subplot(2,2,3);plot(z2)
   xlabel('z2')
   axis([0 22500 -1 1])
soundsc(z3,Fs1)                         %this is mixture of the two first signals
pause(3.5)
soundsc(signal,Fs1)                     %this signal is filtered and tries to retrieve the first signal
pause(3.5)
```

Appendix 5 Matlab Code for Generating the Wav File

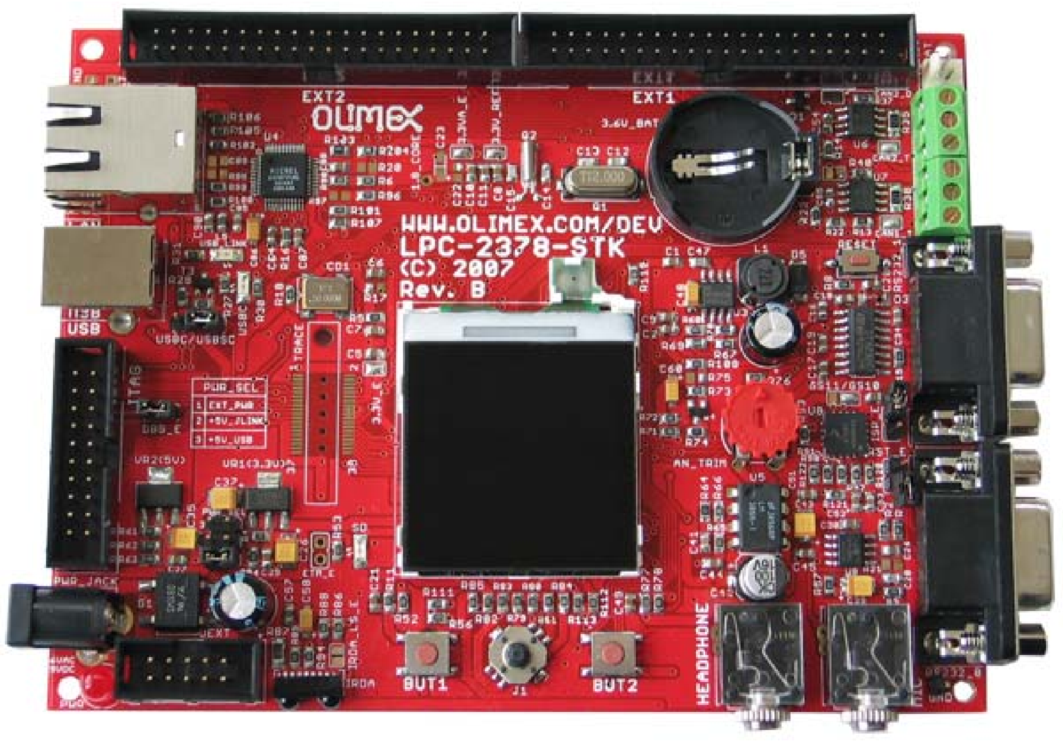
```
file=fopen('test1.txt')                    %opening the saved file
h = fread(file);                           %reading the saved file as data
h3 = (h-128)/128 ;                         %converting the data to a .wav file spectrum
plot(h3)
fclose(file)
wavwrite(h3,sampling_frequency,'test')     %recording the read data as .wav file
                                           %here sampling frequency is 7000 Hz
```

Appendix 6 LPC2378-STK Schematic [1]



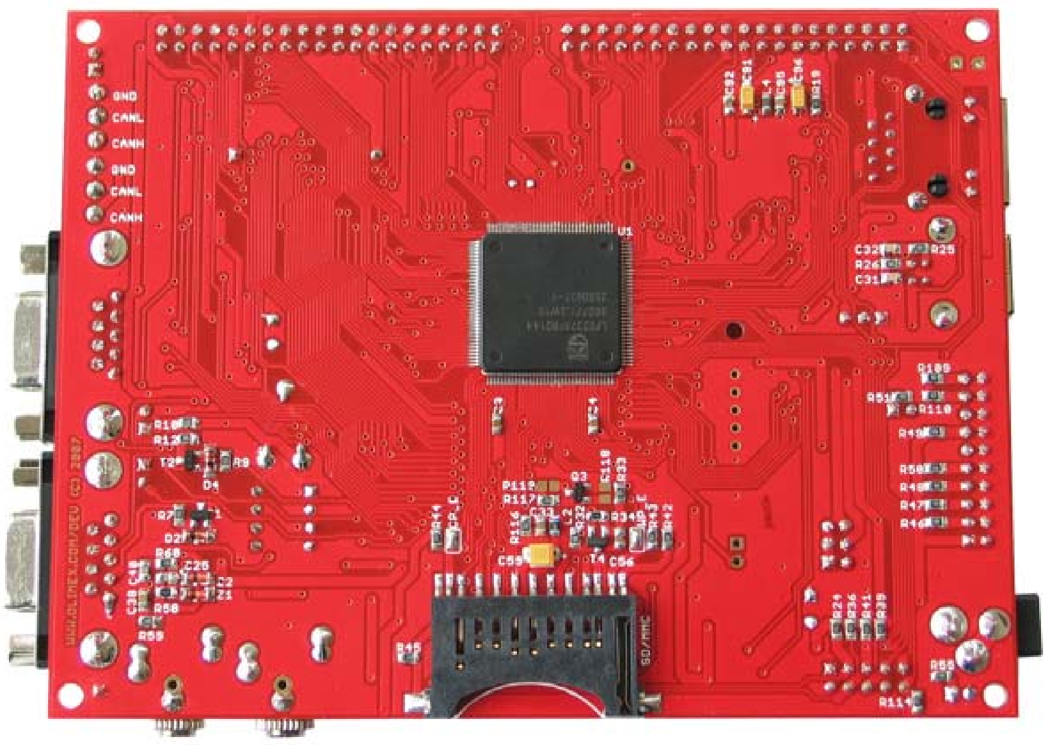
http://olimex.com/dev/images/LPC2378-STK-sch.gi

Appendix 7 LPC2378-STK Circuit Board Front View[2]



The Olimex company website, *LPC2378-STK Development Board.*
http://www.olimex.com/dev/index.html

Appendix 8 LPC2378-STK Circuit Board Back View[2]



The Olimex company website, *LPC2378-STK Development Board.*
<u>*http://www.olimex.com/dev/index.html*</u>

Appendix 9 Analog Circuit