

**University of Southern Queensland**  
**Faculty of Engineering and Surveying**

**A touch-screen controlled “Linear  
Predictive Synthesizer” for  
accessibility applications**

A dissertation submitted by

**Mr. Benjamin W. D. Jordan**

in fulfilment of the requirements of

**Courses ENG4111 and ENG4112 Research Project**

towards the degree of

**Bachelor of Engineering (Computer Systems Engineering)**

Submitted: October, 2009

# Abstract

Numerous voice compression methods are available today for communications over low bandwidth channels. Worthy of note in particular are Linear Predictive Coding (LPC), Mixed Excitation LPC (MELP), and Code Excited LPC (CELP). The channel in these coding schemes is typically a digital transmission line or radio link, such as in cellular telephone communications, but may be other media such as files on a computer hard disk.

Linear Predictive Coding is explored in some detail as a basis for creating a new speech synthesizer that does not convert text to speech (TTS), but rather uses a touch-screen Thin Film Transistor (TFT) panel as user input to create and control voice-like audio sound synthesis.

Research has been carried out to conceptually try different methods for mapping TFT touch panel input (or any 2-dimensional input) to LPC synthesis coefficient vectors for artificial speech reproduction.

To achieve this, various LPC coefficient quantization algorithms have been explored and evaluated using Octave v.3 scripts, resulting in selection and comparison in the final hardware and software implementation.

The hardware and software development platform used for the final implementation is the Altium Nanoboard 3000 Xilinx Edition, along with the Altium Designer EDA package. The Nanoboard 3000 was chosen as it provided a convenient FPGA platform and all the necessary IP, IP Synthesis, and C compilers needed to prototype the design and perform further research.

**University of Southern Queensland**  
**Faculty of Engineering and Surveying**

ENG4111/2 Research Project

**Limitations of Use**

The Council of the University of Southern Queensland, its Faculty of Engineering and Surveying, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Engineering and Surveying or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled “Research Project” is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.



Prof F Bullen

Dean

Faculty of Engineering and Surveying

# Certification

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

**Benjamin W. D. Jordan**

**Student Number: 0031210722**

---

Signature

---

Date

# Acknowledgments

“The preparations of the heart belong to man, But the answer of the tongue is from the LORD... Commit your works to the LORD, and your thoughts will be established.” (Prov. 16:1,3 NKJV)

First and foremost, regardless of the outcomes of any work undertaken I must express my deepest gratitude to God almighty because in spite of my shortcomings, bad habits and weaknesses, He has consistently shown his faithfulness in helping me overcome these barriers.

I would like to extend thanks to all the staff at the University of Southern Queensland, but in particular my project supervisor Mr. Mark Phythian and Assoc. Prof. Dr. John Leis for their help and exceptional care of the pedagogy of Electronics, Computing and Signal Processing.

Due thanks must go to my employer, Altium Limited, for providing me access to corporate resources (including hardware and software) used to implement the designs. I would like to specifically thank my manager Matthew Schwaiger for allowing me to be flexible with work commitments from time to time, and my colleague Dr. Marty Hauff for his encouragement and prayers.

Finally, it is most important to acknowledge the overwhelming love and support extended to me by my wife Erin, and our children Delta, Seth and the new addition Shiloh – born only weeks before submission.

BENJAMIN JORDAN

*The University of Southern Queensland, October 2009*

# Contents

Abstract .....	ii
Limitations of Use.....	iii
Certification .....	iv
Acknowledgments.....	v
Contents .....	vi
Table of Figures .....	xiii
Acronyms .....	xv
Chapter 1.....	1
Introduction .....	1
1.1. Objectives .....	1
1.2. Speech Synthesis Concepts .....	2
1.3. Linear Predictive Coding.....	2
1.4. FPGA System on Chip Implementation .....	3
1.5. Dissertation Structure .....	3
Chapter 2.....	5

---

Literature Review .....	5
2.1. Introduction .....	5
2.2. Linear Predictive Coding (LPC).....	7
2.3. Mixed-Excitation LPC (MELP) .....	8
2.4. Considerations of Language and Accent.....	8
2.5. Parameter Quantisation and Interpolation.....	10
Chapter 3.....	11
Speech Analysis and Modelling.....	11
3.1. Human Speech Organs .....	11
3.2. Insight Into Speech Analysis .....	13
3.3. Synthesis of Speech.....	14
3.4. Speech Transitions.....	15
3.4.1. Frame Interpolation.....	16
Chapter 4.....	18
Voice Coding .....	18
4.1. Introduction .....	18
4.2. Linear Predictive Coding.....	18
4.2.1. Linear Predictor .....	18
4.2.2. Inverse Predictor .....	20
4.2.3. Calculating Predictor Coefficients.....	21
4.3. LPC Initial Results .....	23

---

4.4. LPC Parameter Quantization.....	25
4.5. Conclusion.....	26
Chapter 5.....	27
Morphing Formants across the TFT Panel.....	27
5.1. Introduction .....	27
5.2. User Interface research .....	28
5.3. Morphing Across The TFT.....	29
5.4. Interpolation of Coefficients.....	30
5.5. Introducing LSPs.....	31
5.5.1. Computing the Line Spectrum Pairs.....	31
5.6. Conclusion.....	34
Chapter 6.....	35
Practical Implementation .....	35
6.1. Introduction .....	35
6.2. LPC Analysis Function.....	36
6.3. LSP Calculation.....	37
6.3.1. Roots of the Line Spectrum Pairs .....	37
6.3.2. Interpolation Using LSFs.....	38
6.3.3. Expanding Roots back to LPC Coefficients .....	40
6.4. Tying Interpolation Together .....	41
6.5. Formant Mapping .....	42



---

6.6.	Embedded System Considerations .....	45
6.7.	Fixed Point Implementation .....	45
6.7.1.	Coefficient Scaling .....	46
6.7.2.	Truncation Effects.....	46
6.8.	C Code Development .....	49
6.8.1.	Code for generating Coefficients.....	49
6.8.2.	Drivers and Initialization .....	50
6.8.3.	IIR Synthesis Filter .....	51
6.8.4.	Pulse Source.....	52
6.8.5.	Noise Source.....	53
6.8.6.	User Interface.....	53
6.9.	Conclusion.....	54
	Chapter 7.....	55
	Introduction to the Nanoboard 3000 .....	55
7.1.	Introduction .....	56
7.2.	NB3000 and the Altium Designer Software Platform.....	56
7.3.	Nanoboard Features Utilized.....	57
7.3.1.	Audio Codec .....	57
7.3.2.	I <sup>2</sup> S Interface .....	58
7.3.3.	SPI Interface .....	58
7.3.4.	TFT Interface .....	58

CONTENTS	x
7.3.5. GPIO port, LEDs and Pushbuttons .....	59
7.3.6. SRAM Interface .....	59
7.4. FPGA Hardware Design.....	60
7.5. Project Links and Hierarchy .....	60
7.6. Conclusion.....	61
Chapter 8.....	62
User Interface Research .....	62
8.1. Introduction .....	62
8.2. Robotic Sound .....	62
8.3. User interface problems.....	63
8.4. Conclusion.....	63
Chapter 9.....	64
Conclusion .....	64
9.1. Introduction .....	64
9.2. Further work and research.....	65
9.2.1. Improve LSP interpolation method to include gain.....	65
9.2.2. Find better expression methods .....	65
9.2.1. Implement the LPC and LSP operations in Real-Time .....	65
9.2.2. Adapt the current design to Music generation.....	66
9.3. Conclusion.....	66
References .....	68

CONTENTS	xi
Appendix A	71
Project Specification	71
Appendix B	73
Octave Scripts and Functions	73
B.1 The <code>calc_lpc.m</code> Octave Function	74
B.2 The <code>lpc_gen_figs.m</code> Octave Function	75
B.3 The <code>generate_coeffs.m</code> Octave Function	77
B.4 The <code>ow_pole_mapping_plot.m</code> Octave Script	79
B.5 The <code>lsp_lpc.m</code> Octave Function	81
B.6 The <code>lpclsp.m</code> Octave Function	82
B.7 The <code>expnd.m</code> Octave Function	83
B.8 The <code>lpc_interp.m</code> Octave Function	84
B.9 The <code>lsp_interp.m</code> Octave Function	85
B.10 The <code>gen_all_lpc.m</code> Octave Function	87
B.11 The <code>gen_all_lsp.m</code> Octave Function	90
B.12 The <code>plot_interp.m</code> Octave Function	93
B.13 The <code>lpclsplpc.m</code> Octave Function	95
B.14 The <code>draw_unit_circle.m</code> Octave Function	97
Appendix C	98
Altium Designer FPGA Project Schematic and OpenBus Diagrams	98

CONTENTS	xii
C.1 FPGA Top-level Schematic Diagram .....	99
C.2 FPGA OpenBus System Block Diagram .....	100
C.3 Altium Designer FPGA Project Hierarchy .....	101
Appendix D .....	102
Altium Designer Embedded Project C Code Listings.....	102
D.1 The main.c File.....	103
D.2 The buttons.h Header File .....	115
D.3 The lpc_coeffs.h Header File.....	116
D.4 The lpc_lsp_interpolated_coeffs.h Header File .....	118
D.5 The devices.h Auto-Generated Header File .....	120
Appendix E .....	121
Nanoboard 3000 Data Sheet .....	121
E.1 NB 3000 Data Sheet.....	122

# Table of Figures

Figure 1 Block Diagram of a Formant-based speech synthesizer.....	7
Figure 2 MELP uses formant modeling with <i>mixed</i> sources. ....	8
Figure 3 Formant Spaces of Australian, British and American Accents. ( <i>IEEE Signal Processing Magazine, Vol. 26 No. 3 p72, May 2009</i> ) .....	9
Figure 4 Human Speech Organs (Fu Jen Catholic University Graduate Institute of Linguistics 2007, <i>The biological basis of speech production (2): The Vocal Tract and Related Speech Organs</i> , viewed 22 October 2009, < <a href="http://www.ling.fju.edu.tw/phonetic/mouth.gif">http://www.ling.fju.edu.tw/phonetic/mouth.gif</a> >).....	12
Figure 5 Spectrogram of the voiced sound “iya” .....	14
Figure 6 Linear Prediction FIR Filter.....	19
Figure 7 All-Pole Inverse Prediction Filter.....	21
Figure 8 LPC Analysis Predictor Error and Log Magnitude Spectrum of predictor polynomial. ....	23
Figure 9 Discrete Time waveforms of input and synthesized speech frame.....	24
Figure 10 Mapping a TFT Panel to Vowels and Pitch (Altium Ltd. 2009, NB2 TFT Panel Port Plug-In Library Component) .....	28
Figure 11 Z-plane Poles of a series of LPC frames from /æ/ to /U/. Blue poles mark the LPC coefficient poles of the start frame, red poles are from the final frame, and black in-between. ....	30
Figure 12 Roots of the Palindromic and Anti-palindromic Polynomials: LSPs .....	32

Figure 13 Roots of the Line Spectrum Pairs from /œ/ to /U/. ..... 34

Figure 14 Line Spectrum Frequency Interpolation Using Neighbouring Angles ..... 39

Figure 15 Interpolating Line Spectrum Frequencies over N frames..... 39

Figure 16 Root Expansion Algorithm..... 40

Figure 17 LPC Poles of Interpolation of coefficients using LSPs. .... 41

Figure 18 Log Magnitude Spectrum of LSP Interpolated LPC coefficients..... 42

Figure 19 Log Magnitude Spectrum of LSP interpolations across TFT panel width..... 43

Figure 20 Spectrogram bitmap of interpolation results used as TFT background..... 44

Figure 21 The Software Platform Builder..... 50

Figure 22. The NB3000 running the speech synthesizer. .... 55

Figure 23 Embedded Project Memory Configuration..... 59

Figure 24 Devices View in Altium Designer Software. This is where the FPGA and Embedded projects are downloaded to the target device..... 61

Figure 25 Top Level FPGA Project Schematic..... 99

Figure 26 FPGA OpenBus Block Diagram..... 100

Figure 27 Altium Designer Project Hierarchy ..... 101

# Acronyms

**LPC** Linear Predictive Coding – a method of analysing speech based on linear prediction, and then using the parameters obtained to synthesize it again.

**CELP** Code Excited LPC.

**MELP** Mixed-Excitation LPC.

**LSP** Line Spectrum Pairs – a method of representing LPC parameters using Palindromic and Antipalindromic polynomials

**LSF** Line Spectrum Frequencies – the angle of the roots of the LSPs

**CODEC** EnCOder DECoder – an audio Analogue to Digital converter and Digital-to-Analogue converter in the same package.

**I<sup>2</sup>S** Inter-IC Sound bus – a multiplexed serial interface for streaming audio A-to-D and D-to-A data.

**SPI** Serial Peripheral Interconnect bus – a serial interface used to connect peripherals to host microcontrollers.

**DMA** Direct Memory Acces – used by the TFT panel to gain access to the pixel display buffer.

**TFT** Thin-Film-Transistor (display/panel). The display used in this project is a 320 pixel wide by 240 pixel high panel, 2.4” from corner to coner diagonally.

**RISC** Reduced Instruction Set Computer – the CPU used in this design is a RISC processor, the Altium TSK3000A.

## Chapter 1.

# Introduction

Various speech generation systems are currently available as assistive technology. The majority are simply text-to-speech devices or software, such as the JAWS™, or simple direct speech graphical tablet style devices, such as the Static Display Speech Generators available from LifeTec Queensland.

A few problems with the current products and the approach they employ are:

- They are limited to specific language sounds (i.e. for English, French or German etc.).
- Static Display Speech Generators are limited to a pre-defined set of basic phrases.
- They do not allow for individual vocal creativity, emotions, or accents.
- They cater broadly to visually impaired, and mobility limited users, but are not particularly useful for people with a temporary speech loss.

### 1.1. Objectives

The objectives of this research project are to:



- Create an assistive device that offers an alternative means of translating the user's motor movement to audible speech.
- Develop an input method that allows more generic and abstract sound generation.
- Research the feasibility of touch-screens as the input in terms of ease of use, and flexibility.
- Assess the device's adaptability for multiple languages.

## 1.2. Speech Synthesis Concepts

The advancement of speech synthesis technology has been largely driven by the need to store or transmit voice data on noisy or low-bandwidth media. Applications have ranged from long range telecommunications to children's toys and text-to-speech devices. On the input to any good speech compression system is the analysis of the desired speech. A multitude of methods have been developed and more continue to be developed based on demand for higher quality sound intelligibility and reproduction, as well as improved human – machine user interfaces. The following are perhaps the most pervasively used methods and will be explored in some detail within this dissertation.

## 1.3. Linear Predictive Coding

Linear Predictive Coding (LPC) and Mixed-Excitation LP Coding (MELP) are well-known methods of analysing, coding and then decoding of speech signals for highly compressed telecommunications. The *decoding* used in these methods need not be fed by a bit stream of encoded speech, but could be fed by commands directly input from a user interface. This is the proposed method to be used by this project.

An LCD touch-panel and associated drivers will be used with an FPGA-based microprocessor SoC (System on Chip). This system will contain user interface software as well as the audio synthesis and filtering required for creating the desired sounds. User commands interpreted from the touch screen will control pulse and noise sources in software, which will be fed in turn through time-varying digital filters. This will be output on the system's built-in speakers.

#### **1.4. FPGA System on Chip Implementation**

The idea for a touch-screen controlled synthesizer initially came about when an FPGA development platform, the Altium NanoBoard-2, was being reviewed. This thought lead naturally into the concept of the assistive device for this project. Since the project commencement, Altium Limited has developed a new FPGA development platform called the NanoBoard 3000. Since then, the author has had considerable design experience through tutorial, video and reference design production to support its release to the electronics design industry. The NanoBoard 3000 has all the necessary peripherals for the touch screen synthesizer and therefore is a natural and sensible choice for prototyping the final design.

#### **1.5. Dissertation Structure**

This dissertation is structured in the following manner:

**Chapter 2** discusses the background information researched in the project, covering some historical aspects of speech synthesis as well as the underlying prior art behind the speech encoding and decoding mechanisms explored.

---

**Chapter 3** discusses the analysis of speech by practical means. The problem of modelling transient speech is introduced.

**Chapter 4** covers voice coding mechanisms in more detail. Specifically, it delves deeper into the topic of Linear Predictive Coding. This is then extended to discussion of how the LPC synthesis filter and source signals can be used as part of an assistive device such as the one developed.

**Chapter 5** explores concepts researched for mapping voiced sounds to a touch-screen (or any 2-dimensional) device, and the problems faced relating to time-varying the speech filter.

**Chapter 6** illuminates the practical implementation of the speech synthesizer. Infinite Impulse Response filter theory is discussed in light of the project regarding embedded systems, fixed point arithmetic, and hardware acceleration.

**Chapter 7** introduces the Altium Nanoboard-3000 Xilinx Edition FPGA development board, used in the final design. The Altium Designer software used to develop the FPGA hardware and embedded firmware for the design is also discussed, along with the actual design itself.

**Chapter 8** documents the brief research undertaken into how useable the designed speech synthesizer is, and considers the ability of the system to reproduce languages other than English.

**Chapter 9** concludes the project with a brief summary of what was discovered, what methods were chosen and the final result of the design, including some suggested improvements and further research that could be undertaken.

## Chapter 2.

# Literature Review

### 2.1. Introduction

According to Cole et. al. (1996) there are essentially three classes of speech synthesizer: Articulatory, Formant-based, and Concatenative. However, work has been done since then to develop Hidden Markov Model (HMM) based synthesis by Tokuda et. al. (2000) as well – an adaptation of the concatenative approach.

Articulatory synthesis seeks to mechanically or electronically model the specific movements of the speech organs. While this could potentially provide the most accurate sounding speech, the complexity of the system is somewhat prohibitive for design and use alike.

Formant based modelling attempts to simulate the resonances of the vocal tract and nasal cavities, accepting white noise or pulsed signals as an input to the filtering system.

The third class, concatenative synthesizers, requires a vast library of recorded speech segments (phonemes) which are essentially strung together to form words and sentences.

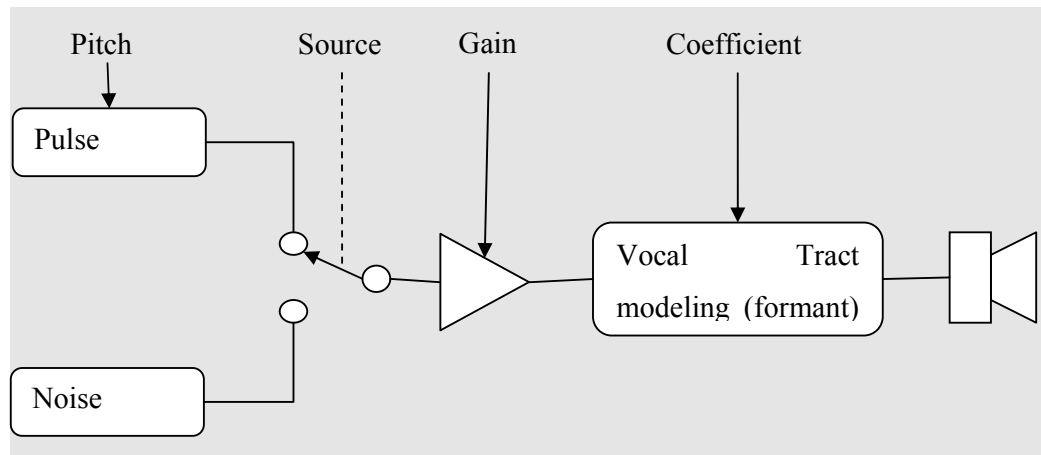
Much work has been done in the past to create mechanisms for synthesizing speech, yet the majority of effort appears to have been placed on finding appropriate ways to encode speech for compact storage or transmission over narrow bandwidth channels, or for text-to-speech (TTS) systems for the visually impaired (Breen, 1992). In terms of assistive devices, it is apparent that little effort has been directed towards those that purely generate speech from the user's motion or command input. One very early contraption of note however, was the VODER ('Voice Operating DEmonstrator') created by Homer Dudley in 1939 (Breen, 1992). The VODER used a noise source, pulse relaxation oscillator and 10 band-pass filters (Synthopia, 2009). While this was an entirely analogue design, it formed the general structure of synthesis used today in many speech coders such as the well-known LPC10 (Linear Predictive Coder, 10 Band).

Arguably, the most elegant approach of the three presented above is the formant-based synthesizer. This is due to the fact that it provides enough flexibility to produce many language sounds within reasonable limits of computing power available today.

Formant synthesizers typically use a pulse generator and a noise generator, and control the pitch and amplitude of these sources, passing the resulting waveform through a digital filter with time-varying response. The pulse source models the human vocal chords/folds whereas the noise source is used to model sounds of sibilants, clicks and pops which would be generated in speech by the constrictions of airflow by the tongue and glottis. The digital filter models the resonances (formants) of the vocal tract, nasal cavity and mouth of the speaker - the position in the

frequency domain of these formants have a profound effect, enabling creation of vowels in various languages. See the block diagram in Figure 1.

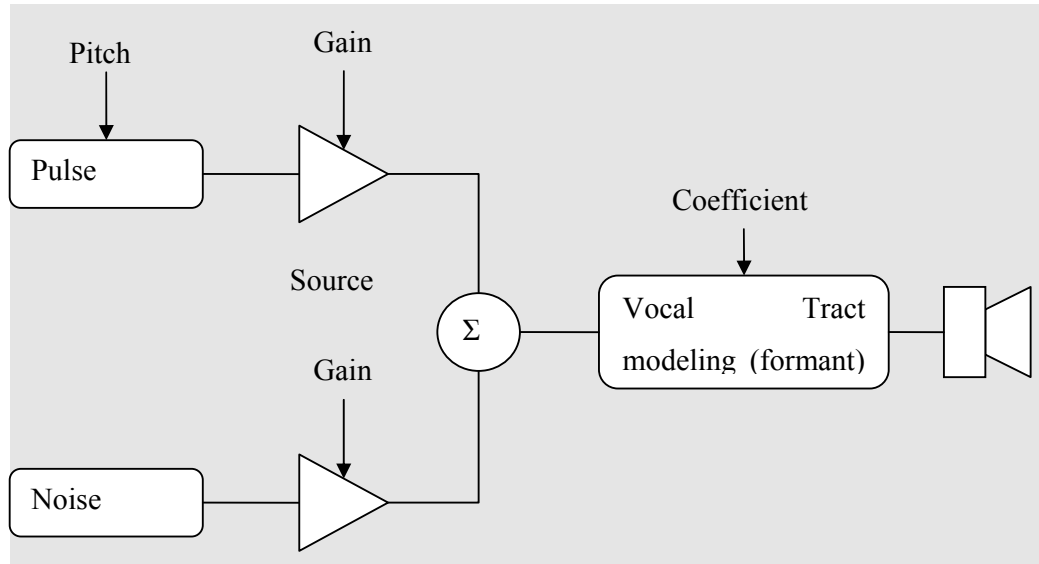
## 2.2. Linear Predictive Coding (LPC)



**Figure 1** Block Diagram of a Formant-based speech synthesizer

In the Linear Predictive Coding speech coder, speech is sampled in wavelets and analysed for vocal tract resonance (short-term autocorrelation is used for this) and pitch (long-term autocorrelation or FFT is used for this). This information is packetized and transmitted over the channel to the synthesizer at the receiving end, as in Figure 1. Leis (unpub) has highlighted the fact that the binary choice between the pulse and noise sources as inputs limits the capability of this synthesis method and the languages it can support. For example, the sound /zh/ as in the French *bon jour* is not truly realizable in LPC coding, because you would need both the noise and pulse sources mixed together.

### 2.3. Mixed-Excitation LPC (MELP)



**Figure 2** MELP uses formant modeling with *mixed* sources.

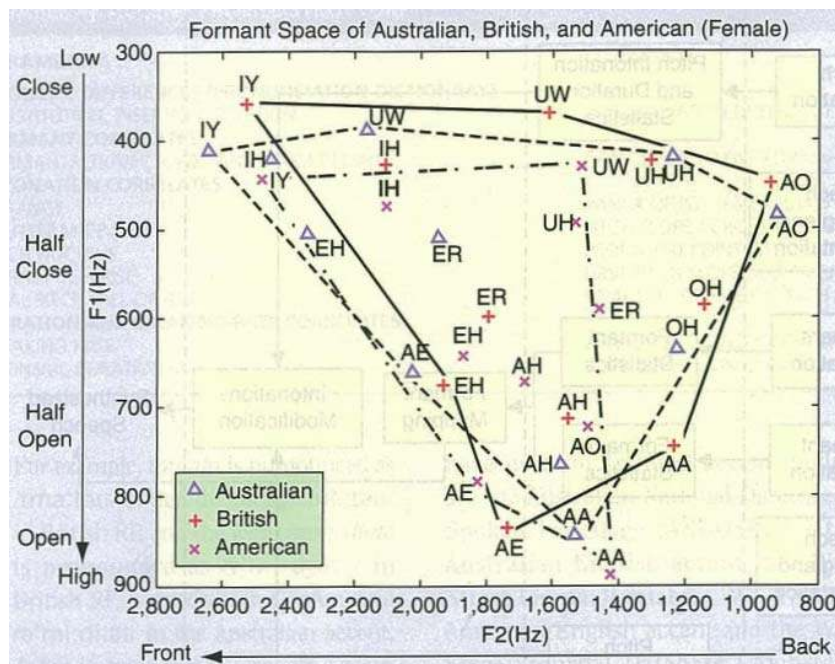
The MELP vocoder, an extension of the LPC vocoder, uses both noise and pulse sources at the same time, mixing them according to the parameters from the analysis (transmission). This makes for a more flexible approach that is better for synthesis of multiple languages, and provides more natural sounds during transitions in speech (ASPi, 1996). While MELP is far more computationally complex on the encoder end, it is not very much more complicated than LPC on the decoder end.

### 2.4. Considerations of Language and Accent

The desire is to produce a synthesizer that could be extended to be able to facilitate any human language. This would possibly imply the use of various code books as in CELP (Code Excited Linear Prediction). Codes for source signal generation would be assembled into books that are each suitable for a specific language, say, and could be interchanged to adapt the synthesizer to different languages. The code book may

also provide a mapping from a 2-dimensional control surface such as the LCD touch screen to the generated sounds – time permitting this concept will be explored.

Vaseghi, Yan & Ghorshi (2009) have recently devised methods for analysing speech accents (in particular, British, American and Australian English), and morphing encoded speech from one accent to another using a Linear Prediction Formant Transformation. In this system, accent databases are used to train HMMs of speech formants for each accent. The HMMs are then used to determine the matching formant set in the target accent, and pitch intonation is also varied. The interesting thing about this work is that in analysing the different accents they developed a formant space showing another 2-dimensional view of speech parameters, shown in Figure 3. This work also highlighted the importance of pitch over time for emphasis and intonation – indicating that pitch control is essential for a good speech synthesis engine.



**Figure 3** Formant Spaces of Australian, British and American Accents. (*IEEE Signal Processing Magazine, Vol. 26 No. 3 p72, May 2009*)



## 2.5. Parameter Quantisation and Interpolation

Speech Parameters (LPC Coefficients, Gain, Error Residual and Pitch Period) can be represented in a number of ways, and various methods seek to quantise them to provide good compression without losing intelligibility.

Kabal and Ramachandran (1986) have presented a way of representing LPC coefficient vectors as Line Spectrum Frequencies the cosine angle of Line Spectrum Pairs (LSFs and LSPs). These can be quantised in terms of their angles and Paliwal (1993) has highlighted their power in this regard.

LSPs and LSFs can be utilized also for interpolating between frames of speech where a set of coefficients may have been lost due to data corruption.

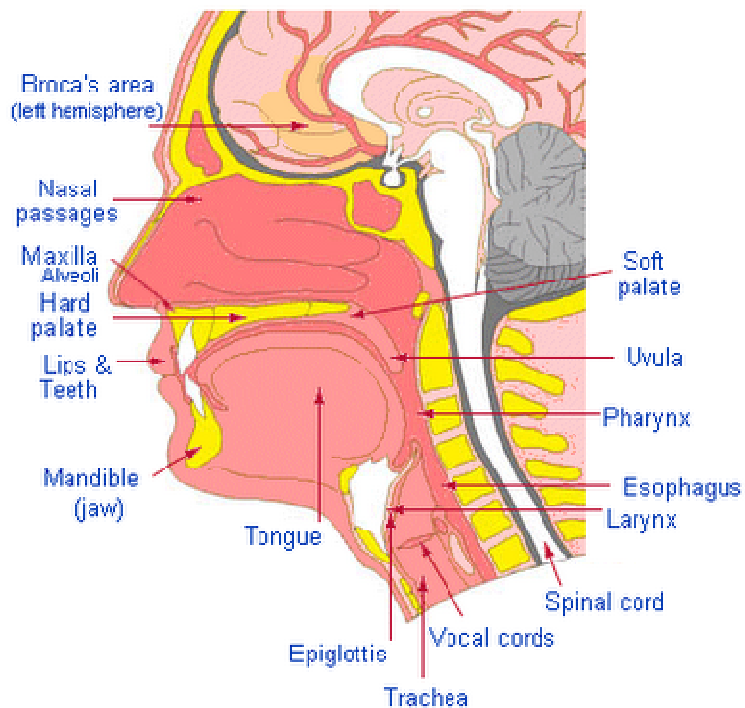
Other methods of interpolating the filter parameters include interpolation of the coefficients directly using Lagrange method as discussed by Hui (1989), morphing and audio flow as discussed by Ezzart, et. al. (2005), and pole shifting (Goncharoff & Kaine-Krolak, 1995).

## Chapter 3.

# Speech Analysis and Modelling

### 3.1. Human Speech Organs

Humans have a unique ability to articulate communication using multiple sound sources and a set of resonant chambers – the throat, tongue, nasal cavity, teeth, lips and glottis all form a part of this complex system, illustrated in Figure 4.



**Figure 4** Human Speech Organs (Fu Jen Catholic University Graduate Institute of Linguistics 2007, *The biological basis of speech production (2): The Vocal Tract and Related Speech Organs*, viewed 22 October 2009, <<http://www.ling.fju.edu.tw/phonetic/mouth.gif>>).

The main sound source within the system is the vocal cords, which operate much like a reed. The muscles surrounding the vocal cords pull them together tightly as the lungs blow air through them, causing a vibration. The sound and air pressure generated moves through the pharynx, mouth and nasal passage past the lips and nostrils respectively. The internal shapes of these cavities form resonant chambers that can arbitrarily change frequency response. Vowel sounds are generated in this way.

By nature of the way vowels are formed, the vocal cord sound source must have a high harmonic content for the frequency shaping of the cavities to have a profound effect.

In addition, air movement through the nose, past the tongue and through teeth and lips is used to create fricative or plosive sounds by restricting airflow or obstructing and releasing it, respectively, which in turn creates noise. Fricative sounds formed by the lips and teeth (such as /f/) relatively white since it occurs towards the outside of the cavities whereas those generated at the back of the tongue (such as /k/) shape the noise through the mouth cavity.

### 3.2. Insight Into Speech Analysis

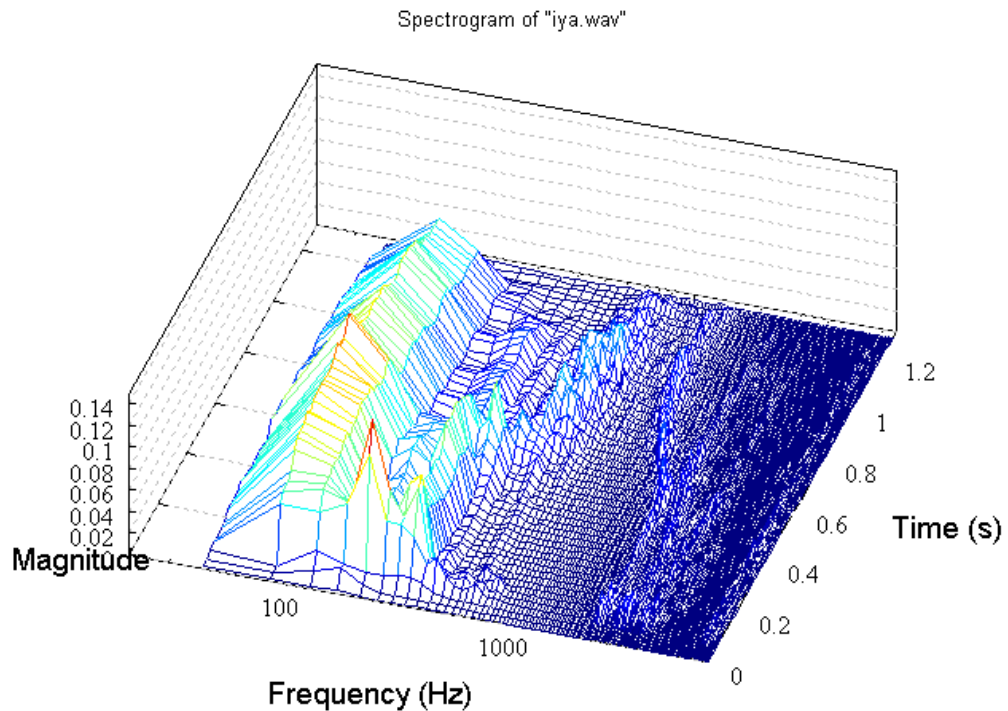
Normal humans are well known to hear sound pressure waves in the frequency range of 20 Hz – 20 KHz, yet our hearing is most sensitive in the midrange frequencies, peaking at around 4 KHz (Bauer and Torick 1966).

It is no surprise then to discover that the majority of the power in a speech signal is in this range, with lower amplitudes in the most sensitive hearing region of the spectrum, and higher amplitudes at the lower end of the spectrum.

A widely used tool for analysis of speech is the spectrogram (Rabiner and Schafer 1978). A spectrogram of a recording of the vowel transition “iya” (as in cornucopia) is shown in Figure 5. The *spectutils* toolbox for Octave was used to generate a spectrogram from the recording “iya.wav”. *Spectutils* can be downloaded from the University of Helsinki’s Music Research Laboratory (University of Helsinki, 2009).

The Magnitude, Frequency and Time axes of the spectrogram are self-explanatory. What is of relevance to this project is the transient nature of the sounds of speech. The foreground shows the “iii” sound, with high energy around 400 Hz, and another lower peak of energy at around 2.5-3 KHz. These represent the two main formants in

English vowels. Their path in the transition to the “ah” sound is clearly visible – they move towards each other in the spectrum to straddle 1 KHz. There is still a lower frequency energy peak just below 100 Hz – this is indicative of the fundamental pitch of the vocal cords which is relatively constant.



**Figure 5** Spectrogram of the voiced sound “iya”.

### 3.3. Synthesis of Speech

The goal of any speech synthesizer is to be able to reproduce human-like sounds and as such must be able to;

- Generate frequencies over the normal human pitch range,
- Model the formants of the resonant cavities, and

- Be able to change parameters dynamically (i.e. interpolate between speech segments).

Each spoken language has its own set of sound combinations with varying degrees of uniqueness. In English, the vowel and semivowel sounds are formed using the two main resonance cavities, and can be modelled using two resonant filters for generating the associated formants. Most speech vowels are better modelled by four formants (Rabiner and Schafer, 1978). Each formant is modelled by two poles in the synthesis filter as they are typically complex and for a real filter to be realized would be conjugate pairs. More formants would be modelled by using more pole pairs in the linear prediction.

### **3.4. Speech Transitions**

Because of the transient nature of speech, the speech synthesis system must be able to adapt pitch, source selections, and gains and filter coefficients with each frame. Filters that adapt in such a way are generally referred to as Linear Shift-Variant (LSV) filters. Hence, speech is sampled into short buffers and treated as stationary for short bursts.

The LSV filters used in speech synthesis are also stationary over each frame of speech (i.e. the coefficients only change between frames). Rabiner and Schafer (1978) and other authors referenced in this dissertation suggest 20-30ms frames are appropriate.

### 3.4.1. Frame Interpolation

Interpolation between frames is necessary if speech frames are encountered with gaps between them (i.e. a frame is lost due to corruption through the transmission channel). This interpolation is a big challenge, especially if it needs to be done in real-time.

Hui (1989) suggests LSV filters can use the Lagrange linear interpolation of filter coefficients, though this method can result in unstable filter kernels.

Goncharoff and Kaine-Krolak (1995) have devised a pole-shifting method whereby filter poles of the first frame are paired with poles of the last, and poles of interpolated frames in between the first and last are interpolated using a frequency-linear relationship. The pole pairing procedure is arduously complicated due to the ambiguity of the pole-pair relationships, and the problem that sometimes real poles must be interpolated with complex conjugate poles.

Ezzat et. al. (2005) present a fairly new method of interpolating frames of speech or music they refer to as ‘audio flow’. The principle behind it is to use a 2-dimensional morphing algorithm that is usually used in computer graphics, but lends itself to morphing the spectral envelope of the audio. It is a complex and computationally expensive algorithm, yet it provides exceptionally natural sounding results.

Pailwal (1993) touts interpolation of Line Spectrum Pairs as the best method for two reasons. The first is that it guarantees stable filter kernels, and the second is that it has the lowest spectral distortion of all the methods tried, which were:

- 
- i. Reflection Coefficient Interpolation
  - ii. Log Area Ratio interpolation
  - iii. Arc-sine Reflection Coefficient Interpolation
  - iv. Cepstral Coefficient Interpolation
  - v. LSP (LSF) Interpolation
  - vi. Autocorrelation Coefficient Interpolation, and
  - vii. Impulse Response Interpolation.

The Impulse Response Interpolation is that presented also by Hui (1989), but Paliwal's results showed it to be the worst due to the instances of instability. This approach was experimented with at first and the result obtained was unsatisfactory, as will be discussed in a later chapter.



## Chapter 4.

# Voice Coding

### 4.1. Introduction

As discussed in section 2.2, Linear Predictive Coding forms the backbone of all the currently popular speech compression mechanisms, including low-bitrate vocoders such as CELP, MELP, G.729 and others.

### 4.2. Linear Predictive Coding

#### 4.2.1. Linear Predictor

The core of Linear Predictive Coding is, as the name suggests, a Linear Predictor (or, FIR filter put to prediction use):

$$\tilde{s}(n) = \sum_{k=1}^P a_k s(n-k) \quad \dots(4.1)$$

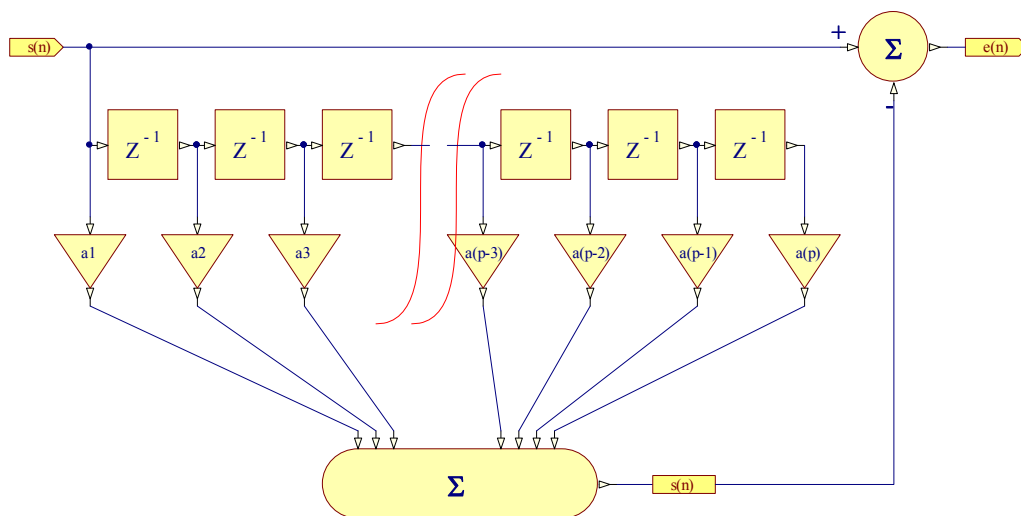
The signal  $\tilde{s}(n)$  models a prediction based on previous samples, and therefore the LPC analysis attempts to find a set of predictor coefficients that minimize the error for each sample within the frame:

$$s(n) - \tilde{s}(n) = e(n) \rightarrow 0 \quad \dots(4.2)$$

Substituting ... ( 4.1 ) into ... ( 4.2 ) gives:

$$e(n) = s(n) - \sum_{k=1}^P a_k s(n-k) \quad \dots(4.3)$$

This is directly realizable in hardware or software as the FIR filter structure shown in Figure 6.



**Figure 6** Linear Prediction FIR Filter

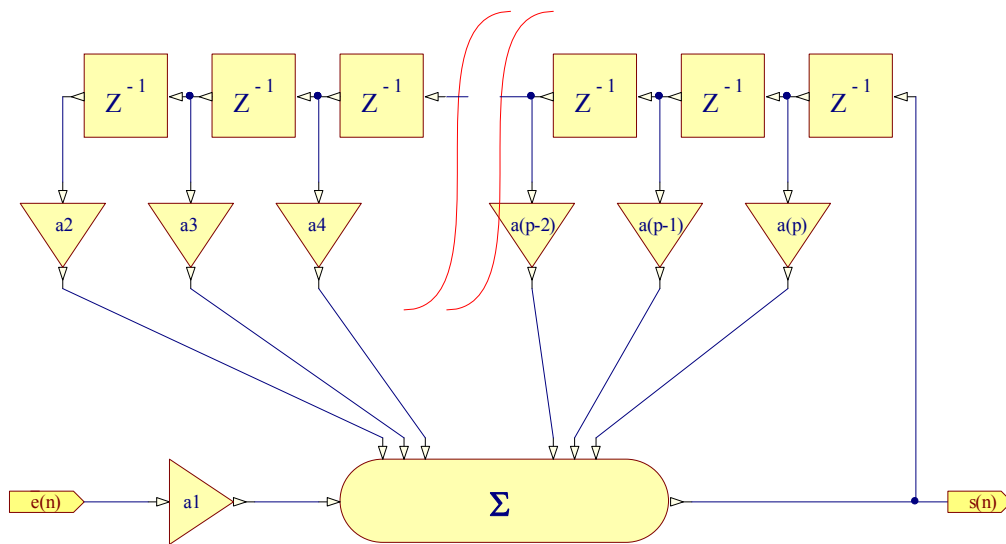
If the linear prediction filter was of infinite length it would allow the error residual  $e(n) \rightarrow 0$ , but since any practical filter will have a finite number of taps (and any practical system could not wait forever for the result), we limit the order of prediction. The limited prediction order means that the error signal will not be zero, but instead resembles a low level noise superimposed with pulses at the fundamental frequency of the original speech frame. The limited predictor order also imposes a limit on the accuracy of the synthesis filter.

### 4.2.2. Inverse Predictor

It is possible to ideally reconstruct the discrete time signal if we have the predictor coefficients and the error signal:

$$s(n) = e(n) + \sum_{k=1}^P a_k s(n - k) \quad \dots(4.4)$$

This “inverse filter” (Rabiner & Schafer 1978) is directly realizable in hardware or software as an all-pole IIR filter, as shown in Figure 7.



**Figure 7** All-Pole Inverse Prediction Filter

### 4.2.3. Calculating Predictor Coefficients

The computation of predictor coefficients for a speech frame can be done using various methods including, but not limited to;

- the Covariance method,
- the Autocorrelation method, and
- the Lattice method

according to Rabiner & Schafer (1978, p397).

In all of these methods, the goal is to efficiently compute the set of coefficients that minimize the mean-square error over the frame:

$$E = \sum e^2(n) \quad \dots(4.5)$$

Substituting ... ( 4.4 ) into ... ( 4.5 ) gives:

$$E_n = \sum_n \left( s(n) - \sum_{k=1}^P a_k s(n-k) \right)^2 \quad \dots(4.6)$$

By taking the derivative of ... ( 4.6 ) and setting it to zero we arrive at:

$$\sum_n s(n-i) = \sum_{k=1}^P a_k \sum_n s(n-i)s(n-k) \quad \text{where } 1 \leq i \leq P \quad \dots(4.7)$$

This can be solved to find the set of  $a_k$  that minimise the error signal. This forms the basis of one of the widely implemented algorithms for LPC coefficient calculation, the Autocorrelation Method. It is computationally more efficient than other methods that were encountered during the course of this project.

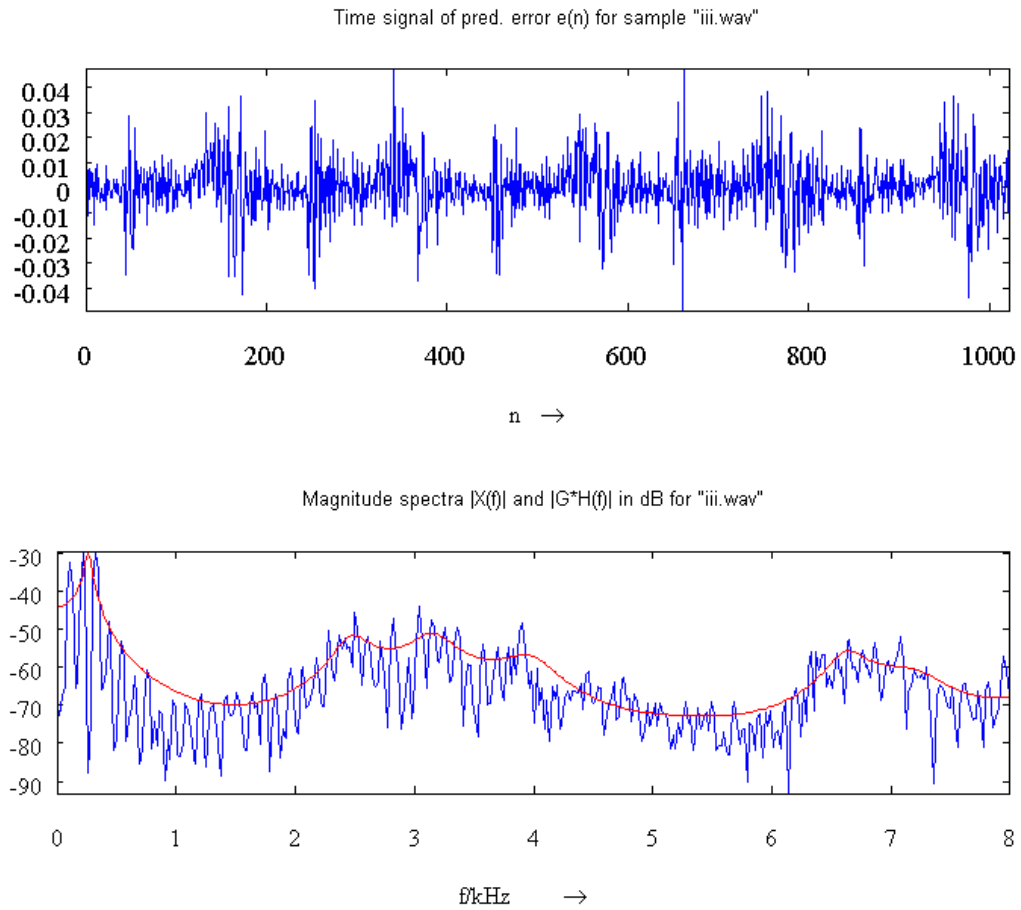
Equation where  $1 \leq i \leq P$  ... ( 4.7 ) can be expressed as the matrix multiplication:

$$\mathbf{R}\mathbf{a} = \mathbf{r}, \text{ where: } \quad r(n) = \sum_{k=1}^P s(n)s(n+k) \quad \dots(4.8)$$

$r(n)$  is the autocorrelation of the input speech frame. Equation ... ( 4.8 ) can be solved using an efficient algorithm known as the Levinson-Durbin Recursion (Keiler & Zölzer (ed.) 2008, p.308). Thus, the top-level algorithm for computing the predictor coefficients from a frame of speech is:

- 1) Input speech to BUFFER[1..N]
- 2) P = Predictor Order
- 3) R[1..P] = cross correlate ( BUFFER, P )
- 4) a[1..P] = Levinson-Durbin Recursion ( R, P )
- 5) e[1..N] = s[1..N] - SUM[1..P] ( a[1..P]\*s[1..N-1..P] )

An Octave (MATLAB) program that performs this algorithm has been provided by Keiler & Zölzer (ed.) (2008, p.308). The code listing is presented for reference in Appendix B.1.



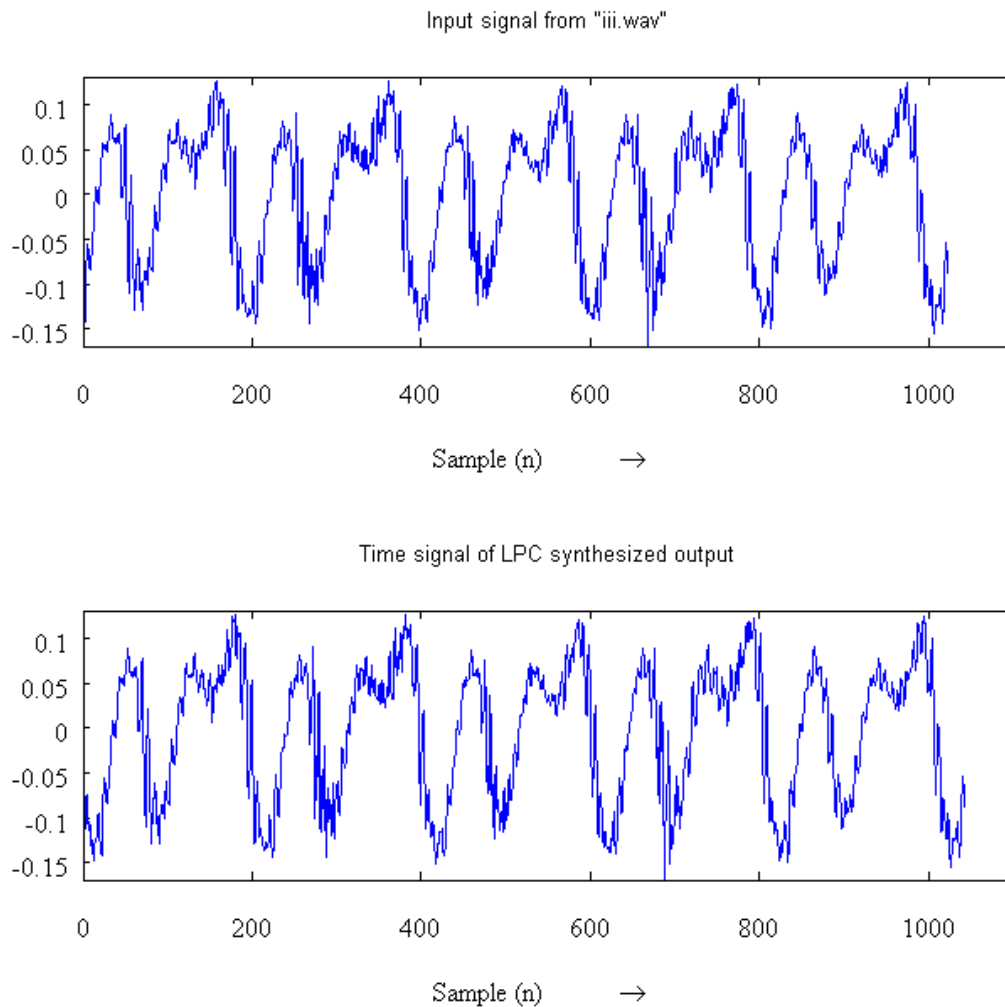
**Figure 8** LPC Analysis Predictor Error and Log Magnitude Spectrum of predictor polynomial.

### 4.3. LPC Initial Results

Figure 8 shows the results of LPC analysis using the above method. This figure was generated by the script `lpc_gen_figs()` listed in appendix B.2. This program

(adapted from Keiler in Zölzer (ed.) (2008, p.306) performs LPC analysis on the frame taken from the named .wav file, and computes the log-magnitude FFT of the speech sample and the LPC filter overlaid. The error residual is shown above.

The `lpc_gen_figs()` script also generates the waveform displays in Figure 9. This provides a visual comparison of the original speech input waveform and that which is reconstructed using the error filtered through the LPC Synthesis filter (Figure 7). In this scenario the sample rate of the voice is 22.05 KHz and the prediction order chosen was 20.



**Figure 9** Discrete Time waveforms of input and synthesized speech frame.

#### **4.4. LPC Parameter Quantization**

LPC and its variants are used for the most part in lossy communications channels, such as the GSM mobile telecommunications system, and in Voice Over IP (VOIP) network communications protocols. The underlying motivation for using vocoders like LPC is to parameterise speech so that the speech frames being transmitted are highly compressed, without perturbing the intelligibility of speech or impairing the listener's ability to identify the speaker.

Once the LPC coefficients are calculated they form a very compact packet that is much smaller than uncompressed speech. Several approaches have been developed for quantising the speech parameters to reduce the storage or transmission load further. Worthy of particular mention in this project is the use of Line Spectrum Frequencies as discussed by Kabal & Ramachandran (1986) and Paliwal (1993).

Although the goal of this project is not to transmit compressed speech through a channel, the concepts behind LSFs will be engaged to solve the interpolation problem discussed in the following chapter.



## 4.5. Conclusion

This shows that if you re-construct the encoded speech frame using the actual residual error you will achieve a close-to-ideal result. In practice the error signal is approximated (or quantized) into a pulse or white noise source, or in the case of the CELP vocoder, a code book. The code book is an array of quantized error approximations that are chosen to reconstruct the speech with minimal error.

In this project, the LPC mechanism of using a pulse source and a noise source has been chosen. While this produces more robotic sounds than those of CELP, it works well enough and provides an expedient solution. It is recognized that some implementations of the LPC vocoder use other waveforms for excitation of the synthesis filter, such as triangle or trapezoidal waves (Vocal Technologies Inc. 2009, MELP (Mixed Excitation Linear Predictive), viewed 3 March 2009, <[http://www.vocal.com/speech\\_coders/melp.html](http://www.vocal.com/speech_coders/melp.html)>).

## Chapter 5.

# Morphing Formants across the TFT Panel

### 5.1. Introduction

The aim of this project was at first somewhat ambiguous in that it sought to map voiced and unvoiced speech sounds to a 2-dimensional control surface. The ambiguity lies in the following facts:

- It is difficult to narrow down the fundamental elements of speech to a handful of simple movements on a screen.
- Some languages produce sounds from the back of the throat and tongue that LPC does not easily reproduce.
- Though we are using a 2-dimensional control surface, speech synthesis in a generic way has more than two dimensions (i.e. not just pitch and tonality, but it also noise and gain).

## 5.2. User Interface research

As a starting point for pragmatic research the design approach taken here is to map specific vowels to the TFT panels X-axis, while the pitch of the synthesizer is controlled by the Y-axis. The choice of using a pulsed source, noise source, or a mix of both is performed by a separate control – pushbuttons. An example of how this implementation will look is illustrated in Figure 10.

Phythian, M. (pers. comm.) has suggested other possibilities involving continuously adjusting the screen display to show a series or circles of symbols that depict speech sounds or code vectors. The display of these glyphs would dynamically update depending on where the pointer to the screen last was. The net result would be that the user constructs strings of phonemes and diphthongs by moving a finger or stylus around the screen. This approach is similar to a text-to-speech system except would be more generic in nature.



**Figure 10** Mapping a TFT Panel to Vowels and Pitch (Altium Ltd. 2009, NB2 TFT Panel Port Plug-In Library Component).

### 5.3. Morphing Across The TFT

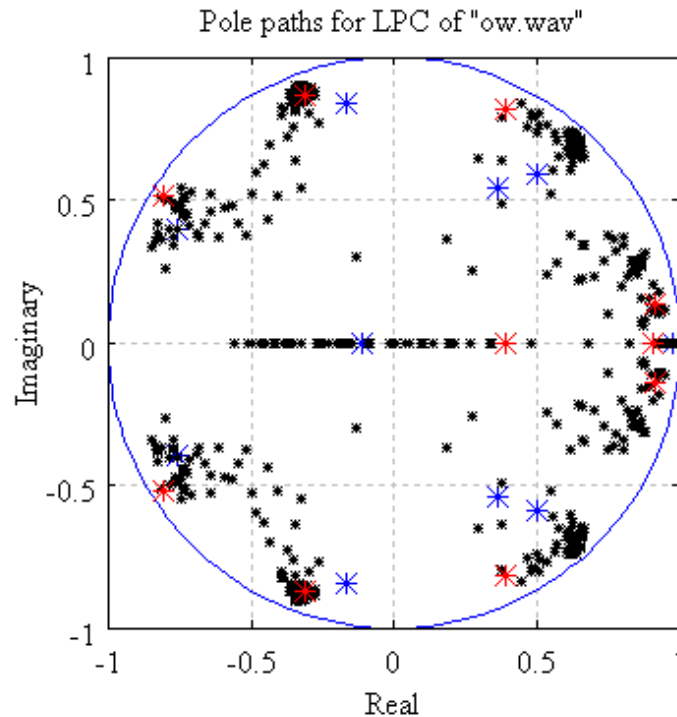
With the decision made to map formants across the X-axis of the screen, the next problems that need solving are:

1. How to generate the LPC synthesis coefficients, and;
2. Once they are mapped to the screen, how to smoothly morph between them as the stylus moves.

It is also worth mentioning that the choosing and ordering of the vowels at this time is arbitrary, but research should be undertaken to gain a better understanding of how to make this choice, and would likely involve using LPC sound corpus from many languages or accents.

The first problem above is easy to solve – in this project we use recorded segments of the author’s speech, and generate the LPC coefficient vectors in non-real-time using Octave scripts.

To get an idea of what to expect in terms of Z-plane pole shifts when vowel sounds dynamically change with respect to speech frames, the Octave script `ow_pole_mapping_plot.m` was used to generate Figure 11. In this image, the simple phrase /æ U/ (as in ‘**ouch**’) was divided into 1024-sample, 50% overlapping frames. The LPC vectors for each were calculated and the roots were plotted on the Z-plane. It is evident that even in natural speech poles can jump around quite a lot.



**Figure 11** Z-plane Poles of a series of LPC frames from /œ/ to /U/. Blue poles mark the LPC coefficient poles of the start frame, red poles are from the final frame, and black in-between.

## 5.4. Interpolation of Coefficients

The second problem mentioned in section 5.3 is not so easy to solve. Two methods were tried and implemented on the Nanoboard. The first of these was the linear interpolation of LPC coefficients, treating the filter as a LSV filter as discussed by Hui (1989). The other was interpolation via LSPs. An algorithm for this has been presented by Morris & Clements (2002), involving computing the Jacobians of the LSPs and using these minima and maxima to detect and modify specific formants of choice in frequency and/or bandwidth. While it is a robust and efficient method it is

overly complex for the needs of this project. Another method, similar but simpler and easier will be used.

## 5.5. Introducing LSPs

Line Spectrum Pairs, as the name suggests, are lines, paired along the spectrum (i.e., around the unit circle on the Z-plane), that describe the characteristics of the LPC filter.

They are computed in the following manner, as shown by many including Kabal and Ramachandran (1986), Soong & Juang (1993), Paliwal & Atal (1993), Stein (2002) and more.

### 5.5.1. Computing the Line Spectrum Pairs

The LPC coefficient vector has the transfer function:

$$H(z) = 1 - A(z),$$

where  $A(z)$  is the polynomial of length (i.e. LPC order)  $p$ :

$$A(z) = 1 + a_1z^{-1} + \dots + a_pz^{-p} \quad \dots(5.1)$$

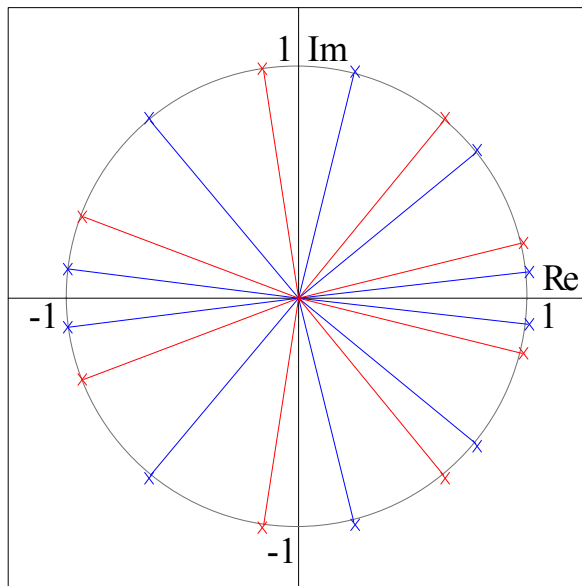
If we take the coefficients of ... ( 5.1 ) and add a mirror-image of them to itself, we arrive at the symmetric (a.k.a. a Palindromic (Stein, 2002)) polynomial:

$$P(z) = 1 + (a_p + a_1)z^{-1} + (a_{p-1} + a_2)z^{-2} + \dots + (a_p + a_1)z^{-p} + 1z^{-p-1} \quad \dots(5.2)$$

Similarly, an antipalindromic equation can be constructed by subtracting the mirrored coefficients:

$$Q(z) = 1 + (a_1 - a_p)z^{-1} + (a_2 - a_{p-1})z^{-2} + \dots + (a_p - a_1)z^{-p} - 1z^{-p-1} \quad \dots(5.3)$$

And, adding ... ( 5.2 ) and ... ( 5.3 ),  $P(z) + Q(z) = 2A(z)$ , so we sum the elements of the palindromic and antipalindromic polynomials and multiply by 0.5 to get back to the original LPC coefficient vector.



**Figure 12** Roots of the Palindromic and Anti-palindromic Polynomials: LSPs

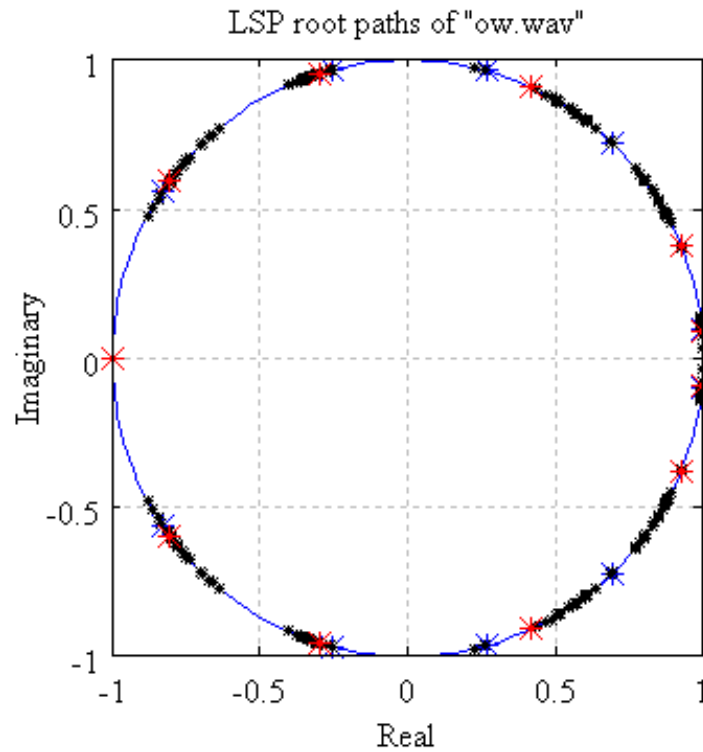
$P(z)$  and  $Q(z)$  are vectors of Line Spectrum Pairs, and have the interesting characteristic that their roots are entirely on the unit circle in the  $Z$ -plane, and the roots of  $P(z)$  are interleaved with those of  $Q(z)$  hence the term Line Spectrum Pairs. These properties are illustrated in Figure 12.

The other useful property these roots possess is that they are always complex-conjugated and if you modify their position as a conjugate pair, you will modify the formants of the LPC vector while guaranteeing a stable filter.

The Octave function `lsp_lpc()`, listed in appendix B.5 obtains the LSPs from an LPC input vector. This function first forms the palindromic and antipalindromic polynomial vectors, then uses Octave's built-in `roots()` function to find their roots. The majority of papers found on LSPs are devoted to finding faster ways of computing their roots to enable their use in real-time systems. A common way is to evaluate the magnitude of the polynomials as excited by cosines of the frequencies around the unit circle and find the zero-crossing points (Kabal & Ramachandran, 1986). There is a speed versus accuracy trade-off in such computations.

Figure 13 shows the roots of  $P(z)$  and  $Q(z)$  evaluated for the same set of LPC coefficient frames as discussed in section 5.3, and generated from the same script.





**Figure 13** Roots of the Line Spectrum Pairs from /œ/ to /U/.

## 5.6. Conclusion

Although several papers mention the method of interpolation using LSPs, few detail anything beyond the computation of the roots of the LSPs – not surprising since the majority of the research has been motivated by the desire to compress speech, as opposed to morph it. The final approach chosen for this project, based on sage advice from Paliwal (1993), Soong & Juang (1993), and others, interpolation of the Line Spectrum Pairs has been chosen.

## Chapter 6.

# Practical Implementation

### 6.1. Introduction

Since the overall idea is to use LPC-style vocal tract modelling for the synthesizer, it makes sense to use the now well-researched LPC synthesis mechanism along with LPC analysis.

However, since the design is for a synthesizer the LPC analysis phase need not be included in the actual final hardware and software. It *does* provide a convenient mechanism for providing the necessary filter coefficients (in non-real-time) for synthesis (in real-time). To that end a number of functions have been developed to produce the coefficients and gains for synthesis in the Octave environment, and in turn these coefficients are written to C code headers for use in the final implementation. This chapter discusses the ensuing design process and outcomes.

## 6.2. LPC Analysis Function

LPC analysis was performed to generate basic vowel coefficients using the Octave (or, MATLAB) script function `calc_lpc()` (Keiler, F & Zölzer, U (ed.) 2008, p.308). This follows the traditional method of using the Levinson-Durbin Recursion, and fortunately Octave comes equipped with the necessary function making the generation of predictor coefficients straightforward. The `calc_lpc()` function returns a vector of coefficients including the 1 in the denominator of the synthesis equation, as well as the gain factor for the analysed frame of speech.

Sample `.wav` files as discussed in Chapter 3 were recorded and clipped for the voiced sounds *iii*, *eh*, *a*, *ah*, *o*, *ue*, *rr*, and *uw*. The format of these samples was mono, 22.05 KHz sample rate and 16-bit quantization. This format provides bandwidth of 11.025 KHz (using the Nyquist theorem) and a theoretical dynamic range of greater than 90dB, which is more than adequate for speech.

The initial version of the design used a set of eight coefficient vectors (from the voice clips mentioned above), which were written out to a C language header file – `lpc_coeffs.h` for use in the test hardware. This was efficiently facilitated through another Octave script which in turn calls `calc_lpc()` for each audio sample file presented, and then saves them using a generic data type and scaling macro for fixed point implementation in C. This function – `generate_coeffs()` – is detailed in appendix B.3. A sample of the `lpc_coeffs.h` file generated by it is listed in appendix D.3.

### 6.3. LSP Calculation

As discussed in Chapter 5, LSPs have been chosen as an experimental method of interpolating LPC coefficient vectors. To perform the conversion of LPC coefficients to LSPs, the function `lsp_lpc()` was developed in accordance with the algorithm presented in section 5.5.1. While most LPC vocoders would transmit only the angles of the LSPs, over one half of the unit circle (as the other is always a mirror image), this function keeps all the roots around the circle in the arrays P and Q. The angles of the LSP roots are usually referred to as Line Spectrum Frequencies (LSFs) as they are represented as angles versus complex numbers. This also reduces frame packet size. For this project, the `lsp_lpc()` function keeps the LSPs in separate vectors for convenience (Ph and Qh) though they are not strictly necessary.

#### 6.3.1. Roots of the Line Spectrum Pairs

The `lsp_lpc()` function also calculates the LSP roots and returns them as a vector with complex numbers. Because the LSP roots lie around the unit circle, they can be expressed as pure angles (LSFs). LSFs are calculated using:

$$\theta_p(k) = \cos^{-1} P(k) \quad \dots(6.1)$$

And conversely, converted back into complex representation with:

$$P(k) = (\cos \theta_p(k) + i \sin \theta_p(k)) \quad \dots(6.2)$$

One important point here is that, because LSFs are mirrored on the bottom-half Z-plane, when converting from LSFs back to LSPs it is necessary to also subtract the imaginary sine term.

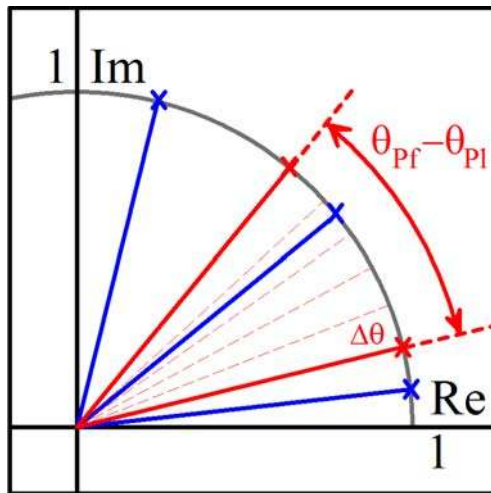
In this project the conversion back to LSPs from LSFs is done in the interpolation function, described in the next sub-section.

### 6.3.2. Interpolation Using LSFs

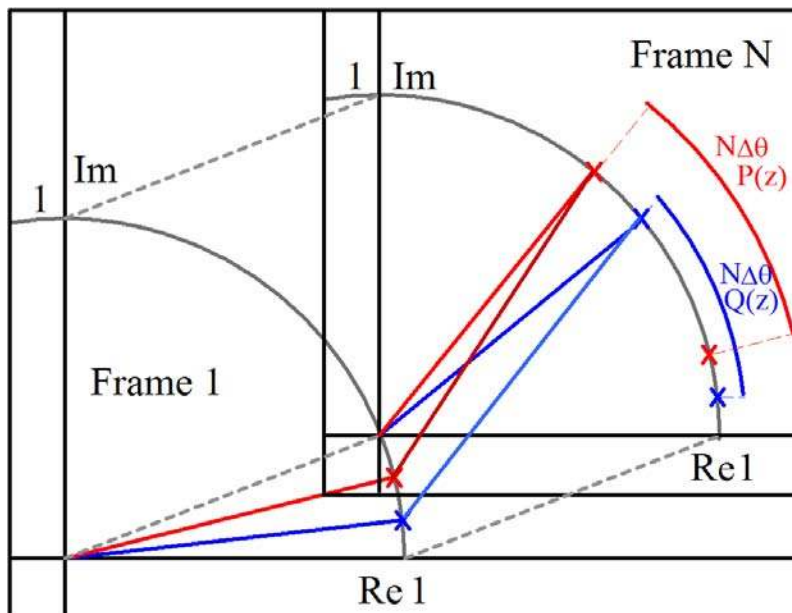
Interpolation of LSFs is as simple as linear interpolation of angles. Figure 14 illustrates this process. The two frames to be interpolated and the number of interpolation steps are given to calculate angle step size:

$$\Delta\theta = \frac{|\theta_f - \theta_l|}{N} \quad \dots(6.3)$$

The critical step here is that LSFs have to be paired from the first frame to the second in the correct (same) order. In this case it is fairly trivial because the LSFs are sorted into an ordered list of values from lowest to highest in magnitude, using Octave's `sort()`. The net result is that the interpolated LSFs do not tend to cross over and behave very well, as illustrated in Figure 15.



**Figure 14** Line Spectrum Frequency Interpolation Using Neighbouring Angles



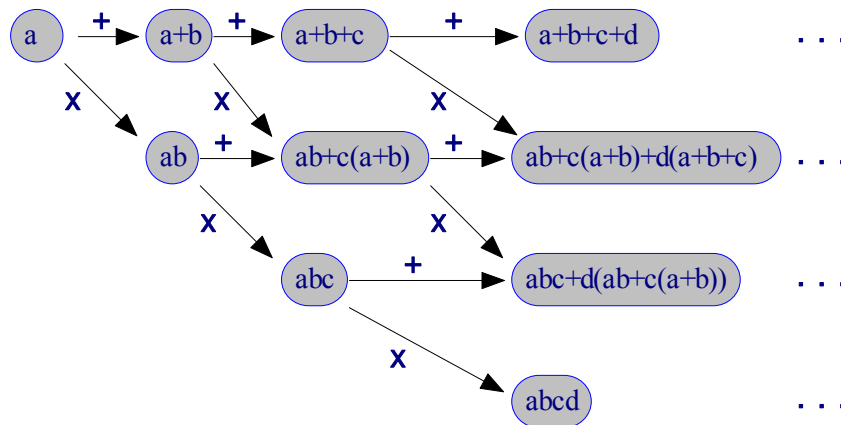
**Figure 15** Interpolating Line Spectrum Frequencies over N frames

The function `lsp_interp()` listed in appendix B.9 performs the LSP/LSF interpolation using this method. Since it takes LSPs as arguments and also returns LSPs, it uses the `arg()` built-in Octave function (equivalent in this case to ... ( 6.1 )) to get the angle from each root, performs the N interpolation steps on the LSF pairs,

simultaneously storing the interpolation results as LSPs (using `cos` and `sin` functions, as in ... ( 6.2 )).

### 6.3.3. Expanding Roots back to LPC Coefficients

The roots are necessary for interpolation, but they make it a more complicated process getting back to LPC coefficients as they have to be expanded. The process one normally undertakes when expanding factored polynomials was examined, and laid out as shown in Figure 16.



**Figure 16** Root Expansion Algorithm

The example shown is for a polynomial of the form:

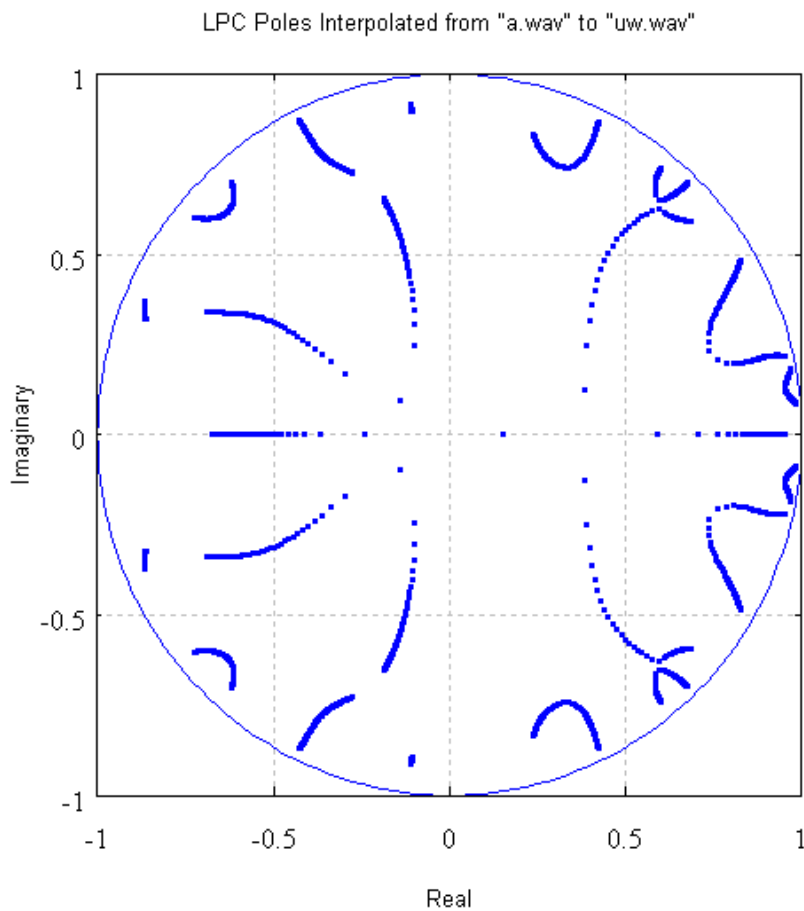
$$A(z) = (z + a)(z + b)(z + c)(z + d)$$

It is easy to see that the process can be broken down into  $\sum_{n=1}^N n$  multiply-add operations. This is simply done in a nested loop, and this has been implemented in the function `expnd()`, listed in appendix B.7.

## 6.4. Tying Interpolation Together

The final step is to write a function that takes two file names of recorded speech segments, builds LPC vectors from each, uses the LSP interpolation as discussed and finally returns a set of LPC vectors from the interpolation – these to be written to a C header file later on for the hardware and firmware design to use.

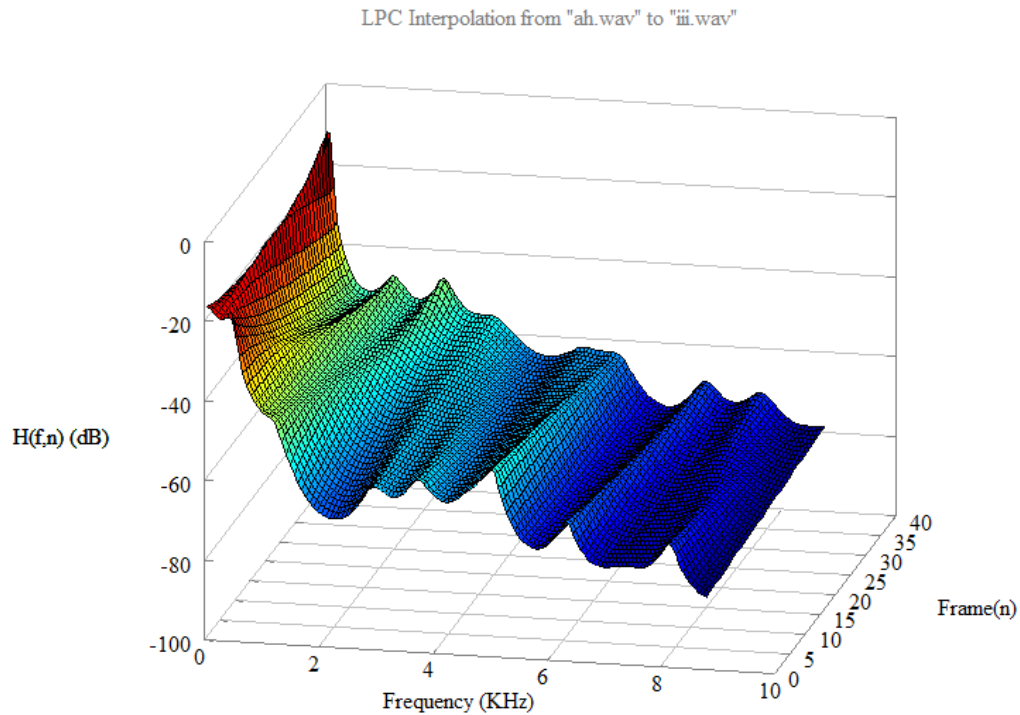
Figure 17 is a Z-plane plot that shows the result of running the function that does this – `plot_interp()`, listed in appendix B.12.



**Figure 17** LPC Poles of Interpolation of coefficients using LSPs.



`plot_interp()` is also written to optionally create a surface plot of the log-magnitude spectrum of the LPC coefficients created by it. Just such a plot is shown in Figure 18.



**Figure 18** Log Magnitude Spectrum of LSP Interpolated LPC coefficients.

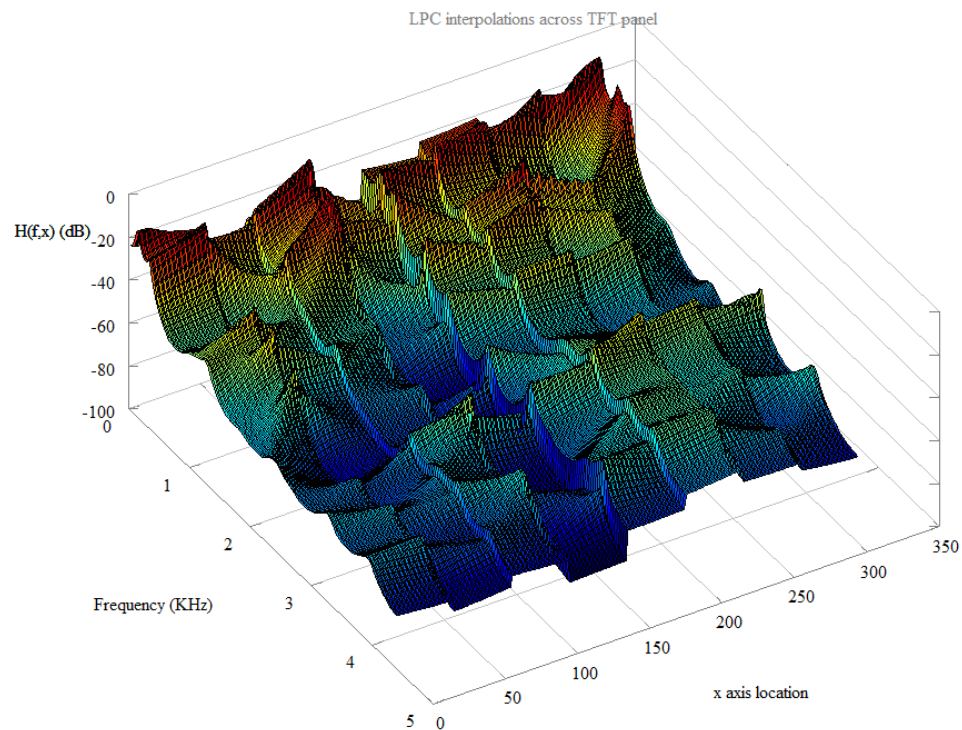
It is very clear from both of these figures that this method of interpolation is a great one. It produces smooth transitions even between very different pole maps.

## 6.5. Formant Mapping

The initial design uses the x-axis of the TFT touch screen to control the formant characteristics, while the y-axis is used to control the pitch period of the pulsed excitation source. The simplest way to map formants to the screen is to divide the screen width into regular segments with each invoking a set of LPC coefficients to be

used for synthesis. This is facilitated by a timer interrupt service routine which regularly checks the status of the pointer driver, and when the user touches the screen the x-location is read and used to point to the appropriate set of coefficients using a C pointer.

There are eight vocal sounds that were used initially to generate the coefficients as pointed out in section 6.2, with interpolation of the coefficient vectors used to fill in the extra spaces between in order to smooth the transitions between them.

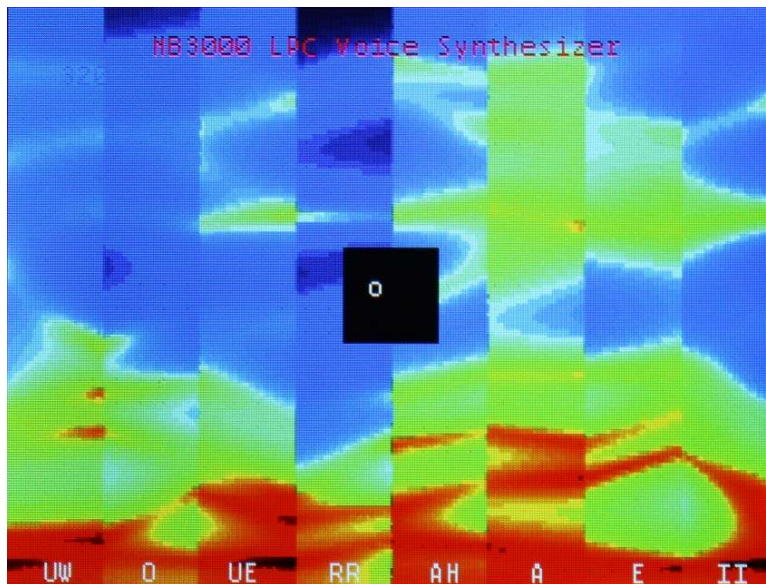


**Figure 19** Log Magnitude Spectrum of LSP interpolations across TFT panel width.

To build an interpolated set of LPC coefficients for every X location on the TFT panel, the `gen_all_lsp()` Octave function was created, and is listed in B.11. The output of this function in the Octave environment is a matrix of coefficients of size 320 by 22. This size reflects the 320 X-resolution of the TFT on the NB3000, and the

20<sup>th</sup>-order LPC coding used (as opposed to the typical 10<sup>th</sup>-order), and the +1 coefficient as well as the gain coefficient for each of the 320 frames. This matrix has been plotted as well, and the 3D surface plot of its log-magnitude spectrum is shown in Figure 19. This image makes obvious the fact that the interpolation of the poles was done well, but the gains are just as important. This is a potential topic for future work on this project.

Overall, the interpolation result was surprisingly good. The graph of Figure 19 was also set to top view in the plot window and a screen-shot of it was taken. The resulting background of the NB3000 TFT panel is shown in Figure 20. This text in this photo is added to the screen by the software of the design.



**Figure 20** Spectrogram bitmap of interpolation results used as TFT background.

## 6.6. Embedded System Considerations

The implementation in the target hardware used 16-bit quantization for the audio data path. This was chosen based on the following facts:

- The .wav files used for coefficient generation were recorded in 16-bit quantization.
- The 32-bit RISC processor used allows scaling operations to be performed from 32-bits down to 16 in a single cycle using its barrel shifter. Contrasting this, if 32-bit data were used the system would require scaling from 64-bits down to 32, which requires at least double the clock cycles.
- As will be shown below, 16-bits allow fixed-point scaling that still has sufficient headroom for the all-pole filtering to work effectively.

Since the system performs LPC synthesis in real-time, there are practical limits imposed by the its architecture which in turn limit the allowable bandwidth, the order of LPC synthesis filter used, and bandwidth of speech generated.

## 6.7. Fixed Point Implementation

Since the target system will be an FPGA System-on-Chip, the design uses fixed-point arithmetic for the signal path and all filtering. This is because signed integer multiplies exist on board the FPGA die which can be used for hardware acceleration of the filter kernel if needed, and in addition the CPU core used for running the main line of code is a RISC CPU that does not include a floating point unit. Therefore implementation in floating point, however convenient from a coding standpoint, would be too slow for such a real-time application.

### 6.7.1. Coefficient Scaling

The synthesis filter kernel will perform  $N$  16-bit multiplies for each sample ( $N$  is the order of the LPC synthesis), which are in turn accumulated in a 32-bit result. The coefficients are generated from Octave in double-precision floating point and are written out to the coefficient header file with 20 digits after the decimal point. This is done using standard formatting in the well-known `fprintf()` function.

Since there is no built-in data type in the C code of the target for fixed point the coefficients are scaled to a suitable integer type, defined as `samp_t` (a 16 bit signed integer), in the `main.c` file, shown on page 103 (appendix D.1).

For all coefficient vectors produced by LPC analysis, the maximum absolute value  $|v|$  encountered is  $2 \leq |v| \leq 3$ , therefore 2 bits at least are required to represent this. However, to ensure that saturation is not as likely to be encountered an extra bit is used. Therefore the scaling of the coefficients has been set to the Q3.13 format (3 bits of integer including the sign, and 13 bits for fractional data). Truncation of data and coefficients was chosen for its ease of implementation and reduced CPU overhead, but as seen in 6.7.2 (next) the adverse effects of this quantisation method are outweighed by its computational benefits for this design.

### 6.7.2. Truncation Effects

With the Q3.13 fixed-point scaling used, it is necessary to take into account the truncation of data and coefficients. Two methods (analytical and experimental) were used to make sure this format is sufficient.

Since we are scaling the coefficients to Q3.13, the resolution is  $\pm 2^{-13} \approx 122 \cdot 10^{-6}$ . This is close to 80dB below a considered peak value of  $\pm 1$  (0dB) for the filtered signal, which is well within an acceptable range considering the typical listener is less sensitive to dynamic range than this.

The mean square error will be higher in terms of filter coefficient accuracy, because the resolution of the coefficient is only  $\pm 2^{-13}$  and the error is multiplied through each stage of filtration, plus back through as a recursive error signal.

As discussed by Schlichthärle (2000, pp.233-238) the mean-square error and variance introduced by single truncation step through which the audio stream passes can be calculated respectively by:

$$\overline{e^2} = \frac{\Delta x_q^2}{3} \left( 1 - \frac{3}{2N} + \frac{1}{2N^2} \right) \quad \dots(6.1)$$

$$\sigma_e^2 = \frac{\Delta x_q^2}{12} \left( 1 - \frac{1}{N^2} \right) \quad \dots(6.2)$$

Where  $N$  is the number of possible error values, and  $\Delta x_q$  is the quantization step after quantization.  $\Delta x_q$  can be expressed in terms of normalized signal resolution step which, for the Q3.13 system design presented is  $2^{-16} - 2^{-13}$ . To quantify the equivalent noise introduced exactly a known set of samples is required. However, it provides some insight into the effects of truncation during filter arithmetic operations. For a set of samples of a complex signal such as speech, there is almost equal probability of the truncation error being any of the values in the range of truncated bits.

Therefore the equivalent noise power added to the signal by truncating a 32-bit result down to 16 bits (assuming full-scale signal values of  $\pm 1$ ) is:

$$\sigma_e^2 = \frac{\Delta x_q^2}{12} \left(1 - \frac{1}{N^2}\right) = \frac{2^{-32}}{12} \left(1 - \frac{1}{65535^2}\right) \approx 19.4 \times 10^{-12}$$

The noise from each truncation step is added to the total filter noise for each output sample. Therefore instead of truncating each filter multiply-add operation, for this design it was decided to accumulate all filter multiplications in 32-bit precision and truncate the result at the end of the kernel loop back to 16-bits. Having just a single truncation in the path keeps the signal to noise ratio fairly high for the filter. If the signal is considered to be  $\pm 1$  full scale then the SNR for the truncation would be:

$$SNR = 10 \log \frac{1}{19.4 \times 10^{-12}} \approx 107dB$$

This is more than sufficient for the design at hand. In essence the SNR is not as good as this, because of the truncation of the filter coefficients. The equivalent truncation noise for the coefficients is greater due to the rounding to 13 bits of precision. This is effectively a reduction in word of 3 bits, giving a SNR of less than 80dB for each coefficient. The noise will add up, but considering that the input source to the filter and the filter are both deactivated when not in use in this design, it is not considered problematic.

## 6.8. C Code Development

This section points out a few pertinent details about the C code implementation on the Nanoboard 3000. The main C code document (`main.c`) is listed in appendix D.1, which will be referred to throughout this session by page number.

### 6.8.1. Code for generating Coefficients

As mentioned in section 6.5, the Octave function `gen_all_lsp()` is used to perform the LSP interpolation, but it also then generates a coefficient header file for use in the NB3000 embedded project. Appendix D.4 lists the `lpc_lsp_interpolated_coeffs.h` C code header file (truncated as it is too large to include in this dissertation). This set of coefficients works well and is used in the final design.

Another earlier Octave script, `gen_all_lpc()` is listed in appendix B.10 for completeness, though this one is not used in the final implementation. It generates a similar LPC coefficient vector array in a C code header, but the key difference is it interpolates the vectors using simple linear interpolation. The results of using these coefficients are that:

1. Many positions across the screen create noises that suggest unstable filter kernel at those vector positions, and
2. The formant changes from one position to the next are unnatural and can not work for generating real speech.

Hence, the LSP interpolated version was chosen.



## 6.8.2. Drivers and Initialization

The advantage of using the Nanoboard 3000 platform for development of this project lies in the IDE environment, which includes all the C code driver libraries for the peripherals used in the design.

The drivers for each part of the hardware platform, along with their corresponding reference documentation, are accessed through a Software Platform Builder file. This file (NB2\_Voice.SwPlatform) is shown in the Software Platform editor in Altium Designer software in Figure 21.

The screenshot displays the 'Device Stacks' and 'Software Services' panels in the Altium Designer Software Platform Builder. The 'Device Stacks' panel shows a hierarchical stack of components:

- Pointer Context** (POINTER\_1)
- Touchscreen Context** (TOUCHSCREEN\_1)
- CS4270 Audio Codec Driver** (DRV\_CS4270\_1)
- AD7843 Touchscreen Digitizer Driver** (DRV\_AD7843\_1)
- Graphics Context** (GRAPHICS\_1)
- GPIO Port Driver** (DRV\_IOPORT\_1)
- I2S Driver** (DRV\_I2S\_1)
- SPI Driver** (DRV\_SPI\_1)
- SPI Driver** (DRV\_SPI\_2)
- Touchscreen Pen Detector** (TOUCH)
- General Purpose I/O Port** (IOPORT)
- I2S Master Controller** (WB\_I2S\_1)
- SPI Master Controller** (SPI\_AUDIO)
- SPI Master Controller** (SPI\_TOUCH)
- VGA TFT Interface Driver** (DRV\_VGA\_IL19320\_1)
- VGA TFT Interface** (IL19320 TFT)

The 'Software Services' panel shows a table of services with their status and descriptions:

Type	Included	Service	Description
System	Always Included	Interrupt Management	Interrupt management routines
System	Always Included	Software Timer Management	Software timing services (doc
Automatic	No Relevant Stacks	Generic Device I/O	Standard POSIX device I/O rc
Optional	<a href="#">Not Currently Included</a>	Message Queues Support	Support for POSIX message c
Optional	<a href="#">Not Currently Included</a>	Signalling Support	Basic support for POSIX signa
Optional	<a href="#">Not Currently Included</a>	Multithreading Support	Basic support for POSIX multi-
System	Always Included	Software Platform Configuration	Global configuration for the S
Automatic	Included	Graphics Services	Graphics routines for line, rec
Automatic	Included	Pointer Services	Generic pointer routines
Automatic	Included	Touchscreen Services	Generic touchscreen routines

At the bottom of the interface, there are buttons for 'API Reference' and 'Compile Project'.

**Figure 21** The Software Platform Builder

In this figure, each hardware peripheral used in the design and its associated C code driver are represented by API stacks that provide different levels of functionality:

- Green blocks represent wrappers that abstract memory maps of peripherals into macros for named access.
- Yellow blocks represent the drivers themselves, with all the necessary structures and functions to initialize and access the hardware.
- Blue blocks represent abstract APIs that add software services to the system, such as graphics and GUIs, touch screen pointer and so on.

The drivers and API stacks included for use in this project are included in the C by way of the `#includes` at the top of the `main.c` file (see page 103).

### 6.8.3. IIR Synthesis Filter

The LPC all-pole synthesis filter is implemented in the function `allpole_kernel()` on page 107 in appendix D.1. This function uses a static global `samp_t` array for the history buffer, and takes as an argument a pointer to the beginning of a coefficient array. The coefficient array passed to it depends on the position of the pointer on the touch screen.

The input sample to the filter  $x_0$  is multiplied by the coefficients at the beginning (equivalent to 1.0) and the end (the gain coefficient for that frame) of the array.

This value then passes into the cumulative summation of all the prior  $N$  samples, multiplied by their corresponding LPC coefficients, according to equation ... ( 4.4 ) where  $e(n)$  is represented by  $x_0$ .

The output of the filter is quantized back from a 32-bit sign-extended multiply-add result, to type `samp_t`, with an arithmetic right-shift of 17 bits and a type cast (shown in appendix D.1 page 108). The right-shift is sign-extended and 17 bits was chosen instead of 16, because it was found that it produced less distortion in the output signal.

#### 6.8.4. Pulse Source

Human speech typically uses fundamental pitch frequencies in the range 100Hz to 1KHz. Since this design is using a sampling frequency of 22.05 KHz, the period of these pitches range from:

$$\frac{Fs}{100} \rightarrow \frac{Fs}{1000}$$

Or, approximately 220 samples to 22 samples. It was chosen to enable the pitch to range from about 50 Hz to 2 KHz so that the user could reach high singing notes with the device, though in a commercial application this makes the range too great to easily control on a small touch panel.

The calculation of pitch period (in buffer size), is taken care of by the `PITCH()` macro listed in appendix D.1, page 104. The pulse source is implemented as a pre-

initialized buffer with a short spike shaped pulse, and is initialized in the `main.c` file (shown on page 105). The buffer length must be longer than `PITCHMIN`.

### 6.8.5. Noise Source

The noise source is created by the initialisation routine, which initializes all the driver handles, sets up the TFT display, and runs a TFT calibration procedure (listed in appendix D.1, page 109). This routine fills the noise buffer with a random number sequence created by the C standard library function `rand()`. There are probably better ways of generating a white noise source, but for the time available this is the best at hand.

### 6.8.6. User Interface

The CPU timer interrupt callback function `handletimer()` (listed in appendix D.1 page 112), checks to see if the user is pressing the TFT screen, by calling the Touchscreen Pointer API function `pointer_update()`.

If there is activity, the Y axis value, which ranges from  $[0, 239]$  is scaled to the range  $[PITCHMIN, PITCHMAX]$  and this sets the length of buffer (either the pulse or noise sources) that will be looped through, thereby changing the pitch.

The X axis value is read also when there is touch activity, and its value is in the range  $[0, 319]$ . These values correspond directly to the coefficient vector array indices included in the header file `lpc_lsp_interpolated_coeffs.h`. This is an array of LPC vectors dimensioned  $(320, 22)$ , and the X value read from the TFT

---

sets the array vector pointer `current_coeffs` to `&coeffs[X][0]` which is passed in turn to the filter kernel.

The timer callback routine also reads in the status of 5 pushbutton switches which are located under the TFT panel on the Nanoboard. If none are selected, just the pulse source is used for synthesis. If the first pushbutton is selected, the noise source is used, and if the second pushbutton is selected, both the pulse and noise sources are used together.

## **6.9. Conclusion**

The theory of LPC and LSP interpolation has been practically applied to generate filter coefficient arrays of morphed vowel sounds. The arrays have in turn been used in a realisation of an LPC synthesizer in an embedded computer system, with the TFT panel touch screen used to control the pitch and of the source, and the vowel sound based on the coefficient array selected by it.

Chapter 7.

## Introduction to the Nanoboard 3000



**Figure 22.** The NB3000 running the speech synthesizer.

## 7.1. Introduction

The Altium Nanoboard 2 was the original choice for the development platform for this project. Since then, a smaller and more appropriate development board has been released – the Nanoboard 3000 or NB3000.

The NB3000 used is shown running the design in Figure 22.

## 7.2. NB3000 and the Altium Designer Software Platform

The NB3000 is primarily design for the design and prototyping of FPGA circuits. However its versatility and usefulness does not merely lie in that alone. The Altium Designer software that is used with it is very closely coupled to its functionality in the following ways:

1. Each and every hardware peripheral available on the NB3000 has an associated driver in the Altium Designer software suite. These drivers make it a trivial task to get inputs and outputs working rapidly.
2. The NB3000 uses a second (non-user) FPGA device for its own internal firmware. This firmware gives it some very useful capabilities, including:
  - a. JTAG download and debugging over high speed USB.
  - b. The ability to auto-load FPGA hardware and firmware on power-up.
  - c. In-system firmware updateability.
  - d. Electronic identification of the board (i.e. a serial number) which also allows FPGA hardware design constraints to be automated (in other words, the pin assignments do not have to be manually entered into the system as they would be in other environments).

3. Altium Designer (summer '09 version and later) provides a graphical method of building software stacks (APIs) to support the design process.
4. The library of FPGA IP cores that comes with Altium Designer software provide a suite of processors and peripherals that can be programmed into the FPGA on the board to suit just about any task. No hardware IP needs to be created by the user unless it forms a core part of their product.

In this case, everything needed for this project is provided out of the box.

### **7.3. Nanoboard Features Utilized**

The NB3000 has many peripherals. A complete list is given in the data sheet, provided in appendix for reference. Only the ones used in this project are discussed here.

#### **7.3.1. Audio Codec**

The audio codec provided on the NB3000 is a Crystal CS4270. It is a 2-channel CODEC (AD/DA converter) that supports I<sup>2</sup>S audio streaming protocol, and sample rates up to 192 KHz, and quantization up to 24-bits. In this project it is configured for a relatively low sample rate (22.05 KHz – ample for speech), and quantization of 16-bits. This is done in the Software Platform Builder which configures the codec drivers to initialise this device over the I<sup>2</sup>C bus.



---

### **7.3.2. I<sup>2</sup>S Interface**

The I<sup>2</sup>S interface and associated IP Core are used to interface to the CS4270 CODEC.

### **7.3.3. SPI Interface**

The NB3000 platform uses SPI bus in numerous ways. Two SPI bus interfaces are used in this design.

The first is for the audio CODEC which, in addition to the I<sup>2</sup>S audio stream interface, uses an SPI bus interface for the host which configures it. In this case the host is the SPI peripheral core in our embedded FPGA System-on-Chip.

The second SPI bus connects to the TFT panel touch screen controller chip, a Texas Instruments TSC2046.

### **7.3.4. TFT Interface**

The TFT Touch screen uses SPI as mentioned above. The TFT video output is a bidirectional 5-bit per pixel digital interface, and uses the TFT controller IP core within the FPGA design. This IP core provides DMA for reading the display buffer memory and supports double buffering.

The buffer is set up using a canvas driver in graphics context within the Software Platform Builder for the project.

### 7.3.5. GPIO port, LEDs and Pushbuttons

The NB3000 has eight RGB LEDs on board. This design makes use of those via the configurable IOPORT peripheral outputs.

The inputs of this peripheral core are used to monitor the user pushbuttons.

### 7.3.6. SRAM Interface

Although the NB3000 sports many memory options, the memory requirements of this project are light, and so only the external SRAM is used. The memory configuration is shown in Figure 23.

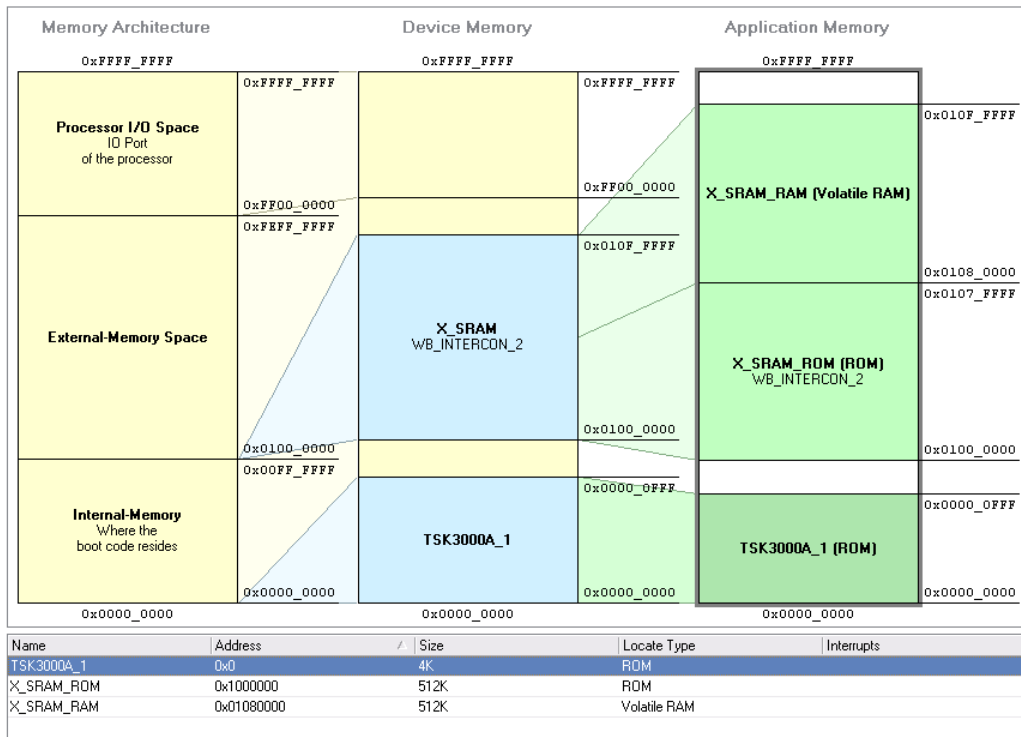


Figure 23 Embedded Project Memory Configuration.

---

In this case the external 1MB (configured as 2, 256K by 16-bit chips) of SRAM is divided into program memory and data memory. A small amount (4KB) of FPGA Block RAM is used within the CPU core as well.

## **7.4. FPGA Hardware Design**

The FPGA hardware design is fairly simple, using only IP cores from the provided libraries.

The physical connections to the peripheral hardware on the NB3000 are made through the top-level FPGA design schematic ports, shown for reference in appendix C.1.

At the core of the system is the TSK3000A 32-bit RISC CPU, connected to the peripheral controller cores by the Wishbone interface, represented by the connecting arrows in the OpenBus System document. This document is provided in appendix C.2.

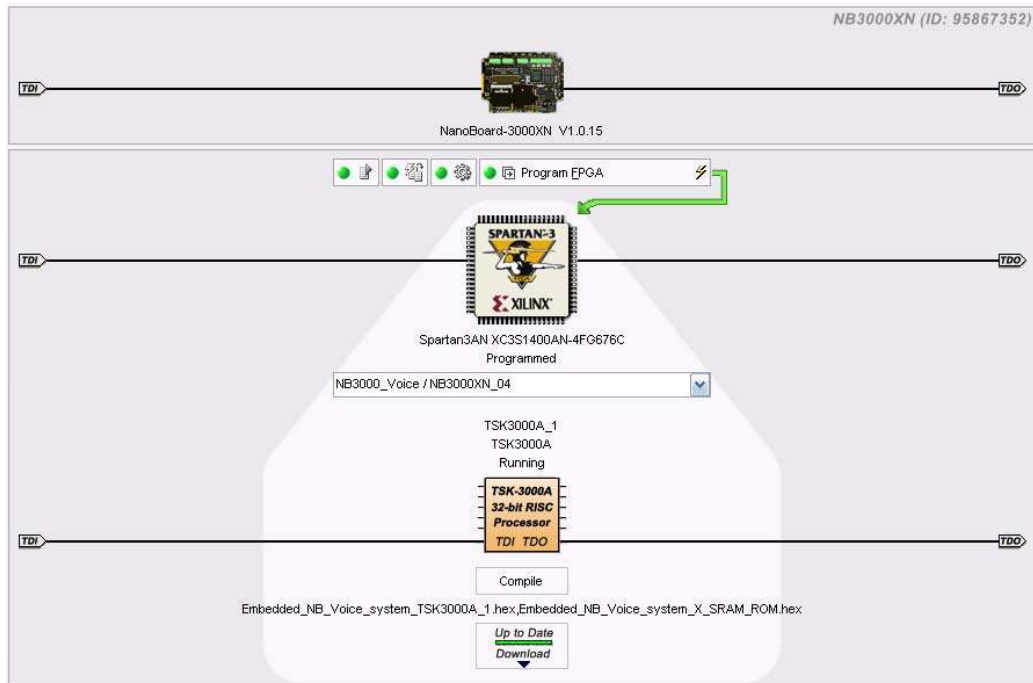
## **7.5. Project Links and Hierarchy**

The FPGA design for the NB3000 speech synthesizer forms the embedded system hardware, and on top of this platform is built the embedded software design – largely the topic of this dissertation.

The embedded project and FPGA project are linked together, and the hierarchy of documentation is shown in appendix C.3 for reference.

## 7.6. Conclusion

The FPGA Project was synthesized, built and downloaded from the Altium Designer Software in the Devices View (shown in Figure 24).



**Figure 24** Devices View in Altium Designer Software. This is where the FPGA and Embedded projects are downloaded to the target device.

After some debugging and fixing of filter kernel code, the design operates and provides a means of looking into the concept of speech generated by movement further.

## Chapter 8.

# User Interface Research

### 8.1. Introduction

One of the original objectives of the project (see the Project Specification in Appendix A) was to research how this system might be used with people whose native languages differ.

Unfortunately, due to project time delays and constraints, it was not possible to conduct a thorough research programme in this regard. However, some anecdotes have been gleaned by people exposed to the project along the way.

### 8.2. Robotic Sound

Although the synthesizer works reasonably well for a first attempt, the first response that has been encountered when showing it to colleagues has been one of bewilderment followed by comments indicating that it sounds very much like a robot

from an old movie. This is probably most due to the pulse source generation and could be alleviated with better error residual signals as a source.

### **8.3. User interface problems**

Problems have been noted with the user interface involved. The most notable are:

1. The TFT screen is too small to be practical.
2. The fact that buttons have to be pressed to choose between voiced or fricative sounds makes it difficult to use.

### **8.4. Conclusion**

Although the idea is novel, it requires a lot more thought and research before it could be turned into a practical commercial product that would be useful to normal users.

However, that is always the case with the first step in exploring a new idea.

## Chapter 9.

# Conclusion

### 9.1. Introduction

Referring again to the project specification (Appendix A) the first aim of the project was to research and implement a Linear Predictive Coding based synthesizer that was to be controlled by a user touch screen panel. The second aim was to look at the feasibility of such a device as a possible means of assisting speech impaired people.

By and large, both of these aims have been achieved, though ideally more work should be done on the user interface study in order to get more ideas of how it could be made to work.

## **9.2. Further work and research**

Further research should be undertaken in the following areas:

### **9.2.1. Improve LSP interpolation method to include gain**

It was noted that the LPC vectors were interpolated nicely, but the gains between each vector interpolated set jumped markedly, and this impaired the performance of the design. This would be a good starting point as it most likely has a straightforward solution.

### **9.2.2. Find better expression methods**

Using the Y-axis to control pitch was primarily motivated by the need to have expression. It does however limit the use of the screen. It would be better to find other methods of controlling the pitch in order to free up TFT space to make the mapping of voiced and affricate sounds easier and better.

### **9.2.1. Implement the LPC and LSP operations in Real-Time**

The final and perhaps most useful extension to this project would be to implement the LSP interpolation and LPC analysis functions in the embedded system for real-time functionality. This would allow the system to be completely stand-alone, and users could speak corpus directly into the device via a microphone in order to bottle their own voice tonality and style within it.



---

### 9.2.2. Adapt the current design to Music generation

This design certainly forms the basis of what potentially could be a music synthesizer. The LPC formant maps do not necessarily have to be models of human speech – given that the order of LPC filtering in the system could be quite high. Other sounds, such as animals, birds, or even different types of musical instrument could be modelled in this system.

The pulse and noise sources could be replaced by inputs that would come from a vocal microphone or instruments such as electric guitars, to extend the usefulness of the device into the musical effects arena.

### 9.3. Conclusion

The project programme of researching and implementing suitable speech processing techniques – namely LPC and LSP – has been explored and implemented.

Voice characteristics were researched and discussed in **Chapter 2**, along with processing techniques that prevail.

**Chapter 3** discussed the characteristics of the human speech organs and how these are modelled in digital systems and introduced the concept of interpolation of speech frames.

---

**Chapter 4** covered details of Linear Predictive Coding as used in this design.

**Chapter 5** detailed the problem of morphing LPC frames and mapping them to the TFT touch screen surface, and covered the theory behind using Line Spectrum Pairs as a means of performing the interpolation.

**Chapter 6** covered the design and implementation of these methods using Octave scripts for the non-real-time part of the design. It went on to discuss the design of the embedded system firmware which runs on the Altium Nanoboard 3000.

**Chapter 7** Introduces the Nanoboard 3000 FPGA/Embedded design platform, and briefly covered the parts that have been put to use in this project.

**Chapter 8** briefly discusses the anecdotal research and feedback gleaned from colleagues, pointing to some useability issues that could be addressed in future projects.

**Chapter 9** Concludes this dissertation.

## References

- Altium Limited, 2009, *Nanoboard 3000 Series English Documentation*, viewed 28<sup>th</sup> October 2009,  
<<http://wiki.altium.com/display/ADOH/NanoBoard+3000+Series>>
- ASPi, 1996, *MELP Vocoder Algorithm*, Atlanta Signal Processors Inc, Atlanta GA, USA
- Breen, A 1992, ‘Speech synthesis models: a review’, *Electronics and Communication Engineering Journal*, February 1992
- Bauer, B and Torick, E 1966, ‘Researches in loudness measurement’, *IEEE Transactions on Audio and Electroacoustics*, Vol. 14, September, pp.141–151
- Cole, RA, Mariani, J, Uszkoreit, H, Zaenen, A and Zue, V 1996, *Survey of the State of the Art in Human Language Technology*, National Science Foundation European Commission, viewed 21<sup>st</sup> May 2009,  
<<http://cslu.cse.ogi.edu/HLTsurvey/HLTsurvey.html>>
- Goncharoff, V and Kaine-Krolak, M 1995, ‘Interpolation of LPC Spectra via Pole Shifting’, *1995 International Conference on Acoustics, Speech, and Signal Processing*, Vol.1 pp.780-783
- Hong, G 2007, *The Vocal Tract and Related Speech Organs*, Fu Jen Catholic University Graduate Institute of Linguistics, viewed 22<sup>nd</sup> October 2009,  
<<http://www.ling.fju.edu.tw/phonetic/base2.htm>>

- Kabal, P and Ramachandran, P 1986, 'The Computation of Line Spectral Frequencies Using Chebyshev Polynomials', *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34, No.6, December, pp.1419-1426
- Keiler, F Arfib, D and Zölzer, U (ed.) 2008, 'Source-Filter Processing', in *DAFX Digital Audio Effects*, pp.299-372, Wiley & Sons, Chichester
- Leis, J 2008, *ELE4607 Advanced Digital Communications: course notes*, University of Southern Queensland, Toowoomba
- McLoughlin, I and Chance, R 1997, 'LSP analysis and processing for speech coders', *Electronics Letters*, Vol.33 No.9, April, pp.743-744
- Morris, R and Clements, M 2002, 'Modification of Formants in the Line Spectrum Domain', *IEEE Signal Processing Letters*, Vol. 9 Issue 1, January, pp.19-21
- Oppenheim, A and Schafer, R 1975, *Digital Signal Processing*, pp.409-438, Prentice-Hall, USA
- Paliwal, K and Atal, B 1991, 'Efficient vector quantization of LPC parameters at 24 bits/frame', *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, IEEE, Toronto, Canada, May, pp. 661-664
- Paliwal, K 1995, 'Interpolation properties of linear prediction parametric representations', *Proceedings of the 4th European Conference on Speech Communication and Technology*, EUROSPEECH-95, Madrid, Spain, September, pp. 1029-1033
- Rabiner, L and Schafer, R 1978, *Digital Processing of Speech Signals*, pp.396-461, Prentice-Hall, USA
- Tokuda, K, Yoshimura, T, Masuko, T, Kobayashi, T & Kitamura, T 2000, 'Speech parameter generation algorithms for hmm-based speech synthesis', Department of Computer Science, Nagoya Institute of Technology, Nagoya Japan
- The University of Helsinki Music Research Laboratory 2008, *Spectutils Home Page*, viewed 2<sup>nd</sup> June 2009, <<http://www.music.helsinki.fi/research/spectutils/>>

- Schlichthärle, D 2000, *Digital Filters Basics and Design*, pp.233-238, Springer-Verlag, Berlin Heidelberg
- Stein, J 2000, 'Digital Signal Processing, A Computer Science Perspective', pp.383-385 Wiley & Sons, USA
- Synthtopia 2009, *The Voder, A Speech Synthesizer From 1939*, Synthesizer online encyclopedia article, viewed 22<sup>nd</sup> May 2009, <<http://www.synthtopia.com/content/2009/02/06/voder-speech-synthesizer/>>
- Vaseghi, S, Yan, Q and Ghorshi, A 2009, 'Speech Accent Profiles: Modeling and Synthesis', *IEEE Signal Processing Magazine*, Vol.26 No. 3, May, pp. 69-74
- Hui, Z 1989, 'The design of linear shift-variant filters using the interpolation technique', *Fourth IEEE Region 10 International Conference, TENCON '89*, Bombay, India, November, pp.263-265

## **Appendix A**

# **Project Specification**

The University of Southern Queensland

FACULTY OF ENGINEERING AND SURVEYING

## ENG4111/4112 Research Project

### PROJECT SPECIFICATION

FOR: Benjamin Warren Dennis JORDAN

TOPIC: A TOUCH-SCREEN CONTROLLED "LINEAR PREDICTIVE SYNTHESIZER" FOR ACCESSIBILITY APPLICATIONS.

SUPERVISOR(S): Mr. Mark Phythian

ENROLMENT: ENG4111 – Semester 1, 2009

ENG4112 – Semester 2, 2009

PROJECT AIM: To design and prototype a synthesizer based on LPC style algorithms, but controlled not by a transmitted bit stream of compressed audio, but rather by a touch-screen LCD based user interface. The secondary aim is then to research such a product's useability as an assistive device for those suffering some form of speech loss.

SPONSORSHIP: Altium Ltd.



PROGRAMME: Issue A, 20<sup>th</sup> March 2009

1. Research appropriate Digital Signal Processing techniques for Speech Synthesis
2. Research characteristics of vocal sounds including formants and sibilants
3. Design an appropriate noise and pulse source and time-varying filter combination for re-creating these sounds
4. Design LCD touch-screen controller for controlling the noise sources and filter(s)
5. Research effectiveness of touch screen for controlling synthesized vocal tract response
6. Submit an academic dissertation on the research


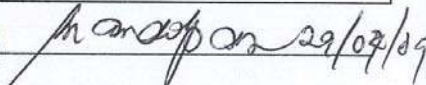
As Time Permits:

7. Develop custom PCB and Enclosure for prototype
8. Extend research of user interface effectiveness with patient studies

AGREED:

 (Student)	 (Supervisor)
DATE: 20-3-2009	DATE: 31/3/09.

Examiner/Co-Examiner:

 28/4/09 /  29/07/09

## **Appendix B**

# **Octave Scripts and Functions**



## B.1 The `calc_lpc.m` Octave Function

The following code from (Keiler, F 2002, ‘Source-Filter Processing’, in Zölzer, U (ed.) 2008) is provided here for clarity. This function was provided to work with other example code in the edited book as the MATLAB `lpc()` function at the time was known to not work correctly all the time.

```
function [a,g]=calc_lpc(x,p)
% calculate LPC coeffs via autocorrelation method
% Similar to MATLAB function "lpc"
% x: input signal
% p: prediction order
% a: LPC coefficients
% g: gain factor
% (c) 2002 Florian Keiler
R=xcorr(x,p); % autocorrelation sequence R(k) with k=-p,...,p
R(1:p)=[]; % delete entries for k=-p,...,-1
if norm(R)~=0
    a=levinson(R,p); % Levinson-Durbin recursion
% a=[1, -a_1, -a_2,..., -a_p]
else
    a=[1, zeros(1,p)];
end
R=R(:)'; a=a(:)'; % row vectors
g=sqrt(sum(a.*R)); % gain factor
end
```

## B.2 The lpc\_gen\_figs.m Octave Function

```

function [a g e] = lpc_gen_figs(fname);
%
% LPC calculation of prediction error and spectra
% adapted from code by Keiler, F (2002)
%
% Adapted by: Ben Jordan
%
% From Reference:
% Keiler, F 2002, 'Source-Filter Processing', in Zölzer, U (ed.)
2008,
% 'DAFX Digital Audio Effects', p.306, Wiley & Sons, Chichester
%
n0=5000; %start index
N=1024; %block length
Nfft=1024; % FFT length
p=20; %prediction order
n1=n0+N-1; %end index
pre=p; %filter order= no. of samples required before n0

[xin,Fs]=wavread(fname,[n0-pre n1]);
xin=xin(:,1)';
win=hamming(N)';
x=xin((1:N)+pre); % block without pre-samples

[a,g]=calc_lpc(x.*win,p); % calculate LPC coeffs and gain
% a=[1, -a_1, -a_2,..., -a_p]
g_db=20*log10(g) % gain in dB

ein=filter(a,1,xin); % pred. error
e=ein((1:N)+pre); % without pre-samples
Gp=10*log10(sum(x.^2)/sum(e.^2)) % prediction gain

Omega=(0:Nfft-1)/Nfft*Fs/1000; % frequencies in kHz
offset=20*log10(2/Nfft); % offset of spectrum in dB
A=20*log10(abs(fft(a,Nfft)));

```

```
H_g=-A+offset+g_db; % spectral envelope
X=20*log10(abs(fft(x.*win,Nfft)));
X=X+offset;

n=0:N-1;
figure(1)
clf
subplot(211)
plot(n,e)
title(strcat('Time signal of pred. error e(n) for sample "', fname,
''))
xlabel('n \rightarrow')
axis([0 N-1 -inf inf])

subplot(212)
plot(Omega,X)
hold on
plot(Omega,H_g,'r','Linewidth',1.5)
hold off
title(strcat('Magnitude spectra |X(f)| and |G*H(f)| in dB for "',
fname, ''))
xlabel('f/kHz \rightarrow')
axis([0 8 -inf inf])

out = filter(1, a, ein);
out = out';
figure(2);
subplot(211);
plot(x);
title(strcat('Input signal from "', fname, ''));
xlabel('Sample (n)\rightarrow');

subplot(212);
plot(out);
title('Time signal of LPC synthesized output');
xlabel('Sample (n)\rightarrow');
g = Gp;
end
```

### B.3 The generate\_coefs.m Octave Function

```

function [a g] = generate_coefs(infile, outfile, lpc_order)
%
% This function creates a file named '<outfile>.h' - a C
% language header file for a 16-bit fixed-point constant array
% of Linear Predictive Coding synthesis filter coefficients
% of order <lpc_order>. <infile> is the input sound sample file
% Microsoft .WAV format.
%
% The LPC prediction coefficients are returned in [a] along with
% the gain coefficient g.
%
% If <outfile> does not yet exist in the current path it will be
% created. If it does exist, it will be appended with the new co-
% efficients.
%
% The source file name, lpc_order and other details will be added
% as comments in the header file before the array declaration.
%
% Author: Ben Jordan
%
    N = 1024;
    [xin, fs] = wavread(strcat(infile, '.wav'), [N N+N]);
    % Set up hamming window
    win = hamming(N)';
    % For each wavelet, get LPC
    x = xin(1:N).*win';
    [a g] = calc_lpc(x, lpc_order);
    Mxa = max(a);
    Mna = min(a);
    if (max(a) >= 3 || min(a) <= -3)
        fid = 0;
        error('LPC coeffs outside comfortable range for fixed
point.');
```

```

if (fid == -1)
    error(msg);
    return;
end

fprintf(fid, '\n/*\n * LPC Coefficients for sample %s', infile);
fprintf(fid, '\n * Date/Time created: ');
fprintf(fid, '%d-', localtime(time()).year + 1900);
fprintf(fid, '%d-', localtime(time()).mon + 1);
fprintf(fid, '%d ', localtime(time()).mday);
fprintf(fid, '%d:', localtime(time()).hour);
fprintf(fid, '%d:', localtime(time()).min);
fprintf(fid, '%d', localtime(time()).sec);
fprintf(fid, '\n * LPC Order: %d', lpc_order);
fprintf(fid, '\n * Sampling rate (Fs): %g', fs);
fprintf(fid, '\n * Minimum LPC coefficient: %g', Mna);
fprintf(fid, '\n * Maximum LPC coefficient: %g', Mxa);
fprintf(fid, '\n * NOTE: LAST element of array is prediction gain,
FIRST is first coefficient.');
```

```

    fprintf(fid, '\n          */\n\n#ifdef          COEFF_LEN\n#define
COEFF_LEN\t\t%d\n#endif\n', length(a));
    fprintf(fid, '\n#ifdef Q\n#error Q must be defined! It is the
floating-point to fixed point scaling factor.\n#endif ');
    fprintf(fid, '\nconst samp_t coeff_%s[COEFF_LEN+1] = \n{\n',
infile);
    for IDX = 1:length(a)
        fprintf(fid, '\t(samp_t) (%1.20f\t* Q),\n', a(IDX));
    end
        fprintf(fid, '\t(samp_t) (%1.20f\t* Q)\t/* <--- Prediction Gain
*/\n};\n', g);
    fclose(fid);
end

```

## B.4 The ow\_pole\_mapping\_plot.m Octave Script

```
%  
% ow_pole_mapping_plot.m  
%  
% Calls calc_lpc to get coefficients of a set of frames of  
transitional  
% speech, then subsequently plots the pole locations of the LPC  
vectors  
% and their corresponding LSP roots - both on unit circles.  
%  
% Author: Ben Jordan  
%  
N = 1024;  
SAMPLES = N*24;  
[xin, fs] = wavread('ow.wav', [0 SAMPLES-1]);  
  
% Set up hamming window and unit circle for z-plane plot  
win = hamming(N)';  
hold off;  
figure(1);  
  
% Draw a unit circle for clarity  
draw_unit_circle;  
% For each wavelet, get LPC and plot poles of LPC reconstruction  
filter  
for IDX = 1:N/2:SAMPLES-N+1  
    x = xin(IDX:IDX+N-1).*win';  
    a = calc_lpc(x, 10);  
  
    R_A = roots(a);  
  
    if (IDX == 1)  
        plot(real(R_A), imag(R_A), '*b', 'markersize', 3);  
    else  
        if IDX < (SAMPLES-N+1)  
            plot(real(R_A), imag(R_A), '*k', 'markersize', 1);
```

```
        else
            plot(real(R_A), imag(R_A), '*r', 'markersize', 3);
        end
    end
end

end

title('Pole paths from "aa" to "uw".');
xlabel('Real');
ylabel('Imaginary');
grid on;
hold off;
figure(2);

% Draw a unit circle for clarity
draw_unit_circle;

% For each wavelet, get LSP and plot poles
for IDX = 1:N:SAMPLES-N+1
    x = xin(IDX:IDX+N-1).*win';
    a = calc_lpc(x, 10);
    [Ph Qh P Q] = lsplpc(a);
    if IDX == 1
        plot(real(P), imag(P), '*b', 'markersize', 3);
    else if IDX < (SAMPLES-(N+1))
        plot(real(P), imag(P), '*k', 'markersize', 1);
    else
        plot(real(P), imag(P), '*r', 'markersize', 3);
    end
end
end

end

title('LSP root paths from "aa" to "uw".');
xlabel('Real');
ylabel('Imaginary');
hold off;
```

## B.5 The lsplpc.m Octave Function

```
function [Ph Qh P Q] = lsplpc(LPC)
% Calculates Line Spectrum Pairs from LPC coefficients of order
% of LPC.
%
% References:
% Kabal P, and Ramachandran R 1986, "The Computation of Line
% Spectral Frequencies Using Chebyshev Polynomials", IEEE
% Transactions on Acoustics, Speech and Signal
% Processing, Vol. ASSP-34, No. 6, December 1986.
%
% Stein, J 2000, "Digital Signal Processing - A Computer
% Science Perspective", pp.383-385, Wiley & Sons, USA.
%
% Author: Ben Jordan
%
    if (nargin ~=1)
        help lsplpc;
        return;
    end
    order = length(LPC)-1;
    Qh = zeros(1,order+2);
    Ph = Qh;
    Ph(1) = LPC(1);
    Qh(1) = LPC(1);
    for ix = 1:order
        Ph(ix+1) = LPC(ix+1) + LPC(order-ix+2);
        Qh(ix+1) = LPC(ix+1) - LPC(order-ix+2);
    end
    Ph(order+2) = 1;
    Qh(order+2) = -1;
    % LSPs are defined as the roots of the above equations.
    P = roots(Ph);
    Q = roots(Qh);
end
```



## B.6 The lpclsp.m Octave Function

```

function [a] = lpclsp(P, Q)
% Calculates LPC Coefficients from Line Spectral Pairs
%
% Arguments: Ph, Qh, == Vector arrays of the LSPs in
%             Coefficient form (not frequencies)
%             order == the order of the LPC system
% Returns:   a == Vector of LPC coefficients
% References:
% Kabal P, and Ramachandran R 1986, "The Computation of Line
% Spectral Frequencies Using Chebyshev Polynomials", IEEE
% Transactions on Acoustics, Speech and Signal
% Processing, Vol. ASSP-34, No. 6, December 1986.
%
% Stein, J 2000, "Digital Signal Processing - A Computer
% Science Perspective", pp.383-385, Wiley & Sons, USA.
%
% Author: Ben Jordan
%
    if (nargin ~= 2)
        help lpclsp;
        return;
    end
    if (length(Q) ~= length(P))
        help lpclsp;
        return;
    end
    order = length(P) - 2;
    o2 = order/2;
    a = zeros(1, order+1);
    a(o2+1:-1:1) = 0.5.*(Q(o2+1:-1:1) + P(o2+1:-1:1));
    a(o2+2:order+1) = 0.5.*(Q(o2+1:-1:2) - P(o2+1:-1:2));
    % compensate for root signs (i.e. every other LSP is in
    % "negative frequency"
    a(2:2:order+1) = -a(2:2:order+1);
end

```

## B.7 The `expnd.m` Octave Function

```
function P = expnd(A)
%
% Assuming A represents roots of a polynomial, expands the
% roots to get to the polynomial P. A and P are row vectors
% and P is length(A)+1;
%
% It assumes that the roots are fully factored, for example:
% (x + a)(x + b)(x + c)...
% but !NOT!:
% (jx + a)(kx + b)(lx + c)...
%
% Author: Ben Jordan
%
    if (nargin ~= 1)
        help expnd;
        return;
    end
    N = length(A);
    P = zeros(1, N+1);
    P(1) = 1;
    %P(2) = A(1);
    for I=1:N
        for J=N+1:-1:2
            P(J) = P(J) + P(J-1)*A(I);
        end
    end
    % just to remove all the residual small imaginary parts...
    P = real(P);
end
```

## B.8 The lpc\_interp.m Octave Function

```

function [A] = lpc_interp(FA, N)
%
% X-dimension LPC Coefficient Interpolator
%
% This function uses the linear interpolation provided by
% the Octave/MATLAB interp1 function. Linear interpolation
% is used to prevent coefficient values from exceeding
% existing coefficient values - thereby reducing likelihood
% of there being an unstable set.
%
% Arguments: FA is a m-order by n-vector matrix of LPC vectors.
%           N is the desired number of output vectors.
% Returns:  [A] is the resulting m-order by N vectors.
%
% Author: Benjamin Jordan
%
    if (nargin < 2)
        help lpc_interp
        return;
    end
    lFA = length(FA(1,:)); % number of input vectors
    wFA = length(FA(:,1)); % number of coefficients in each vector
    A = zeros(wFA, N); % new array will be N by w
    x = 1:lFA;
    step = (lFA-1)/N;
    xi = 1:step:lFA-step;
    for idx=1:wFA
        y = FA(idx,:);
        % linear interpolation across X direction
        yi = interp1(x, y, xi);
        A(idx,:) = yi;
    end
end

```

## B.9 The lsp\_interp.m Octave Function

```

function [IP IQ] = lsp_interp(P1, Q1, P2, Q2, N)
%
% Interpolate between two sets of Line Spectrum Pairs
% using the nearest angular neighbour.
%
% Arguments: P1, Q1 - the first LSP set
%             P2, Q2 - the last LSP set
%             N - the number of intermediate sets req'd.
%
% Returns:   PI, QI - arrays of interpolated LSP vectors
%
% Assumptions: 1. All input LSPs are of equal length.
%               2. All have complex frequency representation
%                  (i.e. there are complex conjugates).
%
% Author: Ben Jordan
%
    if (nargin ~= 5)
        help lsp_interp;
        return;
    end
% sort provides crude mechanism for finding nearest
% neighbouring LSP. BUT WORKS!!
O = length(P1);
IP = zeros(O, N+2);
IQ = IP;
IP(:,1) = P1;
IQ(:,1) = Q1;
IP(:,N+2) = P2;
IQ(:,N+2) = Q2;
[P1] = sort(arg(P1));
[Q1] = sort(arg(Q1));
[P2] = sort(arg(P2));
[Q2] = sort(arg(Q2));
for I = 1:O

```

```
Pstep = (P1(I)-P2(I))/N;  
Qstep = (Q1(I)-Q2(I))/N;  
for J = 2:N+1  
    IP(I,J)=cos(P1(I)-Pstep*J)+i.*sin(P1(I)-Pstep*J);  
    IQ(I,J)=cos(Q1(I)-Qstep*J)+i.*sin(Q1(I)-Qstep*J);  
end  
end  
end
```

## B.10 The gen\_all\_lpc.m Octave Function

```
function [LI] = gen_all_lpc(sOUTFILE, sQ, sT, order, n)
%
% This function uses the calc_lpc function and lpc_interp
% function to generate and save out a C header file
% containing the interpolated LPC coefficient vectors.
%
% Arguments:
% sOUTFILE = name of the C header file to be written (string)
% sQ = C macro name used for fixed-point scaling (string)
% sT = C data type name used for coefficient declaration (string)
% order = LPC filter order
% n = number of desired output vectors (used for interpolation)
%
% Results:
% LI = a matrix sized (order+2, n) of LPC coefficient vectors.
% It is (order+2) rows because a0 coefficient (always 1) for
% the synthesis filter is added, plus the prediction gain is
% suffixed at the end of each coefficient vector.
%
% This function assumes you have a set of basic vowel sounds
% recorded in .wav file format - 16-bits, 22.05Ksps, mono.
% The file names used currently are "uw.wav", "ue.wav", "rr.wav",
% "o.wav", "e.wav", "ah.wav", "a.wav", and "iii.wav".
%
% Author: Ben Jordan.
%
if (nargin < 5)
    usage(['[LI] = gen_all_lpc(sOUTFILE, sQ, sT, order, n)']);
    return;
end
N = 1024;
fst = 1024; % Start a reasonable length into file
lst = fst+N-1;

% read in eight basic vowel sounds:
```

```

[xuw Fs] = wavread('uw.wav', [fst lst]);
[xue]     = wavread('ue.wav', [fst lst]);
[xrr]     = wavread('rr.wav', [fst lst]);
[xo ]     = wavread('o.wav', [fst lst]);
[xe ]     = wavread('e.wav', [fst lst]);
[xah]     = wavread('ah.wav', [fst lst]);
[xa ]     = wavread('a.wav', [fst lst]);
[xii]     = wavread('iii.wav', [fst lst]);

% transpose to column vectors, combine in to an 8 by N array:
X = [xuw(:,1)'; xue(:,1)'; xrr(:,1)'; xo(:,1)'; xe(:,1)';
     xah(:,1)';...
     xa(:,1)'; xii(:,1)'];

% apply a hamming window to all the samples and calculate LPC
vectors:
LP = zeros(8, order+2);
GP = zeros(1, 8); % storage for gains for use later on.
window = hamming(N)';
for I=1:8
    X(I,:) = X(I,:).*window;
    [LP(I,1:order+1), LP(I,order+2)] = calc_lpc(X(I,:), order);
end

% Interpolate using linear interpolation
LI = lpc_interp(LP', n)';
Mxa = max(max(LI));
Mna = min(min(LI));

% Dump LPC coeff. vectors to a outfile:
[fid msg] = fopen(sOUTFILE, 'a+'); %open for append
if (fid == 0)
    error(msg);
    return;
end

fprintf(fid, '\n/*\n * Interpolated LPC Coefficients for vowels');
fprintf(fid, '\n * Date/Time created: ');
fprintf(fid, '%d-', localtime(time()).year + 1900);
fprintf(fid, '%d-', localtime(time()).mon + 1);
fprintf(fid, '%d ', localtime(time()).mday);
fprintf(fid, '%d:', localtime(time()).hour);
fprintf(fid, '%d:', localtime(time()).min);
fprintf(fid, '%d', localtime(time()).sec);

```

```

fprintf(fid, '\n * LPC Order: %d', order);
fprintf(fid, '\n * Sampling rate (Fs): %g', Fs);
fprintf(fid, '\n * Minimum LPC coefficient: %g', Mna);
fprintf(fid, '\n * Maximum LPC coefficient: %g', Mxa);
fprintf(fid, '\n * NOTE: LAST element of each array is prediction
gain, FIRST is first coefficient.');
```

```

fprintf(fid,
        '\n
        */\n\n#ifnndef
        COEFF_LEN\n#define
COEFF_LEN\t\t%d\n#endif\n', length(LI(1,:)));
fprintf(fid, '\n#ifnndef %s\n#error %s must be defined! It is the
floating-point to', sQ, sQ);
fprintf(fid, ' fixed point scaling factor.\n#endif ');
fprintf(fid, '\nconst %s coeffs[%d][COEFF_LEN] = \n{\n', sT, n);
len = length(LI(:,1));
wid = length(LI(1,:))-1;
for JDX = 1:len
    fprintf(fid, '\t{\n');
    for IDX = 1:wid
        fprintf(fid, '\t\t(%s) (%1.20f \t* %s),\n', sT, LI(JDX,
IDX), sQ);
    end
    fprintf(fid, '\t\t(%s) (%1.20f \t* %s)\t/* <--- Prediction Gain
*/\n', sT, LI(JDX, wid+1), sQ);
    if (JDX < len)
        fprintf(fid, '\t},\n');
    else
        fprintf(fid, '\t}\n};\n');
    end
end
fprintf(fid, '\n/* ----- end of coefficients -----
----- */\n');
fclose(fid);
end

```



## B.11 The gen\_all\_lsp.m Octave Function

```

function [C G FS] = gen_all_lsp(sOUTNAME, sQ, sT, order, n)
%
% Generates array of LPC coefficients (LPC of order ORDER)
% Calls lower level functions to interpolate between frames
%
% Author: Ben Jordan
%
[A gx gz S] = plot_interp("uw.wav", "o.wav", order, n-2, 0);
C = A; G = [gx, gz]; FS = S;
[A gx gz S] = plot_interp("o.wav", "ue.wav", order, n-2, 0);
C = [C;A]; G = [G, gz]; FS = [FS;S];
[A gx gz S] = plot_interp("ue.wav", "rr.wav", order, n-2, 0);
C = [C;A]; G = [G, gz]; FS = [FS;S];
[A gx gz S] = plot_interp("rr.wav", "ah.wav", order, n-2, 0);
C = [C;A]; G = [G, gz]; FS = [FS;S];
[A gx gz S] = plot_interp("ah.wav", "a.wav", order, n-2, 0);
C = [C;A]; G = [G, gz]; FS = [FS;S];
[A gx gz S] = plot_interp("a.wav", "e.wav", order, n-2, 0);
C = [C;A]; G = [G, gz]; FS = [FS;S];
[A gx gz S] = plot_interp("e.wav", "iii.wav", order, n-2, 0);
C = [C;A]; G = [G, gz]; FS = [FS;S];
[A gx gz S] = plot_interp("iii.wav", "rr.wav", order, n-2, 0);
C = [C;A]; G = [G, gz]; FS = [FS;S];

% Some edification for the user on array sizes:
disp(size(C));
disp(size(G));

% Find peaks in coefficient values for remarks in header file:
Mxa = max(max(C));
Mna = min(min(C));

% Dump LPC coeff. vectors to a outfile:
[fid msg] = fopen(sOUTNAME, 'a+'); %open for append
if (fid == 0)

```

```

    error(msg);
    return;
end
fprintf(fid, '\n/*\n * LSP Interpolated LPC Coefficients for
vowels');
fprintf(fid, '\n * Date/Time created: ');
fprintf(fid, '%d-', localtime(time()).year + 1900);
fprintf(fid, '%d-', localtime(time()).mon + 1);
fprintf(fid, '%d ', localtime(time()).mday);
fprintf(fid, '%d:', localtime(time()).hour);
fprintf(fid, '%d:', localtime(time()).min);
fprintf(fid, '%d', localtime(time()).sec);
fprintf(fid, '\n * LPC Order: %d', order);
fprintf(fid, '\n * Sampling rate (Fs): 22.05 KHz');
fprintf(fid, '\n * Minimum LPC coefficient: %g', Mna);
fprintf(fid, '\n * Maximum LPC coefficient: %g', Mxa);
fprintf(fid, '\n * NOTE: LAST element of each array is prediction
gain, FIRST is first coefficient.');
```

```

fprintf(fid,
        '\n          */\n\n#ifdef          COEFF_LEN\n#define
COEFF_LEN\t\t%d\n#endif\n', length(C(1,:))+1);
fprintf(fid, '\n\n#ifdef %s\n#error %s must be defined! It is the
floating-point to', sQ, sQ);
fprintf(fid, ' fixed point scaling factor.\n#endif ');
len = length(C(:,1));
fprintf(fid, '\nconst %s coeffs[%d][COEFF_LEN] = \n{\n', sT, len);
for JDX = 1:len
    fprintf(fid, '\t{\n');
    for IDX = 1:order+1
        fprintf(fid, '\t\t(%s) (%1.20f \t* %s),\n', sT, C(JDX, IDX),
sQ);
    end
    Gidx = floor((JDX+n)/n);
    fprintf(fid, '\t\t(%s) (%1.20f \t* %s)\t/* <--- Prediction Gain
*/\n', sT, G(Gidx), sQ);
    if (JDX < len)
        fprintf(fid, '\t},\n');
    else
        fprintf(fid, '\t}\n};\n');
end
end

```

```
end
fprintf(fid, '\n/* ----- end of coefficients -----
----- */\n');
fclose(fid);
end
```

## B.12 The plot\_interp.m Octave Function

```

function [A gx gz S2] = plot_interp(fname1, fname2, LPC_ORDER,
STEPS, plt)
%
% Interpolate between LPC frames taken from sound files FNAME1
% and FNAME2. The interpolation direction is FROM FNAME1 TO
% FNAME2. LPC_ORDER is the linear prediction order used, STEPS is
% the number of steps (desired frames).
%
% If PLT == 1 , the log-magnitude FFT versus frame number
% will be plotted to a 3D figure using surf().
%
% If PLT == 2 , the z-plane plot of interpolated LPC coefficients
% is produced.
%
% Author: Ben Jordan
%
    if (nargin < 4)
        help plot_interp
        return;
    end
    [A IP IQ ax az gx gz FS] = lpclsplpc(fname1, fname2, LPC_ORDER,
STEPS);
    fpts = 256;
    N = length(A(:,1));
    OS= 20*log10(2/fpts);
    G = 20*log10(gx);
    S=zeros(N,fpts);
    for I=1:N
        FT=fft(A(I,:),fpts);
        S(I,:)= -20*log10(abs(FT))+OS+G;
    end
    S2=S(:,1:fpts/2);
    w = (0:2:fpts-1)/fpts*2*FS/10000;
    n = 1:N;

```

```
% If user flags plt then plot this transition:
if (margin > 4)
    if (plt == 1)
        figure();
        surf(w, n, S2);
        title(
            strcat('LPC Interpolation from ', fname1,
                ' to ', fname2, '')
        );
        xlabel("Frequency (KHz)");
        ylabel("Frame (n)");
        zlabel("H(f,n) (dB)");
        % Set azimuth and elevation
        view(-37.5, 30); refresh();
    else
        if (plt == 2)
            figure();
            draw_unit_circle;
            for I=1:N
                R = roots(A(I,:));
                plot(real(R), imag(R), "*", "markersize", 1);
            end
            title(
                strcat('LPC Poles Interpolated from ', fname1,
                    ' to ', fname2, '')
            );
            xlabel("Real"); ylabel("Imaginary"); grid on;
        end
    end
end
end
```

## B.13 The lpclsplpc.m Octave Function

```

function [A IP IQ ax az gx gz FS] = lpclsplpc(fname1, fname2, order,
interp);
%
% Take two sound files, calculate the LPC from a frame in
% each file, then convert these to LSPs. Interpolate the
% LSPs using lsp_interp.m, then expand the interpolated roots
% back into predictor filter coefficients.
%
% Arguments: FNAME1, FNAME2 - two input .WAV files of speech
%           ORDER - LPC Order used
%           interp - number of intermediate frames needed.
%
% Returns: A - a size(interp+2, order) matrix containing the
%           interpolated coefficient vectors.
%           IP, IQ - interpolated LSP vectors.
%           ax, az - original .WAV file LPC coefficient frames
%           gx, gz - corresponding LPC predictor gains.
%
% Author: Ben Jordan
%
% bring in the audio data
[X FS] = wavread(fname1);
[Z] = wavread(fname2);
N = 1024;
win = hamming(N)';
x = X(2048:2048+N-1)'.*win;
z = Z(2048:2048+N-1)'.*win;
% generate LPC vector and gain
[ax gx] = calc_lpc(x, order);
[az gz] = calc_lpc(z, order);
% compute LSPs
[Phx Qhx Px Qx] = lsplpc(ax);
[Phz Qhz Pz Qz] = lsplpc(az);
% Interpolate <interp> inbetween LSPs:
[IP IQ] = lsp_interp(Px, Qx, Pz, Qz, interp);

```

---

```
% compute back to LPC vector:
A = zeros(interp+2, length(ax));
for I = 1:interp+2
    A(I,:) = lpclsplpc(expnd(IP(:,I)), expnd(IQ(:,I)));
end
end
```

## B.14 The draw\_unit\_circle.m Octave Function

```
% Draw a unit circle for clarity
%
% Adapted by: Ben Jordan
%
% From Reference: Leis J, "Digital Signal Processing: A MATLAB
% based tutorial approach", 2002, p108, Research Studies Press,
% Baldock.
%
function draw_unit_circle()
    theta = 0:pi/100:2*pi;
    c = 1*exp(j*theta);
    plot(real(c), imag(c));
    hold on;
end
```



## **Appendix C**

# **Altium Designer FPGA Project Schematic and OpenBus Diagrams**

### C.1 FPGA Top-level Schematic Diagram

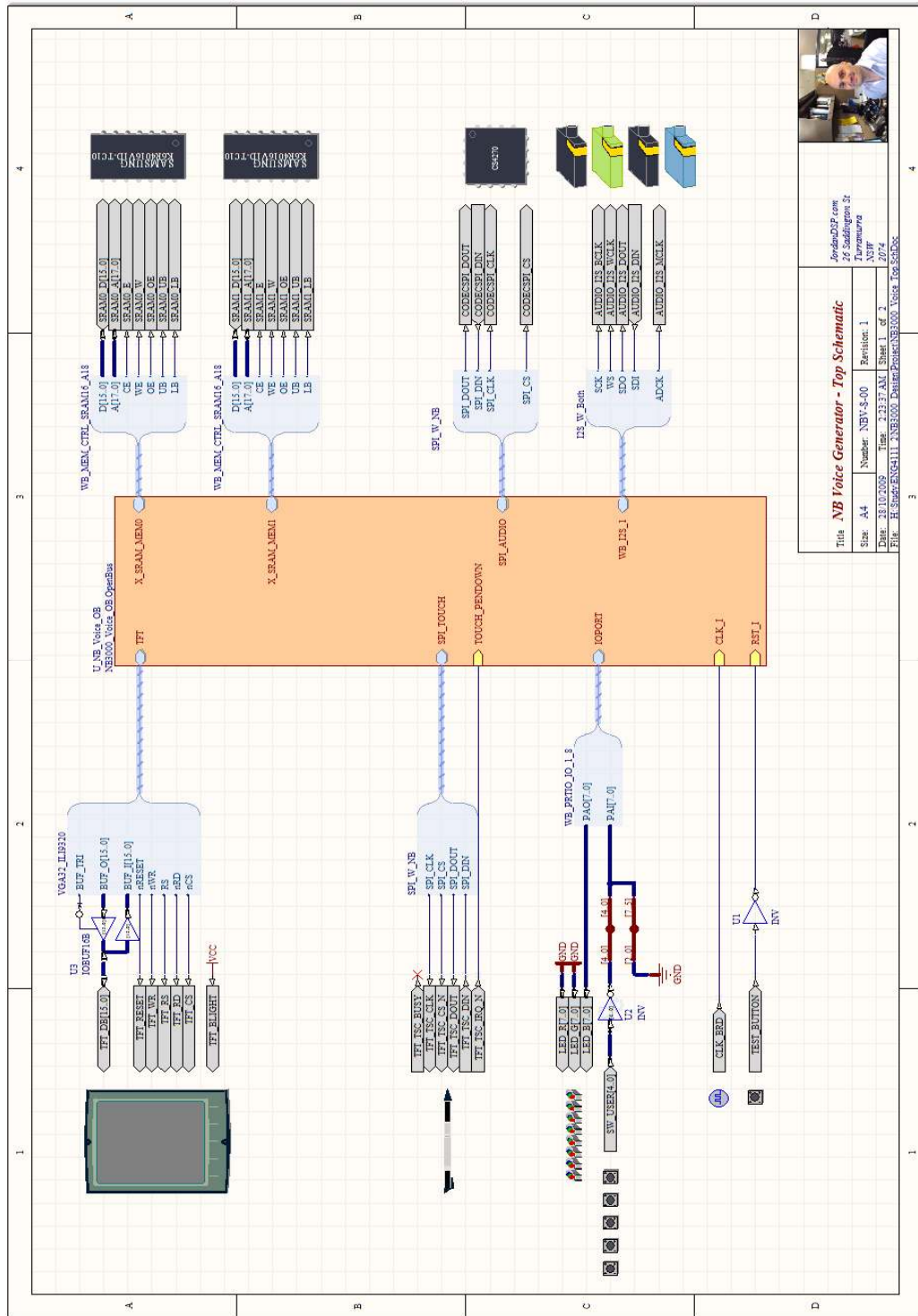


Figure 25 Top Level FPGA Project Schematic

## C.2 FPGA OpenBus System Block Diagram

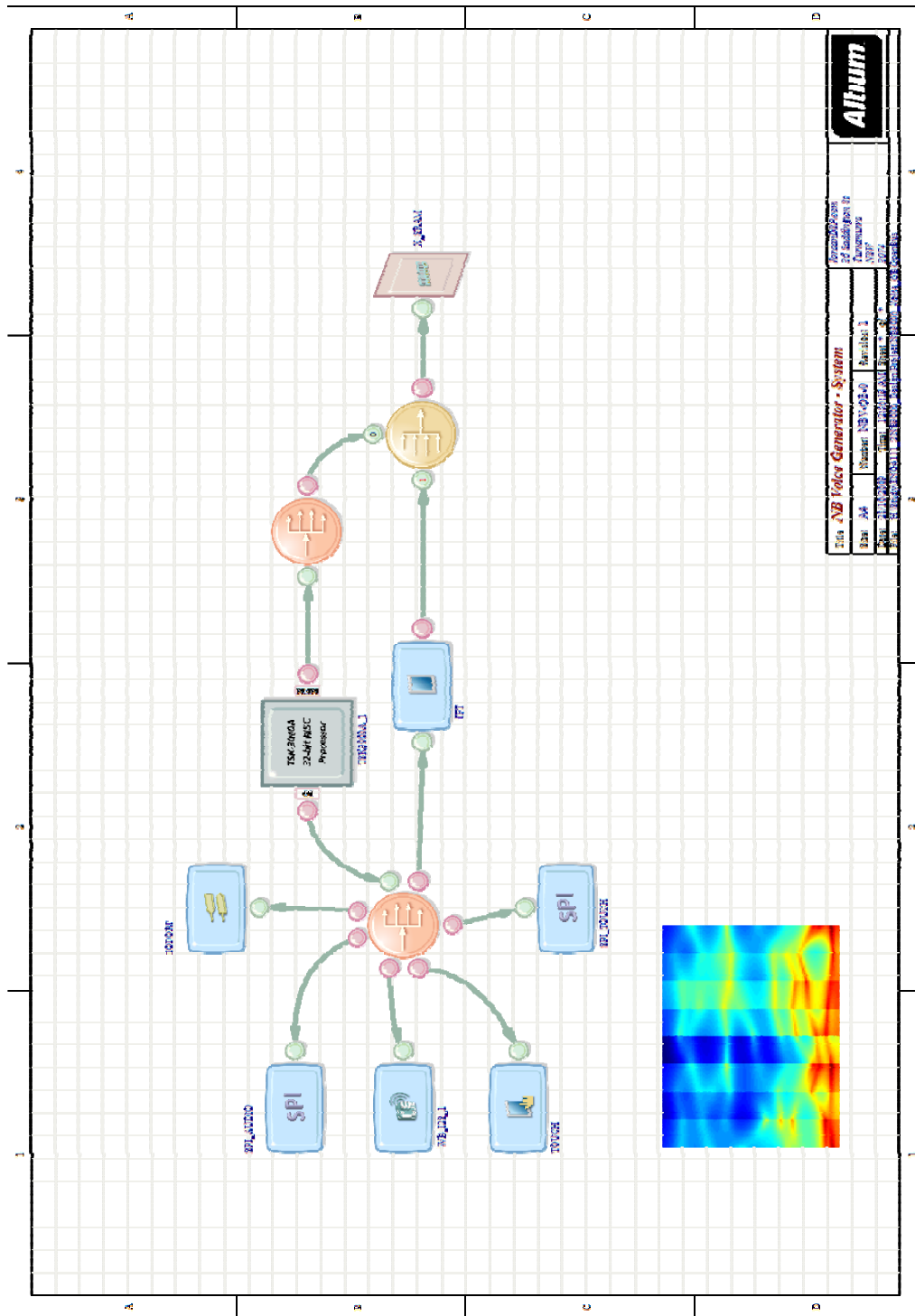


Figure 26 FPGA OpenBus Block Diagram

### C.3 Altium Designer FPGA Project Hierarchy

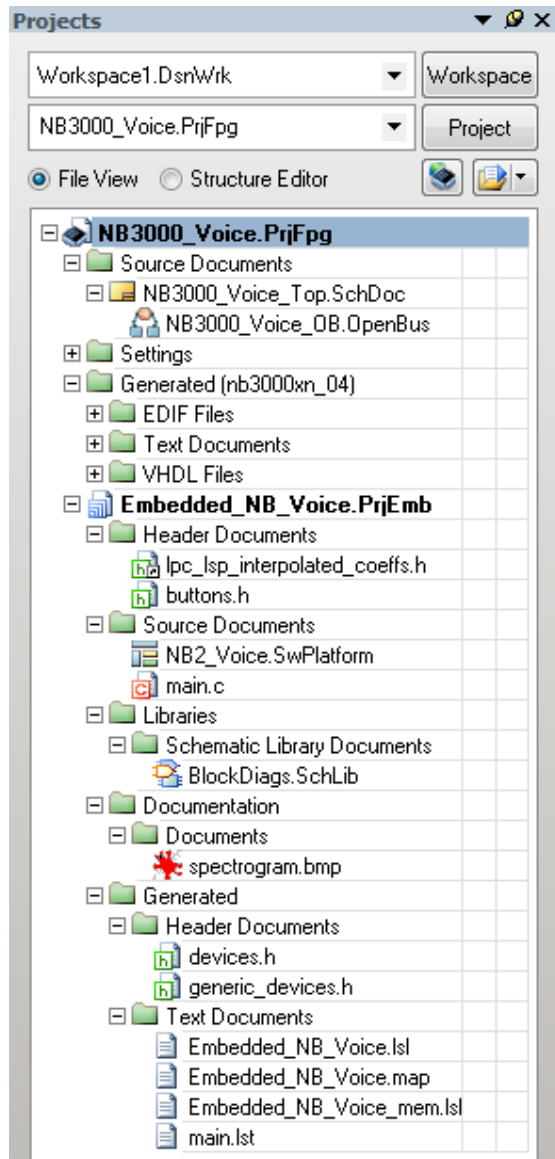


Figure 27 Altium Designer Project Hierarchy

## **Appendix D**

# **Altium Designer Embedded Project C Code Listings**

## D.1 The main.c File

```
/*
 * Copyright (C) 2009 Benjamin W. D. Jordan
 *
 * All source code in this document is copyrighted material
 * though the algorithms used may not be. Work copied with
 * permission (where necessary) or adapted will be cited.
 *
 * Copying this code and using it verbatim is not permitted
 * without prior written consent from the author (consent may
 * be electronic).
 *
 * NB3000 LPC Voice Synthesizer Main Program
 */
#include <timing.h>
#include <timers.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <touchscreen.h>
#include <pointer.h>
#include <graphics.h>
#include <canvas.h>
#include <drv_cs4270.h>
#include <drv_i2s.h>
#include <drv_ioport.h>
#include "devices.h"
#include "generic_devices.h"

#define AUDIO_BUF_LEN 160*2

// Sample rate and frequency used:
#define FS          22050
#define TS          1/FS
#define ORDER 20
```

```

// PITCH period macros
#define PITCH(h)  (int) ((1/h)*TS)
#define PITCHMIN  PITCH(50)
#define PITCHMAX  PITCH(2000)
#define PITCHBLEN 1700

// 0.999999... in int16_t Q3.13
typedef int16_t samp_t;
#define PRES      13
#define ONE       (1<<(PRES-1))
#define Q         (1<<PRES)

// Coefficient scaling - puts data into Q3.29 from float
#ifndef COEFF_SCALE
#define COEFF_SCALE  Q
#endif
#include
"..\\..\\audio\\voicedsounds\\22K05\\lpc_lsp_interpolated_coeffs.h"

#include "buttons.h"

/*
 * Function Prototypes:
 */
// Initialization Functions
void init(void);
void make_background(canvas_t * canvas);
// Timer Interrupt Call-back
void handletimer(void * params);
// Touch Screen Calibrate Call-back
static void touch_callback(int x, int y, int width, int height, void
* vp);
// Filter and Fixed Point Prototypes
samp_t allpole_kernel(samp_t x_n, const samp_t * coeffs);
samp_t allpole_fl2fix(float);
float allpole_fix2fl(samp_t);

/*
 * Software Platform Driver Handles:

```

```
 */
ioport_t * ioprt;
nb_buttons_t userbtms;
graphics_t * display;
canvas_t * canvas;
pointer_t * pointer;
touchscreen_t * touch;
pointer_state_t * pstate;
i2s_t * astream;
cs4270_t * aucodec;

/*
 * Bitmap of interpolated LPC Spectrogram:
 */
extern __no_sdata graphics_bitmap_t _lc_ub_spectrogram_bmp;
extern __no_sdata graphics_bitmap_t _lc_ue_spectrogram_bmp;
graphics_bitmap_t * bmp = & _lc_ub_spectrogram_bmp;

/*
 * Global Variables:
 */
//Pulse source and filter frame counters:
uint16_t count = 0;
uint16_t pulse = 0;
// Output frame buffer:
volatile samp_t outbuf[AUDIO_BUF_LEN];
// Pulse Source frame for Synthesis:
const samp_t pulse_buf[PITCHBLEN] =
{
    0, ONE / 64, ONE / 32, ONE / 16, ONE / 8, ONE / 4, ONE / 2, ONE,
    ONE / 2, ONE / 4, ONE / 8, ONE / 16, ONE / 32, ONE / 64, 0
};
// Noise Source buffer for Synthesis:
volatile samp_t noise[AUDIO_BUF_LEN / 2];
// Pointer to current synthesis filter coefficient set:
static samp_t * current_coeffs = & coeffs[0][0];
// Touchscreen calibration callback strings:
char * cal = "Touch screen at pointer";
char * caldone = "Calibration done.";
```



```
/*
 * Where it all begins!
 */
void main(void)
{
    char    test_str[80];
    int     pulseindex = 0;
    samp_t  audiobuf[AUDIO_BUF_LEN] = {0};
    samp_t  x_0 = 0;

    init();
    sprintf(test_str, "%d ", AUDIO_BUF_LEN);
    graphics_draw_string(canvas, 22, 22, test_str, NULL, BLUE, 0);
    graphics_set_visible_canvas(display, canvas);

    // Start getting and putting audio:
    //i2s_rx_start(austream) - not needed since we are not using
external src;
    i2s_tx_start(austream);
    while (1)
    {
        count = pulse;
        for (int i = 0; (i < count); i++)
        {
            // Set up noise, pulse and mixed noise/pulse sources
            x_0 = 0;
            if (userbtns.switches & SW1) // BT1 means noise only
            {
                x_0 = noise[count];
            }
            else
            {
                if (userbtns.switches & SW2) // BT2 mix noise and
pulse (crude)
                {
                    x_0 += noise[count];
                    x_0 += pulse_buf[pulseindex];
                }
            }
        }
    }
}
```

```

        else
        {
            x_0 = pulse_buf[pulseindex]; // default - pulse
only
        }
    }

    // Execute all-pole filter
    outbuf[i++] = allpole_kernel(x_0, current_coefs);

    // Copy sample output to other audio channel:
    outbuf[i] = outbuf[i - 1];

    //increment pulse source index, reset dependant on pitch
period
    pulseindex = pulseindex >= pulse ? 0: pulseindex + 1;
}
while (i2s_tx_avail(austream) < count);
    i2s_writel6(austream, outbuf, count);
}
}

/*
 * Function allpole_kernel
 *
 * Implementation of the LPC synthesis filter
 *
 * Arguments: x_n - current sample input from source (usually
pulses)
 *             *sample - pointer to filter history buffer - the
buffer must
 *             have length = order-1 or greater
 *             *current_coefs - pointer to coefficient array
 *             n - index of most recent output in sample[]
 *             order - order of LPC synthesis filter
 */
static samp_t sample[ORDER] = {0};
// LPC Synthesis Filter Buffer

```

```

samp_t allpole_kernel(samp_t x_n, const samp_t * coeffs)
{
    int32_t cusum;
    samp_t output;

    // Filter Kernel *could* be inlined for speed:
    // Multiply source by prediction gain (stored in a(P+1)).
    //
    //                                     P
    // y(n) = gain * x(n) + sum [-a(k)*y(P-k)]
    //                                     k = 1
    //
    cusum = x_n * coeffs[0] * coeffs[ORDER + 1];
    for (int k = 1; k <= ORDER; k++)
    {
        cusum += -coeffs[k] * sample[ORDER - k];
    }

    // Truncate and scale output:
    if (cusum > 2147483647)
    {
        cusum = 2147483647;
    }
    else
    {
        if (cusum < - 2147483648)
        {
            cusum = - 2147483648;
        }
    }
    output = (samp_t)(cusum >> 17);

    // Update Sample History Buffer
    for (int k = ORDER - 1; k > 0; k--)
    {
        sample[k] = sample[k - 1];
    }

    // Put latest output in top of buffer

```

```
    sample[0] = output;
    return output;
}

/*
 * function allpole_fl2fix
 * Converts a Single Precision Floating Point to Q1.15
 */
samp_t allpole_fl2fix(float anum)
{
    if (anum >= 1)
        anum = 1 - 2 / 32768;
    if (anum <= - 1)
        anum = - 1;
    return(samp_t)(anum * 32768);
}

/*
 * function allpole_fix2fl
 * Converts a Q1.15 fixed-point to Single Precision Floating Point
 */
float allpole_fix2fl(samp_t anum)
{
    return((float) anum) / 32768;
}

/*
 * functin init
 *
 * Main initialization routine - initializes all drivers from
SWPPlatform
 * and sets up TFT panel, audio IO, UI, and bitmap display.
 */
void init(void)
{
    char    mesg[50];
    // Little test for truncation of fixed point numbers:
    float  flt = 1;
    samp_t  fix = allpole_fl2fix(flt);
```

```

// Initialize SwPlatform Drivers
display = graphics_open(GRAPHICS_1);
canvas = graphics_get_visible_canvas(display);
touch = touchscreen_open(TOUCHSCREEN_1);
pointer = pointer_open(POINTER_1);
ioprt = ioport_open(IOPORT);

// The all important I2S and SPI drivers for the audio codec:
austream = i2s_open(DRV_I2S_1);
aucodec = cs4270_open(DRV_CS4270_1);

// Set up noise and pulse sources
for (int i = 0; i < AUDIO_BUF_LEN / 2; i++)
{
    noise[i] = (samp_t)(rand() << 4) / 64;
}

touchscreen_set_callback(touch, touch_callback, canvas);
while (!touchscreen_calibrate(touch, 320, 240));

make_background(canvas);

sprintf(msg, "NB3000 LPC Voice Synthesizer");
graphics_draw_string(canvas, 60, 10, msg, NULL, RED, 0);
sprintf(msg, " U W   O   U E   R R   A H   A   E   I I ");
graphics_draw_string(canvas, 1, 229, msg, NULL, WHITE, 0);
graphics_set_visible_canvas(display, canvas);
timer_register_handler(NULL, 40000L, handletimer);
}

/*
 * function touch_callback
 *
 * Call back routine for touchscreen calibration driver function
 * It is called during iterative passes of calibration and used
 * to provide instruction to the user.
 */

```

```
static void touch_callback(int x, int y, int width, int height, void
* vp)
{
    if (width && height)
    {
        canvas_t * canvas = (canvas_t *) vp;
        graphics_fill_canvas(canvas, BLACK);
        graphics_draw_circle(canvas, x, y, 10, MAGENTA);
        graphics_draw_line(canvas, x - 15, y, x + 15, y, CYAN);
        graphics_draw_line(canvas, x, y - 15, x, y + 15, CYAN);
        graphics_draw_string(canvas, 50, 40, cal, NULL, RED, 0);
    }
    else
    {
        graphics_draw_string(canvas, 50, 50, caldone, NULL, YELLOW,
0);
    }
    graphics_set_visible_canvas(display, canvas);
}

/*
 * function make_background
 * Sets up background image on TFT
 */
// We will use the LPC coefficient spectrogram to colour the screen
background
void make_background(canvas_t * canvas)
{
    graphics_draw_bitmap(canvas, bmp, 0, 0, 320, 240, 0);
}

/*
 * function handletimer
 *
 * Timer interrupts used to provide user foreground interaction with
TFT
 *
 * User can touch the screen and/or puch buttons.
 */
```

```
* Pitch of pulse source is determined from Y-axis, and pointer to
current set
* of LPC coefficients is updated based on X-axis of touch screen
pressure.
*
* No pressue == no sound.
*/
// Timer interrupt handles user interface:
void handletimer(void * params)
{
    static      uint16_t p_l = 0;
    static      uint16_t p_ll = 0;
    static int   mintime = 5;
    int         y;
    char        * vs;
    // get pushbuttons
    userbtns.switches = ioport_get_value(ioprt, 0);
    ioport_set_value(ioprt, 0, (uint8_t) userbtns.switches);

    // Check TFT touchscreen for pen activity
    if (pointer_update(pointer, pstate))
    {
        if (pstate->x > 280)
        {
            vs = "iii";
        }
        else
        {
            if (pstate->x > 240)
            {
                vs = "e";
            }
            else
            {
                if (pstate->x > 200)
                {
                    vs = "a";
                }
                else
            }
        }
    }
}
```

```
    {
        if (pstate->x > 160)
        {
            vs = "ah";
        }
        else
        {
            if (pstate->x > 120)
            {
                vs = "r";
            }
            else
            {
                if (pstate->x > 80)
                {
                    vs = "ue";
                }
                else
                {
                    if (pstate->x > 40)
                    {
                        vs = "o";
                    }
                    else
                    {
                        vs = "uw";
                    }
                }
            }
        }
    }

    current_coef = & coeffs[pstate->x][0];
    pulse = p_l / 2 + p_ll / 2;
    p_ll = p_l;
    y = pstate->y;
    p_l = (y/PITCHMAX)+PITHCMIN;
    // Set the pitch based on Y location
```



```
    mintime = 5;
    graphics_fill_rect(canvas, 140, 100, 40, 40, BLACK);
    graphics_draw_string(canvas, 150, 110, vs, NULL, WHITE, 0);
    graphics_set_visible_canvas(display, canvas);
}
else
{
    if (--mintime == 0)
        pulse = 0;
}
}
```

## D.2 The `buttons.h` Header File

```
/*
 * Author: Benjamin Jordan
 *         bit field struct for holding pushbutton values.
 */
#ifndef __BUTTONS_H
#define __BUTTONS_H

typedef struct user_btns
{
    uint8_t switches :5;
    uint8_t :3;
} nb_buttons_t;

#define SW1 0x01
#define SW2 0x02
#define SW3 0x04
#define SW4 0x08
#define SW5 0x10

#endif
```

### D.3 The `lpc_coeffs.h` Header File

This header file sample was generated from an earlier script that did not perform interpolation of LPC vectors. This header file was used to test the all-pole filter kernel in the NB3000 FPGA design.

```

/*
 * LPC Coefficients for sample a
 * Date/Time created: 2009-10-10 15:56:46
 * LPC Order: 20
 * Sampling rate (Fs): 22050
 * Minimum LPC coefficient: -1.42714
 * Maximum LPC coefficient: 1.09322
 * NOTE: LAST element of array is prediction gain, FIRST is first
coefficient.
 */

#ifndef COEFF_LEN
#define COEFF_LEN      21
#endif

#ifndef Q
#error Q must be defined! It is the floating-point to fixed point
scaling factor.
#endif
const samp_t coeff_a[COEFF_LEN+1] =
{
    (samp_t) (1.00000000000000000000    * Q),
    (samp_t) (-1.42714376385603336495  * Q),
    (samp_t) (0.59700166372542673443   * Q),
    (samp_t) (-0.53103815249363162110  * Q),
    (samp_t) (1.09321662332023183950   * Q),
    (samp_t) (-0.82875254751330640346  * Q),
    (samp_t) (0.22686828175005249730   * Q),
    (samp_t) (-0.64716245541051964363  * Q),
    (samp_t) (1.08916034142518536321   * Q),
    (samp_t) (-0.58000054188920013853  * Q),
    (samp_t) (0.12958309588605002038   * Q),

```

```
(samp_t) (-0.45851913840098296182 * Q),
(samp_t) (0.53153710818438681951 * Q),
(samp_t) (-0.06503120586575833473 * Q),
(samp_t) (0.17394366018636872595 * Q),
(samp_t) (-0.48091195912796863565 * Q),
(samp_t) (0.22410698122344246963 * Q),
(samp_t) (0.17649710289145784103 * Q),
(samp_t) (0.03990763735604364176 * Q),
(samp_t) (-0.07040305987016133582 * Q),
(samp_t) (-0.12177756292082211886 * Q),
(samp_t) (0.63606232558458797310 * Q) /* <--- Prediction
Gain */
};
```

## D.4 The `lpc_lsp_interpolated_coeffs.h` Header File

This is the file which is auto-generated by the Octave function `gen_all_lsp.m`. An identically formatted header file is generated from `gen_all_lpc.m` except that the coefficients are not interpolated using the LSP method, and therefore markedly different.

```

/*
 * LSP Interpolated LPC Coefficients for vowels
 * Date/Time created: 2009-10-25 17:37:10
 * LPC Order: 20
 * Sampling rate (Fs): 22.05 KHz
 * Minimum LPC coefficient: -2.1922
 * Maximum LPC coefficient: 1.79107
 * NOTE: LAST element of each array is prediction gain, FIRST is 1
coefficient.
 */

#ifndef COEFF_LEN
#define COEFF_LEN      22
#endif

#ifndef Q
#error Q must be defined! It is the floating-point to fixed point
scaling factor.
#endif

const samp_t coeffs[320][COEFF_LEN] =
{
    {
        (samp_t) (1.00000000000000000000    * Q),
        (samp_t) (-1.79698140824335061971 * Q),
        (samp_t) (0.88292388815247579981  * Q),
        (samp_t) (-0.24923128966218513480 * Q),
        (samp_t) (0.46411280796424592143  * Q),
        (samp_t) (-0.60305326959603999804 * Q),
        (samp_t) (0.42545606596659896192  * Q),
        (samp_t) (-0.51230879515425509219 * Q),
        (samp_t) (0.5591182121213345724857 * Q),
    }
}

```

```

        (samp_t) (-0.16785095595795890278      * Q),
        (samp_t) (0.37361527593141219405      * Q),
        (samp_t) (-0.65959996929074682370     * Q),
        (samp_t) (0.65201411268465325755     * Q),
        (samp_t) (-0.46573461349699701861     * Q),
        (samp_t) (0.13835783824904124284     * Q),
        (samp_t) (-0.05474765866315128848     * Q),
        (samp_t) (0.08140394595628824836     * Q),
        (samp_t) (-0.08749443112262278444     * Q),
        (samp_t) (0.00670160817482776117     * Q),
        (samp_t) (-0.07936529079207221837     * Q),
        (samp_t) (0.11610059068015576855     * Q),
        (samp_t) (0.18333513134382359300     * Q) /* <---
Prediction Gain */
    },
    {
        (samp_t) (1.00000000000000000000     * Q),
        (samp_t) (-1.81777011447755798557    * Q),
        (samp_t) (0.91918780876025718563     * Q),
        (samp_t) (-0.23869234917961579256    * Q),

        ...

        (samp_t) (0.13428512777976292503     * Q),
        (samp_t) (-0.14521586669568220529     * Q),
        (samp_t) (-0.30624555284566301605     * Q),
        (samp_t) (0.08774673208746089359     * Q),
        (samp_t) (0.18094884844709002714     * Q),
        (samp_t) (-0.06253544600931593145     * Q),
        (samp_t) (0.05478937079840458246     * Q) /* <---
Prediction Gain */
    }
};

/* ----- end of coefficients ----- */

```

**Truncated Here: File is 7703 Lines Long!**

## D.5 The `devices.h` Auto-Generated Header File

```
// Embedded Framework Generated File:
// Date:26/10/2009
// Time:10:31:57 AM
//

#ifndef _DEVICES_H
#define _DEVICES_H

// instance devices ids macro definitions
#define DRV_AD7843_1      0
#define DRV_CS4270_1     0
#define DRV_I2S_1        0
#define DRV_IOPORT_1     0
#define DRV_SPI_2        0
#define DRV_SPI_1        1
#define DRV_VGA_ILI9320_1 0
#define WB_I2S_1         0
#define IOPORT           0
#define SPI_TOUCH        0
#define SPI_AUDIO        1
#define TOUCH            0
#define TFT              0
#define AD_VGA_ILI9320_1 0
#define GRAPHICS_1       0
#define AD_TOUCHSCREEN_TO_POINTER_1 0
#define POINTER_1        0
#define TOUCHSCREEN_1    0

#endif
```

## **Appendix E**

### **Nanoboard 3000 Data Sheet**



## E.1 NB 3000 Data Sheet

# Altium NanoBoard 3000 Series



### Architectural highlights

- Reprogrammable hardware development platform that harnesses the power of a dedicated high-capacity, low-cost programmable device to allow rapid and interactive implementation and debugging of your designs
- Perfect entry-point to discover and explore the world of FPGA-based embedded systems design. Programmable hardware realm allows you to update the design quickly and many times over without incurring cost or time penalties
- Works seamlessly and in full synchronization with Altium's next-generation electronic design solution, Altium Designer
- High-capacity FPGA located on the motherboard, and provision for a single plug-in peripheral board (Altium or user's own) for additional system flexibility
- Automatic peripheral board detection and configuration
- Dual boot system, allowing the board to update its firmware in the field by itself, over a standard USB connection – no parallel port or USB JTAG Adapter required

### Main board specifications

- Choice of high-capacity FPGAs
  - NanoBoard 3000XN – with fixed Xilinx® Spartan™-3AN device (XC3S1400AN-4FGG676C)
  - NanoBoard 3000AL – with fixed Altera® Cyclone™ III device (EP3C40F780C8N)
  - NanoBoard 3000LC – with fixed LatticeECP2™ device (LFE2-355E-5FN672C)
- Integrated color TFT LCD panel (240x320) with touch screen that facilitates dynamic application interaction
- High-quality stereo audio capabilities including: Line in/out/headphones, audio CODEC with I2S-compatible interface, analog mixer, audio power amplifier and high-quality speakers (located on a separate speaker board attachment)
- USB hub, providing connection of up to three USB 2.0 devices, with interfacing handled by an ISP1760 Hi-Speed USB Host Controller
- SVGA interface (24-bit, 80MHz)
- Variety of standard communications interfaces: RS-232, RS-485, PS/2, 10/100 Fast Ethernet, USB 2.0, S/PDIF, MIDI
- Dual SD card readers – for use by user FPGA and Host Controller respectively
- IR receiver – supports data transmitted using a 38kHz carrier frequency
- Programmable clock (6 to 200MHz) and fixed clock (20MHz) – both available to user FPGA
- 4-channel 8-bit ADC, SPI-compatible – providing maximum sample rate of 200ksps
- 4-channel 8-bit DAC, SPI-compatible – operating at clock rates of up to 40MHz
- 4x isolated IM Relay channels – each channel providing a 5V non-latching DPDT relay with one coil
- 4x PWM power drivers
- 8-way general purpose DIP-Switch, 8 RGB LEDs, 5 PDA-style push button switches and a Test/Reset button – all wired directly to the user FPGA

- User prototyping area
- Dual 18-way (20 pin) I/O expansion headers, with power supply selection links
- On-board memories accessible by user FPGA – 256KB x 32-bit common-bus SRAM (1MB), 16M x 32-bit common-bus SDRAM (64MB), 8M x 16-bit common-bus 3.0V Page Mode Flash memory (16MB), dual 256KB x 16-bit independent SRAM (512KB each)
- Four 8Mbit SPI flash memory devices – one containing Primary boot image for Host Controller, one containing golden boot image for Host Controller, two for use by user FPGA (for boot/embedded purposes)
- SPI Real-Time Clock with 3V battery backup
- Accommodates a single plug-in peripheral board for additional system flexibility
- Board ID memory – 1-Wire® ID system uniquely identifies the motherboard and any attached Altium peripheral board
- Host (NanoTalk) Controller hosts the NanoBoard firmware. Responsibilities include managing JTAG communications (with Altium Designer/User FPGA/connected peripheral board), as well as access to common-bus SPI resources
- 5V DC power connector with power switch, plus testpoints for all major supplies on the board (and GND)
- High-speed PC interconnection through USB 2.0 allows for fast downloading and debugging

### Included in the box

#### Altium Designer

The NanoBoard 3000 includes a 12-month subscription to an Altium Designer Soft Design license which is linked to the NanoBoard in the box. This license option provides functionality to quickly start designing FPGA-based embedded systems, including:

- FPGA design entry in C, OpenBus, Schematic, VHDL and Verilog
- VHDL simulation engine, integrated debugger and waveform viewer
- Support for a range of 32-bit soft processors for use in FPGA design
- A rich set of royalty-free IP core libraries including peripherals and user-configurable custom logic
- Full software development tool chain with libraries and source code
- Programmable FPGA-based instruments for hardware debug and deployment
- Support for importing third-party FPGA IP cores, developing and reusing IP libraries

Additional Altium Designer license options are available for custom board design. For information on Altium Designer licensing options, visit [www.altium.com/altiumdesigner](http://www.altium.com/altiumdesigner)

#### Training and resource materials

Altium provides extensive online resources designed to get you up and running as quickly as possible.

- Everything you need to know to get started and build your proficiency with Altium Designer – [www.altium.com/gettingstarted](http://www.altium.com/gettingstarted)
- Full technical information on the NanoBoard 3000 – [www.altium.com/wiki/nanoboard3000](http://www.altium.com/wiki/nanoboard3000)

For information on all available NanoBoard configurations, visit the Altium website at [www.altium.com/nanoboard](http://www.altium.com/nanoboard)

**Altium.**