Digital Implementation of a Frequency-lowering Channel Vocoder

by

Jeffrey J. Foley

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1996

© Jeffrey J. Foley, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUN 11 1996

LIBRARIES

Author ....
Department of Electrical Engineering and Computer Science
February 7, 1996

Certified by ....
Louis D. Braida
Henry E. Warren Professor of Electrical Engineering
Thesis Supervisor

Eng.

Accepted by ....
F.R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Digital Implementation of a Frequency-lowering Channel Vocoder

by

Jeffrey J. Foley

Submitted to the Department of Electrical Engineering and Computer Science
on February 7, 1996, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

A frequency-lowering channel vocoder reintroduces the high-frequency components of a speech signal, often inaudible to a hearing-impaired listener, into lower frequencies by the addition of select narrow-band noise signals which act as artificial speech cues. Such vocoder systems have met with some success in previous, analog implementations. This study presents a digital implementation of the channel vocoder system, allowing for greater flexibility in the selection of filter banks, the determination of noise levels, and the decision to suppress signal processing when it is shown to be unnecessary. The digital implementation is based on the Motorola DSP96002 processor and Ariel Corporation's DSP-96 Digital Signal Processing Board, using Motorola's DSP96KCC Optimizing C Compiler to facilitate the programming of the DSP board. Adjustable specifications for the system are derived from an interface between the DSP board and a controlling personal computer. The controlling personal computer also allows for both monitoring and dynamic adjustment of run-time system parameters.

Thesis Supervisor: Louis D. Braida
Title: Henry E. Warren Professor of Electrical Engineering

Acknowledgments

First and foremost, I would like to thank my mom and dad, not only for the obviou$ help they've $acrificed over the year$, but also for their incredible patience, understanding, and ability to pull me up when I was slipping into the doldrums. I'm sure they'll agree, though, that this has been quite a step up from the science project on the paper boy BASIC program in fifth grade...

Many thanks to my advisor, Prof. Lou Braida, who exhibited enormous patience and understanding when I "burned out" over the summer. When I came to him for advice or assistance, he always came through, even when I was running way behind schedule.

Dave Lum was invaluable in providing day-to-day assistance with the little quirks and problems I encountered while working in the lab. He never failed to help me out, whether by providing detailed technical advice or just by being in his perpetual good mood.

I can't forget to thank all the people who nagged and kidded and annoyed me about not having graduated yet. Amazing what kind of incentive that provides, just trying to get them to *STOP IT!*

And of course, I truly owe it all to the One Upstairs. All things are possible with the help of the Lord.

Contents

1 Introduction **7**

2 Background **9**

2.1 Linear amplification 10

2.2 Frequency lowering 10

3 System Design **12**

3.1 Previous vocoder systems 12

3.2 Present system 15

3.2.1 Sampling rate 16

3.2.2 Filters 16

3.2.3 Noise 18

3.2.4 PC interface 18

4 Code **23**

4.1 Personal computer code 23

4.1.1 Downloading the DSP board code 23

4.1.2 DSP board interface 26

4.1.3 Monitoring and altering the system 28

4.2 DSP board code 29

4.2.1 Initial setup 29

4.2.2 Sampling I/O 30

4.2.3 Filtering 30

4.2.4 PC interface 32
4.2.5 Noise 33

5 Conclusions **35**

5.1 Functionality of current system 35
5.2 Suggestions for further work 37
5.2.1 Evaluation 37
5.2.2 Improvement 37

A Source Code **39**

A.1 Code for controlling personal computer 39
A.2 Code for Ariel DSP-96 board 47

List of Figures

3-1 Block diagram of vocoder system. 13

3-2 Lippmann's vocoder system. 13

3-3 Posen's modified vocoder system. 14

3-4 Digital implementation of vocoder system. 15

3-5 Design of filter using Matlab's remez function. 18

3-6 Example analysis filterbank designed by Matlab's "remez" function. . 19

3-7 Example synthesis filterbank designed by Matlab's "remez" function. 20

3-8 High-pass (a) and low-pass (b) filters for signal processing cutoff determination. Low pass filter used to simulate hearing loss (c) 21

4-1 Overview of the digital vocoder system. 24

4-2 A sample configuration file. 24

4-3 Interrupt usage for PC to DSP board interface. 27

4-4 DSP board sampling procedure. 31

5-1 RMS levels reported by the PC for constant-tone inputs. 36

Chapter 1

Introduction

Many persons suffering from sensorineural hearing impairments, though left with little or no hearing in higher frequencies, still retain limited hearing in lower frequencies. As the fields of psychoacoustics and signal processing have progressed, various techniques aiming to improve the hearing of such patients have been invented and reinvented. Most often these techniques attempt to improve the speech reception of individuals with high-frequency hearing loss by incorporating the high-frequency components of a speech signal into the individual's hearing range. Such attempts have met with only limited success.

This project is a digital implementation of one of the more promising frequency-lowering systems, a channel vocoder. Previous implementations of vocoder systems, limited by available technology, were based on analog circuits and thus lacked flexibility. Nevertheless, studies of these earlier vocoder implementations have shown encouraging results. The newer system, implemented through an internal digital signal processing board on a personal computer, obviates the problem of inflexibility. With the high speed capabilities of the programmable Motorola DSP96002 processor, modifying system parameters becomes as easy as editing a configuration file and reloading the processor's program. In this manner, a user can customize the system with personal filterbank design and noise level specifications while receiving immediate feedback on additional changes.

This thesis presents the digital implementation of the analog frequency-lowering

channel vocoder system studied by Posen [6], detailing the design and programming issues and improvements from the analog system which he implemented. Chapter 2 offers some background information on other systems which attempt to improve hearing reception, comparing their advantages and drawbacks to those of the channel vocoder system. Chapter 3 focuses on the operation of a vocoder system and the design of both earlier systems as well as design issues and changes for the current system. Chapter 4 details the current system, describing the controlling code for both the personal computer and the DSP board. Chapter 5 summarizes the work and findings of the present study with suggestions for further improvements. An appendix includes a listing of the code for the DSP board and the controlling personal computer.

Chapter 2

Background

The key difficulty for persons with sensorineural hearing impairments lies in the fact that certain speech elements which make possible the mental transformation from "acoustic signal" to "intelligible speech" are imperceptible. For example, the cues may fall below the listener's threshold of hearing, or they may be masked by internally generated sounds such as tinnitus. For whatever the reason, the failure to detect these speech cues is what impairs the decoding of acoustic speech signals.

Acoustic systems designed to improve speech reception focus on recovering these inaudible speech cues for the listener suffering from high-frequency hearing loss by amplifying, frequency shifting, remapping, or otherwise reintroducing the cues back into the acoustic speech signal. One difficulty inherent in this task, however, is to incorporate the lost speech cues while still preserving other lower-frequency cues already present in the speech signal. Often these cues are distorted, masked, or otherwise obliterated by the process of amplifying and transposing the higher-frequency speech cues. [1]

This chapter's overview of these acoustic systems outlines not only their advantages and disadvantages, but also how the channel vocoder system described in this project overcomes some of the other systems' problems.

2.1 Linear amplification

Effective and elegant in its simplicity, linear amplification of an incoming speech signal is a common approach to combatting hearing loss. It recovers the previously undetectable speech cues by increasing their level to one that is above the listener's threshold of hearing. Most mild and moderate hearing losses can be corrected by means of linear amplification. Often such an amplification system is improved by tailoring its frequency-gain characteristic to match the audiogram of a patient or by introducing limiting factors, such as an automatic volume control, to avoid painful peak sound levels or "holes" in the speech signal.

Because of the general success of linear amplification in treating the hearing-impaired, studies of other systems often compare their efficacy to that of amplification. Despite its proven worth, however, linear amplification has its own difficulties. Since the elevation of a subject's absolute threshold is not accompanied by an equivalent rise in the discomfort threshold of the subject, the auditory area available between the hearing and pain thresholds diminishes, leaving less available space for the amplified signal to fall. Additionally, sensorineural impairments are often accompanied by resolution loss or sound distortion, problems which amplification will only aggravate. Finally, a listener with no hearing in a given frequency range will never be able to benefit from amplification techniques. [1]

2.2 Frequency lowering

If a listener has no hearing in a given frequency range, frequency lowering can be used to move spectral components to regions of better hearing. In this manner a listener with high-frequency hearing loss may still benefit from high-frequency speech cues sufficiently lowered in frequency to perceptible frequencies.

Many frequency-lowering systems, including slow playback, frequency shifting, transposition, and spectral warping, have been studied; an analysis of such systems can be found in [1], [3], and [8]. In general, these systems performed poorly compared

to linear amplification. The shifting, imposition, or compression of spectral components to recover imperceptible high-frequency characteristics distorted temporal and rhythmic patterns, fundamental frequency contours, and phoneme durations.

A promising frequency-lowering technique based on channel vocoding has recently been studied. Channel vocoders create a controlled, artificial version of high-frequency speech elements in the lower frequency range. This idea takes advantage of the fact that the majority of lost higher-frequency speech cues are bands of noise associated with consonants. An initial short burst of noise, for example, distinguishes a stop consonant, while sustained periods of higher-frequency noise are indicative of fricatives or affricates. The vocoder system takes the input signal, analyzes it for the presence of high-frequency energy, and then adds in a corresponding lower-frequency signal approximating the buzz or hiss of the imperceptible high-frequency cues. This process gives the listener a sense of the lost information without masking the low-frequency components of the speech, as most frequency lowering techniques do, and without risking pain or distortion, as amplification does.

Both Lippmann [4] and Posen [6] studied vocoder-based frequency-lowering systems, each concluding that the vocoder processing improved the recognition of stop, fricative, and affricate consonants. Posen, whose study noted that the processing also degraded the perception of semivowels and nasals, modified Lippmann's system to reduce these degradations. His results indicated that the vocoder system was successful in improving intelligibility, given sufficient subject training to become accustomed to the perceptively different processed speech. The positive results of Posen's system suggest that a vocoder-based system holds promise for improving the speech reception of hearing-impaired subjects.

Chapter 3

System Design

Channel vocoders operate by analyzing the speech signal with a bank of band-pass filters whose outputs control the addition of low-frequency noise signals to the signal (Figure 3-1). In this manner lost high-frequency information regarding stop, affricate, and fricative consonants is recovered without deeply affecting the rest of the speech signal. This chapter briefly examines the designs of Lippmann's and Posen's vocoder system before presenting an in-depth look at design issues and specifications for the current project.

3.1 Previous vocoder systems

This project's channel vocoder was based on the analog systems previously constructed by Lippmann [4] and Posen [6].

In Lippmann's system (Figure 3-2), speech sounds passed through a bank of eight bandpass filters with center frequencies ranging from 1000–8000 Hz. Lippmann grouped the eight filters into two 2/3-octave-wide low-frequency analysis bands, and two octave-wide high-frequency analysis bands. The output of these analysis bands controlled the levels of corresponding low-frequency noise in the 400–800 Hz range. Lippmann evaluated the vocoder system using CVC nonsense syllables formed from 16 consonants and 6 vowels.

Posen made some adjustments to Lippmann's system in his preliminary study, the

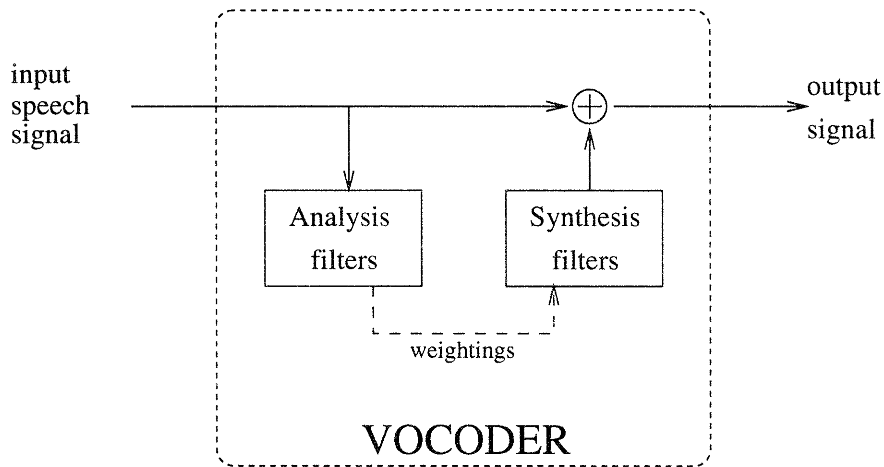

Figure 3-1: Block diagram of vocoder system.

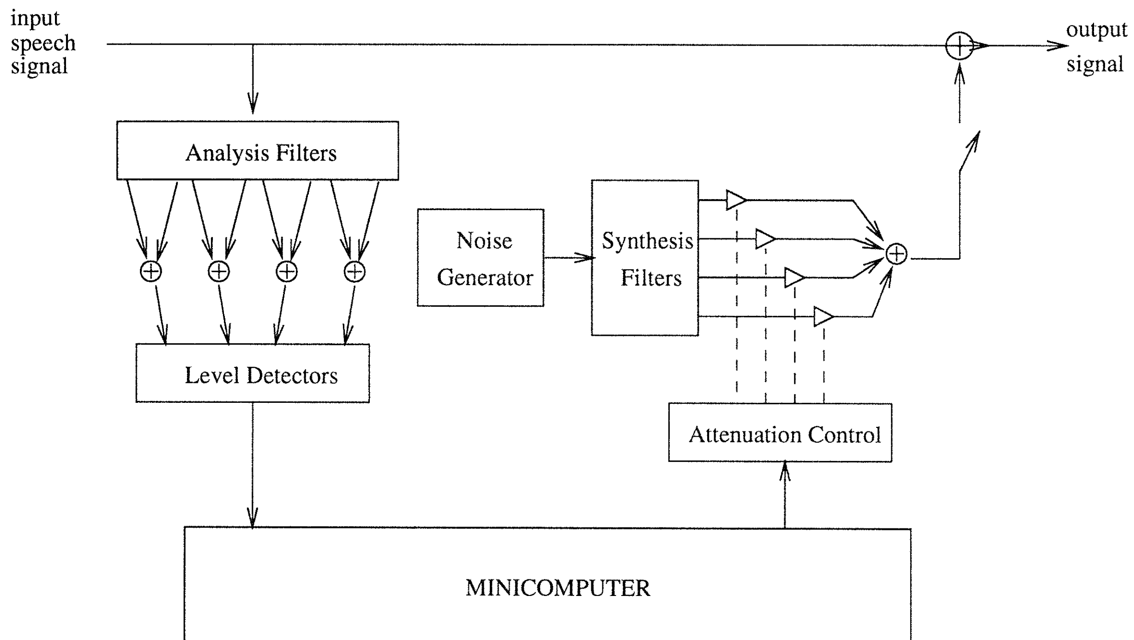

Figure 3-2: Lippmann's vocoder system.

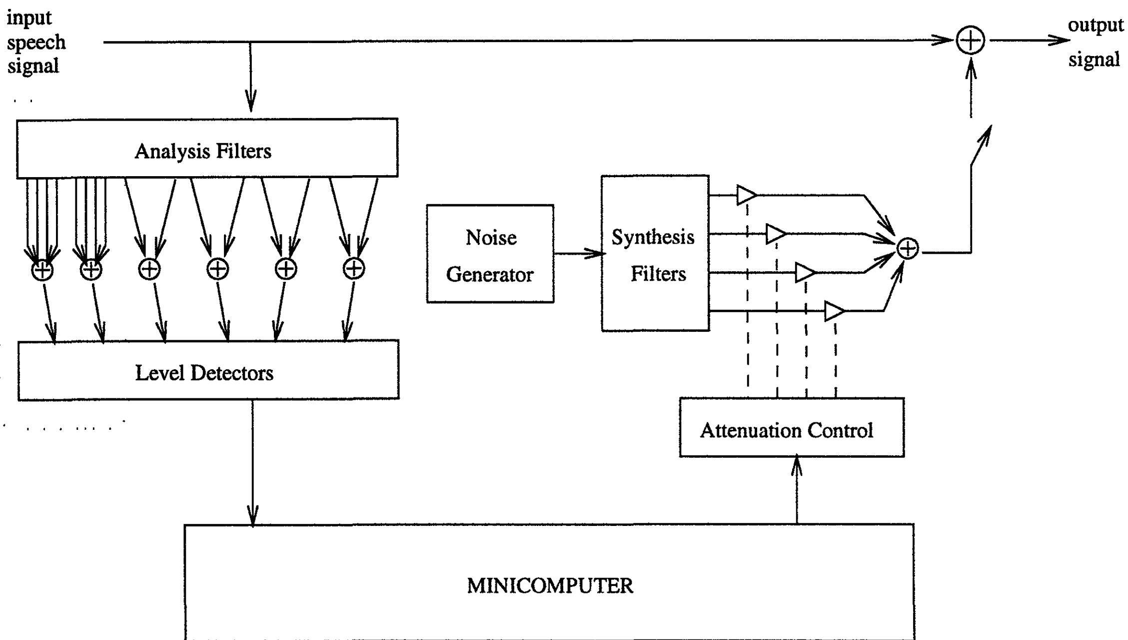

Figure 3-3: Posen's modified vocoder system.

most significant adjustment being a narrowing of the analysis range. Eight contiguous 1/3-octave filters were combined to form four 2/3-octave wide analysis bands with center frequencies spanning a range of 1000 to 5000 Hz. RMS level detectors were used to measure the output levels of the analysis bands and thus control the output levels of the low-frequency noise signals. The noise-band signals were generated by passing wide-band noise through a bank of 1/3-octave synthesis filters with center frequencies ranging from 400 to 800 Hz. A 1 dB increase in a given analysis band signal caused a 1 dB increase in the corresponding noise-band signal. The noise-band level was also partially controlled by the level of the low-frequency sounds present in the signal so that low-frequency cues would not be masked by the noise. Posen evaluated his vocoder system using CV nonsense syllables formed from 24 different consonants and 3 vowels.

After evaluating this system, Posen noticed that the intelligibility of nasals and semi-vowels, which were not evaluated under Lippmann's study, suffered under the vocoder signal processing. These consonants, which by their nature exhibit both noisy and periodic qualities, are characterized by low-frequency and mid-range energy.

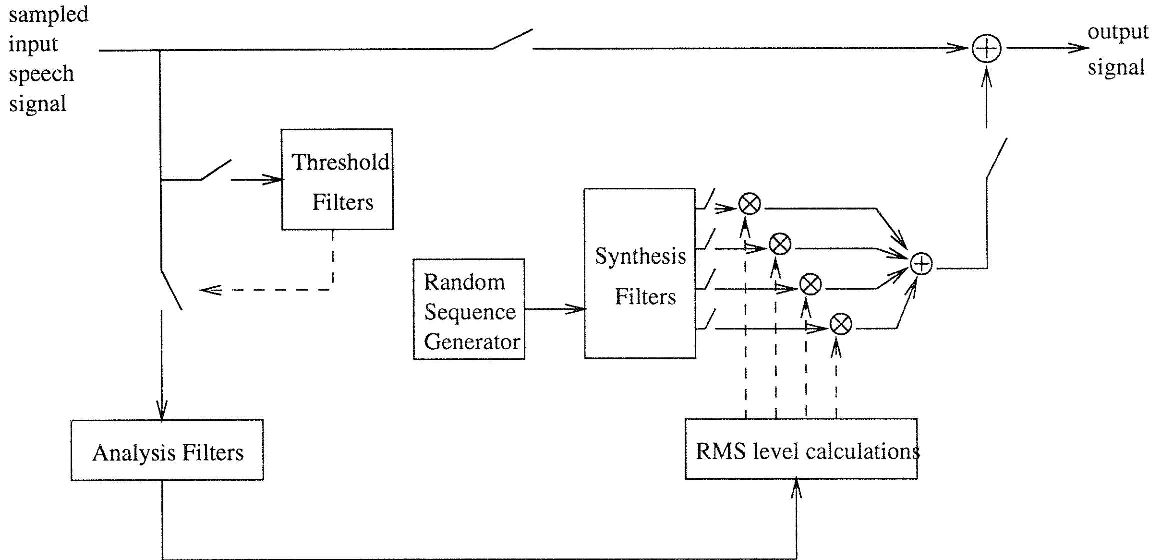

Figure 3-4: Digital implementation of vocoder system.

Posen corrected this problem by adding an additional qualifier to further improve intelligibility: a means of suppressing the addition of low-frequency noise signals if the input signal had significantly more low-frequency energy than high-frequency energy (Figure 3-3). He first determined a corner frequency separating "high" from "low" frequency, roughly about 1400 Hz, based on spectral plots of consonant-vowel pairs from male and female speakers. The system would then, with two more filters, determine the ratio of high-frequency energy E_H to low-frequency energy E_L based on resultant RMS values. This ratio was then compared to a threshold T to determine whether to suppress the signal processing. If $E_H/E_L < T$, then the processing would be suppressed.

Subjects listening to sets of consonant-vowel syllables processed by the vocoder independently agreed upon the threshold value which would suppress processing for as many nasals and semivowels as possible.

3.2 Present system

The present vocoder system, though modeled after Posen's analog system, has been modified to better fit the needs and abilities of the digital equipment available (Fig-

ure 3-4). The digital implementation of the system is based on the Motorola DSP96002 processor and Ariel Corporation's DSP-96 Digital Signal Processing Board, using Motorola's DSP96KCC Optimizing C Compiler to code the processor's program. Some aspects of Posen's system were modified in order to accomodate an important constraint on the DSP-96 board: speed. The DSP-96 must be able to sample the speech signal, perform the necessary signal processing, and produce an output signal in real-time. Other modifications have been made to take advantage of the flexibility of a digital implementation. Though the frame of Posen's system remains, every attempt has been made to generalize the vocoder specifications such that an experimenter can continuously adjust system parameters without modifying, recompiling, and reloading the processor's source code.

3.2.1 Sampling rate

The DSP-96 board can only sample the input signal at one of several specific rates. For the sake of speed, slower sampling rates are beneficial; the fewer data points the board has to manipulate each second, the fewer calculations the system will need to perform in order to maintain a continuous output. However, since the system is to detect speech features as high as 5 kHz, the Nyquist sampling theorem states that the system's sampling rate must be at least 10 kHz. The DSP board samples at the lowest possible rate which is equal to or greater than the Nyquist rate–in this case, 11.025 kHz.

3.2.2 Filters

The filters comprising the vocoder system include a group of analysis filters, a group of synthesis filters, and two threshold filters. The analysis-synthesis filter pairs determine the amplitude of the low-frequency noise added to the speech signal. The threshold filters compute the high- and low-energy in the speech signal in order to implement Posen's threshold ratio.

Lippmann's and Posen's vocoder systems both used eight analysis filters. Rather

than combining the results of each pair of digital filters to control a given noise band, thus requiring extra filtering calculations, the digital vocoder is designed for four filters, each representing one analysis band and controlling one noise band.

For filtering calculations, discrete Fourier transforms (DFTs) effectively represent the finite-duration filter sequences used by the vocoder. A fast Fourier transform (FFT) algorithm provided in the Ariel C library calculates these DFTs. Since it is impossible to represent the indefinite-length input sequence with a single DFT, the system segments the input sequence into smaller blocks of finite sequences and uses the overlap-add block convolution method to perform the filtering.

The overlap-add method and the FFT algorithm impose constraints on the selection of the N-point FFT, input segment length L, and filter length P of the system. The FFT algorithm requires that N be a power of 2. By the overlap-add method, $N \geq L + P - 1$. Though longer filter sequences allow for better filter design, they require more computation time and memory on the DSP board.

Initially, the system used 512-point FFTs with filters 128 points long. After implementing only a few analysis filters, however, it became apparent that the Motorola processor could not keep up with the filtering calculations. Constant-tone input signals analyzed by just two filters before being passed through the system would come through with discontinuities, as the system simply could not fill the output buffer quickly enough. The final system used 256-point FFTs with filters 64 points in length; for these values the tones passed without distortion.

A filterbank based on Posen's system was designed to help test the digital vocoder's performance. Four contiguous 2/3-octave filters with center frequencies from 1000–5000 Hz formed the analysis filterbank (Figure 3-6), and four contiguous 1/3-octave filters with center frequencies ranging from 400–800 Hz comprised the synthesis filterbank (Figure 3-7). A high-pass filter and a low-pass filter, each with a cutoff frequency of 1400 Hz, served as the threshold filters, and a low-pass filter with a cutoff of 1000 Hz was designed to simulate high-frequency hearing loss (Figure 3-8).

Filter sequences for the vocoder system can be designed by any process so long as the end result is an ASCII file of the proper length and format. The filters should take

```
>> A1_prep = [0 794.3 891.3 1413 1585 5512.5];
>> M = [0 0 1 1 0 0];
>> N=128;
>> samp_freq = 11025;
>> A1 = A1_prep * 2 / samp_freq;
>> aa1 = remez(N-1, A1, M);
>> AA1 = fft(aa1, N);
```

Figure 3-5: Design of filter using Matlab's remez function.

into account the sampling rate of 11025 Hz, and the FFT of each filter should have a bandpass magnitude of approximately 1.0 to avoid any scaling errors. Several filter design techniques are discussed in [5] and [9]. Test filterbanks for the digital system were designed with the help of MATLAB's `remez` function, which approximates a desired frequency response with an optimal equiripple FIR filter using the Parks-McClellan algorithm (Figure 3-5).

3.2.3 Noise

The artificial low-frequency cues added to the speech signal are produced by filtering wide-band noise. The outputs of the synthesis filters are then scaled to appropriate levels based on the ouptut levels of each of the corresponding analysis filters. The DSP-96 produces its own noise to generate the low-frequency artificial cues by generating a pseudorandom sequence of numbers. The sequence is passed through each of the synthesis filters in turn to produce narrow-band noise. The noise, representing the low-frequency cues, is then scaled and added to the input signal.

Rather than generate four random number sequences, one for each synthesis filter, the DSP-96 generates one random sequence for all four filters.

3.2.4 PC interface

Though the entire vocoder system could be run solely from the DSP board, the addition of a personal computer allows a user to control the operation of the system without recompiling the board code after each modification. The PC handles the

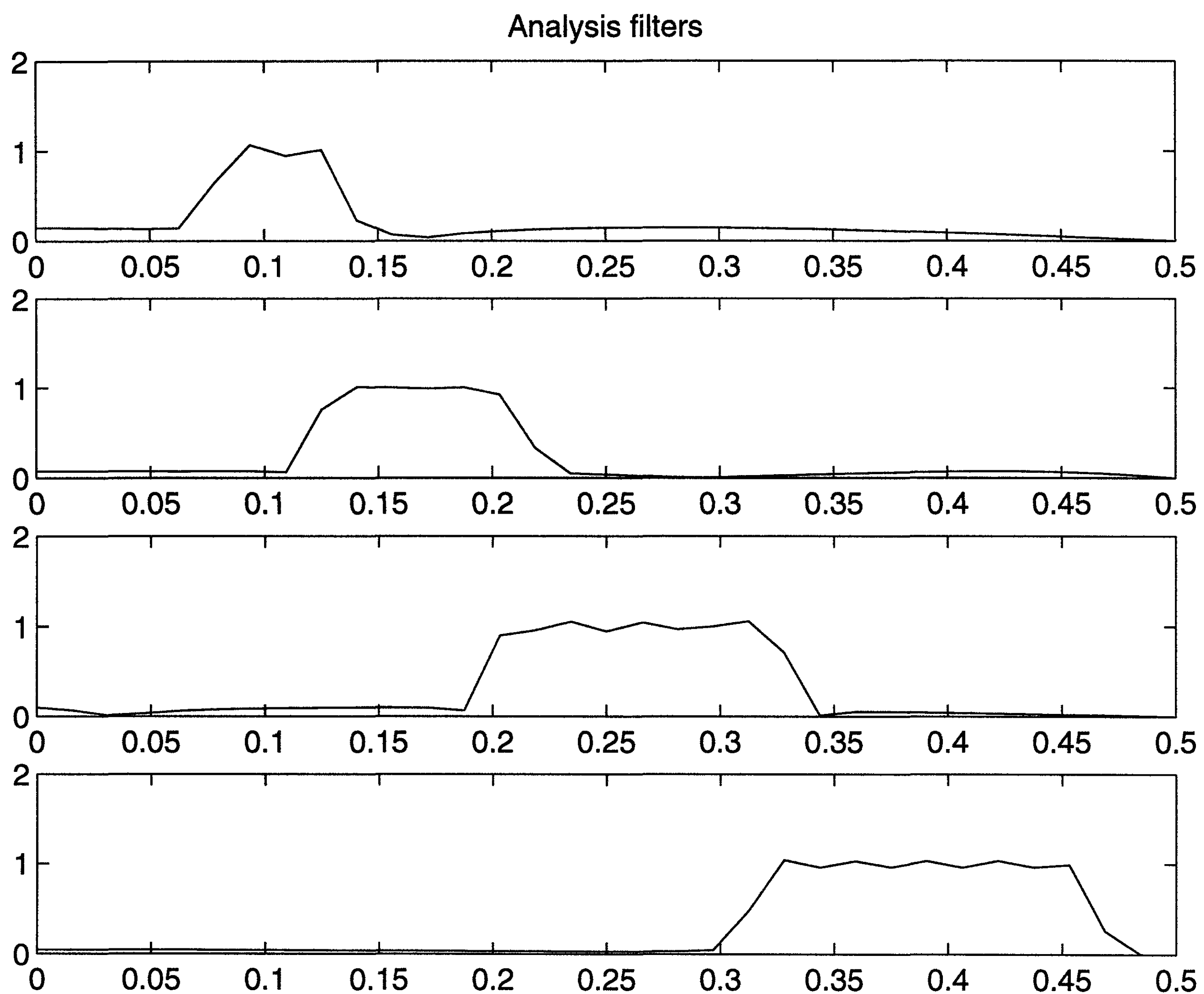

Figure 3-6: Example analysis filterbank designed by Matlab's "remez" function.

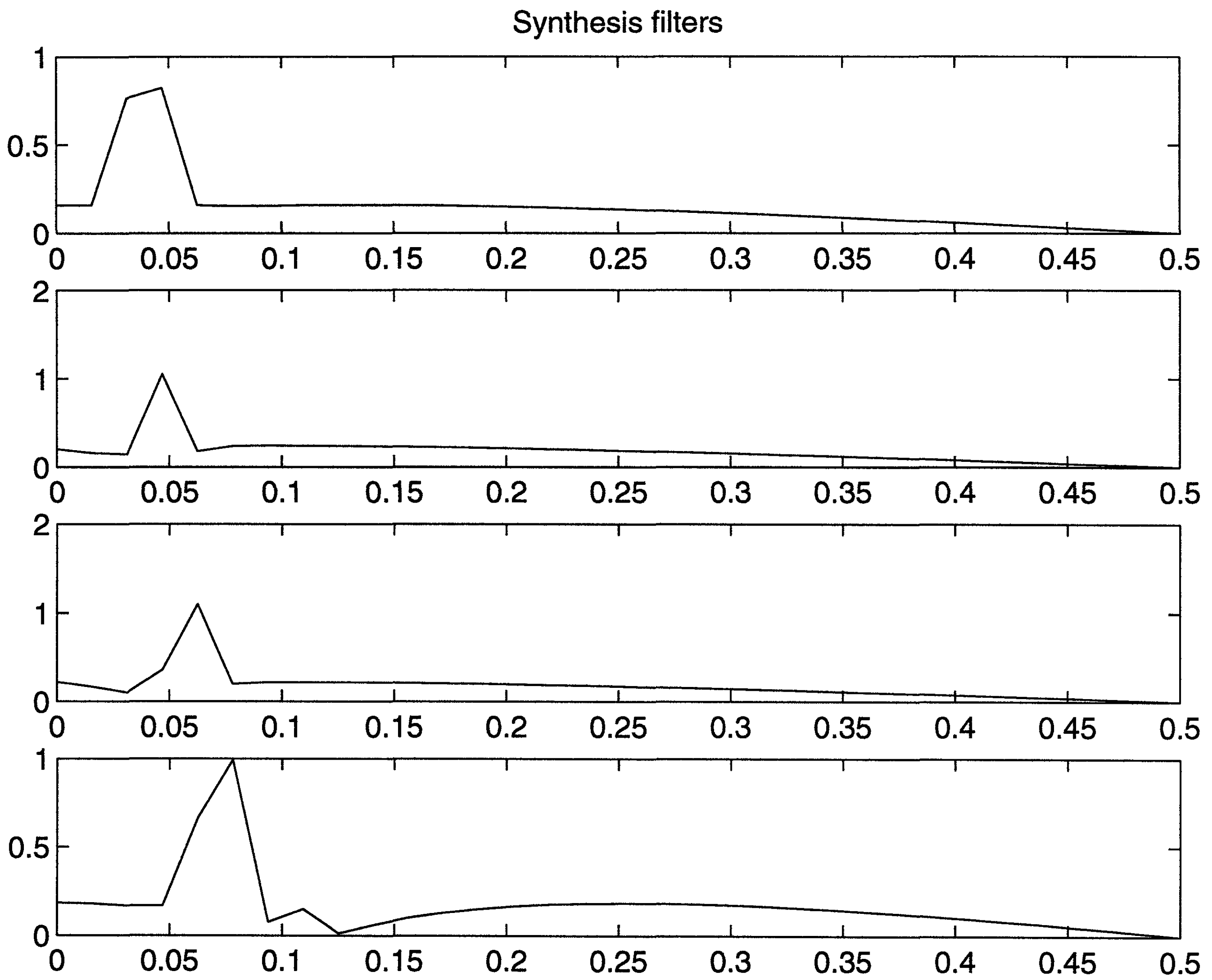

Figure 3-7: Example synthesis filterbank designed by Matlab's "remez" function.

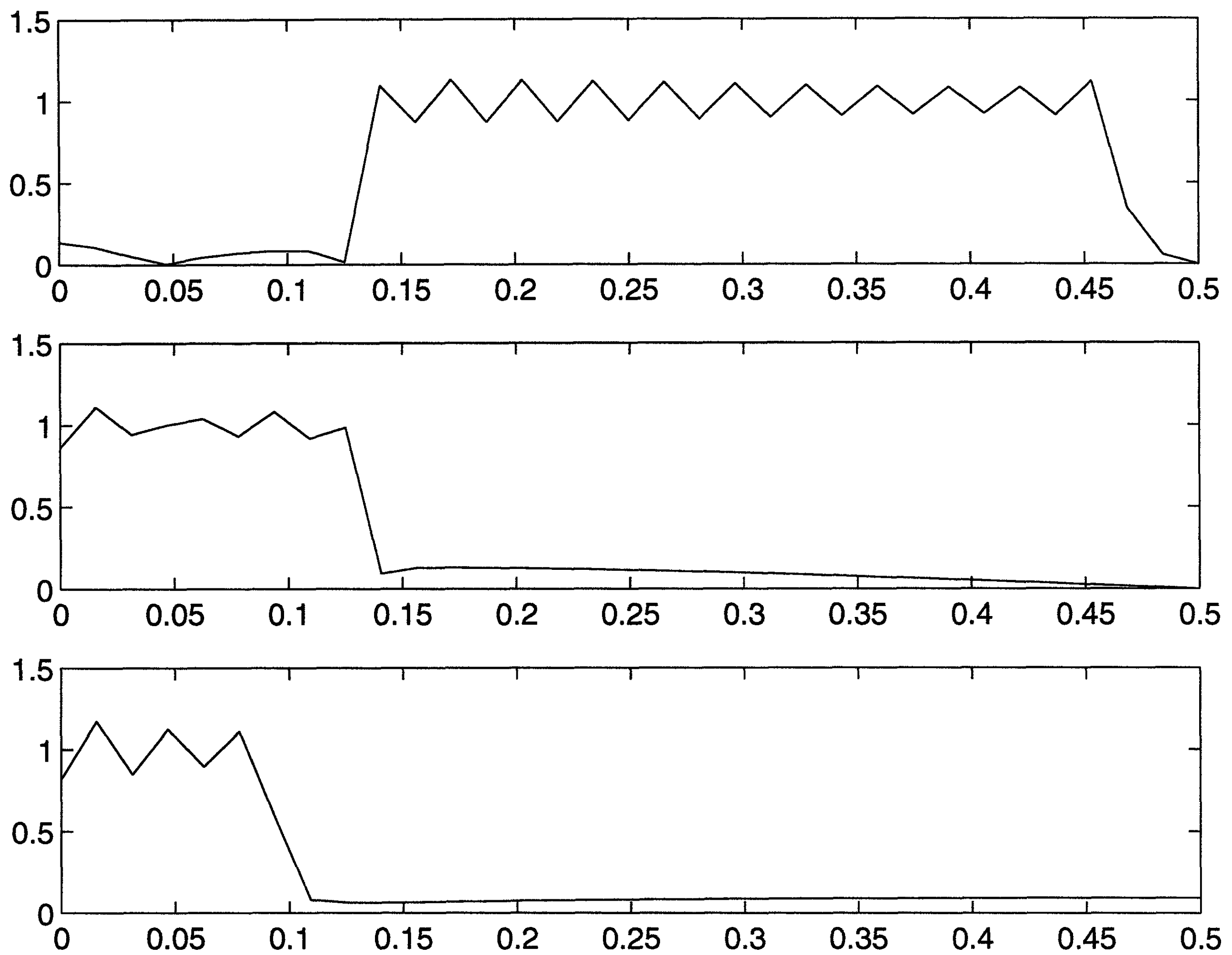

Figure 3-8: High-pass (a) and low-pass (b) filters for signal processing cutoff determination. Low pass filter used to simulate hearing loss (c) .

downloading of not only the board's program but also the appropriate parameters and filters. While the board program executes, the PC can request status information and vary run-time parameters without disturbing the continuous operation of the DSP board.

Chapter 4

Code

This chapter takes an in-depth look at both the personal computer program and the DSP board program.

The PC and the DSP board each require its own program and compiler. The PC code was compiled with the computer's local ANSI-C compiler. The code for the DSP board was compiled using the Motorola DSP96KCC compiler and optimizer on a SUN workstation. A `Make` file steps through the code compilation, code parallelization, interrupt routine preparation, and linking to standard files and libraries. [10]

4.1 Personal computer code

The PC code has two functions. First, it initializes the DSP board, providing it with its previously compiled program and passing it the user-specified values for operation. Second, it allows the user to monitor and control the output of the DSP board while the system operates by modifying system variables, including or bypassing various components of the system, displaying the results of the filterbank analysis, and checking the board for errors. The PC code can be found in Section A.1.

4.1.1 Downloading the DSP board code

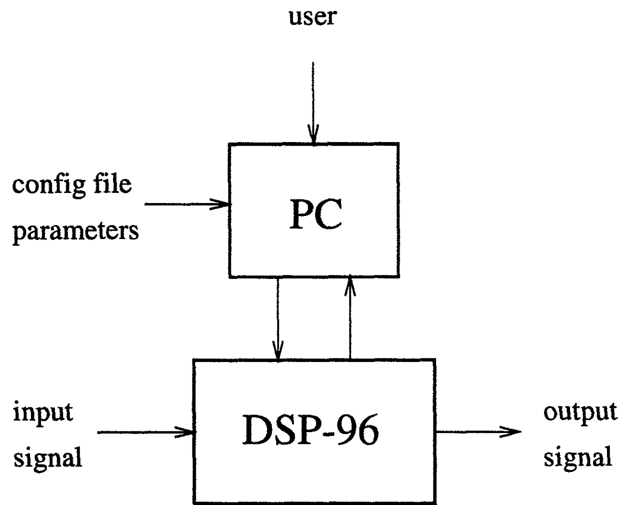

Figure 4-1: Overview of the digital vocoder system.

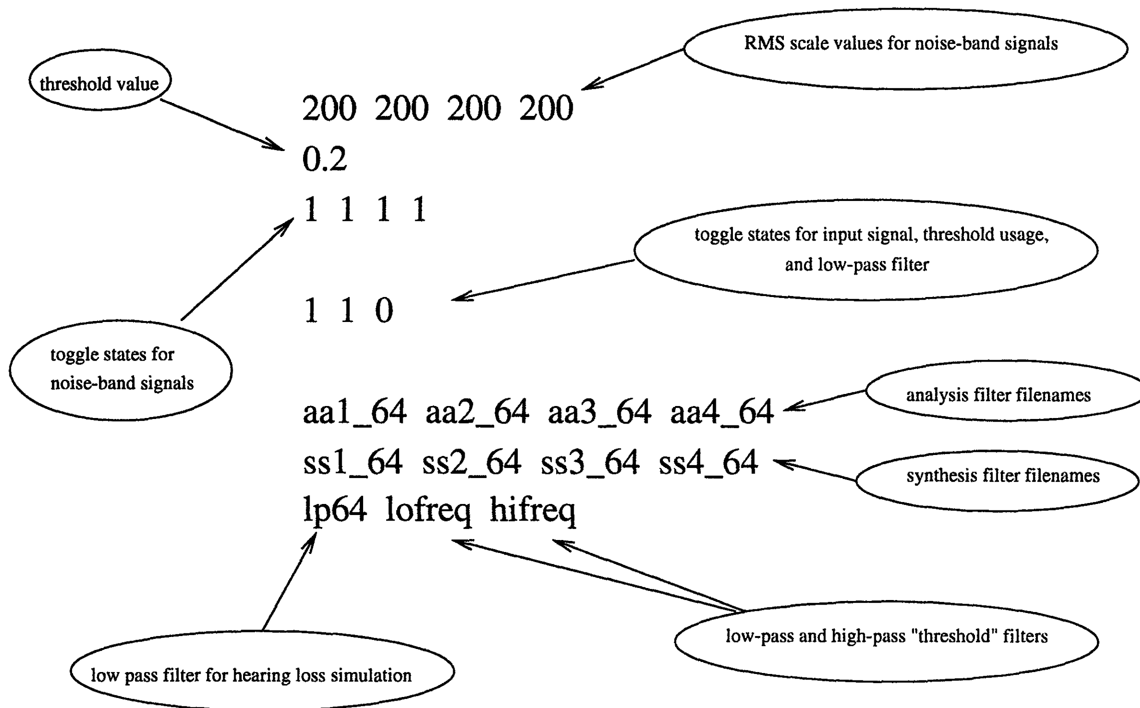

Figure 4-2: A sample configuration file.

Upon execution of the PC code, the program calls the `DspRunCompiled()` function to load the Ariel Janus monitor and the compiled board code onto the Ariel board. Janus is a bootloadable port monitor program for the 96002 processor which supports program control, data transfer, monitor relocation, and other functions. As Janus is responsible for preparing the DSP board to accept a non-bootloadable program, it must be downloaded prior to downloading the board code. The Ariel program is then loaded into the P-memory of the DSP board where it can be quickly accessed by the processor. Execution of the Ariel program begins immediately after downloading; the Janus monitor is relocated to run in the background. See [2] for more information about the Janus monitor and downloading DSP programs to the DSP-96 board.

Rather than using a long list of command line parameters to set each variable, the executable accepts only one argument: a configuration file specifying all the appropriate filter filenames and values for the system. A sample configuration file is shown in Figure 4-2. The file must specify, in the following order:

- the analysis filters
- the corresponding synthesis filters
- the high-pass and low-pass filters for suppression determination
- the low-pass filter
- the multiplication factors for noise-level amplitude
- the threshold ratio for suppression determination
- whether to include the input and noise-band signals in the output signal
- whether to use the low-pass filter
- whether to use the threshold ratio for suppression determination

When the configuration information has been loaded by the controller program, it feeds the information to the DSP board.

4.1.2 DSP board interface

Communication between the PC program and the DSP board during program execution is handled completely by interrupt service routines. The PC sends a hardware signal to the DSP-96 redirecting control to a particular address in memory. This address contains instructions which direct the program flow to a specified subroutine. Once the subroutine completes its operations, control returns to the program's point of execution just before the interrupt signal was received. Special instructions are necessary to initialize the interrupt-driven routines; see [2] and [10].

All interrupt routines used by this system follow the same basic handshake format (Figure 4-3). The PC, after verifying that the DSP board is ready to receive data, first sends values to the DSP board while the Ariel program is running (Figure 4-3a). The PC then calls the interrupt via a signal from `qcmd_m96()`. The DSP-96, notified by the signal, suspends its program to run the interrupt routine (Figure 4-3b). In the routine, the DSP-96 receives the values sent by the PC and then sends back an acknowledgement. Based on the identity of the interrupt routine that has been called, the DSP-96 knows both how to interpret the newly received value and how to respond to the PC. The board program then returns to its earlier point of execution, and the PC interprets the success of the routine based on the response from the board (Figure 4-3c).

The PC opens up each ASCII file in turn, reads the filter data point by point, and sends the information to the DSP-96 via `SendValue()`. The PC calls `GetValue()` to verify receipt of the data. After loading the filter data onto the DSP-96, the PC program calls a series of interrupt routines via `ToggleSignal()`, `ToggleNoise()`, `ToggleRatioUsage()`, and `ToggleLowPass()` to initialize certain variables used by the DSP board with the values read in from the configuration file. The PC keeps its own copy of these toggle variables to avoid unnecessary requests to the DSP board for their status. Once the initialization of these variables is accomplished, the PC commands the DSP-96 to begin its operation, and the PC program moves on to its second function.

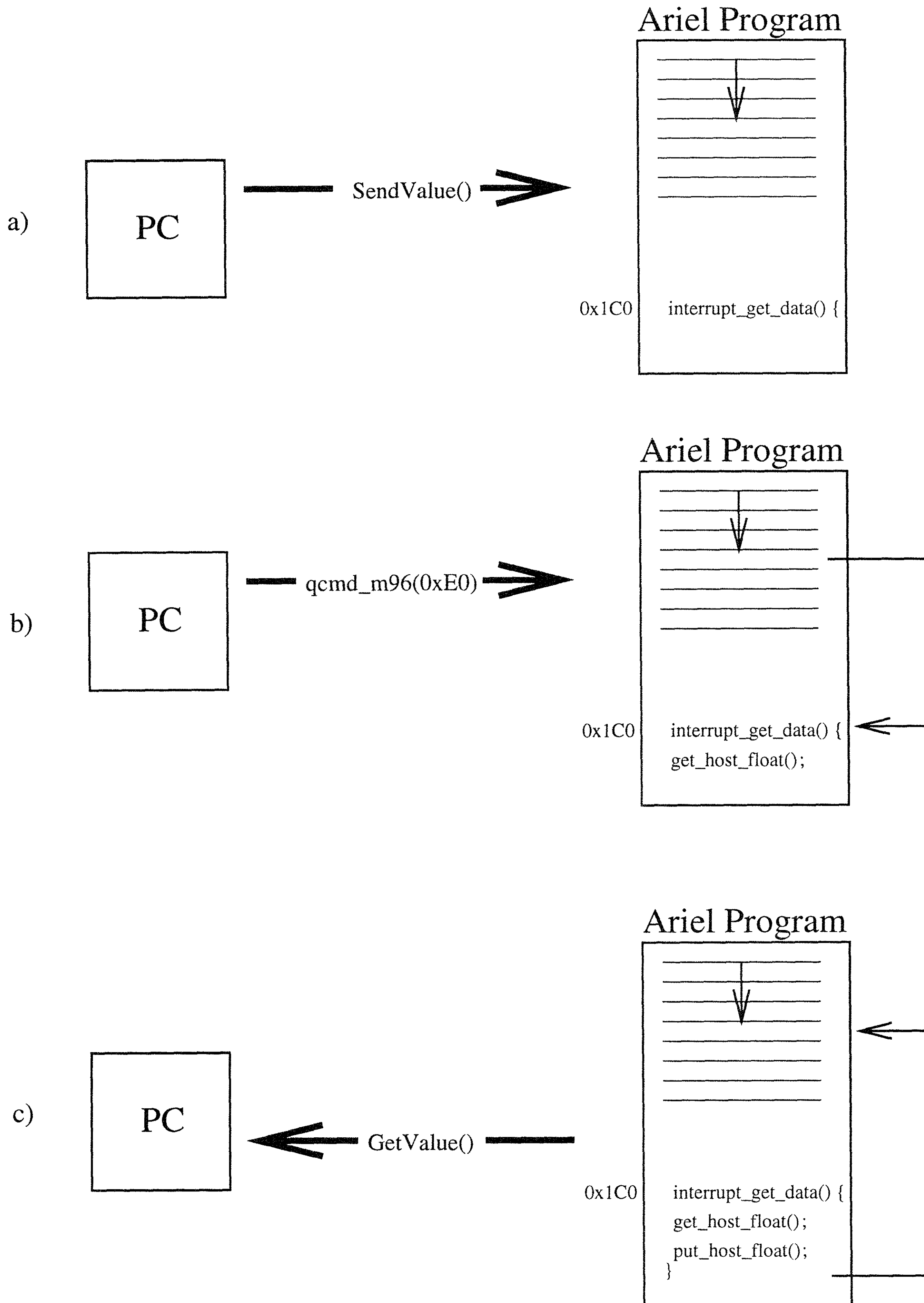

Figure 4-3: Interrupt usage for PC to DSP board interface.

4.1.3 Monitoring and altering the system

Once the board program has received the filter data, it enters a loop of sampling input and generating output, and running independently of the PC program, continuing even after the PC program terminates. However, by calling some of the same interrupt routines which initialized the DSP board, the PC can change key variables used by the DSP-96. Functions such as `CheckRMS()` and `CheckErrors()` call the appropriate DSP board interrupt routines to obtain information on system performance.

To facilitate the debugging of system parameters, the user can suppress certain components of the vocoder system by selecting from a list of menu options. These components include:

- the addition of the input signal to the output
- the addition of any or all noise-band signals to the output
- the low-pass filtering of the output
- the use of the threshold ratio to suppress processing

The same toggle functions used to initialize these parameters are called for this purpose.

The text menu also allows the user to:

- view the output levels of the analysis and synthesis filter
- view the total number of sampling errors (See 4.2.2)
- change the levels of the noise-band signals
- change the threshold ratio for suppression determination
- quit the PC program (though the board continues to operate)

These options place calls to `CheckRMS()`, `CheckErrors()`, `ChangeNoise()`, `ChangeRatio()`, and `Quit()`, which in turn call the appropriate DSP board interrupt routines.

4.2 DSP board code

The DSP board is the core of the vocoder system. It simultaneously samples the input signal, filters the input signal, generates narrow-band noise signals, adds the noise-band signals back into the input at levels dependent on information gleaned from the filtering operations, and produces an output signal. While executing all these real-time operations the board remains alert for information requests or parameter alterations by the controlling PC program. The code for the DSP board can be found in Section A.2.

4.2.1 Initial setup

Since the DSP board code is not bootloadable, the Janus port monitor program must be downloaded onto the DSP board. Once Janus is resident on the board and the DSP board code has been downloaded, the program immediately begins execution.

A few data buffers must be initialized prior to proceeding with the program. In order to use the sampling circuitry of the DSP-96 board, two buffers of C-type `iobuffer` must be set up and reserved for input and output from the board with the `init_input_buffer()` and `init_output_buffer()` procedures. To employ the signal processing procedures in the `ariel.h` library, the board declares buffers specifically set up to hold complex numbers. The data type `complex`, along with procedures for accessing the real and imaginary parts of the buffer data, handles the necessary memory organization. To make these buffers readily accessible by the board, hence increasing its speed, the `#pragma` command declares storage for the buffers in available space on the board's SRAM rather than its slower DRAM. For more information on memory organization refer to [2] and [10].

Other variable declarations and initializations by the board code that need only be performed once are executed prior to the main loop of the program. This includes the zero-padding of buffers, the reception of filter data from the personal computer, and the FFT calculation for each of these filters.

4.2.2 Sampling I/O

After setting up the input and output buffers and making a call to `set_sample_rate()`, a call to `start_sampling()` prepares the DSP-96 for sampling with appropriate interrupt routines, then causes the DSP board to store samples into the input buffer and playing samples from the output buffer. The DSP board continues to place and play samples until the program terminates. This high-level interaction between the program and the DSP board's A/D and D/A converters abstracts the complex real-time processes of sampling, coordinating all synchronization, wait-states, and interrupt procedure calls without further code. The `ariel.h` library procedures `get_input()` and `put_output()` retrieve or pass sample data to these input/output buffers (Figure 4-4).

The DSP board program's main loop pulls `CHUNK_SIZE` samples from the input buffer and places them in the buffer `data_in`. The DSP board then performs the necessary filtering operations upon the sampling information in `data_in` without corrupting the buffer data. When the DSP board has prepared a `data_out` buffer complete with narrow-band noise and the original input data, the `data_out` samples are passed to the output buffer, and the loop repeats. Any samples either written to an overflowing input buffer or played from an exhausted output buffer will increment the number of errors returned by the `get_input()` or `put_output()` functions. The number of these errors is reported to the PC by a call to `interrupt_errors_check()`.

4.2.3 Filtering

In the implementation of the vocoder system, the bulk of the DSP board's computations is devoted to filtering. Filtering requires the implementation of a linear convolution of the input signal with the given filter sequence. This linear convolution may be obtained by performing a circular convolution of sufficient length using the Discrete Fourier Transforms (DFTs) of the two sequencess. This section details the implementation of the filtering process in the DSP board code.

The `ariel.h` library includes signal processing functions to compute both the

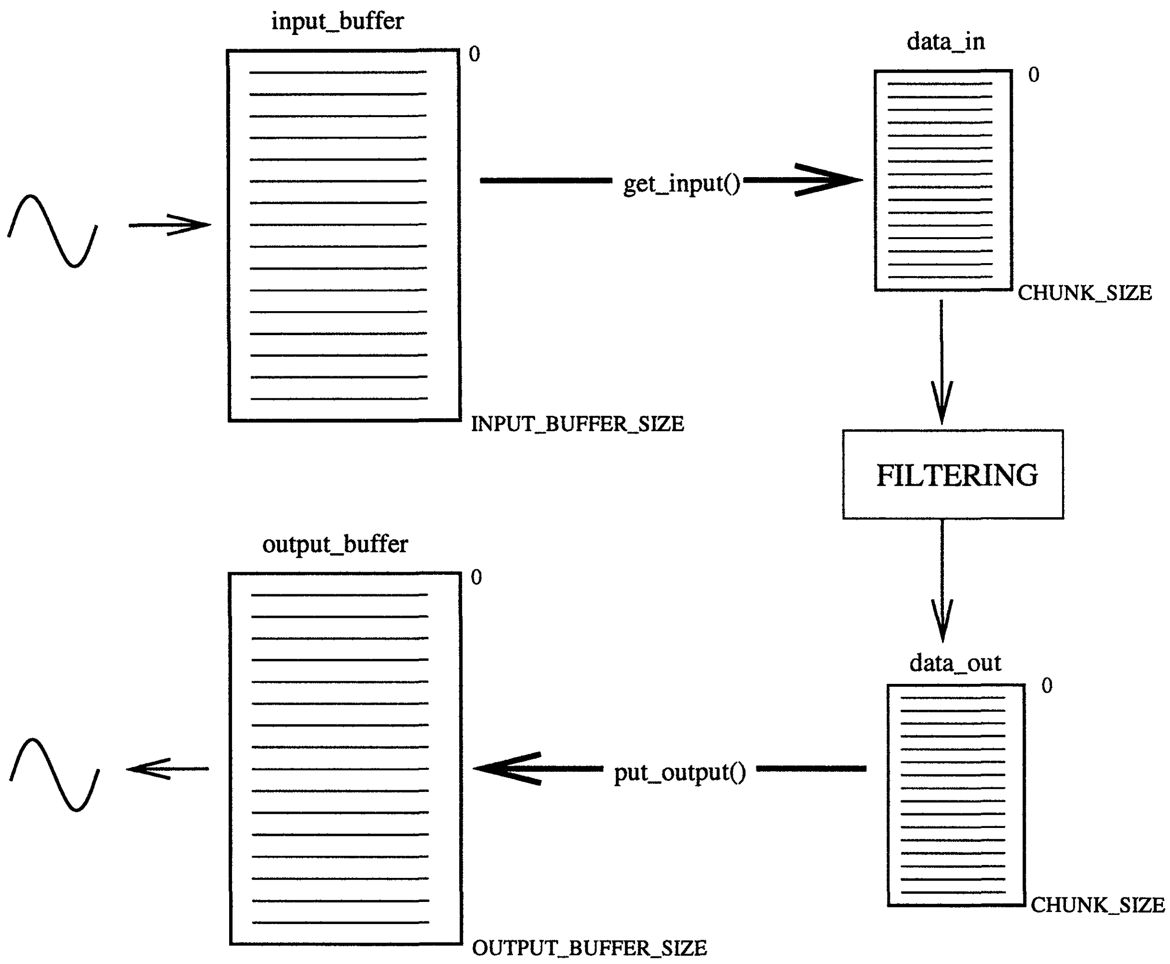

Figure 4-4: DSP board sampling procedure.

N-point fast Fourier transform (FFT) and the N-point inverse FFT of a complex buffer. Both `real_fft()` and `inv_fft()` require two buffers of data type `complex`; the complex array `tempbuffer`, used as an intermediate buffer for each of these functions, contains corrupted data after the call. The FFTs of the noise, input, and filter sequences are stored into `noisebuffer`, `input_signal`, or one of the `filter` data arrays, respectively.

As each filter is downloaded to the DSP board it is temporarily placed in the zero-padded `data_in` buffer until all `FILTER_SIZE` points have been received. The program next performs the FFT of the filter response sequence and stores it in the appropriate `filter` buffer, then repeats the procedure for the next filter. This process improves the speed of the program by computing the FFTs just once.

The overlap-add method of block convolution is implemented using several buffers. The indefinite-length input signal is segmented into smaller finite sequences by reading in `CHUNK_SIZE` data points from the input buffer. To save time, the sequence is zero-padded prior to the main loop by assigning the remaining `(FFT_SIZE - CHUNK_SIZE)` points in the `data_in` array to zero. The program calculates the FFT of the input and stores it in `input_signal`. A call to `multbuff()` then performs an element by element multiplication of `input_signal` with the precalculated FFT of the given filter. The program calculates the inverse FFT of the result and stores it in the `data_out` array. The overlapping `FILTER_SIZE` points at the end of the resultant sequence `data_out`, representing time-aliased points from the circular convolution, are copied in the `overlap` buffer corresponding to that filter and then added into the next convolution performed with that filter. The information in `data_out` is only used to calculate an RMS level and store it in the `rms` array; the data is written over when calculating the next filtering operation.

4.2.4 PC interface

As described above, all communication between the DSP board and the PC is accomplished by the use of interrupt service routines. Upon execution, the DSP board loops in a wait state until it has received all of filter information from the PC program.

As the PC sends a value to the DSP board, it calls the `interrupt_get_data()` routine. The DSP board temporarily leaves its wait state, places the received data in the `data_in` array (since it is not needed for sampling data at the time), increments the `index` counter, and returns the counter value back to the PC. When the `Check()` function reports that `FILTER_SIZE` data points have been received, the program proceeds with the FFT calculations for that filter and begins accepting values for the next filter.

The PC can initialize and later control various run-time parameters by calling other interrupt routines. The `interrupt_toggles()` function toggles the input signal, use of the threshold ratio, and use of the low pass filter, depending on which signal the PC sends. The `interrupt_toggle_noise()` function toggles the presence of each of the noise-bands in the output signal. The `interrupt_set_value()` function accepts a signal and a value from the PC, the signal indicating whether the threshold ratio or one of the noise-band scale factors should be set to the value provided.

4.2.5 Noise

The DSP board code is also responsible for the generation of narrow-band noise. This is accomplished by first generating wide-band noise, then passing the wide-band noise signal through the synthesis filters. It is the vocoder system's addition of the narrow-band noise back into the input signal which creates the artifical lower-frequency speech cues for an impaired listener.

To create the wide-band noise, the code uses a variation of the Box-Muller method for generating random deviates with a normal (Gaussian) distribution, as described in [7]. The `gasdev()` function implements this procedure, using the `ran1()` function to generate random numbers quickly. The data placed in the `noise` buffer, created by a series of calls to `gasdev()`, is then filtered through the synthesis filter to create narrow-band noise signals in the `noise_out` buffers. The overlap-save method is again used for this filtering, with `noise` replacing `data_in` and `noise_out` replacing `data_out`.

The amplitude of the noise signals is controlled by the RMS level calculated from

the analysis filterbank output. Because the set of noise signals is of an arbitrary unitless amplitude, the `noise_out` signals are multiplied by the appropriate RMS values (stored in `rms`) after dividing by a preset factor, `rms_scale_value`. This factor can be specified in the configuration file or modified during run-time.

Chapter 5

Conclusions

5.1 Functionality of current system

The current system functions as described in chapter 4. Specifications from a configuration file are successfully read in by the personal computer and downloaded to the DSP board. The board responds to PC status queries. A constant-tone input produces the expected results: the RMS value reported to the PC for the appropriate frequency range increases, and low-frequency noise for the corresponding band is added to the signal (Figure 5-1). At a threshold level of roughly 0.2, the DSP-96 added noise for the the /z/ and /t/ phonemes in the consonant-vowel syllables /zu/ and /ta/ but suppressed noise for the /l/ in /li/.

Problems surfaced with maintaining enough speed to avoid sampling errors under certain conditions. The random sequence generator could not generate a new random sequence for each loop of the program without causing sampling errors; consequently, the generator was moved outside of the main loop and the synthesis filterings were performed on the same noise sequence. Other combinations of conditions also caused sampling errors to occur: if the PC requested information from the DSP board while the low-pass filter simulating hearing loss was active, or if the threshold filters were active and the input signal never fell below the threshold level, then the DSP-96 would be unable to feed data to the output sampling buffer with enough speed to avoid errors.

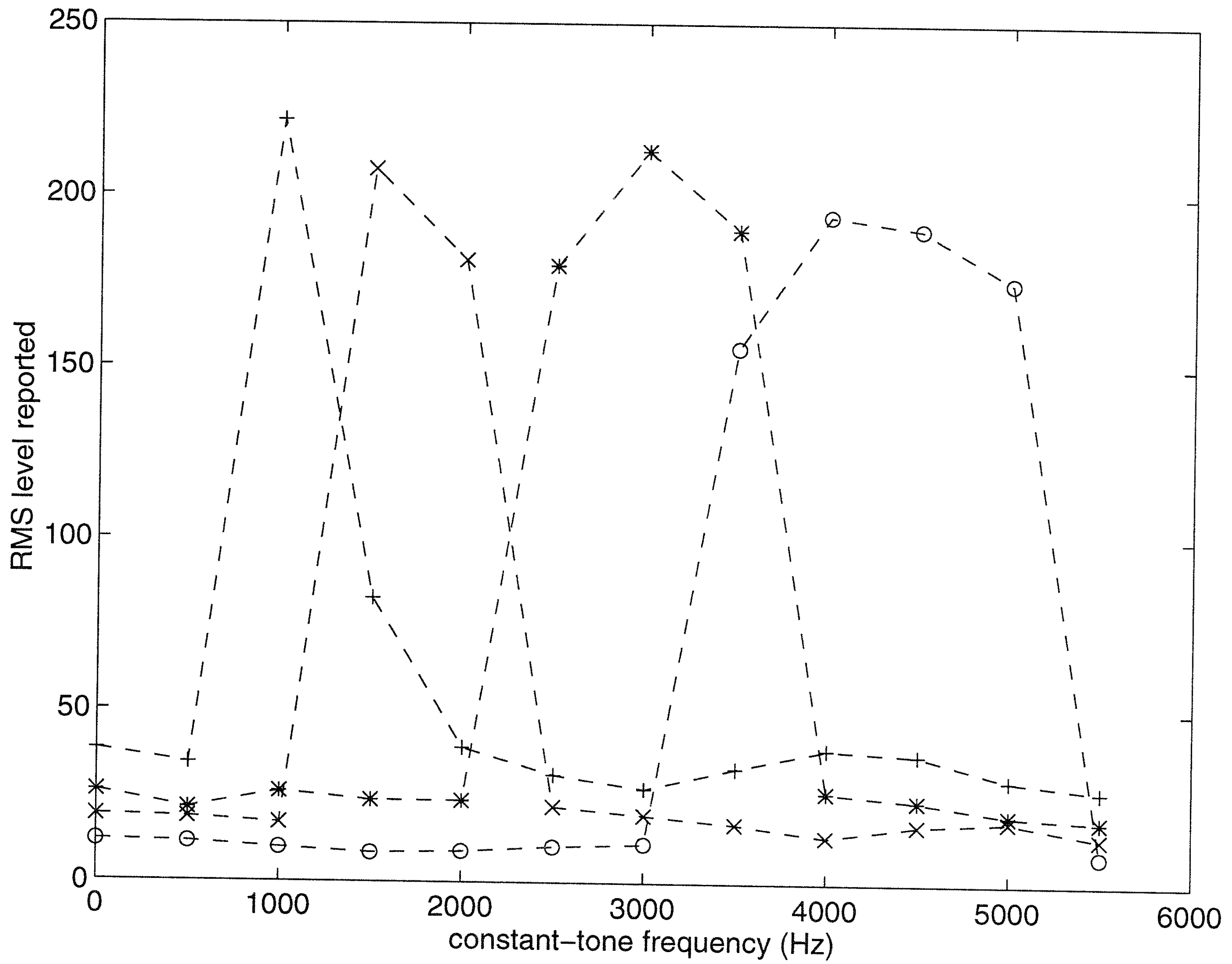

Figure 5-1: RMS levels reported by the PC for constant-tone inputs.

5.2 Suggestions for further work

5.2.1 Evaluation

Extensive testing of the viability of the vocoder system as a hearing aid still remains. Such testing would help optimize vocoder specifications, such as filter design, noise scaling, and the suppression threshold, that were previously limited by the available analog equipment. Experiments could include subject feedback in determining system settings.

Analysis and synthesis filterbanks which differ from the standards set by the analog vocoder systems could also be implemented. These filterbanks, for example, might feature filters that are not contiguous, of unequal bandwidths, or that control more than one noise-band, in order to explore the effect of weighting different frequency ranges of speech. Filterbanks could be designed based on the audiogram of a hearing-impaired listener.

5.2.2 Improvement

Some modifications to the source code might improve the system further.

The sampling errors necessitate more modifications to the program to improve its speed. Loading up a pregenerated array of random numbers rather than generating them during run-time would improve speed, as would reducing the sizes of the FFTs and the filters.

More functionality could be added to the PC controlling program: for example, a way to reset the board in case of errors, a means of replacing a filter "on the fly," or perhaps even a more user-friendly, informative interface for a subject to use instead of the crude text menu currently implemented.

More complicated methods of controlling changes to the amplitude of noise signals might also be implemented. The current system does not exert control over the noise-band levels based on the level of low-frequency sounds already present in the input signal, as Posen's system did. Noise levels could be specified as a percentage of the

signal amplitude rather than as a multiplicative factor to make the determination of RMS scale values more intuitive for a user.

Appendix A

Source Code

A.1 Code for controlling personal computer

The code for the personal computer side of the system was compiled using the standard C compiler available on the machine. The code consists of a header file and main file.

```
/* pc.h -- last update 2/1/96 */
/* Jeff Foley */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <m96/dsp.h>
#include "/ollie/jjfoley/Code/lib/run_compiled.h"
#include "/ollie/jjfoley/Code/lib/wait_m96.h"
                                                            10
#define JANUS_MONITOR "janusl.lod"
#define BOARD_PROGRAM "board.lod"
#define MAX_WAIT 10
#define MAX_STR 80
#define NUM_FILTERS 4
#define NUM_MORE_FILTERS 3
#define NUM_ARGS 2
```

```
#define DEBUG 0
#define OFF 0
#define ON 1                                                    20

/* menu options */
void ToggleNoise(int *current_value);
#define SIGNAL_TOGGLE 1
void ToggleSignal(int *current_value);
#define RATIO_TOGGLE 2
void ToggleRatioUsage(int *current_value);
#define LPF_TOGGLE 3
void ToggleLowPass(int *current_value);
void CheckRMS(void);                                            30
void CheckErrors(void);
void ChangeRatio(void);
void ChangeNoise(void);
void Quit(void);

/* other useful functions */
void Help(void);
void FileError(char *);
void SendValue(float);
void GetValue(int);                                             40
```

```
/* pc.c -- last update 2/1/96 */
/* Jeff Foley */

#include "pc.h"

main(int argc, char * argv[])
{
  FILE *fp; FILE *fp1; int i; float value, retrieved, t;
  char choice;
  char filtfile[MAX_STR];                                       10
```

```
int signal_on=1, noise_on[NUM_FILTERS], ratio_on=0, lpf_on=0;
float threshold;

/* ------INITIALIZE BOARD CONFIGURATION------ */

if (argc != NUM_ARGS) Help();
get_config_m96(NULL);
DspRunCompiled(BOARD_PROGRAM, JANUS_MONITOR, 0, 1);

if (!(fp = fopen(argv[1], "r"))) FileError(argv[1]);                    20

/* read in rms levels and threshold ratio*/
for(i=0; i<NUM_FILTERS+1; i++) {
  fscanf(fp, "%f", &value);
  qsend_float_m96(i);
  qsend_float_m96(value);
  qcmd_m96(0xE0);  /* calls interrupt service routine at 0x1C0 */
}
/* read in initial toggle states */
for(i=0; i<NUM_FILTERS; i++) {                                          30
  fscanf(fp, "%f", &value);
  if (value) noise_on[i]=ON;
  else {
    noise_on[i]=OFF; qsend_float_m96(i); qcmd_m96(0xD0);
  }
}
fscanf(fp, "%f", &value);
if (!value) {
  signal_on=OFF; qsend_float_m96(SIGNAL_TOGGLE); qcmd_m96(0xC0);
}                                                                       40
fscanf(fp, "%f", &value);
if (value) {
  ratio_on=ON; qsend_float_m96(RATIO_TOGGLE); qcmd_m96(0xC0);
}
fscanf(fp, "%f", &value);
if (value) {
```

```
  lpf_on=ON; qsend_float_m96(LPF_TOGGLE); qcmd_m96(0xC0);
 }

 /* Send the analysis, synthesis, lowpass, and high- and low-freq filters                                   50
    to the DSP board */
 for (i=0; i<NUM_FILTERS*2 + NUM_MORE_FILTERS; i++) {
  fscanf(fp, "%s", filtfile);
  if (!(fp1 = fopen(filtfile, "r"))) FileError(filtfile);
  printf("File '%s' -->",filtfile);
  if (i < NUM_FILTERS) printf(" Analysis filter #%d\n", i+1);
  else if (i < NUM_FILTERS*2) printf(" Synthesis filter #%d\n", (i%4)+1);
  else if (i == NUM_FILTERS*2) printf(" Low Pass filter\n");
  else if (i == NUM_FILTERS*2+1) printf(" Low Frequency filter\n");
  else if (i == NUM_FILTERS*2+2) printf(" High Frequency filter\n");                                        60
  else printf(" Additional (unknown) filter\n");

  while (fscanf(fp1, "%f", &value) != EOF) {
   SendValue(value);
   GetValue(i);
  }
 }

/* -----MAIN MENU----- */
                                                                                                          70
 while (1) {
  printf("\nOptions:\n");
  printf("(1) turn input signal");
  if (signal_on) printf(" off\n"); else printf(" on\n");
  printf("(2) toggle noise-band signals on and off\n");
  printf("(3) turn threshold ratio usage");
  if (ratio_on) printf(" off\n"); else printf(" on\n");
  printf("(4) turn low-pass filtering");
  if (lpf_on) printf(" off\n"); else printf(" on\n");
  printf("(5) check filter outputs\n");                                                                   80
  printf("(6) check total sampling errors reported\n");
  printf("(7) change threshold ratio\n");
```

```
    printf("(8) change levels of noise-band signals\n");
    printf("(q) quit\n");
    choice = getchar();
    if (choice == '\n') choice = getchar();
    if (choice == '1')
      ToggleSignal(&signal_on);
    else if (choice == '2')
      ToggleNoise(noise_on);                                    90
    else if (choice == '3')
      ToggleRatioUsage(&ratio_on);
    else if (choice == '4')
      ToggleLowPass(&lpf_on);
    else if (choice == '5')
      CheckRMS();
    else if (choice == '6')
      CheckErrors();
    else if (choice == '7')
      ChangeRatio();                                            100
    else if (choice == '8')
      ChangeNoise();
    else if (choice == 'q')
      Quit();
    else printf("* Not a menu option *\n");
  }
}

/* -----MENU FUNCTIONS----- */
                                                                110
void ToggleSignal(int *current_value){
  printf("\n<< Input Signal ");
  if (*current_value) {
    printf("Off >>\n"); *current_value=OFF;
  } else {
    printf("On >>\n"); *current_value=ON;
  }
  qsend_float_m96(SIGNAL_TOGGLE);
```

```
  qcmd_m96(0xC0);  /* calls interrupt service routine at 0x180 */
}                                                                                   120

void ToggleNoise(int *current_value){
  int which;
  printf("\nWhich noise-band (1-%d)?", NUM_FILTERS);
  scanf("%d",&which);
  if (which < 1 || which > NUM_FILTERS) {
    printf("\n* Invalid noise-band *\n");
  } else {
    printf("\n<< Noise Signal %d ", which);
    if (current_value[which]) {                                                     130
      printf("Off >>\n"); current_value[which]=OFF;
    } else {
      printf("On >>\n"); current_value[which]=ON;
    }
    qsend_float_m96(which-1);
    qcmd_m96(0xD0);   /* calls interrupt service routine at 0x1A0 */
  }
}

void ToggleRatioUsage(int *current_value){                                          140
  int new_value;
  printf("<< Threshold Ratio Usage ");
  if (*current_value) {
    printf("Off >>\n"); *current_value=OFF;
  } else {
    printf("On >>\n"); *current_value=ON;
  }
  qsend_float_m96(RATIO_TOGGLE);
  qcmd_m96(0xC0);  /* calls interrupt service routine at 0x180 */
}                                                                                   150

void ToggleLowPass(int *current_value) {
  int new_value;
  printf("\n<< Low Pass Filtering ");
```

```
  if (*current_value) {
    printf("Off >>\n"); *current_value=OFF;
  } else {
    printf("On >>\n"); *current_value=ON;
  }
  qsend_float_m96(LPF_TOGGLE);                                          160
  qcmd_m96(0xC0);  /* calls interrupt service routine at 0x180 */
}

void CheckRMS(void) {
  int i;
  qcmd_m96(0xA0);  /* calls interrupt service routine at 0x140 */
  for (i=0; i<NUM_FILTERS; i++) {
    printf("Filter #%d:  RMS level %g\t", i+1, qget_float_m96());
    printf("RMS scale value:  %g\t", qget_float_m96());
    printf("Noise:  %g\n", qget_float_m96());                           170
  }
  printf("Threshold ratio set to %g\n", qget_float_m96());
}

void CheckErrors(void) {
  qcmd_m96(0xB0);  /* calls interrupt service routine at 0x160 */
  printf("\nTotal number of sampling errors:  %.1f\n", qget_float_m96());
}

void ChangeRatio(void) {                                                180
  float new_ratio;
  printf("\nWhat should the new threshold ratio be?");
  scanf("%f", &new_ratio);
  qsend_float_m96(NUM_FILTERS);
  qsend_float_m96(new_ratio);
  qcmd_m96(0xE0);  /* calls interrupt service routine at 0x1C0 */
}

void ChangeNoise(void) {
  int which; float newlevel;                                            190
```

```
  printf("\nWhich noise-band (1-%d)?", NUM_FILTERS);
  scanf("%d",&which);
  printf("\nWhat should the new level be?");
  scanf("%f",&newlevel);
  if (which > NUM_FILTERS || which < 1) {
    printf ("\n* Invalid noise-band *\n");
  } else {
    printf("\n<< Noise Signal Level %d set to %g", which, newlevel);
    qsend_float_m96(which−1);
    qsend_float_m96(newlevel);                                                200
    qcmd_m96(0xE0); /* calls interrupt service routine at 0x1C0 */
  }
}

void Quit(void) {
  /* if anything needs to be cleaned up before exiting, do it now */
  exit(−1);
  }

/* ------ADDITIONAL USEFUL FUNCTIONS------ */                                  210

void Help(void) {
  printf("\nUsage:  pc configfile\n");
  printf("where configfile is a text file with specs for the vocoder.\n");
  printf("See sample.cfg for an example of a valid config file.\n");
  exit(−1);
}

void FileError(char *filename) {
  printf("\nCouldn't open file \"%s\", exiting...\n", filename); exit(−1);    220
}

void SendValue(float value) {
  if (DEBUG) printf("\nValue:  %f", value);
  if (WaitCanSendM96(0,DFLT,MAX_WAIT) == −1) {
    printf("\n*** Timed out waiting to send to DSP board ***\n");
```

```
    exit(-1);
  } else {
    qsend_float_m96(value);
    qcmd_m96(0x90);                                                    230
  }
}

void GetValue(int i) {
  float retrieved;
  if (WaitCanGetM96(0,DFLT,MAX_WAIT) == -1) {
    printf("\n*** Timed out waiting to receive from DSP board ***\n");
    exit(-1);
  } else {
    retrieved = qget_float_m96();                                      240
    if (DEBUG) printf("\tGot value:  %f ",retrieved);
  }
  if (retrieved == -1) {
    printf("** Board rejected value loading filter #%d", i);
  }
}
```

A.2 Code for Ariel DSP-96 board

The code for the digital signal processing board was compiled using the g96k compiler on a SUN workstation A Make file, as specified by Sexton in his document, ensures that the proper compiler, parser, and optimizer are invoked on the C code.[10]. This code also consists of a header file and main file.

```
/* board.h -- last update 2/3/96 */
/* Jeff Foley */

#include <ariel.h>
#include <math.h>

/* system parameters */
```

```
#define INPUT_BUFFER_SIZE 5000
#define OUTPUT_BUFFER_SIZE 5000
#define SAMP_RATE 11025.0                                                        10
#define RMS_INIT_SCALE_VALUE 750
#define FILTER_SIZE 64 /* 32 */
#define CHUNK_SIZE 193 /* 97 */ /* FILTER_SIZE < CHUNK_SIZE */
#define FFT_SIZE 256 /* 128 */  /* FFT_SIZE = FILTER_SIZE + CHUNK_SIZE - 1 */
#define NUM_FILTERS 4  /* number of analysis-synthesis filter pairs */
#define NUM_MORE_FILTERS 3  /* for the lowpass and 2 threshold filters */
#define TOTAL_FILTERS NUM_FILTERS*2+NUM_MORE_FILTERS

#define SIGNAL_TOGGLE 1
#define RATIO_TOGGLE 2                                                           20
#define LPF_TOGGLE 3
#define ON 1
#define OFF 0
#define LPF_FILTER NUM_FILTERS*2
#define LO_FILTER NUM_FILTERS*2+1
#define HI_FILTER NUM_FILTERS*2+2

/* definitions for "quick and dirty" random generator,
   taken from _Numerical_Recipes_in_C_ */
unsigned long myidum, itemp;                                                     30
float rand;
#ifdef vax
  static unsigned long jflone = 0x00004080;
  static unsigned long jflmsk = 0xffff007f;
#else
  static unsigned long jflone = 0x3f800000;
  static unsigned long jflmsk = 0x007fffff;
#endif

/* interrupt service routine prototypes -- IMPORTANT: Names must begin           40
   with "interrupt" for parser to detect and setup interrupts properly */
void interrupt_get_data(void);
void interrupt_rms_check(void);
```

```
void interrupt_errors_check(void);
void interrupt_toggles(void);
void interrupt_toggle_noise(void);
void interrupt_set_value(void);

int check(void);
float ran1(void);
float gasdev(void);
void GenerateNoiseSignal(void);
void multbuff(complex[], complex[], complex[]);

static iobuffer input_buffer[INPUT_BUFFER_SIZE];
static iobuffer output_buffer[OUTPUT_BUFFER_SIZE];

/*complex numbers put into l-mem for greater speed in filtering calculations*/
#pragma l_run l:$108000
static complex tempbuffer[FFT_SIZE];
static complex input_signal[FFT_SIZE];
static complex noisebuffer[FFT_SIZE];
static complex filter[NUM_FILTERS*2+3][FFT_SIZE];
#pragma l_run

float data_in[FFT_SIZE];
float data_out[FFT_SIZE];
float threshold_out[FFT_SIZE];
float noise_out[NUM_FILTERS][FFT_SIZE];
float noise[FFT_SIZE];
float rms[TOTAL_FILTERS];
float overlap[TOTAL_FILTERS][FFT_SIZE];

float timer; int errors=0; /* sampling variables */

/* Parameters which can be changed by config file */
int index=0, lpf_on=0, signal_on=1, ratio_on=0, noise_on[NUM_FILTERS];
float rms_scale_value[NUM_FILTERS];
float ratio=1.0;
```

```
int process=ON;
```

```
/* board.c -- last update 2/3/96 */
/* Jeff Foley */

#include "board.h"

main()
{
  int i, j, k;
  init_input_buffer(input_buffer,INPUT_BUFFER_SIZE);
  init_output_buffer(output_buffer,OUTPUT_BUFFER_SIZE);
  /* specify user interrupt vectors (between 0x120 and 0x1FE) */
  set_interrupt(0x120,(void (*)(void))interrupt_get_data);
  set_interrupt(0x140,(void (*)(void))interrupt_rms_check);
  set_interrupt(0x160,(void (*)(void))interrupt_errors_check);
  set_interrupt(0x180,(void (*)(void))interrupt_toggles);
  set_interrupt(0x1A0,(void (*)(void))interrupt_toggle_noise);
  set_interrupt(0x1C0,(void (*)(void))interrupt_set_value);

  /* initialize noise levels, overlap buffer; zero-pad input-signal for FFTs */
  for (i=0; i<NUM_FILTERS; i++) {
    rms_scale_value[i] = RMS_INIT_SCALE_VALUE;
    noise_on[i] = ON;
  }
  for (i=0; i<TOTAL_FILTERS; i++) {
    for (j=0; j<FILTER_SIZE-1; j++) {
      overlap[i][j]=0;
    }
  }
  for (j=FILTER_SIZE; j<FFT_SIZE; j++) {
    data_in[j]=0;
  }

  /* FFT the filters and stuff them into "filter" buffers */
```

```
for (i=0; i<TOTAL_FILTERS; i++) {
  while(check() == 0);  /* wait for filters to be loaded in and padded */
  real_fft(data_in, tempbuffer, filter[i], FFT_SIZE);
  index=0;
}

/* preset narrow-band noise signal */                                    40
GenerateNoiseSignal();
real_fft(noise, tempbuffer, noisebuffer, FFT_SIZE);
for (i=0; i<NUM_FILTERS; i++) {
  multbuff(noisebuffer, filter[i+NUM_FILTERS], tempbuffer);
  inv_fft(tempbuffer, noise_out[i], FFT_SIZE);
}

set_sample_rate(SAMP_RATE, NORMAL_MODE, ANALOG_IO);
start_sampling();
                                                                         50
/* -----MAIN LOOP----- */

for (;;){
  reset_timer();
  errors += get_input(CHUNK_SIZE,0,data_in);

  real_fft(data_in, tempbuffer, input_signal, FFT_SIZE);
  if (ratio_on) {
    rms[HI_FILTER]=0;
    rms[LO_FILTER]=0;                                                    60
    for (i=LO_FILTER; i<HI_FILTER+1; i++) {
        multbuff(input_signal, filter[i], tempbuffer);
        inv_fft(tempbuffer, threshold_out, FFT_SIZE);
        for (j=0; j<FILTER_SIZE-1; j++) {
          threshold_out[j] += overlap[i][j];
          overlap[i][j] = threshold_out[j+CHUNK_SIZE];
        }
        for (j=0; j<CHUNK_SIZE; j++) {
          rms[i] += threshold_out[j] * threshold_out[j];
```

```
      }                                                                 70
      rms[i] = sqrt(rms[i] / CHUNK_SIZE);
  }
  if (rms[HI_FILTER] / rms[LO_FILTER] > ratio)
      process = ON;
  else process = OFF;
} else process = ON;

if (!signal_on) {
  for (i=0; i<CHUNK_SIZE; i++) {
      data_in[i] = 0;                                                   80
  }
}
if (process == ON) {
  for (i=0; i<NUM_FILTERS;i++) {
      /* multiply input_signal buffer with each filter buffer,
         store in tempbuffer */
      multbuff(input_signal, filter[i], tempbuffer);
      inv_fft(tempbuffer, data_out, FFT_SIZE);

      /* data_out[0] thru data_out[CHUNK_SIZE-1] are "good points";       90
         data_out[CHUNK_SIZE] thru data_out[FFT_SIZE] are "aliased points" */

      /* add old overlap information, save new info for next iteration */
      for (j=0; j<FILTER_SIZE-1; j++) {
        data_out[j] += overlap[i][j];
        noise_out[i][j] += overlap[i+NUM_FILTERS][j];
        overlap[i][j] = data_out[j+CHUNK_SIZE];
        overlap[i+NUM_FILTERS][j] = noise_out[i][j+CHUNK_SIZE];
      }
                                                                        100
      /* calculate RMS value from filter */
      rms[i]=0;
      for (j=0; j<CHUNK_SIZE; j++) {
        rms[i] += data_out[j] * data_out[j];
      }
```

```
        rms[i] = sqrt(rms[i] / CHUNK_SIZE);

        /* multiply noise signal based on rms value and add to data_in */
        if (noise_on[i]) {
          for (j=0; j<CHUNK_SIZE; j++) {                                      110
            data_in[j] += noise_out[i][j] * rms[i] / rms_scale_value[i];
          }
        }
      }
    }
    /* lp filter data_in, if lpf_on flag is set */
    if (lpf_on) {
      real_fft(data_in, tempbuffer, input_signal, FFT_SIZE);
      multbuff(input_signal, filter[LPF_FILTER], tempbuffer);
      inv_fft(tempbuffer, data_in, FFT_SIZE);                                 120
      for (j=0; j<FILTER_SIZE-1; j++) {
          data_in[j] += overlap[LPF_FILTER][j];
          overlap[LPF_FILTER][j] = data_in[j+CHUNK_SIZE];
      }
    }
    errors += put_output(CHUNK_SIZE,0,data_in);
  }
}

/* -----INTERRUPT SERVICE ROUTINES----- */                                   130

void interrupt_get_data(void) {
  if (index < FILTER_SIZE) {
    data_in[index] = get_host_float();
    index++;
    put_host_float(index);
  } else {
    get_host_float();
    put_host_float(-1.0);
  }                                                                           140
}
```

```
void interrupt_rms_check(void) {
  /* check rms values */
  int i;
  for (i=0; i<NUM_FILTERS; i++) {
    put_host_float(rms[i]);
    put_host_float(rms_scale_value[i]);
    put_host_float(noise_on[i]);
  }                                                              150
  put_host_float(ratio);
}

void interrupt_errors_check(void) {
  /* check total errors */
  put_host_float(errors);
}

void interrupt_toggles(void) {
  float which;                                                   160
  which=get_host_float();
  if (which == SIGNAL_TOGGLE) {
    signal_on = !signal_on;
  }
  else if (which == RATIO_TOGGLE) {
    ratio_on = !ratio_on;
  }
  else if (which == LPF_TOGGLE) {
    lpf_on = !lpf_on;
  }                                                              170
}

void interrupt_toggle_noise(void) {
  int which;
  which=(int) get_host_float();
  noise_on[which] = !noise_on[which];
}
```

```
void interrupt_set_value(void) {
  int which;                                                        180
  float value;
  which=(int) get_host_float();
  value=get_host_float();
  if (which == NUM_FILTERS) ratio = value;
  else rms_scale_value[which] = value;
}

/* ------NOISE GENERATION FUNCTIONS------ */

void GenerateNoiseSignal(void) {                                    190
  int i;
  for (i=0; i<CHUNK_SIZE; i++) {
    noise[i] = gasdev();
  }
  for (i=CHUNK_SIZE; i<FFT_SIZE; i++) {
    noise[i] = 0;
  }
}

/* A "quick and dirty" random generator, from _Numerical_Recipes_in_C_ */    200
float ran1(void) {
  myidum = 1664525L*myidum + 1013904223L;
  itemp = jflone | (jflmsk & myidum);
  rand = (*(float *)&itemp)-1.0;
  return(rand);
}

/* returns random deviate with normal (Gaussian) distribution using
   Box-Muller transformation method, also taken from
   _Numerical_Recipes_in_C_, using random generator above */        210
float gasdev(void)
{
  float ran1(void);
```

```
  static int iset=0;
  static float gset;
  float fac, rsq, v1, v2;

  if (iset == 0) {
    v1=2.0*ran1()-1.0;
    v2=2.0*ran1()-1.0;                                                  220
    rsq=v1*v1+v2*v2;
    while (rsq >= 1.0 || rsq == 0.0) {
      v1=2.0*ran1()-1.0;
      v2=2.0*ran1()-1.0;
      rsq=v1*v1+v2*v2;
    }
    fac=sqrt(-2.0*log(rsq)/rsq);
    gset=v1*fac;
    iset=1;
    return v2*fac;                                                      230
  } else {
    iset=0;
    return gset;
  }
}

/* ------ADDITIONAL USEFUL FUNCTIONS------ */

/* This do-nothing "check" function is necessary to prevent the
   optimizer from misrepresenting the while() loop after compiling. */   240
int check(void) {
  if (index < FILTER_SIZE)
    return(0);
  else {
    return(1);
  }
}

/* Performs element by element multiplication of two complex buffers */
```

```
void multbuff(complex first[], complex second[], complex target[])                250
{
  int k;
  for (k=0; k<FFT_SIZE; k++) {
    put_complex_real(&(target[k]),
                     get_complex_real(&(first[k])) *
                     get_complex_real(&(second[k])) −
                     get_complex_imag(&(first[k])) *
                     get_complex_imag(&(second[k])));
    put_complex_imag(&(target[k]),
                     get_complex_real(&(first[k])) *                              260
                     get_complex_imag(&(second[k])) +
                     get_complex_imag(&(first[k])) *
                     get_complex_real(&(second[k])));
  }
}
```

Bibliography

[1] Louis D. Braida, Nathaniel I. Durlach, Richard P. Lippmann, Bruce L. Hicks, William M. Rabinowitz, and Charlotte M. Reed. Hearing aids–a review of past research on linear amplification, amplitude compression, and frequency lowering. *American Speech-Language-Hearing Association Monographs Number 19*, 1979.

[2] Ariel Corporation. *User's Manual for the DSP-96 DSP96002 Floating-Point Attached Processor Board with Dual-Channel Analog I/O*, 1993.

[3] B.L. Hicks, L.D. Braida, and N.I. Durlach. Pitch invariant frequency lowering with nonuniform spectral compression. In *Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 121–124, New York, March 1981. IEEE.

[4] R.P. Lippmann. Perception of frequency lowered consonants. *J. Acoust. Soc. Am.*, 67:S78, 1980.

[5] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing.* Prentice Hall, 1989.

[6] Miles P. Posen. Intelligibility of frequency-lowered speech produced by a channel vocoder. Master's project, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1984.

[7] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*, chapter 7.1-7.2. Cambridge University Press, 1992.

[8] Charlotte M. Reed, Kenneth I. Schultz, Louis D. Braida, and Nathaniel I. Durlach. Discrimination and identification of frequency-lowered speech in listeners with high-frequency hearing impairment. *J. Acoust. Soc. Am.*, 78(6):2139–2141, December 1985.

[9] C. Britton Rorabaugh. *Digital Filter Designer's Handbook: Featuring C Routines.* McGraw-Hill, 1993.

[10] Matthew G. Sexton. C language development on the dsp-96 dsp96002 floating-point attached processor board, June 1995.