FSK Modem Modules For SDR Using Different Linux API's

Patrick Khaile

Thesis presented in partial fulfilment of the requirements
for a MScEng(Sci) in Electronics Engineering at the
University of Stellenbosch.

Supervisor: Prof. J.G.Lourens

December 2004

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature:

Date:

Abstract

The purpose of this thesis is to implement the Frequency Shift Keying (FSK) modem modules for the Software Defined Radio (SDR), using different Linux sound Application Programming Interfaces (API's).
The FSK modulation scheme, uses coherent detection with matched filters.
The modules are implemented using the Linux operating system and the programming language used is `C/C++`. Simulation is performed first and then followed by implementation in real-time, using APIs.
The APIs used are Open Sound System (OSS), Advanced Linux Sound Architecture (ALSA), Jack Audio Connection Kit (JACK) and PortAudio (PA).

In real-time two computers are used, one acting as the modulator and the other as the demodulator. The two sound cards are connected by an audio cable.

Results obtained using OSS, ALSA and JACK are satisfactory.
Recommendations are subsequently made for the suitable API(s). Both ALSA and JACK are the best APIs for the implementation.

Opsomming

Die doel van hierdie tesis is om modules vir 'n frekwensieskuif-sleuteling (FSK) modem vir 'n Sagteware-gedefinieerde Radio (SDR) te implementeer, deur gebruik te maak van verskillende Linux klankargitekture (API's). Die FSK modulasieskema gebruik koherente deteksie met aangepaste filters.

Die modules is geïmplementeer met behulp van die Linux bedryfstelsel en C/C++ is gebruik as programmeringstaal. Simulasies is eers gedoen, gevolg deur 'n reële-tyd implementasie. Die klankargitekture wat gebruik is, is Open Sound System (OSS), Advanced Linux Sound Architecture (ALSA), Jack Audio Connection Kit (JACK) en PortAudio (PA).

Twee rekenaars is gebruik vir die reële-tyd stelsel, waar een as die modulator optree en die ander een as die demodulator. Die twee klankkaarte is verbind deur 'n klankkabel.

Bevredigende resultate is verkry met behulp van OSS, ALSA en JACK. Aanbevelings is gevolglik gemaak vir toepaslike API's. Beide ALSA en JACK is die geskikste API's vir die implementasie.

Acknowledgments

I would like to thank the following people who made a contribution to my thesis:

Prof. J.Lourens, my supervisor, whose guidance and enthusiasm I appreciate.
Mr. L.Schwart, who made me aware of different Linux Sound APIs in early stages of this thesis.
Mr. J.Cronje for technical support.
Department of Communications for financial support.
My family and friends for their courage and support.

Glossary

ADC	:Analog to Digital Converter
ALSA	:Advanced Linux Sound Architecture
API	:Application Programming Interface
ASCII	:American Standard Code for Information Interchange
bps	:bits per second
CPU	:Central Processing Unit
DSP	:Digital Signal Processing
FSK	:Frequency Shifting Key
ISA	:Industry Standard Architecture
JACK	:Jack Audio Connection Kit
MF	:Matched Filter
OSS	:Open Sound System
PA	:Port Audio
Pe	:Probability of error
PCM	:Pulse Code Modulation
PCI	:Peripheral Component Interconnect
R_x	:Receiver
SDR	:Software Defined Radio
SNR	:Signal-to-Noise Ratio
SR	:Sampling Rate
T_x	:Transmitter

Contents

1 INTRODUCTION **1**
- 1.1 Software Defined Radio (SDR) . . . 1
- 1.2 SDR Open System Architecture . . . 2
- 1.3 Thesis Objectives . . . 2
- 1.4 Thesis Overview . . . 2

2 BACKGROUND THEORY **4**
- 2.1 Introduction . . . 4
- 2.2 FSK Modulation . . . 4
- 2.3 Sampling Rate . . . 5
- 2.4 Sample Resolution . . . 6
- 2.5 Quantization Noise . . . 6
- 2.6 Performance Analysis of Coherent FSK . . . 7
- 2.7 Summary . . . 8

3 FSK MODULES DESIGN **9**
- 3.1 Introduction . . . 9
- 3.2 Specifications . . . 9
  - 3.2.1 Buffer Sizes . . . 9
  - 3.2.2 Samples Per Bit . . . 9
  - 3.2.3 Sampling Rate . . . 10
  - 3.2.4 Bit Duration . . . 10
  - 3.2.5 Frequencies . . . 10
  - 3.2.6 Audio Formats . . . 10
- 3.3 Modulator Processes . . . 10
- 3.4 Demodulator Processes . . . 11
- 3.5 Conclusion . . . 11
  - 3.5.1 Summarised Specifications . . . 11

4 SIMULATION **12**
- 4.1 Introduction . . . 12
- 4.2 FSK Modulator . . . 12
  - 4.2.1 Input of Characters . . . 13
  - 4.2.2 Converting Characters to Bits . . . 14
  - 4.2.3 FSK Modulation . . . 15
  - 4.2.4 Writing To File (Transmission) . . . 16

4.3 FSK Demodulator . . . 17
4.3.1 Initialising variables . . . 17
4.3.2 Matched Filters . . . 18
4.3.3 Buffer lengths . . . 19
4.3.4 Recording (Capturing) . . . 19
4.3.5 Convolution Process . . . 20
4.3.6 Bit Synchronisation . . . 22
4.3.7 Decision Device . . . 24
4.3.8 Shifting of samples . . . 25
4.3.9 Output Display . . . 27
4.4 Conclusion . . . 27

5 IMPLEMENTATION USING LINUX SOUND APIs 28
5.1 Introduction . . . 28
5.1.1 Native APIs . . . 28
5.1.2 Advantages of ALSA over OSS . . . 28
5.1.3 Low-Latency Higher-Level APIs . . . 29
5.2 Implementation Using OSS . . . 29
5.2.1 OSS Overview and Introduction . . . 29
5.2.2 Opening the Device . . . 29
5.2.3 Modulator Processes . . . 33
5.2.4 Demodulator Processes . . . 36
5.2.5 Closing the Device . . . 39
5.2.6 OSS Summary . . . 39
5.3 Implementation Using ALSA . . . 40
5.3.1 ALSA Overview and Introduction . . . 40
5.3.2 Interrupt-Driven Modulator . . . 40
5.3.3 ALSA Demodulator . . . 44
5.3.4 ALSA Summary . . . 46
5.4 Implementation Using JACK . . . 47
5.4.1 Jack Overview and Introduction . . . 47
5.4.2 JACK Demodulator . . . 48
5.4.3 Jack Callback Functions . . . 48
5.4.4 Demodulation Process . . . 52
5.4.5 JACK summary . . . 52
5.5 Implementation using PortAudio . . . 53
5.5.1 PA Overview and Introduction . . . 53
5.5.2 PA Modulator . . . 53
5.5.3 PA Demodulator . . . 55
5.5.4 PA Summary . . . 56

6 RESULTS 57
6.1 Introduction . . . 57
6.2 Simulated FSK . . . 57
6.2.1 Modulation results . . . 57
6.2.2 Demodulation results . . . 58

6.3 Performance Analysis 60
6.4 Real-Time FSK 62
6.4.1 T_x/R_x Sampling Rate Mismatch Range 62
6.4.2 Tests Across APIs 63
6.4.3 Profiling Demodulation Process 64
6.5 Summary 64

7 DISCUSSIONS AND CONCLUSIONS 66
7.1 Implementation Options 66
7.2 Concluding Remarks 66
7.3 Outstanding Tasks 67
7.4 Uses of Modules Outside SDR Architecture 67
7.5 Final Conclusion 67

A THESIS TOOLS 68
A.1 Software Used 68
A.2 Hardware Used 69

B DISCRETE CONVOLUTION 70

C SYSTEM CALLS USED BY OSS 72
C.1 The ***open()*** System Call 72
C.2 The ***ioctl()*** System Call 72
C.3 The ***read()*** System Call 73
C.4 The ***write()*** System Call 73
C.5 The ***close()*** System Call 74

D SOUND CARD 75
D.1 Sound Card Devices 75
D.2 Duplexing 76
D.3 Sound Card Jitter 76
D.4 Linux Supported Sound Cards 77

E JACK LAYOUT 78

List of Figures

1.1 SDR Open System Architecture 2

2.1 An idealized FSK waveform of character M 5
2.2 Matched-filter detection of FSK waveforms 5
2.3 Under Sampling 6
2.4 Theoretical probability of error for coherent FSK. 8

4.1 Modulator Flowchart 13
4.2 FSK Modulation Process 15
4.3 Demodulation Process 17
4.4 Demodulator buffers 19
4.5 Buffer Underruns 22
4.6 Buffer Overruns 22
4.7 Shifting of samples 26

5.1 OSS modulator flowchart 33
5.2 Start/End bits 34
5.3 OSS demodulator flowchart 36
5.4 JACK implementation steps 48

6.1 Modulated signal 58
6.2 Low Frequency MF Output 58
6.3 High Frequency MF Output 59
6.4 Sum Of MF Outputs 59
6.5 Difference Of MF Outputs 60
6.6 Performance Analysis Test 62
6.7 OSS Demodulator Profile Results 65

A.1 Modulator figure 69

C.1 read() system call 73
C.2 write() system call 73

D.1 Sound Card Block Diagram 76

E.1 JACK Diagram 79

List of Tables

3.1 Buffer sizes 9
3.2 Audio Formats 10

5.1 OSS Audio Formats 30

6.1 Performance Analysis Table 61
6.2 SR Mismatch Ranges 62
6.3 Tests Across APIs 63

C.1 Flags For The open() Call 72

Chapter 1

INTRODUCTION

1.1 Software Defined Radio (SDR)

SDR is an emerging technology intended for the conception and design of flexible radio systems, which are "multi-service", "multi-standard", "multi-band", re-configurable and re-programmable by software.
With it the following are possible:

1. It has the ability to receive and transmit using various modulation methods and a common set of hardware.

2. Its functionality can be completely redefined by downloading and running new software at will.

3. It minimises the costs involved in the building of hardware as some tasks can be performed using software, resulting in simplification of radio architecture and improved performance.

One of the first software radios was a military project named SpeakEasy. The primary goal of the SpeakEasy project is to utilize programmable processing to emulate more than 10 existing military radios, operating in frequency bands between 2 and 200MHz. Another design goal was to be able to easily incorporate new coding and modulation standards that can be used in the future, so that military communications can keep pace with advances in coding and modulation techniques [19].

SDR is currently being used to implement radio modem technologies.
In the long run, it is expected to become the dominant technology in radio communications.

1.2 SDR Open System Architecture

Figure 1.1 illustrates the hardware of a software-defined radio [16]. It typically consists of a superheterodyne radio frequency (RF) front end which converts RF signals to and from analog intermediate frequency (IF) signals, and analog to digital converter and digital to analog converters which are used to convert a digitised IF signal to and from analog form.

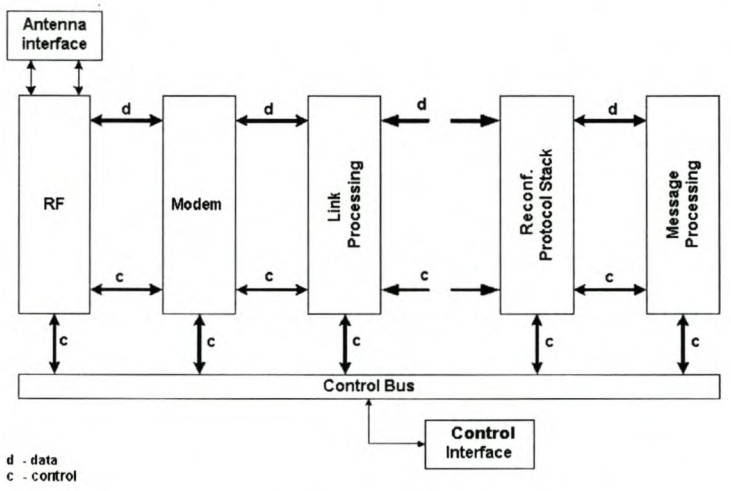

Figure 1.1: SDR Open System Architecture

1.3 Thesis Objectives

The objectives of this are:

- to implement the FSK modem modules for the SDR, using a sound card and different Linux sound drivers.
- to choose the API that works best.

1.4 Thesis Overview

The structure of the thesis is as follows:

Chapter One: Introduction
This chapter introduces Software Defined Radio and objectives of this thesis.

Chapter Two: Background Theory
This chapter explains the background theory required for the implementation of the FSK modem modules.

Chapter Three: FSK MODULES DESIGN

This chapter outlines the design that is to be followed in the implementation stages of the modules.

Chapter Four: Simulation
The simulation of both the modulator and demodulator is implemented in this chapter.

Chapter Five: Implementation Using Linux Sound APIs
In this chapter, modules are implemented using OSS, ALSA, JACK and PA.

Chapter Six: Results
Results of simulation and real-time implementation are interpreted in this chapter.

Chapter Seven: Discussions and Conclusions
Recommendations are made for the best API(s).

Appendices

Included CD-ROM
The CD contains the source code of Simulation and all four APIs used.

Chapter 2

BACKGROUND THEORY

2.1 Introduction

This section gives some background theory on FSK modulation using matched filters. The theory covers both modulation and demodulation processes. The expected performance for the chosen modulation scheme is also worked out. Factors affecting sound quality are also discussed.

2.2 FSK Modulation

FSK modulation is the process corresponding to switching or keying the frequency of the carrier signal between two frequencies that correspond to binary symbols 0 and 1.

Let $s_1(t)$ and $s_2(t)$ represent the two signals used to represent the binary symbols 0 and 1, respectively [1].

$$s_1(t) = \begin{cases} A\cos 2\pi f_1 t & 0 < t \leq T, \\ 0 & \text{elsewhere,} \end{cases} \tag{2.1}$$

$$s_2(t) = \begin{cases} A\cos 2\pi f_2 t & 0 < t \leq T, \\ 0 & \text{elsewhere.} \end{cases} \tag{2.2}$$

For example, Figure 2.1 shows the modulated waveform of the character "M". Intervals of high frequency denotes "1" bits and those of low frequency denotes "0" bits.

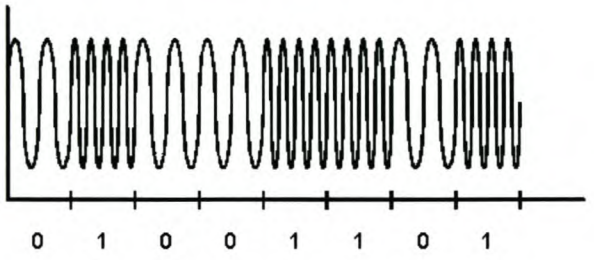

Figure 2.1: An idealized FSK waveform of character M

Coherent detection is used in the demodulating process, In this method the receiver has a perfect knowledge of the two signals $s_1(t)$ and $s_2(t)$ transmitted by the modulator system. The received binary FSK signal is passed through two different matched filters, each one containing the exact shape of one of the signals transmitted.
The matched filter (MF) receiver is shown in Fig. 2.2. The outputs of the matched filters are added and subtracted for the purpose of bit synchronisation and decision making, respectively.
The bit synchronisation process determines peaks from the signal created by the sum of the output of the two matched filters. The peaks positions are used in the decision threshold block.
The decision threshold block uses the signal created from the subtraction of the outputs. It reads the signal at peak positions and decides on whether a bit is a one or zero by using a threshold of zero.

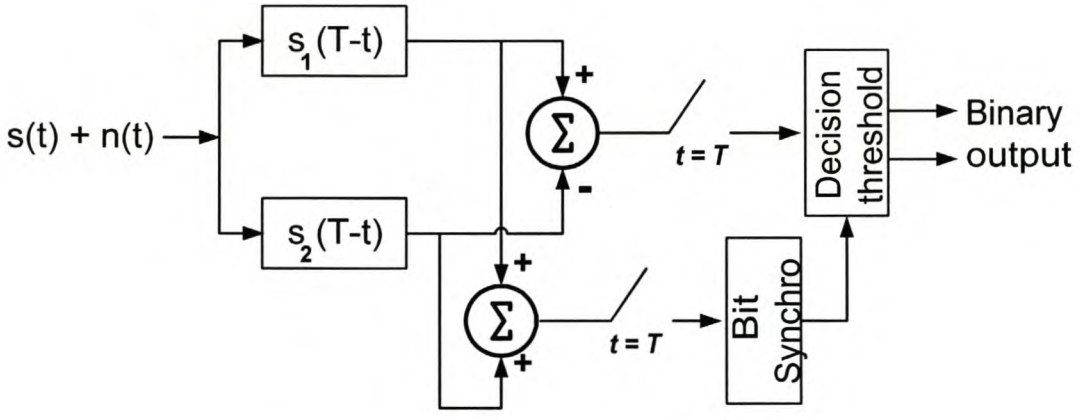

Figure 2.2: Matched-filter detection of FSK waveforms

2.3 Sampling Rate

Sampling Theorem: If the analog input is sampled instantaneously at regular intervals at a rate that is at least twice the highest analog frequency then the samples contain *all* of

the information of the analog signal [5].

$$f_s \geq 2f_A(max) \tag{2.3}$$

Figure 2.3 shows the results of under sampling. Joining the dots gives a different picture about the original waveform. The observed frequency is also different.

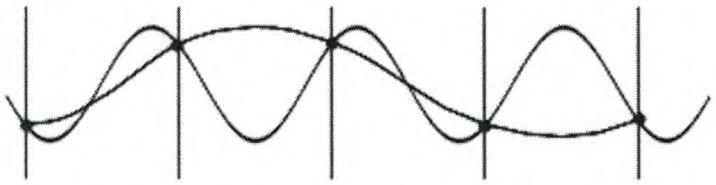

Figure 2.3: Under Sampling

2.4 Sample Resolution

This refers to how many different values the samples can take on. The higher the sample resolution, the more accurate the representation of the level of each sample, but again, the more memory is required to store each sample.
Digital audio is normally found in one of two resolutions:

1. 8-Bit: 8-bit resolution was used in the earliest sound cards, and is used for some lower-quality recording formats as well. Here, each sample can take one of 256 different values ($2^8 = 256$).

2. 16-Bit: This is the standard for compact disc audio and newer sound cards. Each sample can take one of 65,536 different values. The resolution is considered to accurately represent music audio.

2.5 Quantization Noise

Another important factor in any source encoding scheme is the amount of noise or distortion introduced, the difference between the sample value and the original waveform received [1].
The signal-to-quantization-noise ratio increases by 6dB for every additional bit used in a binary system. There is a difference of about 48dB between 8-bit and 16-bit resolution.

2.6 Performance Analysis of Coherent FSK

The average energy per binary digit for analog FSK is [1]

$$E = \frac{A^2 T}{2} \tag{2.4}$$

where A and T are the amplitude and bit duration, respectively.

In digital demodulation, sampling of the analog signal takes place. The average energy per bit is given by

$$E = \frac{A^2 N}{2} \tag{2.5}$$

where N is the number of samples per bit.

If one signaling frequency is present in the absence of noise, it is assumed that the one MF output is zero and the other output is at E. Conversely, if the second signaling frequency is present, the first MF output is zero and as a result of the subtraction the net output is at -E [1].

The sum of the noise output variance from the two matched filters is given by,

$$\sigma_{out}^2 = \sigma_{in}^2 A^2 N \tag{2.6}$$

For Gaussian-distributed noise and equiprobable ones and zeros, the probability of error is given by [1],

$$P_e = \int_0^\infty \frac{1}{\sqrt{2\pi\sigma_{in}^2 E}} e^{-\frac{(y+E)^2}{2\sigma_{in}^2 E}} dy \tag{2.7}$$

and this gives the probability of error

$$P_e = \frac{1}{\sqrt{2\pi}} \int_{\frac{A\sqrt{N}}{2\sigma_{in}}}^\infty e^{-\frac{z^2}{2}} dz \tag{2.8}$$

and for Gaussian-distributed noise [1],

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{z^2}{2}} dz \tag{2.9}$$

$$= \frac{1}{2}\left[1 - erf\left(\frac{x}{\sqrt{2}}\right)\right] \tag{2.10}$$

From equations 2.8 and 2.9,

$$P_e = \frac{1}{2}\left[1 - erf\left(\frac{A\sqrt{N}}{2\sqrt{2}\sigma_{in}}\right)\right] \tag{2.11}$$

Figure 2.4 is the expected probability of error curve for the coherent FSK, using equation 2.11.

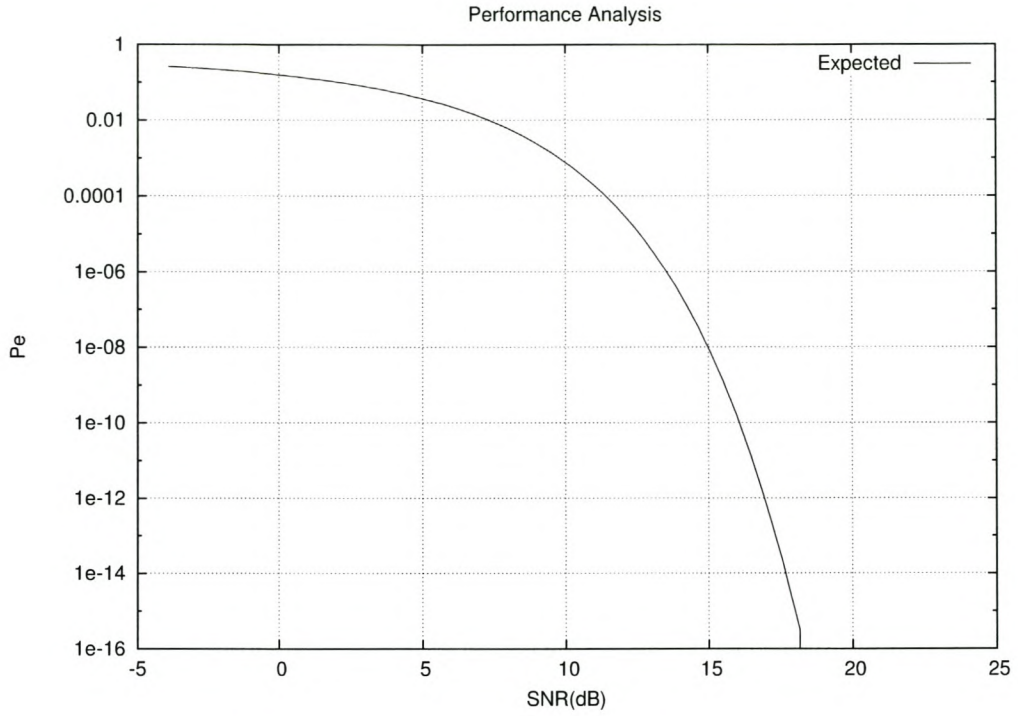

Figure 2.4: Theoretical probability of error for coherent FSK.

2.7 Summary

This chapter discusses the background theory that is used in the implementation of the modules. The FSK modulation scheme using coherent detection is discussed. Factors affecting the quality of sound are stated. Equation 2.11 is used to calculate the expected probability of error for the implemented modulation scheme.

Chapter 3

FSK MODULES DESIGN

3.1 Introduction

This chapter outlines the proposed specifications for modules design.

3.2 Specifications

The following is a list of proposed specifications for the module.

3.2.1 Buffer Sizes

Buffer sizes should be in integer powers of 2, that is 2^n, where n is an integer. This works best with the buffering used internally by the sound drivers.
Buffer sizes differs from one API to another. The OSS API prefers buffer sizes between 1024 and 4096 for normal use [7], while ALSA works best with much bigger buffers.
Table 3.1 shows kernel buffer sizes that will be used.

API	Periods	Period size	Buffer Size
OSS	1	1024	1024
ALSA	8	1024	8192
JACK	8	1024	8192
PA	1	1024	1024

Table 3.1: Buffer sizes

3.2.2 Samples Per Bit

A full buffer contains 8 transmitted bits. A buffer of 1024 (128×8) samples will have 128 samples per bit.

3.2.3 Sampling Rate

The chosen sampling rate is 44kHz. This is to allow use of full sound card bandwidth later.

3.2.4 Bit Duration

With a sampling rate of 44kHz and samples per bit equal to 128, bit duration will be 2.91 ms. This gives a baud rate of 343 bits per second.

3.2.5 Frequencies

Low and high frequencies chosen are 1300Hz and 2100Hz, respectively. These are frequencies used in 1200 or 2400 baud modems.

3.2.6 Audio Formats

Table 3.2 shows audio formats to be used for different APIs.

API	Audio Format
OSS	8-bit unsigned
ALSA	8-bit unsigned
JACK	32-bit floating point
PA	8-bit unsigned

Table 3.2: Audio Formats

3.3 Modulator Processes

Methods that are used in the modulator are the following:

inputMessage() :-writes entered message from keyboard, to an array.
init() :-initialises variables and memory allocation of all arrays.
modulator() :-performs modulation process.
playBack() :-writes modulated signal to kernel buffer.

3.4 Demodulator Processes

Methods that are used in the demodulator are the following:

init()	:-initialises variables and memory allocation of all arrays.
matchedFilters()	:-prepares matched filters.
recording()	:-captures data from the kernel buffer.
convolution()	:-performs discrete convolution.
bitSynchro()	:-synchronises bits.
shiftingSamples()	:-shifts samples left out after synchronisation to the front of the buffer.
decisionDevice()	:-makes decision on bits.
displayMessage()	:-displays message on the screen.

3.5 Conclusion

All the necessary specifications are given in this section. Processes that will take place inside the modulator and demodulator modules are stated and Audio formats to be used in different APIs are also stated.

3.5.1 Summarised Specifications

Low Frequency Signal	:1300 Hz
High Frequency Signal	:2100 Hz
Bit Duration	:2.91 ms
Baud Rate	:343 bps
Samples Per Bit	:128
Sampling Rate	:44 kHz
Duplexing	:half-duplex

The simulation will use a buffer size of 1024, that contains about 8 bits.

Chapter 4

SIMULATION

4.1 Introduction

In this chapter simulation of both the modulator and demodulator takes place. The language used in simulation is C++. Theory discussed in Chapter 2 is used. The modulator write an FSK modulated signal to file. The demodulator read the FSK modulated signal from a file and demodulate it using matched filters, bit synchronisation and threshold for deciding. The output of the demodulator should be the original message that was typed as the input of the modulator.
The aim of simulation is to get the software working properly on files before implementation with sound cards in real-time takes place.

4.2 FSK Modulator

The modulator consists of four stages:

- Input of Characters
- Converting Characters to Bits
- FSK Modulation Process
- Writing Analogue Signal to File

Figure 4.1 shows a flowchart of the modulator.

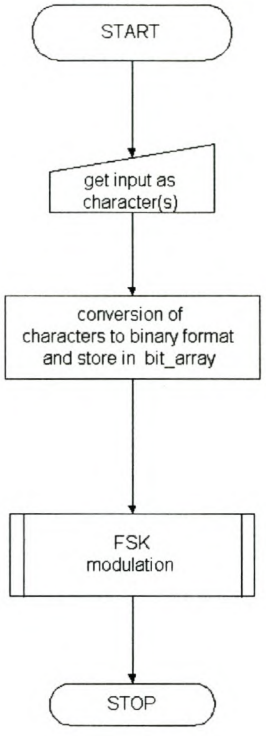

Figure 4.1: Modulator Flowchart

4.2.1 Input of Characters

The simulation uses characters that are entered through the keyboard. The following C++ code fragment shows how characters are entered:

```
int ch;
cout<<"Enter message: ";
numOfChars = 0;
//the ASCII code for newLine is 10
while((ch=cin.get()) != 10)
{
   message.push_back((char)ch);
   numOfChars++;   //character counter
}
```

Characters are entered, with enter key "new line" marking the end of the message. The ASCII code for the enter key is 10. As the length of the message is not known, a vector is used instead of an array to store characters.

4.2.2 Converting Characters to Bits

Each number stored in the message vector gets converted to binary by repeatedly dividing the number by 2 until the answer is 0 and by retaining the remainder each time division takes place. These remainders, in reverse order, give the binary value of the character.

The code is as follows:

```
for(int i = 0; i<numOfChars;i++)
{
   tempoInt.push_back((int)message[i]);
}
//Converting integers to bits
int reverseBitsArray[8], number;
int charCounter,bitCounter,remainder;
//Loop as long as charCounter < numOfChars
charCounter = 0;
do
{
   for(int i =0;i<numOfChars;i++)
   {
      number = tempoInt[i];
      bitCounter = 0; //reset bit counter
      do
      {
         remainder = number%2;
         number = number/2;
         reverseBitsArray[bitCounter] = remainder;
         bitCounter++;
      }while(bitCounter<8);
      //Reversing the order
      for(int i=7; i >= 0;i--)
      {
         bitArray.push_back(reverseBitsArray[i]);
      }
   }
   charCounter++;
}while(charCounter<numOfChars);
```

All characters at this stage have been converted into binary format and all the bits are stored in the "bitArray" vector.

4.2.3 FSK Modulation

Figure 4.2 shows a flowchart for the modulation process.

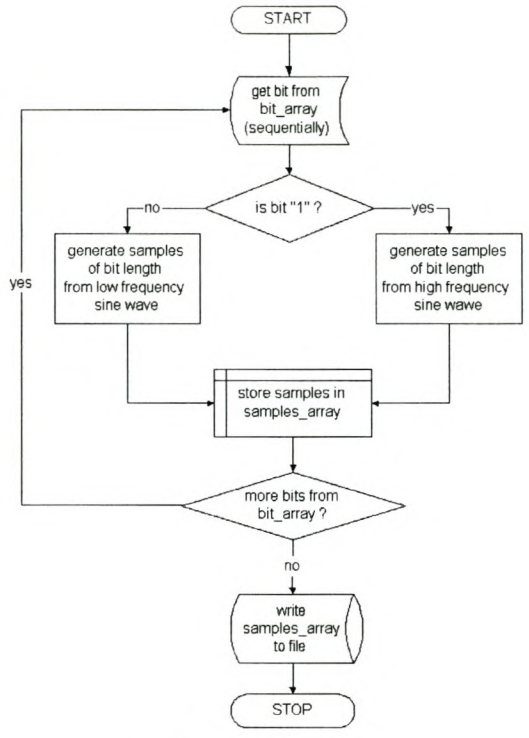

Figure 4.2: FSK Modulation Process

This code fragment shows how bits are converted to an analog signal. Zero and one bits are converted to low and high frequency, respectively .

```
int temp;
int zeroSamples = 0;
int oneSamples = 0;
bufferSize = numOfChars * samplesPerBit;
////////////////////////////////////
//unsigned int allSamples;
//allSamples = 8 * numOfChars * samplesPerBit;
//float tempBuffer1[allSamples];
////////////////////////////////////
unsigned int z = 0;
for (int i = 0; i < totalBits; i++)
{
   if(bitArray[i] == 0)
   {
      for(int t=0; t < samplesPerBit; t++)
      {
         float x = t * lowFreq/samplingRate * 2 * M_PI;
         temp = (int)(100*cos(x));
         tempBuffer.push_back(temp);
         zeroSamples++;
      }
   }
   else
   {
```

```
        for(int t=0; t < samplesPerBit; t++)
        {
           float x = t * highFreq/samplingRate * 2 * M_PI;
           temp = (int)(100*cos(x));
           tempBuffer.push_back(temp);
           oneSamples++;
        }
     }
};
```

4.2.4 Writing To File (Transmission)

All the samples contained in the "tempBuffer" vector are in a waveform and are now written to a file.

```
FILE * fin;
if((fin = fopen("/temp/fskfile.dat", "wb")) == NULL)
{
   cout << "Cannot open file" << endl;
   exit(1);
}
unsigned long numSamples;
numSamples = totalSize;
if(fwrite(tempBuffer, sizeof(int), numSamples, fin) != numSamples)
{
   cout << "Error writing to the file" << endl;
   exit(1);
}
fclose(fin);
```

4.3 FSK Demodulator

The modulated signal stored in a file (transmitted by the modulator) is now going to be demodulated.
The demodulator entails:

1. matched filtering
2. bit synchronisation
3. threshold decisions

Figure 4.3 shows a flowchart of the demodulator.

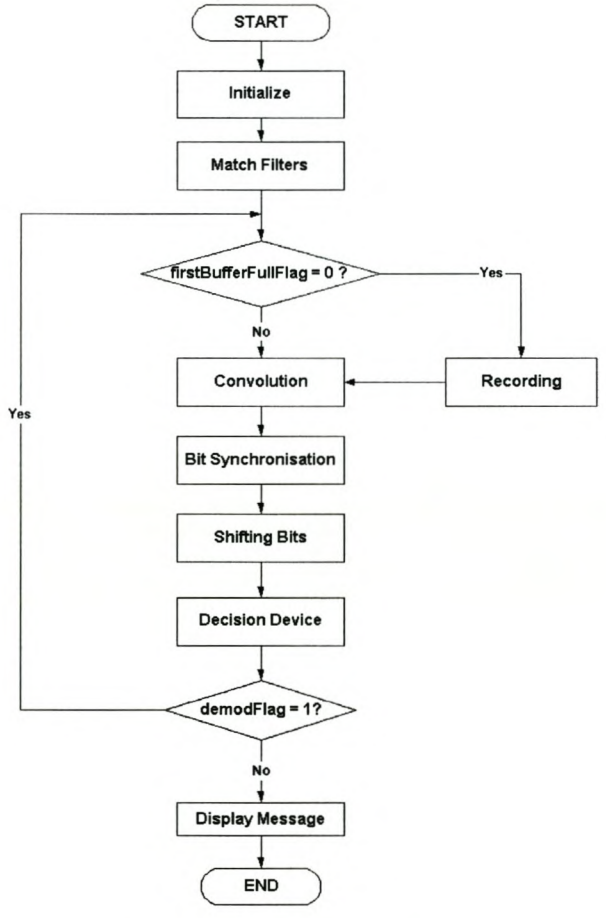

Figure 4.3: Demodulation Process

4.3.1 Initialising variables

Before the demodulation process starts, variables are initialised and memory is allocated to application arrays. The file containing the modulated signal is opened, its size determined, and then closed. This is done for the purpose of allocating memory to hold the

contents of the file. The code fragments below shows the retrieval of data from the file.

```
FILE *ptr;
/*****************************************
* Opening a file to count number of samples *
* and then closing it                       *
*****************************************/
ptr = fopen("/temp/fskfile.dat","rb");
while ((c = fgetc(ptr)) != EOF) charRead++;
fclose(ptr);
```

The audioStream array acts as audio from the outside world. Retrieved data from the file is read into the audioStream array.

```
//Dividing by 4 to get num of samples in type int
fileSize = charRead/4;
dataSize = charRead/4;
flipIndex = dataSize-1;
numOfSamples = dataSize;
audioStream = new int[fileSize];
//Reopening file again,this time for reading in binary
FILE * fp;
if((fp = fopen("/temp/fskfile.dat","rb")) == NULL)
{
   cout<<endl"Data file does not exist;
}
if(fread(audioStream, sizeof(int), fileSize, fp)!=fileSize)
{
   cout<<endl"Error reading from file;
   exit(1);
}
fclose(fp);
```

All the data has now been retrieved from the file and copied into the audioStream array. The audioStream now contains the analog signal that was transmitted by the modulator.

4.3.2 Matched Filters

Matched filters should be tailored to the two signal waveforms of the modulator, to achieve the maximum signal-to-noise ratio.
The filters use the same formulae as those used in the modulation process, one for low and one for high frequency signal.

Low Frequency Matched Filter

```
//low frequency matched filter
for(int i=0; i < samplesPerBit;i++)
{
   float x = i *(lowFreq/samplingRate * 2 * M_PI);
   temp =(int)(Amplitude*cos(x));
   //Flipping the match filter by using push_front
   lowFreqMF.push_front(temp);
}
```

High Frequency Matched Filter

```
//high frequency matched filter\\
for(int i=0; i < samplesPerBit; i++)
{
   float x = i * (highFreq/samplingRate * 2 * M_PI);
   temp =(int)(Amplitude*cos(x));
   //Flipping the matched filter by using push_front
   highFreqMF.push_front(temp);
}
```

The two matched filters contain the phases and frequencies of the two original signals modulated.

4.3.3 Buffer lengths

The internal kernel buffers use sizes of buffers that are in integer powers of two, as explained in Chapter 3. The kernel buffer (kernelBuffer[]) to be used must have sizes of this magnitude.
The application buffer (applicationBuffer[]) must always be twice the size of the kernel buffer in order to avoid losing bits. This is explained clearly in Section 4.3.6. Figure 4.4 shows how buffer sizes are related.

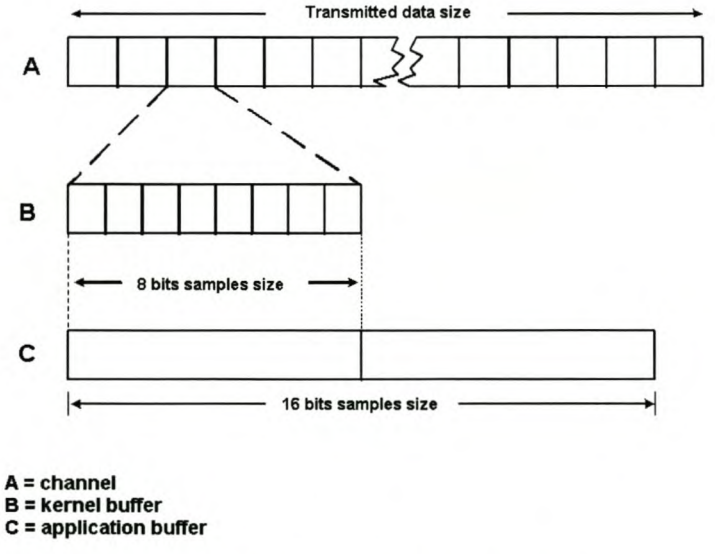

Figure 4.4: Demodulator buffers

4.3.4 Recording (Capturing)

This part of the program emulates the process of capturing analog signals from the outside world, with the use of the sound card. The audio stream is captured sequentially in lengths of eight bit samples size. The captured stream is stored in the kernel buffer.

```
for(i=0;i<eightBitSamples;i++)
{
   kernelBuffer[i] = audioStream[sampleCounter++];
}
```

The kernel buffer is then accessed by the application and contents of the kernel buffer are *read* to the application buffer.

```
for(i=0;i<eightBitSamples;i++)
{
   applicationBuffer[tailIndex++] = kernelBuffer[i];
}
//Monitoring the buffer
if(tailIndex >maxTailIndex)  maxTailIndex = tailIndex;
```

The role played by "tailIndex" is explained in Section 4.3.8. It is used to monitor the position of the last sample in the application buffer.
Samples in the application buffer are now ready to be processed.

4.3.5 Convolution Process

Samples equivalent to eight bits are convolved at a time. The function of the MF_input[] array is to pass samples through the matched filters. The MF_input[] array is firstly filled with "ground valued samples" of one bit length (equal to the size of the matched filter).

```
counter = 0;
for(i = 0;i < samplesPerBit;i++)
{
   MF_input[counter++] = 0; //ground position
}
```

The ground valued samples are then followed by the first eight bits samples, taken from the application buffer.

```
for(i = 0; i < eightBitSamples; i++)
      {
         MF_input[counter++] = applicationBuffer[i];
      }
```

Samples in the MF_input array are passed through the matched filters. The process of discrete convolution is explained in Appendix B. The code below shows the convolution process that takes place inside the matched filters.

```
float output_low, output_high; //MF outputs
counter = 0;
for(int i=samplesPerBit; i<(samplesPerBit+eightBitSamples);i++)
{
   output_high = 0;  //reset sums to zero\\
   output_low = 0;
   for(int j=1;j <= samplesPerBit;j++)
   {
      output_low += MF_input[i-j+1] * lowFreqMF[j];
      output_high += MF_input[i-j+1] * highFreqMF[j];
   }
   MF_output_sum[counter] = (output_low+output_high);
   MF_output_subtract[counter] = (output_high-output_low);
   counter++;
}
```

The variables outputZero and outputOne are outputs of both the low and high frequency MF, respectively.
The sum of the two outputs from the filters are added and stored in the MF_output_sum[]. This array is used in bit synchronisation to determine peak positions.
The MF_output_subtract[] stores the difference of the two outputs from the matched filters. This array is used in deciding whether a bit is a zero or one bit.

4.3.6 Bit Synchronisation

The purpose of synchronisation is to determine the positions, where decisions on bits should be taken.

In ideal cases, bit synchronisation should occur at intervals of 128 samples, since samples per bit are chosen to be 128.
This is not always the case. Synchronised bit positions can occur at intervals that are more or less than 128. This happens when the clock frequencies of the two sound cards slightly differ.

Buffer Underruns ($f_{clock_{TX}} > f_{clock_{RX}}$)
Figure 4.5 illustrates the signal created by the sum of the output of the two matched filters for the situation where synchronisation takes place for every 127 samples or less. The arrow points at a position of the 8^{th} synchronised position, which lies inside the buffer, between position 896 and 1024. If synchronisation occurs for every 127 samples, the 8^{th} position will be at position 1016.

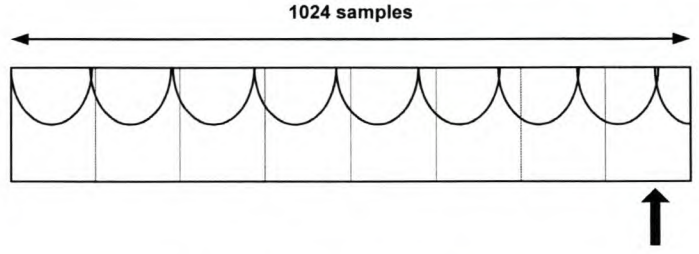

Figure 4.5: Buffer Underruns

Buffer Overruns ($f_{clock_{TX}} < f_{clock_{RX}}$)
Figure 4.6 illustrates a situation where synchronisation takes place after every 129 samples or more. The arrow indicates the eighth bit synchronised position, which is at position 1032, beyond the 1024 samples size. This will result in losing samples beyond 1024.

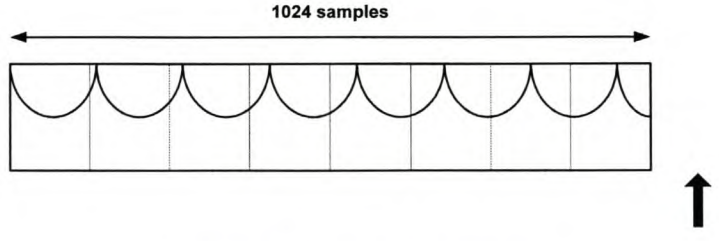

Figure 4.6: Buffer Overruns

From the situation that occurs under buffer overruns, the application can only synchronise seven bits. The 8^{th} bit lies beyond the 1024 buffer size. This will actually force the application to synchronise only seven bits at a time. The rest of the samples after the 7^{th}

bit will be processed in the next cycle. Section 4.3.8 explains how the application handles the rest of the buffer.

Performing Bit Synchronisation

Bit synchronisation is performed by locating peaks in the signal created by the sum of the output of the two matched filters. Signal samples are stored in MF_output_sum[] array. The application starts at the front of the MF_output_sum[] array and then takes the sum of samples as it moves in step size. Seven steps are performed. The sum of samples is stored. For the next cycle, the application starts at the second sample of the array and perform the operation again in step sizes. The new sum is compared with the previous one. The larger one is taken as the new sum. The process will go on until the application determines the right step size of locating peak positions. The step size will be used in the decision logic method.
The code fragment below shows how bit synchronisation is performed.

```
\\newStartPos = 0;newStep = 0;newSum = 0;
for(int step = (samplesPerBit - R);step<(samplesPerBit + R);step++)
{
   for(int startPos = 0; startPos < (samplesPerBit + R); startPos++)
   {
      float sum = 0;
      for(int bitNum = 0; bitNum < 7; bitNum++)
      {
         sum = sum + MF_output_sum[startPos + bitNum * step];
      }
      if(sum > newSum)
      {
         newSum = sum;
         newStartPos = startPos;
         newStep = step;
      }
   }
}
```

Knowing the starting position of synchronisation and the step to be taken for the next synchronised bit, positions are stored in synchroPosition array.

```
for(i = 0;i < 7;i++) //seven bits to be synchronised\\
{
   synchroPosition[i] = newStartPos;
   newStartPos += newStep;
}
```

There are cases where peaks are detected within the first bit length. Such cases occur when a lot of Gaussian noise is added to the modulated signal in the analysis of bit error rate. These peaks start to show up at around a SNR of 12dB. To correct the situation, bit shifting has to be performed.
The code fragment below controls the shifting of bits.

```
if(indexPosition[0] < R)
{
   for(i = 0;i < 7;i++)
   {
      indexPosition[i] += (newStep);
   }
```

```
}
if((indexPosition[0] > R)&& (indexPosition[0]<(samplesPerBit-2*R)))
{
   for(i = 0;i < 7;i++)
   {
      indexPosition[i] = (i+1)*newStep;
   }
}
```

The next step after bit synchronisation is the decision logic block.

4.3.7 Decision Device

At this stage, synchronised bit positions are known. The synchroPosition array contains positions of synchronisation. Decisions are made by reading the value of the MF_output_subtract array at positions of synchronisation. If the value is less than zero, it means that the targeted bit is "0"; otherwise it is "1".
Bits are then stored in the demodulatedBits vector for further processing.
The code fragment is as follows:

```
int i,j;
int bitCounter = 0;
for(i = 0; i < 7; i++)
{
   j = synchroPosition[i];
   if(MF_output_subtract[j] < 0) //condition for zero bit
   {
      zeroBitCounter++;
      oneBitCounter = 0; //reset
      demodulatedBits.push_back(0); //storing zero bit in a vector
   }
   else    //condition for one bit
   {
      oneBitCounter++;
      zeroBitCounter = 0; //reset
      demodulatedBits.push_back(1); //storing one bit in a vector
   }
   bitCounter++;
   if(zeroBitCounter >= 8) demodFlag = 0;
}
```

4.3.8 Shifting of samples

The buffer has now been processed and it should be prepared for the next intake of new samples from the kernel buffer.
Samples from the start to the 7^{th} position of synchronisation (end of seventh bit), inside the application buffer array, are erased. The rest are shifted to the front of the application buffer array.
Note that the application buffer is double the size of the kernel buffer.
Figure 4.7 shows processes that takes place in the applicationBuffer. It shows how shifting occurs inside the buffer in three stages. These stages occur for every demodulation process cycle, and are illustrated as follows:

- Stage A: This is the state just after bit synchronisation.
- Stage B: Samples up to the 7^{th} synchronised bit are deleted.
- Stage C: Remaining samples are shifted to the front of the array.

When number of samples in Stage C are greater than or equal to 1024, the application does not go to the kernel buffer to collect new data. It processes samples that have been accumulated in Stage C and at the same time the application must not compromise new data that it must collect from the kernel buffer.
To tackle this problem, the application must be fast enough to complete two cycles of processing before the kernel buffer is ready to deliver new data. If it takes 23.28ms of recording to fill one buffer, the application has to take less than $\frac{23.28}{2} = 11.64$ms to complete one process cycle. Failure to do so will result in the application not coping with new data that it must collect from the kernel buffer.

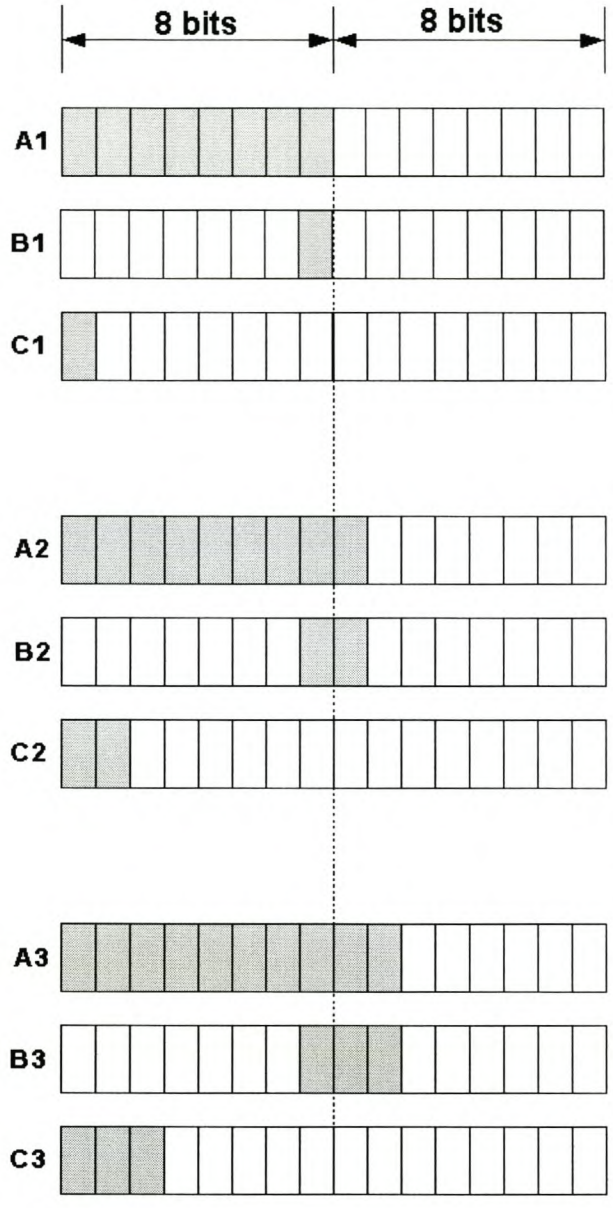

Figure 4.7: Shifting of samples

This code fragment shows how shifting of samples takes place. The firstBufferFullFlag is assigned a value of "1" when the application buffer is holding samples that are more than 1024.

```
position =synchroPosition[6]; //seventh bit position
//Deleting the first seven synchro bits samples of applicationBuffer
index = 0;
for(i = (position + 1);i < tailIndex; i++)
{
      applicationBuffer[index++] = applicationBuffer[i];
}
tailIndex = tailIndex - (position + 1);
if(tailIndex >= eightBitSamples)
```

```
{
   firstBufferFullFlag = 1;
   firstCounter = 0; //reset
}
else
{
   firstBufferFullFlag = 0;
}
```

The variable "tailIndex" monitors the position of the last sample in the application buffer. If the variable is at a position greater or equal to eight bits samples size, the flag "firstBufferFullFlag" is set to one. This tells the application to use data it has in the application buffer before the application can go and collect new data from the kernel buffer.

4.3.9 Output Display

The purpose of this method is to check whether the application is demodulating correctly. Converting demodulated bits to characters makes it easier to check the correctness of the results.
At this stage, all bits that are stored in the demodulatedBits array, are converted back to characters that are displayed on the screen.

```
size = demodulatedBits.size();
bitCounter = 0;
do
{
   for(int i = 7;i >= 0;i--)
   {
      //reversing Bits
      finalMessage[i] = demodulatedBits[bitCounter++];
   }
   sum = 0;
   for(int i=0;i<8;i++)
   {
      sum = sum + finalMessage[i]*pow(2,(float)i);
   }
   cout<<(char)sum; //character format
}while(bitCounter < size);
```

4.4 Conclusion

The simulation of both the modulator and the demodulator is described in this chapter. The modulated signal is written to file. The demodulator reads the file and demodulate the read signal. The output of the demodulation process is then displayed on the screen. Results showed that software modulates and demodulates successfully. All the results of simulation, including the bit error rate, are explained in Chapter 6.
The simulation has proven that software algorithm is fine and the sound card, API's and real-time issues can be brought into the chain.

Chapter 5

IMPLEMENTATION USING LINUX SOUND APIs

5.1 Introduction

In this chapter, different Linux sound drivers are used to implement the modulator and demodulator in real time. All four APIs are used to determine the API with the best performance.

Application Programming Interfaces used are:

- Open Sound System (OSS)
- Advanced Linux Sound Architecture (ALSA)
- Jack Audio Connection Kit (JACK)
- PortAudio (PA)

5.1.1 Native APIs

OSS and ALSA are two different low-level APIs for Linux. They are the kernel sound drivers of Linux.
Older kernels of Linux distributions come with OSS. For the Red Hat (RH) distribution, kernels prior to release 2.6 come with OSS, while the current releases come with ALSA. The RH distribution used in this thesis is Fedora Core 1 and its kernel uses OSS, the last kernel from RH to come with OSS kernel drivers.
ALSA is backward compatible with OSS/Free (via OSS emulation). For this thesis, the ALSA kernel module was downloaded from [8]. Both OSS and ALSA implement native blocking calls. ALSA comes ahead of OSS in many things.

5.1.2 Advantages of ALSA over OSS

1. ALSA supports the OSS API as a subset of its own API.

2. In cases where there might be a driver for both APIs, the ALSA one is often better.
3. ALSA has a callback function while OSS does not.
4. ALSA supports most sound card drivers that are sometimes not supported by OSS.

Pros and Cons of ALSA vs OSS can be found at [18].

5.1.3 Low-Latency Higher-Level APIs

Port Audio and JACK are low-latency APIs that simplify the work of the programmer in communicating with the audio hardware. They form an interface between the native APIs and the programmer. PortAudio comes in two versions, the OSS and ALSA. JACK uses only ALSA drivers.

5.2 Implementation Using OSS

5.2.1 OSS Overview and Introduction

Open Sound System (OSS) is the first attempt to unify the digital audio architecture for UNIX. OSS is a set of device drivers that provide a uniform API across all the major UNIX architectures. It supports SoundBlaster or Windows Sound System compatible sound cards which can be plugged into any UNIX workstation supporting the ISA or PCI bus architecture. OSS also supports workstations with on-board digital audio hardware[9]. OSS uses Linux system calls that are explained in Appendix C.
Topics that are to be covered in this section are:

- Opening the device file "/dev/dsp".
- Setting the sound parameters.
- Accessing the kernel buffer.
- Modulator and demodulator processes.
- Closing the device

5.2.2 Opening the Device

The following line of code opens the device `/dev/dsp` for writing only. The modulator will be writing samples to the kernel buffer.

```
fd_dsp=open("/dev/dsp",O\_WRONLY,0);
```

For the demodulator, the device is opened for read only. The application will be reading data from the kernel buffer.

```
fd_dsp=open("/dev/dsp",O\_RDONLY,0);
```

Setting Sound Parameters

After opening "*/dev/dsp*", the next step is to set the sampling parameters. These parameters determine the quality of sampled audio data.
The parameters are:

- Sample format (no. of bits)
- Number of channels (mono/stereo)
- Sampling rate (speed)

The OSS manual guide [7] prefers the order above, when setting these parameters. Setting sampling rate before the number of channels does not work with all devices.
Sampling parameters can only be changed between open and the first *read*(), *write*() or other *ioctl*() call made to the device. The effect of changing sampling parameters when the device is active is undefined. The device must be reset using the ioctl `SNDCTL_DSP_RESET` before it can accept new sampling parameters.

Sample Format

This parameter affects the quality of audio data. Table 5.1 shows formats that are supported by the OSS API.

Audio Format	Description
AFMT_MU_LAW	logarithmic `mu-Law` audio encoding
AFMT_U8	standard unsigned 8 bit audio encoding
AFMT_S16_LE	standard 16 bit signed little endian
AFMT_S16_BE	big endian variant of the 16 bit signed format
AFMT_S8	signed 8 bit audio format
AFMT_U16_LE	unsigned little endian 16 bit format
AFMT_U16_BE	unsigned big endian 16 bit format

Table 5.1: OSS Audio Formats

These audio formats are defined in the kernel *soundcard.h* header file. In this thesis, 8-bit unsigned samples are used.
The settings are as follows:

```
arg = AFMT_U8;
status = ioctl(fd_dsp, SNDCTL_DSP_SETFMT, &arg);
if(status == -1)
{
   close(fd_dsp);
   throw "SNDCTL_DSP_SETFMT";
}
if(arg != fmt)
{
   close(fd\_dsp);\\
   throw "unable to set the requested format";
}
```

Channels

Both mono and stereo modes are supported and only values "0" and "1" are allowed, respectively.
Here the application will select the number of channels, calling ioctl *SNDCTL_DSP_STEREO* with an argument specifying the number of channels. The variable *arg* will have a value of zero if mono is requested. The settings are as follows:

```
arg = channel;
status = ioctl(fd_dsp, SNDCTL_DSP_STEREO, &arg);
if(status == -1)
{
   close(fd_dsp);
   throw "SNDCTL_DSP_STEREO ioctl failed";
}
if(arg != channel)
{
   close(fd\_dsp);
   throw "unable to set requested channel";
}
```

Sampling Rate

The sampling rate for the sound card is set as follows:

```
arg = samplingRate;
status = ioctl(fd_dsp, SNDCTL_DSP_SPEED, &arg);
if(status == -1)
{
   close(fd\_dsp);
   throw "SNDCTL_DSP_SPEED ioctl failed";
}
if(arg != samplingRate)
{
   close(fd_dsp);
   throw "unable to set requested sampling rate";
}
```

Fragments Size

The setting of the fragment size is not compulsory. If not set, the default fragment size of the sound card buffer will be used. However caution should be taken. If the default fragment size is smaller than the size of the application buffer, buffer under-runs are likely to occur. So it is advisable to set it according to the application needs. The application makes a request by calling ioctl *SNDCTL_DSP_SETFRAGMENT* with an argument specifying the number of frames. The request is to have a buffer that has one fragment that can accommodate 1024 frames. In return the ioctl will return argument containing the frame size. If the value of the argument returned is not equal the requested one, the application has to give an error message.

```
int numOfFragments,fragmentSize, intPowerOfTwo;
intPowerOfTwo = 10; //2^10 = 1024
numOfFragments = 1;
arg = intPowerOfTwo;
arg = arg + numOfFragments * 65536;
int fragSize = arg;
status = ioctl(fd_dsp,SNDCTL_DSP_SETFRAGMENT,&arg);
if(status == -1)
{
   close(fd_dsp);
```

```
    throw "SNDCTL_DSP_SETFRAGMENT ioctl failed";
}
if(arg != fragSize)
{
    close(fd_dsp);
    throw "unable to set the requested fragment";
}
```

NB: The application should always check the returned argument and compare it with the requested one.

Determining Fragment Size

The fragment size can be determined by the following code fragment:

```
status = ioctl(fd_dsp,SNDCTL_DSP_GETBLKSIZE,&arg);
if(status == -1)
{
    close(fd_dsp);
    throw "SNDCTL_DSP_GETBLKSIZE ioctl failed";
}
```

All the required sampling parameters have been set. The application is now ready for the modulation process.

5.2.3 Modulator Processes

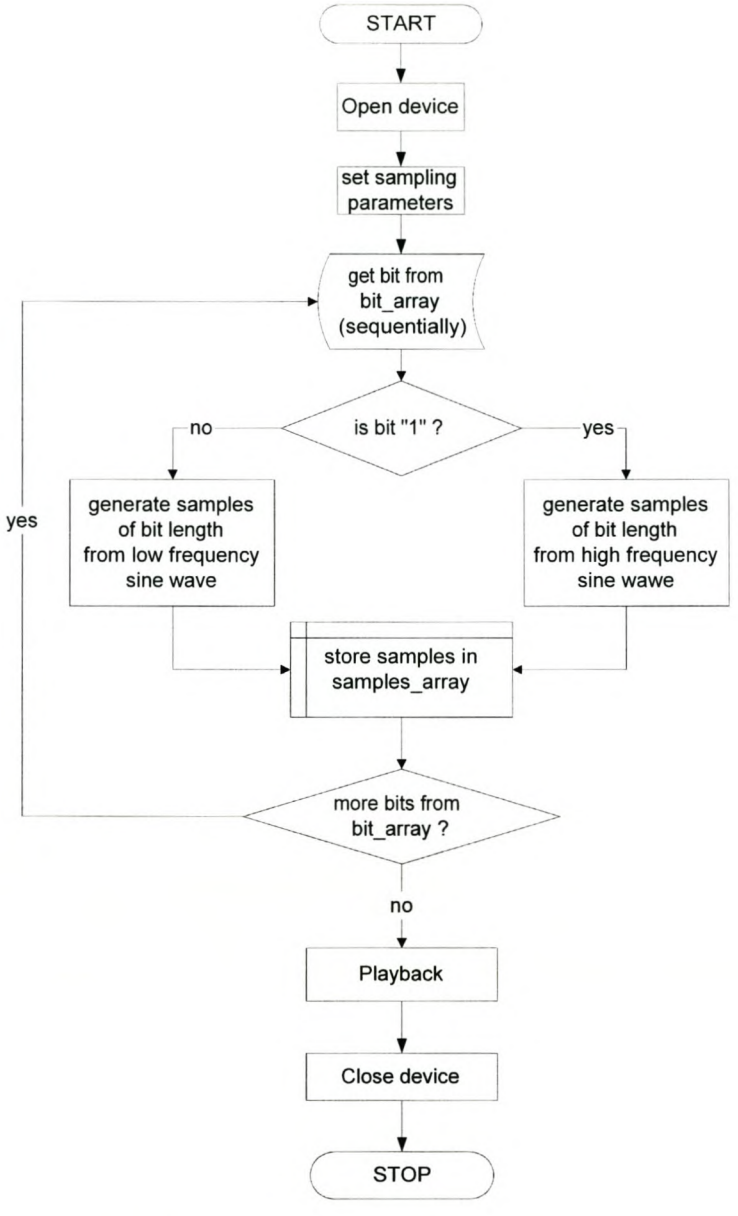

Figure 5.1: OSS modulator flowchart

Figure 5.1 shows a flowchart of the implementation of the modulator.
The modulated signal is written to "/dev/dsp" and transmitted using a sound card.

Start/End Bits

For the demodulator to detect the message transmitted, the modulator must have both start and end bits, before and after the message, respectively.
The message that was entered using the keyboard is converted to bits and stored in the

bitsMessageArray[]. Before the message is modulated, twelve leading zero bits followed by a one bit as the 13^{th} bit are placed in front of the message bits. Figure shows how start and end bits are implemented.

Figure 5.2: Start/End bits

For example, the message "@7" in binary is

```
01000000 00110111
```

The message above has eight zero bits in a row. Showing that if only eight zero bits were used as both start and end bits, the modem was going to mistaken this as end bits. Thirteen "zero" bits in a row are chosen as part of start bits, since there is no combination of characters that will have a sequence of more than eight "zero" bits.

The code fragment that implement this is as follows:

```
messageArray = new int[messageSize];
int p = 0;
for(int i= 1; i <= messageSize; i++)
{
    temp = (int)inputMessage[i];//(type string)
    messageArray[p++] = temp;
}
//////////////////////////////////////////
//s = String(SpinEdit->Text);
//leadingBits = StrToInt(s);
int inputBit;
startBits = 13;
endBits = 13;
int totalBits = 8*messageSize + startBits + endBits;
p = 0;
bitsMessageArray = new int[totalBits];
```

Inserting start bits:

```
for(int i = 1;i <= startBits; i++)
{
   if( i == 13)
      inputBit = 1;
   else
      inputBit = 0;
   bitsMessageArray[p++] = inputBit;
}
```

Converting integers to bits:

```
int reverseBitsArray[8];
int number,charCounter,bitCounter,remainder;
for(int i =0;i<messageSize;i++)
{
   number = messageArray[i];
   bitCounter = 0;
```

```
    do
    {
       remainder = number%2;
       number = number/2;
       reverseBitsArray[bitCounter++]=remainder;
       //bitCounter++;
    }while(bitCounter<8);
    for(int i = 7; i >=0;i--)
    {
       bitsMessageArray[p++]=(reverseBitsArray[i]);
    }
}
```

Inserting end bits:

```
for(int i =0;i < endBits; i++)
{
   inputBit = 0;
   if (i == 0) inputBit = 1;
   if (i == 12) inputBit = 1;
   bitsMessageArray[p++]= inputBit;
}
numOfBits = totalBits;
```

Bits that are stored in the bitsMessageArray[] are the ones that are modulated. The modulation procedure is the same as the one used in the simulation process.
The modulated signal is stored in the audioBuffer[] array.

Playback

Samples in the audioBuffer are now written to `/dev/dsp`, the kernel buffer using `write()` system call. The modulated signal is played using the sound card.
Samples are written to the kernel buffer as follows:

```
write(fd_dsp, audioBuffer, totalBuffer);
```

The modulated signal gets played and transmitted using an audio cable, to the input port of the sound card of the second computer. The second computer will demodulate the signal.

5.2.4 Demodulator Processes

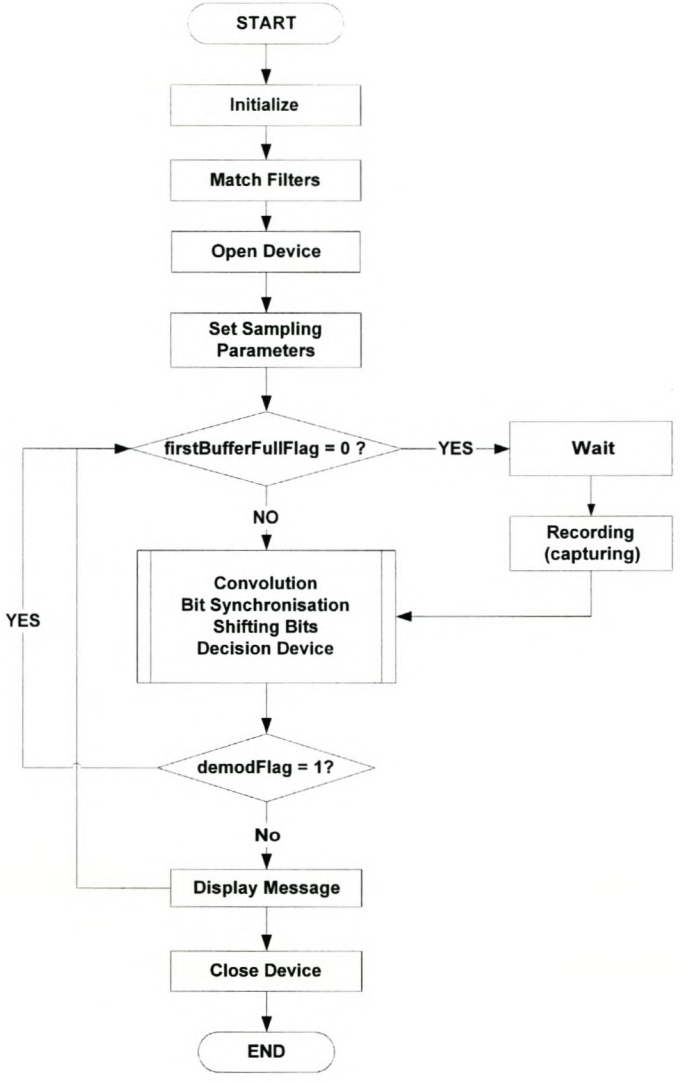

Figure 5.3: OSS demodulator flowchart

Figure 5.3 shows a flowchart of the modulator. All processes that take place in the demodulation process are just the same as those explained in Chapter 4.

Reading from the kernel buffer

Data captured through the sound card is transferred to the application using the *read*() system call.

```
read(snd.audio_fd,snd.applicationBuffer,snd.bufferSize);
```

Data is copied to the application buffer through the `audio_fd` file descriptor. The number of bytes to be read per call must be equal to the kernel buffer size.

OSS uses `double-buffering`. The buffer is divided into two. The first buffer is accessed by the kernel, while the second buffer is accessed by the application. When the kernel has completely filled the first buffer, swapping takes place. The application now access the first and the kernel the second buffer.
All that the application has to do, is to copy all the data from the kernel buffer it has accessed, go and process it and return back. This process is repeated as long */dev/dsp* is still open.
There are two problems the application is faced with:

1. It should be extremely fast in processing data and return to collect new data on time.

2. Returning early will result in collecting the same data from the same buffer that it had just accessed before.

The first problem can only be solved by optimising the functionality of the code. The application must be able to process data in less than half the time required to record one buffer.

The second problem can be solved by using buffer pointers. The OSS API provides the pointer that can be used to point to the current recording position of a buffer. Knowing the current recording position, the application can utilize this to monitor swapping of buffers.
At this stage, the application simply idles inside an infinite loop while at the same time it monitors the current recording position. If the present byte offset (the field info.ptr) of the current position happens to be smaller than the previously read one, the application knows that swapping has taken place. The application then gets out of the loop and access the new kernel buffer to collect new data.

The code fragment below shows how this can be done.

```
tempPtr = 0;
for(;;)
{
   if(ioctl(audio_fd,SNDCTL_DSP_GETIPTR,&info) !=0)
   {
      perror("Unable to query buffer space");
      exit(1);
   }
   if(tempPtr < info.ptr)
   {
      tempPtr = info.ptr;
   }
   else
   {
      break;
   }
};
```

This process is repeated as long the device is open.

Bits Detection

As the demodulator converts modulated signals to bits, this process can continuously supply bits for further processing or bits can be passed on to the next processing stage, only when data considered to be valuable is detected by the demodulator. For the purpose of testing the functionality of the modem module, the modem keeps idling even when no transmission is taking place. The module idles with one bits and as soon it detects start bits of the message, it then stores them in a vector. A vector is used instead of an array, since the length of the message is not known. After detecting end bits, the module proceeds to the "Display Message" block. The transmitted message will then be displayed on the screen.

```
for(int i= 0; i < 7; i++)
{
    j = synchroPosition[i];
    if(MF_output_difference[j] < 0) //zero bit is detected
    {
           detectedBit = 0;
           zeroBitCounter++;
    }
    else   //one bit is detected
    {
           detectedBit = 1;
           oneBitCounter++;
    };
    //*****************************************************
    //* The conditional statement below checks for successive
    //* "0" leading bits more than ten and being followed by a
    //* "1" bit. The condition will set on the capture flag.
    //*****************************************************
    if((zeroBitCounter > 10) && (detectedBit == 1))
    {
         //if(captureFlag ==1) outputFlag = 1;
         captureFlag = 1;
         zeroBitCounter = 0;
    };
    //*****************************************************
    //* The conditional statement below checks for
    //* successive "1" that are more than ten and while the
    //* capturing flag is set on. If the condition is true,
    //* capturing is stopped and display process is started.
    //*****************************************************
    if((oneBitCounter >10)&&( captureFlag == 1))
    {
         outputFlag = 1;
         captureFlag = 0;
    }
    //*****************************************************
    //* Conditional statement below  controls
    //* capturing of bits to be used to display message.
    //*****************************************************
    if(captureFlag ==  1) // capturing condition
    {
         demodulatedBits.push_back(detectedBit);
    };
    //*****************************************************
    //* Conditional statements below resets
    //* bit counters.
    //*****************************************************
    if(detectedBit == 0) oneBitCounter = 0;
    if(detectedBit == 1) zeroBitCounter = 0;
    //Controlling oneBitCounter not to overflow
    //Not at all restricted to use 50
    if (oneBitCounter >50) oneBitCounter = 0;
}
```

5.2.5 Closing the Device

To shut down the demodulator, the application releases any locks that are held by the process on the device.
The device is closed by calling the *close*() system call. The line of code for closing is:

```
close(fd_dsp)
```

5.2.6 OSS Summary

Both the modulator and the demodulator were implemented. The combination deliver expected results. The output is equivalent to the input.

The only serious drawback is the way the modulator is designed. If the modulator has to modulate very long messages. All the message has to be modulated first, and be played all at once. The design is not practically viable.
The modulator should be able to collect data from the data table in fixed fragments for playing, without disturbing play.
The solution to this problem is to come up with an interrupt-driven modulator, but unfortunately OSS does not have a callback mechanism.

The demodulator performs very well. Synchronisation between the application and the kernel buffer was achieved with the pointer that is provided by OSS. The pointer monitors current recording positions in the kernel buffer.

5.3 Implementation Using ALSA

5.3.1 ALSA Overview and Introduction

Jaroslav Kysela and others started writing an alternate sound driver for the Gravis Ultra-Sound Card. The project was renamed Advanced Linux Sound Architecture (ALSA) and has resulted in, what they believe, a more generally usable sound driver that can be used as a replacement for the built-in kernel drivers. The ALSA drivers support a number of popular sound cards, are full duplex, fully modularised, and compatible with the sound architecture in the kernel.

Topics that are covered in this section are:

- Opening the device
- Setting sound pcm parameters
- The callback process
- Delivering data to callback
- Closing the device

5.3.2 Interrupt-Driven Modulator

Handles to Structures

Before the application starts, handles to structures are created.

```
snd_pcm_hw_params_t *hw_params;        //for software params
snd_pcm_sw_params_t *sw_params;        //for hardware params
snd_pcm_sframes_t frames_to_deliver;   //for callback
```

Opening The Device

```
char *pcm_name;
pcm_name = strdup("plughw:0,0");
if ((err = snd_pcm_open (&playback_handle, pcm_name,
                    SND_PCM_STREAM_PLAYBACK, 0)) < 0)
{
   fprintf (stderr, "cannot open audio device %s
           (%s)\n",argv[1],snd_strerror (err));
   exit (1);
}
```

The next step is to set hardware parameters.

Setting Sound PCM HW Parameters

The steps below set hardware parameters.

1. Allocate the hardware parameter structure, snd_pcm_hw_params_t.

```
if ((err = snd_pcm_hw_params_malloc (&hw_params)) < 0)
{
   fprintf (stderr, "cannot allocate hardware parameter structure
   exit (1);
}
```

2. Fill parameters with full configuration space for a PCM.

```
if ((err = snd_pcm_hw_params_any (playback_handle, hw_params)) < 0)
{
   fprintf (stderr, "cannot initialize hardware parameter structure
            (%s)\n",snd_strerror (err));
   exit (1);
}
```

3. Configure space to contain only one access type.

```
if ((err = snd_pcm_hw_params_set_access (playback_handle, hw_params,
                                         SND_PCM_ACCESS_RW_INTERLEAVED)) < 0)
{
   fprintf (stderr, "cannot set access type (%s)\n",snd_strerror(err));
   exit (1);
}
```

4. Set the sample format.

```
if ((err = snd_pcm_hw_params_set_format (playback_handle, hw_params,
                                         SND_PCM_FORMAT_U8)) < 0)
{
   fprintf (stderr, "cannot set sample format (%s)\n",
                    snd_strerror(err));
   exit (1);
}
```

5. Configure space to have sampling rate nearest to the targeted one.

```
unsigned int sampling_rate = data.samplingRate; int dir = 0;
if ((err = snd_pcm_hw_params_set_rate_near (playback_handle,
                            hw_params, &sampling_rate, &dir)) < 0)
{
   fprintf (stderr, "cannot set sample rate (%s)\n",
                    snd_strerror(err));
   exit (1);
}
```

6. Set configuration to contain only mono channel.

```
if ((err = snd_pcm_hw_params_set_channels (playback_handle,
                                           hw_params, 1)) < 0)
{
   fprintf (stderr, "cannot set channel count (%s)\n",
                                      snd_strerror(err));
   exit (1);
}
```

7. Set the above hardware parameter settings.

```
if ((err = snd_pcm_hw_params (playback_handle, hw_params)) < 0)
{
   fprintf (stderr, "cannot set parameters (%s)\n",
            snd_strerror(err));
   exit (1);
}
```

8. Free previously allocated snd_pcm_hw_params_t.

```
snd_pcm_hw_params_free (hw_params);
```

The next step is to tell ALSA when to wake the application up whenever frames_per_buffer or more frames of playback data can be delivered. Also to tell ALSA that the application itself will start the device.

Setting Sound PCM SW Parameters

The steps below set the sound software parameters.

1. Allocate the software parameter structure.

```
if ((err = snd_pcm_sw_params_malloc (&sw_params)) < 0)
{
   fprintf (stderr, "cannot allocate sw param struct (%s)\n",
                                      snd_strerror (err));
   exit (1);
}
```

2. Initialise the current software parameters structure.

```
if ((err = snd_pcm_sw_params_current(playback_handle,sw_params))< 0)
{
   fprintf (stderr, "cannot initialize sw  param struct (%s)\n",
                                          snd_strerror (err));
   exit (1);
}
```

3. Set the minimum available frames (frames_per_buffer) to consider PCM ready.

```
if ((err = snd_pcm_sw_params_set_avail_min (playback_handle,
        sw_params, data.frames_per_buffer)) < 0)
{
   fprintf (stderr, "cannot set min available count (%s)\n",
        snd_strerror (err));
   exit (1);
}
```

4. Start a threshold inside a software configuration container, threshold start value is 0U (unsigned int) in frames.

```
if ((err = snd_pcm_sw_params_set_start_threshold (playback_handle,
      sw_params, 0U)) < 0)
{
   fprintf (stderr, "cannot set start mode (%s)\n",
         snd_strerror(err)) ;
   exit (1);
}
```

5. Set software parameters.

```
if ((err = snd_pcm_sw_params (playback_handle, sw_params)) < 0)
{
   fprintf (stderr, "cannot set sw parameters (%s)\n",
                                snd_strerror(err));
   exit (1);
}
```

6. After setting all the required parameters, the audio interface is prepared for use as follows:

```
if ((err = snd_pcm_prepare (playback_handle)) < 0)
{
   fprintf (stderr, "cannot prepare audio interface (%s)\n",
                                    snd_strerror (err));
   exit (1);
}
```

Callback Process

The interface is already set at this stage to interrupt the kernel for every frames_per_buffer frames, and ALSA will wake up this program very soon after that.

The "Callback Process" procedure takes place inside the *while(1)* loop as follows:

1. Application waits until the interface is ready for taking data.

```
if ((err = snd_pcm_wait (playback_handle, 1000)) < 0)
{
   fprintf (stderr, "poll failed (%s)\n", strerror (errno));
   break;
}
```

2. Application finds out how much space is available for playback data.

```
if ((frames_to_deliver = snd_pcm_avail_update (playback_handle)) < 0)
{
   if (frames_to_deliver == -EPIPE)
   {
      fprintf (stderr, "an xrun occurred\n");
      break;
   }
   else
   {
      fprintf (stderr,"unknown avail update return value (%d)\n",
                           frames_to_deliver);
      break;
   }
}
```

3. If space available for playback is greater than the calculated frames per buffer, the calculated number of frames (by the application) will be delivered. Otherwise only the number of frames required for playback will be send. The code fragment is:

```
frames_to_deliver = (frames_to_deliver > data.frames_per_buffer) ?
                     data.frames_per_buffer : frames_to_deliver;
```

4. When valuable message has been played, the application is interrupted by breaking the loop.

```
// ==== deliver data to callback ====//
if(data.index > data.valuable_samples + 3*data.samplesPerBit)
   break;
if (data.playback_callback (frames_to_deliver) != frames_to_deliver)
{
   fprintf (stderr, "playback callback failed\n");
   break;
}
```

Playback

At callback, data is transferred from the audioBuffer array in fixed frames for playback.

```
int err;
/* ... fill buffer with data ... */
for(int i = 0; i < nframes; i++)
{
   Buffer[i] = audioBuffer[index++];
}
if ((err = snd_pcm_writei (playback_handle,Buffer, nframes)) < 0)
{
   fprintf (stderr, "write failed (%s)\n", snd_strerror (err));
}
```

5.3.3 ALSA Demodulator

The implementation of the demodulator resembles that of the OSS. The exception is that ALSA has a callback feature that OSS does not have.
In OSS some means of using buffer pointers are used to monitor the exact position of the

recording position. Which in turn makes it possible to know when the right time avails to collect captured data (i.e Wait process in Figure 5.3 on page 36).

Focus will be put on demodulator processes that are different from those of the modulator. Processes like "Closing the device" will not be discussed as they are the same as those of the modulator.

Opening the device

In opening the device for capturing only; the stream to use is, SND_PCM_STREAM_CAPTURE.

```
snd_pcm_open (&capture_handle, pcm_name, SND_PCM_STREAM_CAPTURE, 0) < 0)
```

Setting the parameters

```
dir = 0;
totalSamples = periods * periodsize;
audioBuffer = new unsigned char [totalSamples];
//audioBuffer = new unsigned char [periodsize];
if ((err = snd_pcm_hw_params_malloc (&hw_params)) < 0)
{
   fprintf (stderr, "cannot allocate hw param structure (%s)\n",
                    snd_strerror (err));
   exit (1);
}
//==== Initialize hardware parameter with full configuration space ====//
if ((err = snd_pcm_hw_params_any (capture_handle, hw_params)) < 0)
{
   fprintf (stderr, "cannot initialize hw param, snd_strerror (err));
   exit (1);
}
//==== Set access type ====//
if ((err = snd_pcm_hw_params_set_access (capture_handle, hw_params,
                                         SND_PCM_ACCESS_RW_INTERLEAVED)) < 0)
{
   fprintf (stderr, "cannot set access type ",snd_strerror (err));
   exit (1);
}
//==== Setting the sample format ====//
if ((err = snd_pcm_hw_params_set_format (capture_handle, hw_params,
                                         SND_PCM_FORMAT_U8)) < 0)
{
   fprintf (stderr, "cannot set sample format (%s)\n",
            snd_strerror (err));
  exit (1);
}
//===== Setting  sampling rate  =====//
if ((err = snd_pcm_hw_params_set_rate_near (capture_handle, hw_params,
                                            &samplingRate, &dir)) < 0)
{
   fprintf (stderr, "cannot set sample rate (%s)\n",
                    snd_strerror (err));
   exit (1);
}
//==== Setting channels =====//
if ((err = snd_pcm_hw_params_set_channels (capture_handle,
                                           hw_params, 1)) < 0)
{
   fprintf (stderr, "cannot set channel count (%s)\n",
                                snd_strerror (err));
   exit (1);
```

```
}
//===== Setting number of periods (fragments) =========//
if (snd_pcm_hw_params_set_periods(capture_handle, hw_params,
                                  periods, 0) < 0)
{
   fprintf(stderr, "Error setting periods.\n");
   exit(1);
}
//==== Set buffer size (in frames). The resulting latency is given by
//==== latency = periodsize * periods / (samplingRate * bytes_per_frame)
if ((err = snd_pcm_hw_params_set_buffer_size(capture_handle,
                                 hw_params, (periods*periodsize))) < 0)
{
   fprintf(stderr, "Error setting buffersize (%s)\n",
                    snd_strerror (err));
   exit(1);
}
//====== Setting hw parameters =====//
if ((err = snd_pcm_hw_params (capture_handle, hw_params)) < 0)
{
   fprintf (stderr, "cannot set parameters (%s)\n",snd_strerror (err));
   exit (1);
}
//====== Freeing a previously allocated snd_pcm_hw_params_t =====//
snd_pcm_hw_params_free (hw_params);
if ((err = snd_pcm_prepare (capture_handle)) < 0)
{
   fprintf (stderr, "cannot prepare audio interface for use (%s)\n",
            snd_strerror (err));
   exit (1);
}
```

Capturing

As explained before,the application is controlled by callback process that informs it when data becomes available. Data is then copied into the audioBuffer array.
The code fragment below shows how the capturing process takes place.

```
while ((pcmreturn = snd_pcm_readi(capture_handle, audioBuffer,
                                  (periodsize * periods)))< 0)
{
   snd_pcm_prepare(capture_handle);
   fprintf(stderr, "<<<<<<<<<< Buffer Overrun >>>>>>>>>>\n");
};
```

5.3.4 ALSA Summary

Both the ALSA modulator and the demodulator were implemented in this section.
ALSA has a callback function that is called at the right time when data is needed from the kernel buffer.
The modulator is interrupt-driven. This makes it possible for the modulator to collect modulated samples from the data table in fixed fragments, while at the same time the modulator is writing samples to the sound card, for playing.
In the demodulator design, no difficulties were experienced. A callback function synchronises capturing of new data between the the application and the kernel buffer.

5.4 Implementation Using JACK

5.4.1 Jack Overview and Introduction

Most Unix APIs are based on the read/write. OSS is such an API. This is the abstraction that is used in Unix:, "Everything is a file". According to developers of audio applications, the problem with this design is that it fails to take the real-time nature of audio interfaces into account. It becomes rather difficult to facilitate inter-application audio routing when different programs are not all running synchronously. JACK solves all this.

JACK is a low-latency audio server, written for operating systems that use POSIX threads such as GNU/Linux and Apple's OS X.
JACK's layout is shown in Figure E.1 on page 79.

JACK Capabilities

JACK is capable of the following [15]:

1. It provides a high level abstraction for programmers that removes the audio interface hardware from the picture and allows them to concentrate on the core functionality of their software.

2. It allows applications to send and receive audio data to/from each other as well as the audio interface. There is no difference in how an application sends or receives data regardless of whether it comes from another application or an audio interface.

3. It provides a "callback" function in your program that will be executed at the right time. Your callback can send and receive data as well as do other signal processing tasks. The programmer is not responsible for managing audio interfaces or threading, and there is no "format negotiation": all audio data within JACK is represented as 32-bit floating-point values normalised to range [-1;1].

4. It can connect several client applications to an audio device, and allow them to share audio with each other.

5. Clients can run as separate processes like normal applications, or within the JACK server as "plug-ins".

Using JACK within a program typically consists of:

1. calling jack_client_new to connect to the JACK server.

2. registering "ports" to enable data to be moved to and from your application.

3. registering a "process callback" which will be called at the right time by the JACK server.

4. telling JACK that your application is ready to start processing.

5.4.2 JACK Demodulator

Figure 5.4 shows steps that are taken in implementing the demodulator using JACK.

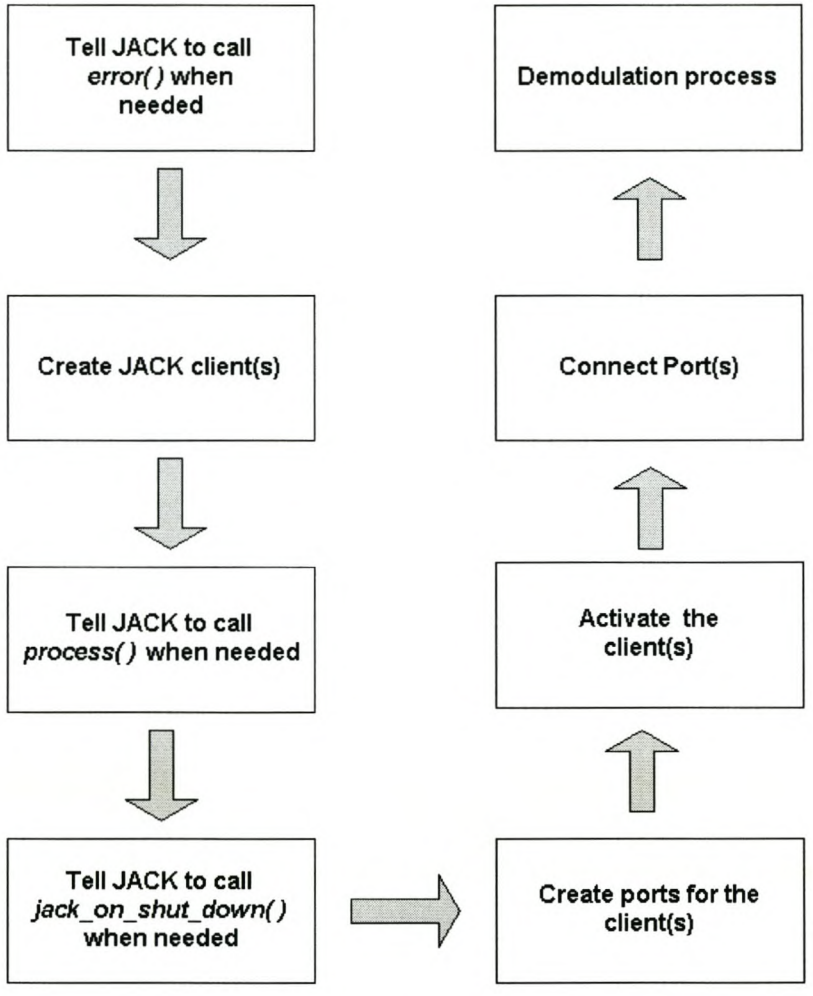

Figure 5.4: JACK implementation steps

5.4.3 Jack Callback Functions

The following callback functions are called by the *main()* function.

Process callback

This is the function that JACK calls whenever data is needed for processing.
The function has the parameter, *nframes*, which is the number of frames available on the clients input port, that the client expects to receive.

Data is then received by using `jack_port_get_buffer`, which returns a pointer to the buffer associated with the given port.
The function *memove()* is used to move data of length *sizeof(jack_default_audio_sample_t) *nframes* that is available, from the associated buffer to the application (client) buffer.
The problem is now how to make the demodulation process know when the data is ready to be captured.
The use of semaphores solves this problem. Semaphores can be used to coordinate operations between the callback process() and the demodulation process.
By having the statement *sem_wait*(&*sem*) in the demodulation process and *sem_post*(&*sem*) in the callback, the two can be synchronised as follows:

- *sem_wait*(&*sem*) - will pause the demodulation process to wait till the data is ready. This should come just before the capturing process.
- *sem_post*(&*sem*) - will release demodulation process to continue with its activities. This should come just after the filling of the application buffer.

The callback *process*() code is as follows:

```
int process (jack_nframes_t nframes, void *arg)
{
   /*grab our input buffer */;
   sample_t *in = (sample_t *)jack_port_get_buffer(input_port,
   memmove (app_buffer, in,sizeof(jack_default_audio_sample_t)* nframes);
   for(int i = 0; i< 8192; i++)
   {
      app_buffer[i] = 100 * app_buffer[i];
   }
   sem_post(&sem);
   return 0;
}
```

Sampling Rate callback

This is also used as a callback. It is only called back when the sampling rate changes. The rate (*samplingRate*) at which the demodulation process samples with, will also have to be updated.
The first parameter (*nframes*) is the number of frames per second (the new sampling rate). The second parameter is the arg pointer.

```
int srate(jack_nframes_t nframes, void *arg)
{
   printf("the sample rate is now %lu/sec\n", nframes);
   samplingRate = nframes;
   return 0;
}
```

Errors callback

This is another callback function. It gets called when problem(s) arise. It even describes the nature of error (desc is a description of the error).

```
void error(const char *desc)
{
   fprintf(stderr,"JACK error: %s\n", desc);
}
```

Shutdown callback

This function gets called when the jack client is shut down. The code fragment is:

```
void jack_shutdown (void *arg)
{
   exit (1);
}
```

Preparing Client for JACK

To prepare a client for Jack is easy and the procedure takes place in the *main*() function of the application.

Catching Errors

Before starting the connection process, JACK is set to call *error*() whenever it experiences an error. This callback is global to this process, and not specific to each client.

```
jack_set_error_function (error);
```

Creating a JACK client

The next step is to create a client of the JACK server.

```
if ((client = jack_client_new (argv[1])) == 0)
{
   fprintf (stderr, "jack server not running?\n");
   return 1;
}
```

The argument is passed when the application is run.

JACK Callback Process

Tell the JACK server to call *process*() whenever there is work to be done.

```
jack_set_process_callback (client, process, 0);
```

JACK server shutting client down

Tell the JACK server to call jack shut_down if ever it needs to be shut down, either entirely, or if it just decides to stop calling the client.

```
jack_on_shutdown (client, jack_shutdown, 0);
```

Creating Ports

A port is created as "input" port of the client. It is the one that delivers audio data. To the JACK server it is the output port of the server.

```
input_port = jack_port_register (client, "input",
             JACK_DEFAULT_AUDIO_TYPE, JackPortIsInput, 0);
```

Activating the client

After ports are created, the next task is to activate the client by telling the JACK server that the client is ready to perform its tasks.

```
if (jack_activate (client))
{
   fprintf (stderr, "cannot activate client");
   return 1;
}
```

Connecting Ports

After activating the client, ports are connected. This step cannot be done before clients are activated. Clients have to run first.

```
if ((ports = jack_get_ports (client, NULL, NULL,
             JackPortIsPhysical|JackPortIsOutput)) == NULL)
{
   fprintf(stderr, "Cannot find any physical capture ports\n");
   exit(1);
}
if (jack_connect (client, ports[0], jack_port_name (input_port)))
{
   fprintf (stderr, "cannot connect input ports\n");
}
```

5.4.4 Demodulation Process

JACK is now running. The demodulation process starts to run inside a *while* loop. The semaphore *sem_wait(&sem)* pauses the demodulation process to wait until data is ready. This just comes before the capturing process.

```
while(1)
{
   if(mf.firstBufferFullFlag == 0)
   {
      sem_wait(&sem);
      mf.kernel_to_application(app_buffer);
      for(int i = 0; i < mf.periods;i++)
      {
         mf.convolution();
         mf.bitSynchro();
         mf.shiftingSamples();
         mf.decisionDevice();
      }
   }
   else
   {
      mf.convolution();
      mf.bitSynchro();
      mf.shiftingSamples();
      mf.decisionDevice();
   }
   if(mf.outputFlag == 1) break;
}
mf.displayMessage();
```

5.4.5 JACK summary

No difficulties were experienced with the implementation of JACK at all. JACK takes care of setting sound parameters.
Semaphores are used to synchronise the callback function with the demodulation process. The only big problem experienced was to understand how JACK works, before starting with modem code. James Shuttleworth [6] explains JACK very well. Jack documentation is found at [14].

5.5 Implementation using PortAudio

5.5.1 PA Overview and Introduction

PortAudio is a library that provides streaming audio input and output. It is a cross-platform API that works on Windows, Macintosh, Unix running OSS, ALSA, JACK, SGI and BeOS[17]. The OSS version of PA is used in this implementation.

Using PA within a program typically consists of the following steps[17]: Here are the steps to writing a PortAudio application:

1. Write a callback function that will be called by PortAudio when audio processing is needed.

2. Initialize the PA library and open a stream for audio I/O.

3. Start the stream. Your callback function will now be called repeatedly by PA in the background.

4. In your callback you can read audio data from the inputBuffer and/or write data to the outputBuffer.

5. Stop the stream by returning 1 from your callback, or by calling a stop function.

6. Close the stream and terminate the library.

5.5.2 PA Modulator

The modulation process is again the same as in other used APIs.
A callback function that is called by PA when audio processing is needed is coded as follows:

```
static int Modu_Callback (
              void *inputBuffer,             //this array is not in use.
              void *outputBuffer,            //array of interleaved samples
              unsigned long framesPerBuffer, //sample frames to be
              PaTimestamp outTime,
              void *userData )
{
   Modu_Data *data = (Modu_Data*)userData;
   char *out = (char*)outputBuffer; //
   unsigned long i;
   int finished = 0;
   (void) outTime; /* To prevent unused variable warnings. */
   (void) inputBuffer;
   for( i=0; i<framesPerBuffer; i++ ) // Data collected in frames
   {
```

```
            *out++ = data->data_array[data->index];
            data->index += 1;
            if (data->index >= data->total_samples) finished = 1;
        }
        return finished;
    }
```

Functions that are called by main() are listed below in order.

1. input_message(): receives entered characters.

2. init(): initialises required variables and arrays.

3. process(): generates a modulated signal.

4. Samples of the modulated signal are transferred to the `data_array`. This is the data table of the modulator.

```
for(i=0; i < data.total_samples; i++)
{
   data.data_array[i] = audio.audioBuffer[i];
}
```

5. Pa_Initialize(): initialise PA. It is called before any other calls to PA. This will trigger a scan of available devices which can be queried later.

6. Pa_OpenStream(): sets sound parameters for the audio stream.
   The code fragment below shows the parameters:

```
Pa_OpenStream(
      &stream,    /* passes back stream pointer */
      paNoDevice, /* default input device  */
      0,          /* no input */
      paUInt8,    /* unsigned 8 bit format */
      NULL,
      OUTPUT_DEVICE,
      1,          /* mono output */
      paUInt8,    /* unsigned 8 bit format */
      NULL,
      SAMPLE_RATE,
      FRAMES_PER_BUFFER,
      0,          /* number of buffers, if zero then use default minimum */
      paClipOff,  /* we won't output out of range samples so don't bother
      Modu_Callback, /* specify custom callback */
      &data);       /* pass data through to callback */
```

7. Pa_StartStream(stream): starts the stream to run. The variable stream points to the stream.

8. Time required to play the modulated signal is calculated.

9. Pa_Sleep(allocated_Time * 1000): sets the modulator to sleep for the allocated time to play the modulated signal.

10. Pa_StopStream(stream): stops the stream after the allocated play time has elapsed.

11. Pa_CloseStream(stream): closes the stream.

12. Pa_Terminate(): terminates PA.

5.5.3 PA Demodulator

A callback function called by the demodulator is as follows:

```
static int recordCallback( void *inputBuffer, void *outputBuffer,
                           unsigned long framesPerBuffer,
                           PaTimestamp outTime, void *userData )
{
   Modu_Data *data = (Modu_Data*)userData;
   SAMPLE *rptr = (SAMPLE*)inputBuffer;
   SAMPLE *wptr = &data->recordedSamples[data->frameIndex * NUM_CHANNELS];
   long i;
   (void) outputBuffer; /* Prevent unused variable warnings. */
   (void) outTime;
   if( inputBuffer == NULL )
   {
      for( i=0; i < data->total_frames; i++ )
      {
         *wptr++ = SAMPLE_SILENCE;
      }
   }
   else
   {
      for( i=0; i < data->total_frames; i++ )
      {
         *wptr++ = *rptr++;
      }
   }
   data->frameIndex -= data->total_frames; //reset index
   data->frameIndex = 0;                   //reset index
   sem_post(&sem);
   return 0;       // to run continuously
}
```

Semamphores are used again to synchronise the demodulation with a callback function. Functions called by main() are listed below in order.

1. init()

2. matchedFilters()

3. Pa_Initialize()
4. Pa_OpenStream()
5. Pa_StartStream()
6. Demodulation Process:

```
while(Pa_StreamActive(stream))
{
   if(mf.firstBufferFullFlag == 0)
   {
      sem_wait(&sem);
      for( i=0; i < data.total_frames; i++ )
      {
          data.processArray[i] = data.recordedSamples[i];
      }
      mf.kernel_to_application(data.processArray);
   }
   mf.convolution();
   mf.bitSynchro();
   mf.shiftingSamples();
   mf.decisionDevice();
   if(mf.outputFlag == 1) break;
   //sem_wait(&sem);
}
```

7. displayMessage()
8. Pa_CloseStream().
9. free(data.recordedSamples):
10. Pa_Terminate().

5.5.4 PA Summary

No serious problems in the implementation of both the modulator and demodulator were experienced. The implementation so far is the simplest if it is compared to the first three APIs.
Like in JACK, semaphores are used to synchronise the callback function with the demodulation process.
PA implementation appears not to be stable at all. Results obtained from it are not reliable. Both tests were conducted using both 8-bit unsigned format and 16-bit format. Results obtained are either completely error free, or with all characters in error. The fact that the demodulator can sometimes display error free messages, indicates that the problem could be related to latency issues or the demodulator could be failing to detect the start and end bits of the message. Buffer sizes ranging from 1024 to 8192 were also tried to solve the problem and mixer settings were monitored carefully for cases like clipping. Other options to solve this problem are:

- to recompile the kernel for low-latency again.
- to use the ALSA version of Port Audio.

Chapter 6

RESULTS

6.1 Introduction

This chapter shows results obtained in simulation and real-time implementations.

6.2 Simulated FSK

6.2.1 Modulation results

Tests on the simulation are performed by modulating the character "P" and then followed by the demodulation of the same character. The ASCII code for the character is "01010000".
The modulated signal is shown in Figure 6.1. The signal has a higher frequency in regions between 128^{th} and 256^{th} samples; and between 384^{th} and 512^{th} samples , while at other positions it has a lower frequency.
Demodulation results of the same signal are shown in the next section.

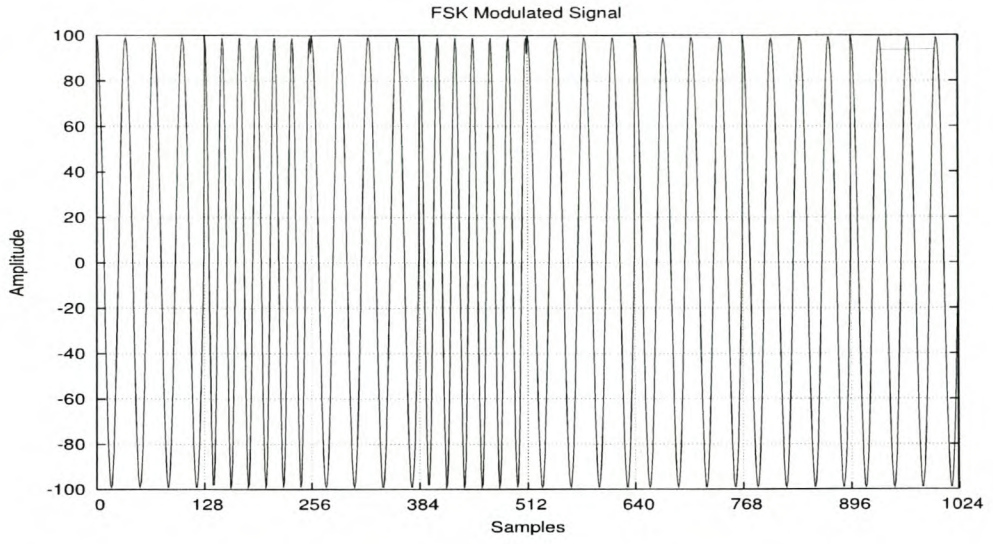

Figure 6.1: Modulated signal

6.2.2 Demodulation results

MF Outputs

Figure 6.2 is the output signal of the low frequency matched filter.
The signal has its peaks at positions 128, 384, 640, 768, 896 and 1024. These peaks denotes "zero" bits.

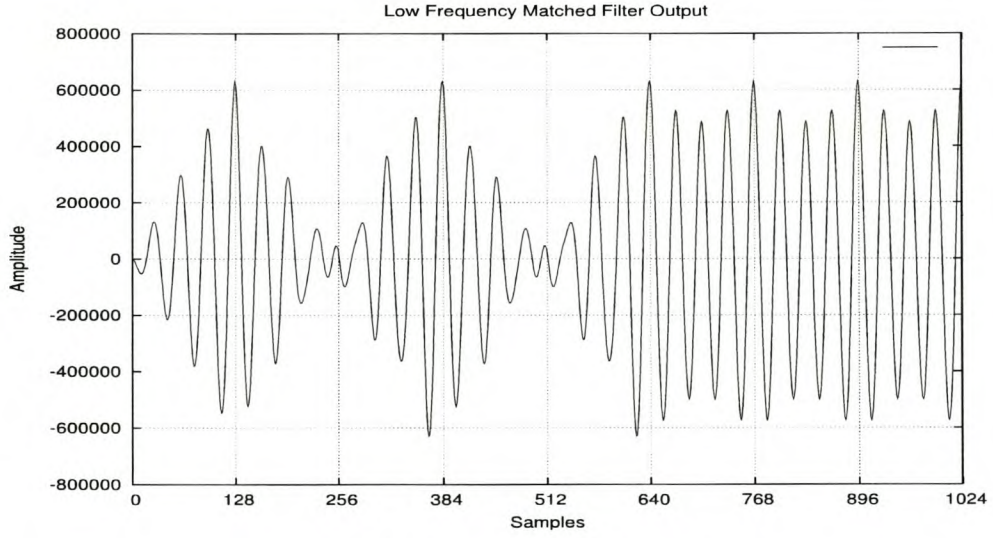

Figure 6.2: Low Frequency MF Output

Figure 6.3 is the output signal of the high frequency matched filter.
The signal has its peaks at position 256 and 512. These peaks denotes "one" bits. Both the two matched filter outputs have energy per bit values that agrees well with the expected

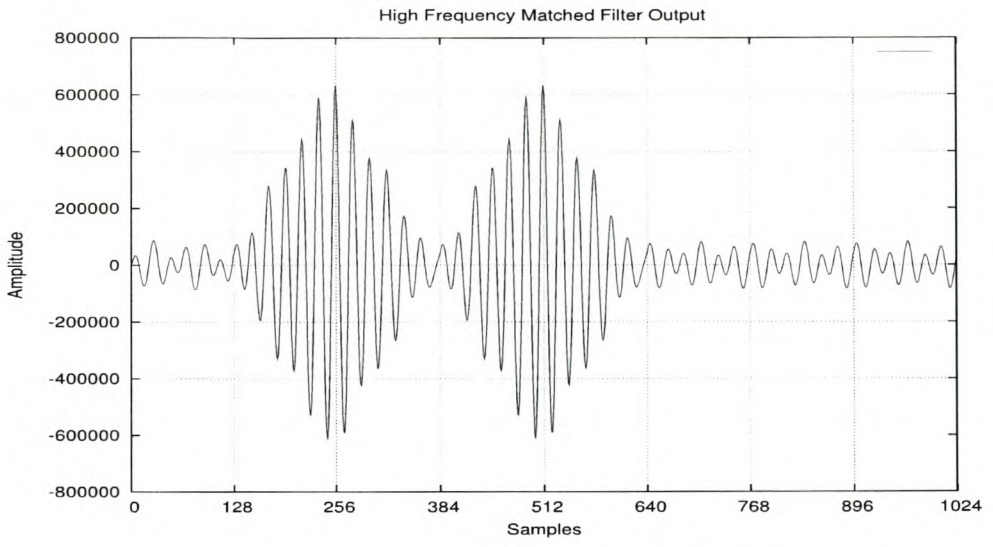

Figure 6.3: High Frequency MF Output

value of $E_b = 0.64 \times 10^6$.

Summation of MF Outputs

Figure 6.4 is the summation of output signals of the two matched filters for the purpose of bit synchronisation. Peaks takes place for every 128 samples. Bit detection takes place at these samples positions.

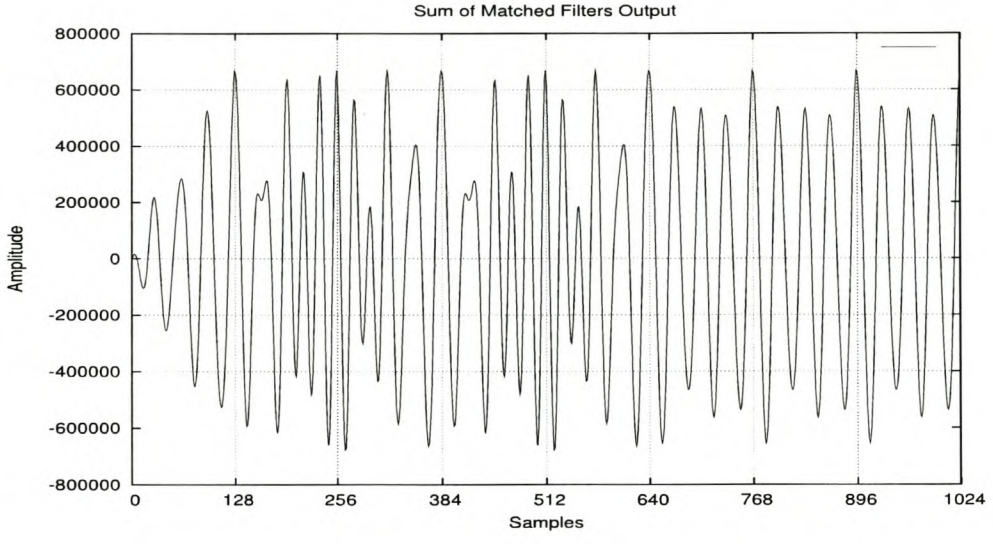

Figure 6.4: Sum Of MF Outputs

Bit Detection

Figure 6.5 is the difference of output signals of the two matched filters for the purpose of decision making. According to the application threshold, bit synchronised positions that have negative values are considered to be "zero bits" otherwise "one bits". From figure 6.5 it is clear that bits detected at positions 256 and 512 are "ones", while at the other five positions (excluding 1024) "zero" bits are detected. The demodulator is designed to synchronise only seven bits at a time, therefore the peak at 1024 is left for the next round.

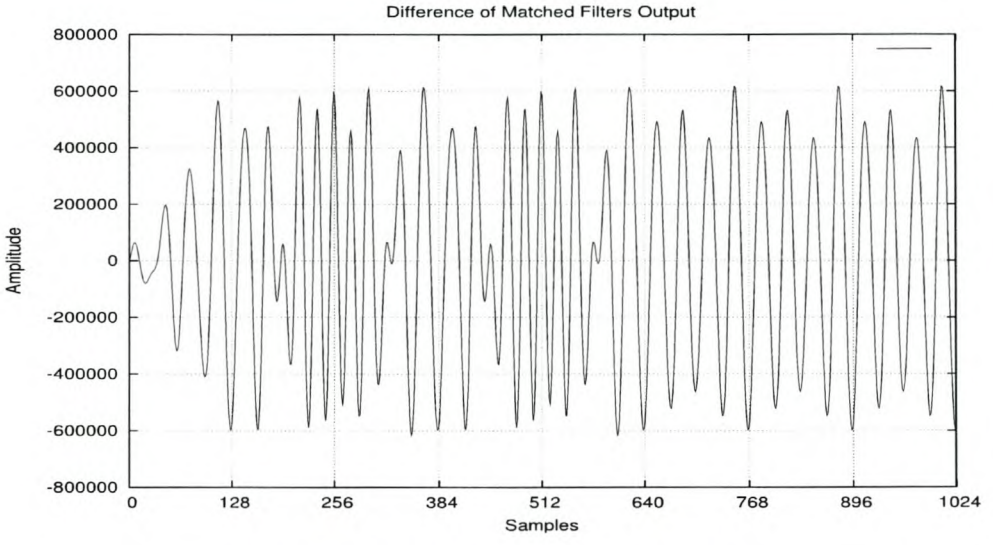

Figure 6.5: Difference Of MF Outputs

6.3 Performance Analysis

The performance analysis test of the module is conducted in simulation. A message consisting of 32200 bits is both modulated and demodulated. The demodulation process is repeated, each time increasing the noise level and adding it into the system. Table 6.1 shows the results obtained.

Noise Level	Error Bits	Error Bits	Error Bits	Error Bits	Bit Error Rate
120	0	0	0	0	0
150	1	4	5	4	108.7×10^{-6}
155	8	8	3	8	209.6×10^{-6}
160	12	11	13	5	318.3×10^{-6}
165	21	22	14	18	582.3×10^{-6}
170	26	26	13	28	722.0×10^{-6}
175	28	23	28	30	846.3×10^{-6}
180	54	43	48	54	1.545×10^{-3}
185	53	70	72	53	1.925×10^{-3}
190	74	71	68	79	2.267×10^{-3}
195	109	115	105	110	3.408×10^{-3}
200	114	105	120	124	3.595×10^{-3}
205	166	269	130	133	5.419×10^{-3}
210	266	204	231	179	6.832×10^{-3}
215	204	214	306	241	7.492×10^{-3}
220	249	474	478	306	11.70×10^{-3}

Table 6.1: Performance Analysis Table

Figure 6.6 shows the plot of both practical and the expected probability of error for the module. The implementation loss comes out to be roughly 0.5dB.

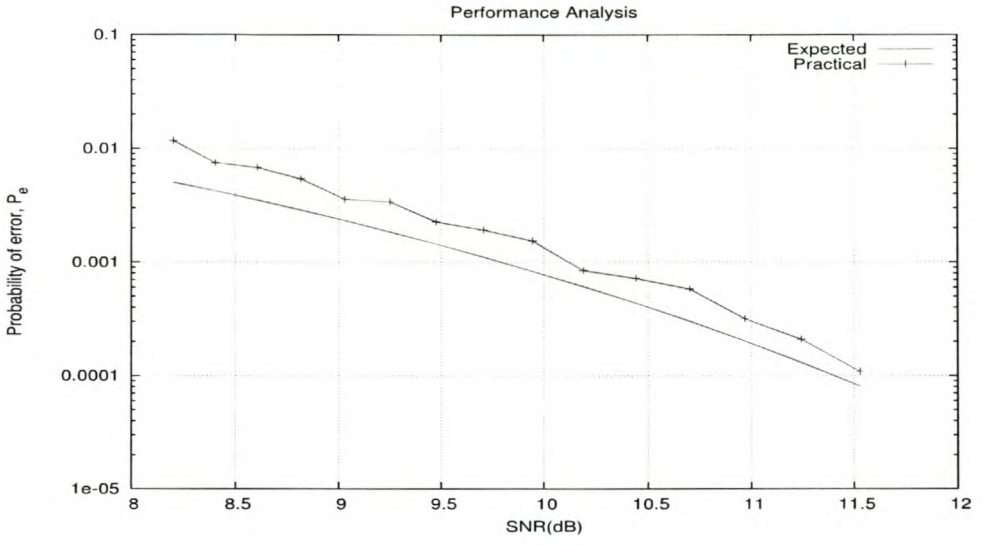

Figure 6.6: Performance Analysis Test

6.4 Real-Time FSK

6.4.1 T_x/R_x Sampling Rate Mismatch Range

The aim of this section is to test the performance of the modem under buffer overruns and underruns. The sampling rate of the application and that used for the sound card settings are mismatched. The mismatch range is determined. This is the range where bit synchronisation and decision functions performs without any errors. All sampling rate mismatch took place in the modulator.

First the test is conducted by starting with the application sampling rate of 44kHz and gradually decreasing it till errors occur.
The second test is to start with the application sampling rate of 44kHz and gradually increase it till errors occur.

Table 6.2 shows ranges that appeared to give stable correct results. Beyond the ranges, there is a probability of errors appearing in the output results.

Mismatch Range (kHz)		% Error in sampling rate
Min Rate	Max Rate	
42.9	44	2.5
44	46.7	6.1

Table 6.2: SR Mismatch Ranges

6.4.2 Tests Across APIs

Tests were conducted using all possible combinations of APIs and results compiled as follows:

From Table 6.3, PortAudio appears not to be stable at all.

Modulator	Demodulator	Comments
OSS	OSS	Error free
OSS	ALSA	Error free
OSS	JACK	Error free
OSS	PA	Not reliable
ALSA	OSS	Error free
ALSA	ALSA	Error free
ALSA	JACK	Error free
ALSA	PA	Not reliable
PA	OSS	Not reliable
PA	ALSA	Not reliable
PA	JACK	Not reliable
PA	PA	Not reliable

Table 6.3: Tests Across APIs

6.4.3 Profiling Demodulation Process

The code is profiled to determine how much time is taken in calling different subroutines of the program and also to get the overall processing time of the code. Only the OSS was profiled. All four APIs are using the same methods for the demodulation process.
According to program specifications, a bit duration is 2.91ms. Since a buffer has a length of eight bits, it will take about 23.28ms to fill one buffer with data.
For 2000 buffer recordings, it will take 45.56 seconds of recording time. Figure 6.7 shows the profiling results obtained. The overall processing time is roughly 2.9 seconds for the process of a recording that lasted 45.56 seconds. The results obtained clearly show that the program is capable of processing data at durations far less than half the recording time of just filling one buffer.
From calculations, the processing time is roughly 6.4% of the time required to do one buffer recording. Profiling results were compiled using Intel Celeron 850MHz CPU in the demodulator.

6.5 Summary

Results of both simulation and real-time implementations are discussed in this chapter. Simulation results discussed are:

- Waveforms at different stages of the demodulator.
- Bit error rate of the modem.
- Implementation loss.

Real-time results discussed are:

- Tests across different APIs, that is the modulator and demodulator operating with different APIs.
- T_x/R_x Sampling Rate Mismatch Range.
- Profiling of the demodulator.

2004-10-11 #1

Function/Method	Count	Total (s)	%	Self (s)	Total ms/call	Self ms/call
global constructors keyed to matDemod::matDemod	1	2.930	0.000	0.000	0.000	0.000
global constructors keyed to sound::wait_for_kernel_buffer	1	2.930	0.000	0.000	0.000	0.000
matDemod::bitSynchro	2290	2.890	8.190	0.240	0.100	0.100
matDemod::clrscr	1	2.930	0.000	0.000	0.000	0.000
matDemod::convolution	2290	2.650	90.440	2.650	1.160	1.160
matDemod::decisionDevice	2290	2.930	0.000	0.000	0.000	0.000
matDemod::init	1	2.930	0.000	0.000	0.000	0.000
matDemod::kernel_to_application	2000	2.920	1.020	0.030	0.010	0.010
matDemod::matchedFilters	1	2.930	0.000	0.000	0.000	0.000
matDemod::~matDemod	1	2.930	0.000	0.000	0.000	0.000
matDemod::matDemod	1	2.930	0.000	0.000	0.000	0.000
matDemod::shiftingSamples	2290	2.930	0.340	0.010	0.000	0.000
sound::soundSettings	1	2.930	0.000	0.000	0.000	0.000
sound::wait_for_kernel_buffer	2000	2.930	0.000	0.000	0.000	0.000
__static_initialization_and_destruction_0(int, int)	1	2.930	0.000	0.000	0.000	0.000
__static_initialization_and_destruction_0(int, int)	1	2.930	0.000	0.000	0.000	0.000

Figure 6.7: OSS Demodulator Profile Results

Chapter 7

DISCUSSIONS AND CONCLUSIONS

7.1 Implementation Options

In the bit synchronisation method, there is another efficient way of implementing synchronisation.
In the method used, accumulated samples are processed if their number is greater than or equal to the kernel buffer size. That means that at some stages processing time will have to be double. The performance of this method depends entirely on the processor's speed. The other method is to switch between 7 and 8 bit synchronisation. In this method, the processing time will only increase by 14.3% when 8 bits are synchronised and this actually gives the processor some time to perform other tasks. The method used in the project is simpler to implement.

7.2 Concluding Remarks

From the results obtained during tests of different APIs, both the native ALSA and JACK are considered the best APIs to work with, mainly for the following reasons:

1. ALSA has a callback function.

2. The modulator is interrupt driven, and data can be collected from the data table in fixed buffer sizes for playing.

3. ALSA can easily port to JACK.

4. Both ALSA and JACK support bigger buffers.

5. Both support full duplex.

6. Implementation loss of less than 0.5 dB is obtained.

7.3 Outstanding Tasks

Outstanding tasks to be performed are:

1. A full-duplex system should be implemented. Each computer should be capable of both modulating and demodulating and be able to switch between the two processes.

2. The application program should be able to adjust the mixer automatically, to control cases of clipping.

7.4 Uses of Modules Outside SDR Architecture

The modules are not restricted to be used only in SDR architecture. Other areas of use are:

- Software modems, especially the development of linmodems (Linux modems). There is a project to turn winmodems into linmodems.

- Wireless local area networks, which can reduce the number of cables running around.

7.5 Final Conclusion

- Modem works.
- ALSA & JACK are best API's to use.

Appendix A

THESIS TOOLS

A.1 Software Used

Linux Distribution	:Fedora Core 1 (Red Hat)
Kernel	:2.4.22-1.2115.nptl
C++ compiler	:gcc 3.3.2
IDE	:KDevelop 2.1.5 & Kylix 3
Profiler	:Kprof & gprof
Plotter	:gnuplot (interfaced)

Figure A.1 shows the modulator that is used for both Simulation and OSS API.

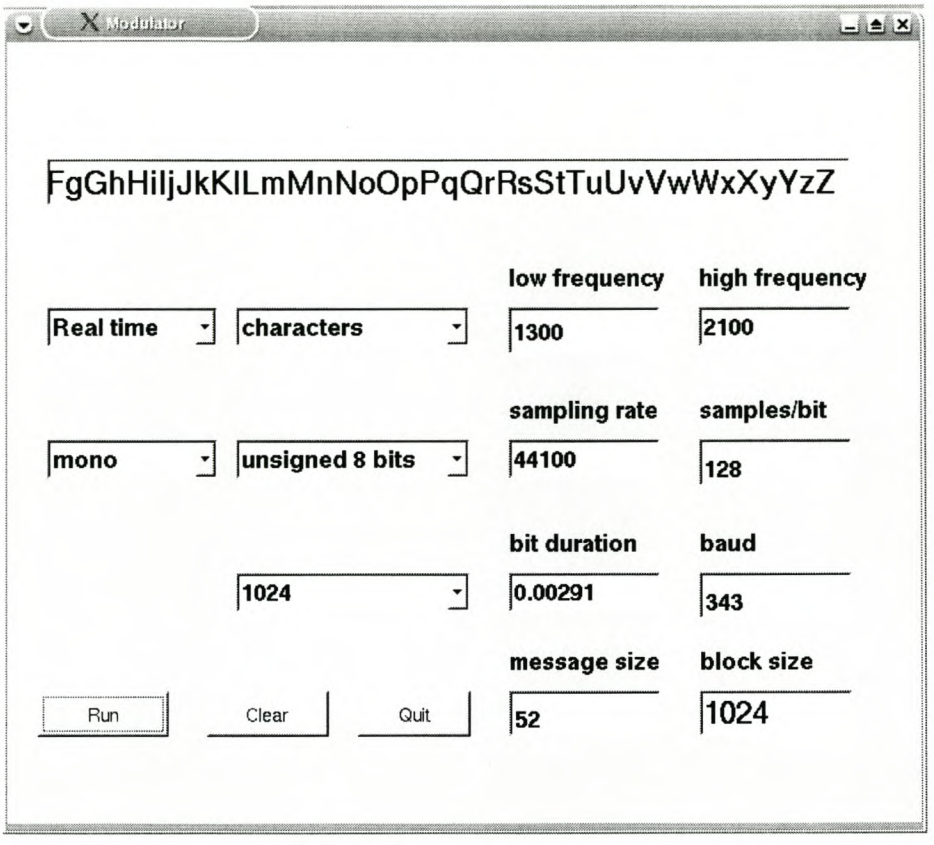

Figure A.1: Modulator figure

A.2 Hardware Used

Two computers with sound cards are used. The two are connected by an audio cable, between the two sound cards. One computer acts as the modulator, while the other one acts as the demodulator.

$\underline{1^{st} PC}$

CPU	:Intel Celeron 850MHz
RAM	:128MB
Sound Card	:Trident, ALI M5451

$\underline{2^{nd} PC}$

CPU	:Intel Celeron 800MHz
RAM	:128MB
Sound Card	:VIA 82C686

Appendix B

DISCRETE CONVOLUTION

Convolution is the ordered combination of multiplication followed by summation. Since sampling is involved, discrete convolution is used.

The convolution of the input signal *x(n)* and the impulse response *h(n)* is achieved by reversing the impulse response under the sample of current interest.

The process of performing the ordered multiplication is[3],

t = 0	t = T	t = 2T	t = 3T	⋯	t = (n-1)T	Response due to
x_0h_0	x_0h_1	x_0h_2	x_0h_3	⋯	x_0h_{n-1}	x_0
	x_1h_0	x_1h_1	x_1h_2	⋯	x_1h_{n-2}	x_1
		x_2h_0	x_2h_1	⋯	x_2h_{n-3}	x_2
			x_3h_0	⋯	x_3h_{n-4}	x_3
				⋱	⋮	⋮
					$x_{n-1}h_0$	x_{n-1}

followed by summation...

Column	Sum
1	$y_0 = x_0h_0$
2	$y_1 = x_0h_1 + x_1h_0$
3	$y_2 = x_0h_2 + x_1h_1 + x_2h_0$
4	$y_3 = x_0h_3 + x_1h_2 + x_2h_1 + x_3h_0$
5	$y_4 = x_0h_4 + x_1h_3 + x_2h_2 + x_3h_1 + x_4h_0$
⋮	⋮
n	$y_n = x_0h_{n-1} + x_1h_{n-2} + x_2h_{n-3} + x_3h_{n-4} + \cdots + x_{n-1}h_0$

From the two processes above, the convolved output is given by:

$$y(i) = \sum_{k=0}^{i} x_k h_{i-k} \tag{B.1}$$

and the pseudo code is as follows:

```
m is the number of terms in the impulse response
output[max]
impulse[max]

FOR i ← (max/2) TO ((max/2) + n - 1) STEP 1
{
   output ← 0; (reset sum to zero)
   FOR j ← 1 TO m STEP 1
   {
      output ← output + input[i-j+1] * impulse[j]
   }
}
```

Appendix C

SYSTEM CALLS USED BY OSS

C.1 The *open()* System Call

The *open*() system call is used to gain access to a device file (/dev/dsp) for reading,writing or both [4]. The prototype for *open*() is as follows:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname,int flags)
int open(const char *pathname,int flags, mode_t mode)
```

The flag defines how the file should be opened. Table C.1 lists flags that can be used for opening the device.

Flag	Description
O_RDONLY	open device for capturing only
O_WRONLY	open device for playback only
O_RDWR	open device for both capturing and playback

Table C.1: Flags For The open() Call

C.2 The *ioctl()* System Call

The *ioctl*() system call is a catch all for setting or retrieving various parameters associated with a file or to perform other operation on the file. The prototype for *ioctl*() is as follows:

```
#include <sys/ioctl.h>
int ioctl(int fd,int request, ... )
```

The argument fd is the file descriptor that was returned from the previous *open*() call.

C.3 The *read()* System Call

The *read*() system call is used to read data from the device corresponding to a file descriptor `fd` that was returned from the previous open() call. Figure C.1 shows a conceptual diagram of this process.

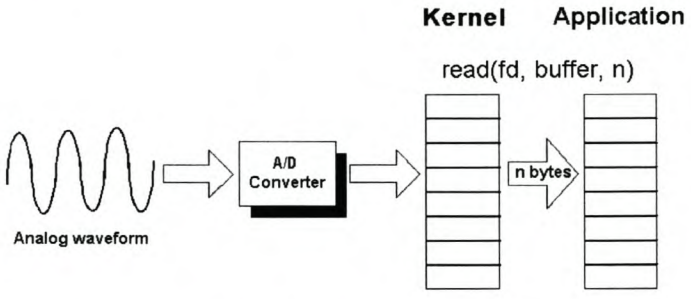

Figure C.1: read() system call

The prototype for *read*() is as follows:

```
#include <unistd.h>
ssize_t read(int fd,void *buf, size_t count)
```

The second argument is a pointer to a buffer to copy data from, and the third argument returns the number of bytes read or a value of -1 if an error occurs.

C.4 The *write()* System Call

The *write*() system call is used to write data to the device corresponding to the file descriptor. Figure C.2 shows a conceptual diagram of this process.

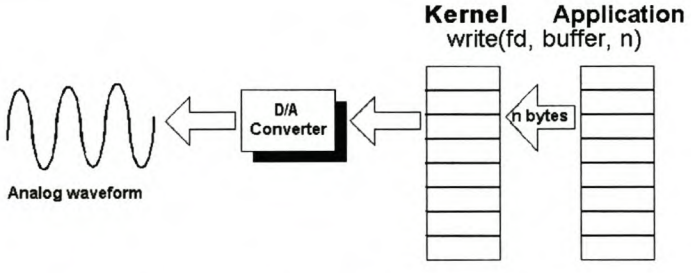

Figure C.2: write() system call

The prototype for *write*() is as follows:

```
#include <unistd.h>
ssize_t write(int fd,void *buf, size_t count)
```

C.5 The *close()* System Call

Any locks held by the process on the device are released, even if they were placed using a different file descriptor.

The prototype for *close*() is as follows:

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
int close(int fd)
```

Appendix D

SOUND CARD

A typical sound card can do the following things:

1. play pre-recorded music (from CDs or sound files, such as wav or MP3), games or DVDs.
2. record audio in various media from external sources (microphone or tape player).
3. synthesise sounds.

D.1 Sound Card Devices

Figure D.1 is a block diagram of an idealised sound card that the Linux sound driver provides [10]. The DAC and ADC provide the means for getting the audio in and out of the sound card while the DSP oversees the process. The DSP also takes care of any alterations to the sound, such as echo or reverb. Because the DSP focuses on the audio processing, the computer's main processor can take care of other tasks.

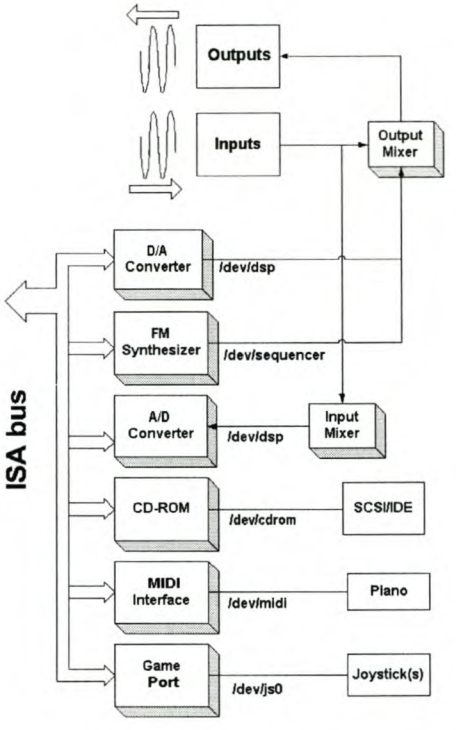

Figure D.1: Sound Card Block Diagram

D.2 Duplexing

A half-duplex sound card is capable of either transmitting or receiving data, but not both, while a full-duplex sound card can transmit data in both directions simultaneously. According to the Free OSS documentation, the Free OSS drivers do not support full duplexing. Only the commercial OSS supports full duplex.
ALSA supports full duplex, and is free [13].

D.3 Sound Card Jitter

One problem of Sound Cards is "jitter" [11]. It is the deviation in or displacement of the sampling rate. If a sound card is recording at a sample rate of 44.1kHz, it does not exactly take one sample every 1/44100 second. There is always a tiny timing error which causes the sample to be slightly too late or early.
The really bad part is that jitter is frequency dependent. Because it is related to the timing of the sample, it can change the recorded frequencies just a little. If it records a sample just a little too fast, the card thinks that the recorded frequency is a little lower than it really is.
Typical jitter-times go between 1.0×10^{-9} and 1.0×10^{-7} seconds.

D.4 Linux Supported Sound Cards

Sound cards supported by the Linux kernel sound driver can be found at [20].

Appendix E

JACK LAYOUT

Figure E.1 is the schematic diagram of JACK.

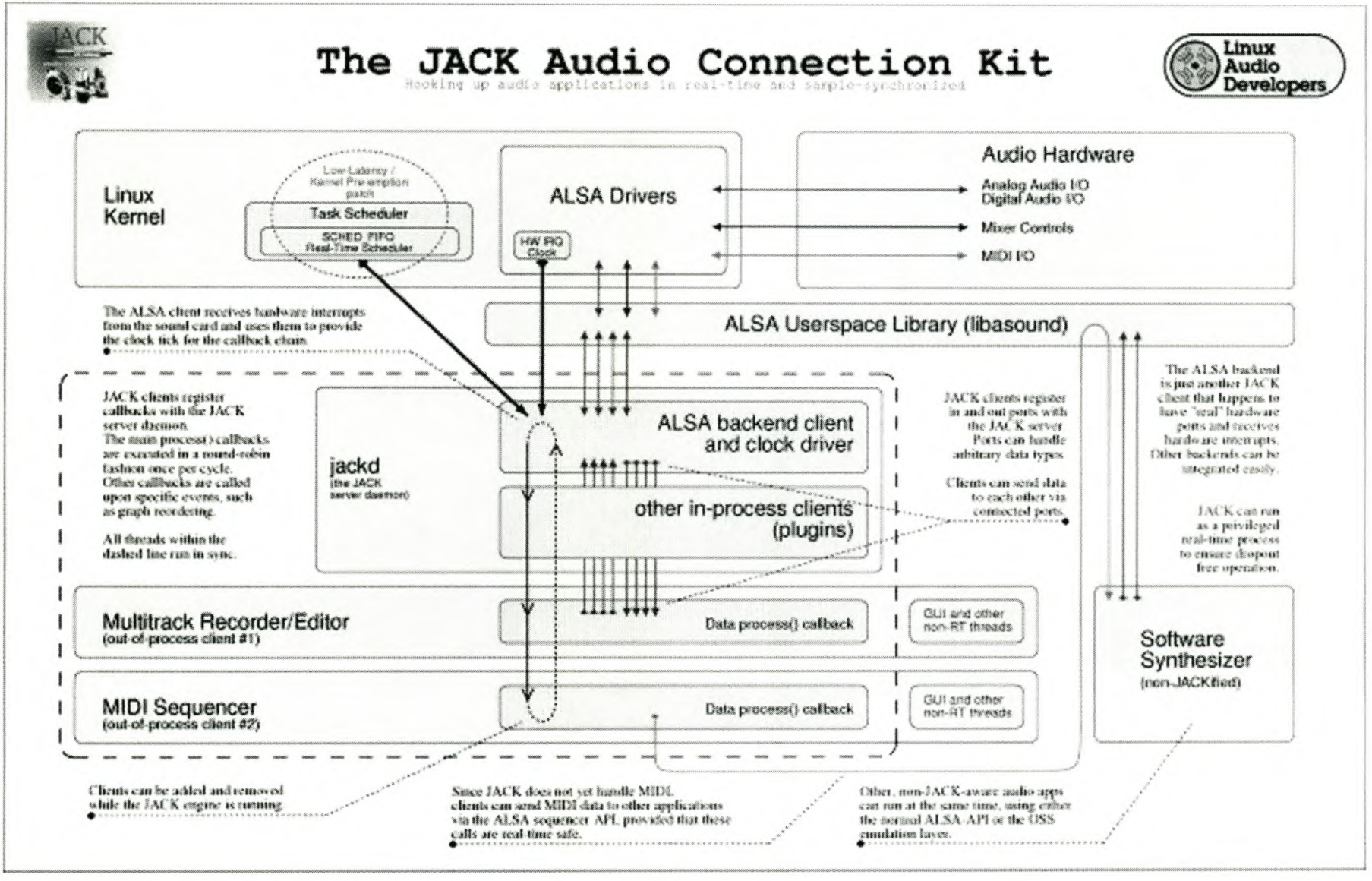

Figure E.1: JACK Diagram

Bibliography

[1] Stremler, F.G. *Introduction to Communication Systems*, Addison-Wessley, third edition, 1990.

[2] Haykin, S. *An Introduction To Analog And Digital Communications*, Wiley, 1989.

[3] Hutchings, H. *Interfacing with C*, Newnes, second edition, 2001.

[4] Wall, K. *Linux Programming Unleashed*, Sams, 1999.

[5] Young, P.H. *Electronic Comunication Techniques*, Prentice Hall, fourth edition.

[6] Shuttleworth, J.*dis-dot-dat.net*, http://dis-dot-dat.net/jacktuts/starting/

[7] Tranter, J. *OSS Progammers Guide*, www.opensound.com/pguide/index.html

[8] *Site for searching RPMs*, http://rpm.pbone.net

[9] *Open Sound Systems*, www.opensound.com/oss.html

[10] Tranter, J. *Linux Multimedia Guide*, O'Reilly & Associates, first edition, 1996.

[11] *Audio Basics*, http://www.hammersound.net/audiobasics/audiobasics.html,

[12] *Linux Modems*, http://linmodems.org,

[13] *Duplexing*, http://www.germane-software.com/SpeakFreely/Duplexing.html

[14] *Installing Jack*, http://jackit.sourceforge.net/docs/,

[15] *JACK Audio Connection Kit*, http://www.joq.us/jack/reference-devel/html

[16] *SDR Open System Architecture*, http://www.ofcom.org.uk/static/archive/ra/topics/research/topics/convergence-new-emerging/sdr/3-moessner.pdf

[17] *PortAudio Tutorial*, http://www.portaudio.com/docs/pa_tutorial.html

[18] *ALSA vs OSS*, http://www.linuxhardware.org/article.pl?sid=01/03/06/179255

[19] *SpeakEasy Military Project*, http://en.wikipedia.org/wiki/Software-defined-radio

[20] *Supported Sound Cards*, http://www.linux.com/howtos/Sound-HOWTO/x96.shtml