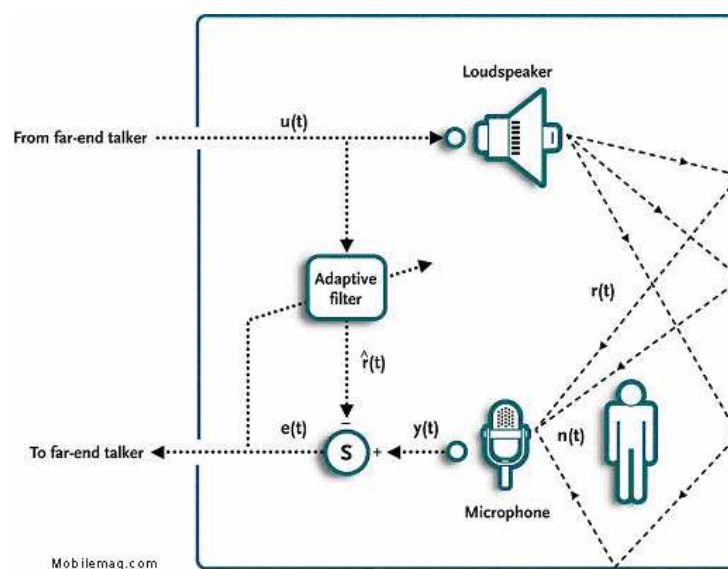


Semesterarbeit  
Digitale Signalverarbeitung

# Acoustic Echo Canceler

Autoren: Martin Rösch  
Berni Imfeld  
Betreuer: Prof. Dr. G. Schuster



## **Zusammenfassung**

Zielsetzung dieser Arbeit war ein adaptives Filter zu implementieren, das das akustische Echo unterdrückt, das bei einem Telefongespräch mit einer Freisprechanlage durch die Einkopplung des Lautsprechersignals in das Mikrofon entsteht.

Anders als bei gebräuchlichen Echo Canceler, die eine Adaption im Zeitbereich berechnen, arbeitet der hier implementierte Algorithmus komplett im Frequenzbereich.

Aufgrund der hohen Rechenkapazität, die auf handelsüblichen PCs zur Verfügung steht, war eine weitere Vorgabe, den Algorithmus in C++ zu implementieren. Die Sprachaufnahme und -wiedergabe sollte mit der DirectSound-Schnittstelle realisiert werden.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
1.1	Aufgabenstellung . . . . .	2
1.2	Problembeschreibung . . . . .	3
<b>2</b>	<b>Theorie</b>	<b>5</b>
2.1	Overlap-Save Filter . . . . .	5
2.2	Adaptive Filter im Zeit- und Frequenzbereich . . . . .	6
2.3	LMS / BLMS . . . . .	7
2.4	PFLMS-Algorithmus von Dr. Pius Estermann . . . . .	8
2.4.1	Übersicht . . . . .	8
2.4.2	Filterung . . . . .	8
2.4.3	Adaption . . . . .	9
2.4.4	Rechenaufwand . . . . .	11
2.4.5	Speicheraufwand für Variablen . . . . .	13
<b>3</b>	<b>Implementation in Matlab</b>	<b>14</b>
3.1	PFLMS . . . . .	14
3.2	BLMS . . . . .	15
<b>4</b>	<b>Implementation in C/C++</b>	<b>16</b>
4.1	Benötigte Software-Bibliotheken . . . . .	16
4.1.1	DirectX/DirectSound . . . . .	16
4.1.2	Die FFTW Bibliothek . . . . .	23
4.2	IP Telefon . . . . .	25
4.2.1	Aufbau der Software . . . . .	25
4.2.2	Das Controller-Actor Design Pattern . . . . .	25
4.2.3	Das Full Duplex Modul . . . . .	26
4.2.4	Das Filter Modul . . . . .	27
4.2.5	Das Net I/O Modul . . . . .	28
4.2.6	Das GUI des IP Telefons . . . . .	29
<b>5</b>	<b>Messungen und Resultate</b>	<b>31</b>
5.1	Erzielte Resultate . . . . .	31
5.2	Benötigte Ressourcen . . . . .	33
5.3	Probleme . . . . .	34
<b>6</b>	<b>Fazit</b>	<b>35</b>
6.1	Aktueller Stand der C++ Implementation . . . . .	35
6.2	Erweiterungen . . . . .	35
6.3	Schlusswort . . . . .	35
6.4	Danksagung . . . . .	35
<b>A</b>	<b>Inhalt der CD</b>	<b>37</b>

# Kapitel 1

## Einleitung

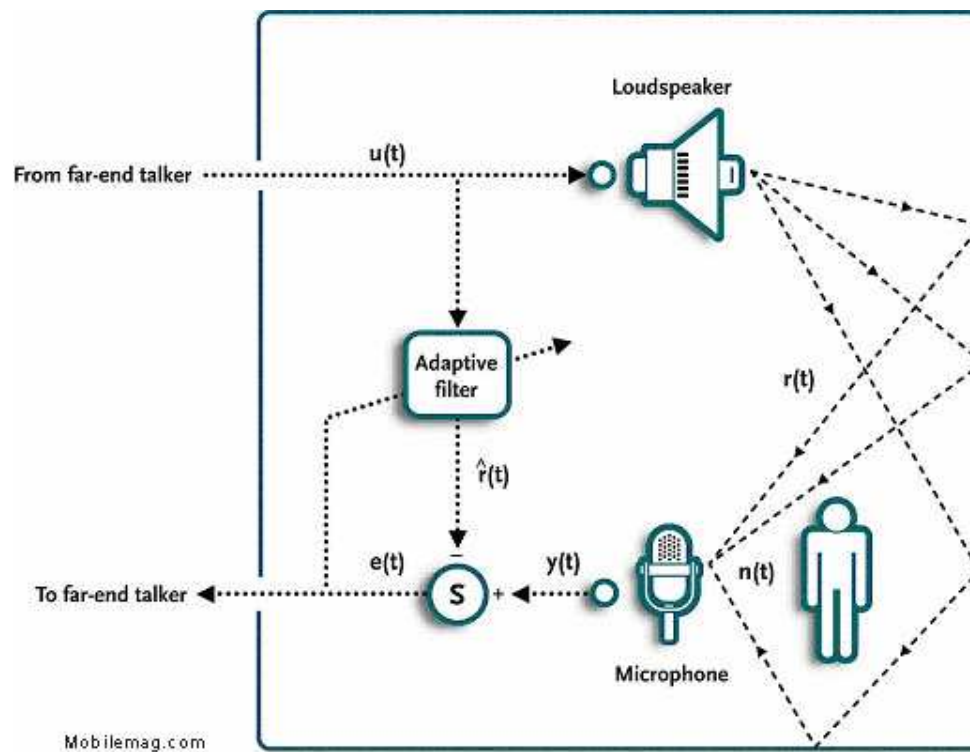
### 1.1 Aufgabenstellung

#### Thema: VoIP Acoustic Echo Cancellation

Studenten: Berni Imfeld, Martin Rösch und Philip Mahler  
Betreuer: Prof. Dr. Guido M. Schuster

#### Kurzbeschreibung

Das Ziel dieser Studienarbeit ist es das akustische Echo einer VoIP Applikation auf einem PC auszulöschen. Mit der letzten Generation wurden handelsübliche PCs leistungsfähig genug, eine Echtzeitauslöschung des akustischen Echos vorzunehmen.



## Aufgabenstellung

- Erarbeitung der erforderlichen Grundlagen
- C/C++ Implementierung des geeigneten Verfahrens auf einem PC

## Erwartete Ergebnisse

- C/C++ Programm auf PC
- Dokumentation der Theorie und der Implementierung

## Arbeitsweise

- Sie führen ein persönliches Laborbuch, wo Sie aufschreiben **wann** Sie **was** für **wie lange** machen und was die Ergebnisse sind.
- Sie schicken mir vor jedem Treffen eine Agenda und nach dem Treffen ein Protokoll.

## 1.2 Problembeschreibung

Beim Telefonieren mit einer Freisprecheinrichtung kann der entfernte Gesprächspartner (Far-End Speaker) mit einer Zeitverzögerung ein Echo hören, das (je nach Lautstärke und Verzögerung) unangenehm ist. Das Echo entsteht dadurch, dass sich das vom Far-End Speaker gesendete Signal durch die Lautsprecher bei seinem Gesprächspartner (Near-End Speaker) im Raum ausbreitet, durch direkte Einkopplung und mehrfache Reflexionen (an Wänden und Objekten im Raum) wieder vom Mikrophon aufgenommen wird und so wieder zum Far-End Speaker zurückgesendet wird. Der eben beschriebene Echo-Pfad ist in Abbildung 1.1 dargestellt. Das Echo wird umso störender

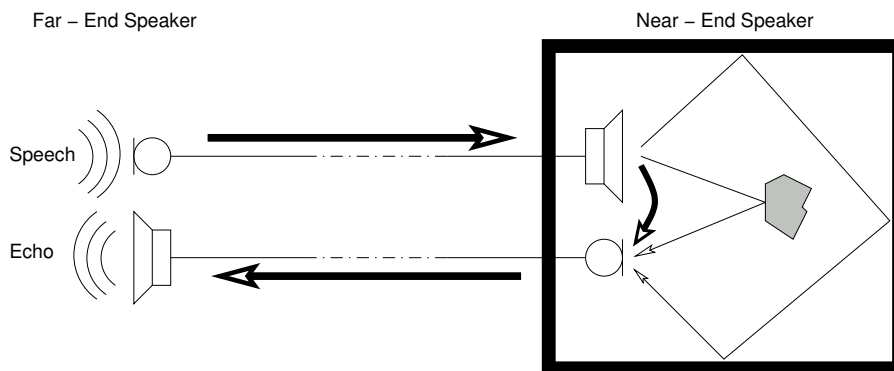


Abbildung 1.1: Echo-Pfad

empfunden, je lauter das Echo ist und je grösser die Zeitverzögerung zwischen dem Signal und dem Echo ist. Der Zusammenhang zwischen Echolautstärke und Zeitverzögerung ist multiplikativ, d.h. ein lautes, wenig verzögertes Echo ist gleich störend wie ein leises, stark verzögertes Echo. Gerade bei VoIP<sup>1</sup>-Anwendungen können grosse Verzögerungen auftreten, da die Signale durch ein Netzwerk geschickt werden.

Signaltheoretisch betrachtet, kann der Raum beim Near-End Speaker durch ein LTI<sup>2</sup>-System beschrieben werden, welches das Lautsprechersignal  $x(t)$  verzerrt. Dessen Impulsantwort  $h(t)$  ist

<sup>1</sup> Voice over IP

<sup>2</sup> Linear TimeInvariant

abhängig von der Reflexionswirkung der Wände, der Anzahl Echo-Pfade und deren Länge. Das Mikrofonsignal  $y(t)$  kann im Zeitbereich mathematisch durch

$$y(t) = x(t) * h(t) = \int x(t - \tau)h(\tau)d\tau \quad (1.1)$$

beschrieben werden.

Genau betrachtet ist dieses System aber nicht zeitinvariant, da sich die Impulsantwort durch sich bewegende Objekte im Raum oder durch Veränderung der Lausprecher-Mikrofon Anordnung ändern kann. Die Änderungen sind aber genügend langsam, sodass die Impulsantwort über einen Zeitabschnitt als konstant betrachtet werden kann.

# Kapitel 2

## Theorie

### 2.1 Overlap-Save Filter

Da ein normales Filterverfahren im Frequenzbereich zu einer zyklischen Faltung im Zeitbereich führt, wird ein erweitertes Verfahren benötigt: das ‘Overlap-Save’ Verfahren.

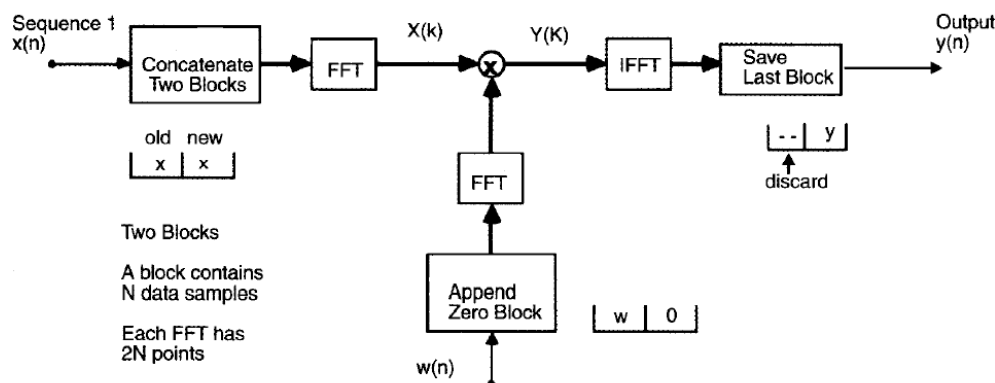


Abbildung 2.1: Aufbau eines Overlap-Save Filters (Quelle: [SHY92] Seite 20)

Es werden jeweils zwei Blöcke des Eingangssignals zusammengefasst. Der erste Block entspricht dem alten Datenblock, der Zweite dem neuen. Bei der Impulsantwort steht die erste Hälfte für die Filterkoeffizienten, die zweite Hälfte muss 0 sein. Diese beiden Signale werden jetzt fouriertransformiert und anschliessend multipliziert. Das Ausgangssignal wird zurücktransformiert; die zweite Hälfte entspricht dem Ausgangssignal. Die erste Hälfte enthält zum Teil Resultate der zyklischen Faltung und muss daher vernachlässigt werden. Wenn der nächste Datenblock ankommt, wird der bis anhin neue Datenblock zum alten Block, der ankommende Block wird zum neuen. Eine genauere Beschreibung des Verfahrens kann in [SHY92] gefunden werden.

Durch dieses Verfahren wird erreicht, dass keine Überlappung des zyklischen Anteils mit den Nutzdaten des Ausgangssignals entsteht. Verglichen mit dem Verfahren ohne Overlap steigt der Rechenaufwand an, da FFTs der doppelten Grösse benötigt werden um gleichviele Samples  $N$  zu verarbeiten. Geht man davon aus, dass sich die Impulsantwort jedes mal ändert, werden jeweils drei FFTs benötigt; ansonsten nur zwei.

Eine FFT braucht

$$\frac{C}{2} \log_2 C \quad (2.1)$$

komplexe Multiplikationen und

$$C \log_2 C \quad (2.2)$$

komplexe Additionen (Quelle: [ZT94] Seite 148). Der Rechenaufwand in komplexen Operationen pro Block ist in Tabelle 2.1 zu finden.

Rechenaufwand	ohne Overlap-Save	mit Overlap-Save
FFT-Grösse	$N$	$2N$
Anzahl FFTs	$3$	$3$
Multiplikationen	$(N/2) \log_2 N$	$N \log_2 (2N)$
Additionen	$N \log_2 N$	$2N \log_2 (2N)$
zusätzliche Multiplikationen	$N$	$2N$

Tabelle 2.1: Rechenaufwand beim Overlap-Save Verfahren

Der Rechenaufwand an Multiplikationen steigt somit um den Faktor:

$$\frac{3 \cdot N \log_2 (2N) + 2N}{3 \cdot \frac{N}{2} \log_2 N + N} = \frac{6(\log_2 N + 1) + 4}{3 \log_2 N + 2} = 2 \cdot \frac{\log_2 N + \frac{5}{3}}{\log_2 N + \frac{2}{3}}$$

Der Rechenaufwand an Additionen steigt um den Faktor:

$$\frac{3 \cdot 2N \log_2 (2N)}{3 \cdot N \log_2 N} = 2 \frac{\log_2 N + 1}{\log_2 N}$$

Bei grösseren Blöcken betragen die Faktoren ungefähr 2.

Umgerechnet in reelle Operationen ergibt dies einen Aufwand pro Sample von:

$$M_{reell} = \frac{4 \cdot (N * \log_2 (2N) + 2N)}{N} = 4(\log_2 (2N) + \frac{1}{2}) \quad (2.3)$$

$$A_{reell} = \frac{2(2 + 1)N * \log_2 (2N) + 2 \cdot 2N}{N} = 6 \left( \log_2 (2N) + \frac{2}{3} \right) \quad (2.4)$$

(Die Umrechnungen von komplexen in reelle Operationen befindet sich in Tabelle 2.5.)

Bei einer Faltung werden pro Sample  $N$  reelle Multiplikationen und Additionen benötigt. Der Vergleich bei einer Blockgrösse von  $N = 1024$  ergibt, dass das ‘Overlap-Save’ Verfahren rund 20 mal schneller ist, als eine Faltung im Zeitbereich.

Vergleich bei $N = 1024$	Faltung	Overlap-Save
reelle Multiplikationen	1024	46
reelle Additionen	1024	70

Tabelle 2.2: Vergleich: Faltung / Overlap-Save

Das ‘Overlap-Save’ Verfahren ist somit besonders gut geeignet, wenn die Verarbeitung ohnehin blockweise erfolgen muss. Ansonsten bezahlt man den kleineren Rechenaufwand mit einer zusätzlichen Verzögerung von rund einem Block.

## 2.2 Adaptive Filter im Zeit- und Frequenzbereich

In diesem Projekt werden adaptive Filter eingesetzt, um die Übertragungsfunktion zwischen dem Lautsprecher- und dem Mikrofonsignal zu finden. Die Filterung und die Adaption kann im Zeitbereich (MSE<sup>1</sup>, LMS<sup>2</sup>, BLMS<sup>3</sup>) oder auch im Frequenzbereich (PFLMS) vorgenommen werden. In den folgenden 2 Unterkapiteln wird je ein Verfahren genauer erklärt. Weitere Informationen zu adaptiven Filter im Zeitbereich können auch aus dem Script von A. Schüeli [SCH02] bezogen werden.

<sup>1</sup>MSE = Mean Squared Error

<sup>2</sup>LMS = Least Mean Squares

<sup>3</sup>BLMS = Block LMS

## 2.3 LMS / BLMS

Der klassische Ansatz für adaptive Filter ist die Minimierung des quadratischen Fehlers ( $\rightarrow MSE$ ). Bei diesem Verfahren wird der quadratische Fehler als Funktion der Impulsantwort und des Eingangssignals dargestellt.

$$Q = [e(i)]^2 = \left[ r(i) - \sum_{k=0}^N c_k s(i-k) \right]^2 \quad (2.5)$$

Wobei  $r$  das Referenzsignal,  $s$  das Eingangssignal und  $c_k$  die Filterkoeffizienten sind. Um das globale Minimum der Funktion  $Q$  zu finden, werden die Ableitungen nach den einzelnen Filterkoeffizienten nullgesetzt. Dies führt zu folgendem Gleichungssystem:

$$\frac{\partial Q}{\partial c_n} = -2\overline{e(i)s(i-n)} = 0 \quad , \quad n = 0, 1, \dots, N \quad (2.6)$$

Die direkte Lösung dieser Gleichung wird als ‘Least Squares’-Verfahren bezeichnet. Da die direkte Lösung oft zuviele Ressourcen benötigt, wird häufig der LMS-Algorithmus eingesetzt. Anstatt gleich die ganze Funktion zu minimieren wird jeweils nur ein Schritt entgegen dem Gradienten gemacht. Dies führt dazu, dass das Minimum schrittweise gefunden wird. Durch eine Normalisierung der Schrittgröße bezüglich der Leistung des Eingangssignals erreicht man einen NLMS<sup>4</sup>-Algorithmus.

Beim BLMS-Verfahren wird die Filterung sowie die Adaption blockweise durchgeführt. Das Vorgehen bleibt das Gleiche wie beim LMS. Der Rechenaufwand wird dadurch reduziert, dass die Adaption nicht mehr nach jedem Sample, sondern nur noch nach jedem Block vorgenommen wird. In Tabelle 2.3 wird der Rechenaufwand für den BNLS-Algorithmus analysiert. Die Variable  $N$  bezeichnet die Blockgröße. Dies entspricht auch der Länge der Impulsantwort, sowie der Anzahl Ausgangs-Samples, die pro Block generiert werden.  $M$ ,  $A$  und  $D$  stehen für Multiplikation, Addition und Division.

Art der Operation	Anzahl	Rechenschritt / Kommentar
<b>Filtern:</b>		
$M_{reell}$	$N^2$	$N$ Eingangswerte à $N$ Filterkoeffizienten
$A_{reell}$	$N^2$	Aufsummierung
<b>Adaptieren:</b>		
$M_{reell}$	$N^2$	$N$ Koeffizienten à $N$ Eingangswerte
$A_{reell}$	$N^2$	Aufsummierung
$D_{reell}$	$N$	Leistungsnormierung
<b>Total:</b>		
$M_{reell}$	$2N^2$	
$A_{reell}$	$2N^2$	
$D_{reell}$	$N$	

Tabelle 2.3: Rechenaufwand beim ‘Overlap-Save’ Verfahren

### Aufwand pro Sample

Länge der Impulsantwort:  $N$

$$M_{reell} = \frac{2N^2}{N} = 2N \quad (2.7)$$

$$A_{reell} = \frac{2N^2}{N} = 2N \quad (2.8)$$

<sup>4</sup>Normalized LMS

$$D_{reell} = \frac{N}{N} = 1 \quad (2.9)$$

$$(2.10)$$

Eine weitere Reduzierung des Aufwands kann durch Filtern im Frequenzbereich (z.B. Overlap-Save) erreicht werden( $\rightarrow$  *FastBLMS*).

## 2.4 PFLMS-Algorithmus von Dr. Pius Estermann

### 2.4.1 Übersicht

Der PFLMS<sup>5</sup> -Algorithmus ist ein adaptives Filter das Eingangsdaten blockweise filtert und die Filterkoeffizienten adaptieren kann.

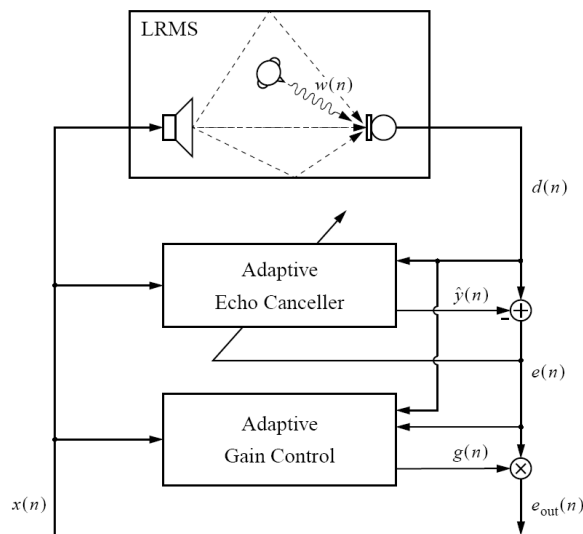


Abbildung 2.2: Aufbau eines Echo-Cancelers  
Quelle: [EST96]

Als Eingangssignale dienen  $x(n)$  (Signal an den Lautsprechern) und  $d(n)$  (Signal beim Mikrofon). Das Filter erstellt mit der geschätzten Impulsantwort  $\hat{h}(n)$  aus  $x(n)$  die Schätzung  $\hat{y}(n)$  des Mikrofonsignals. Die Differenz zwischen dem Mikrofonsignal und der Schätzung entspricht dem Fehler  $e(n)$ . Die Adaption wird aus dem Lautsprechersignal und dem Fehler berechnet. Die Adaptive Verstärkung (Adaptive Gain Control) des Signals wurde in dieser Arbeit nicht implementiert und erklärt.

Die folgenden Gleichungen stammen alle aus der Publikation [EST96] von P. Estermann. Jedoch wurden einige in der Schreibweise abgeändert, um sie besser verständlich zu machen. Die Vektoren werden jeweils elementweise multipliziert.

### 2.4.2 Filterung

Der Filter verwendet das ‘Overlap-Save’-Verfahren, das bereits in Kapitel 2.1 beschrieben wurde. Speziell beim PFLMS ist, dass die Impulsantwort in einzelne Segmente aufgeteilt wird. Das Ausgangssignal berechnet sich folgendermassen( $\hat{Y}$ ,  $X$  und  $\hat{H}$  entsprechen jeweils den Signalen im

<sup>5</sup>PFLMS steht für Partitioned Frequency-Domain Least Mean Square

Frequenzbereich.):

$$\hat{Y}[k] = \sum_{l=0}^{L-1} X[k-lm]\hat{H}[k] \quad (2.11)$$

Dadurch kann die Länge der Impulsantwort vergrössert werden. Die Verzögerung, die durch das blockweise Verarbeiten entsteht, bleibt jedoch konstant. Beim Filtern entsteht kein grosser Mehraufwand, da gleichviele DFT und IDFT Operationen benötigt werden. Lediglich die Additionen im Frequenzbereich vor der Rücktransformation kommen hinzu.

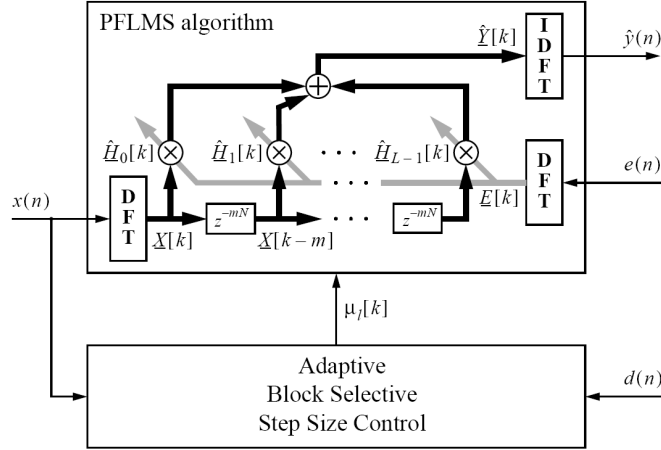


Abbildung 2.3: Schema des PFLMS Echo-Cancelers  
Quelle: [EST96]

### 2.4.3 Adaption

Die folgenden Erklärungen berücksichtigen nur den Fall wenn  $m = 1 \rightarrow C/N = 2$ , wobei  $C$  die Blockgrösse und  $N$  die Anzahl Samples pro Block bezeichnet.

Ebenso wie die Filterung wird auch die Adaption im Frequenzbereich berechnet. Der Fehler berechnet sich wie folgt:

$$E = \mathcal{F}\left(\text{ZeroOne} \cdot \mathcal{F}^{-1}(D[k] - \hat{Y}[k])\right) \quad (2.12)$$

$\mathcal{F}$  und  $\mathcal{F}^{-1}$  stehen für Fouriertransformation und -rücktransformation. Die erste Hälfte des Fehlers im Zeitbereich wird durch die Maske *ZeroOne* 0 gesetzt. Dies ist bedingt durch das ‘Overlap-Save’-Verfahren, bei dem nur die zweite Hälfte des Ausgangsvektor relevant ist.

Der Kern der Adaption ist durch folgende Gleichungen gegeben:

$$d\hat{H}_l[k] = \mathcal{F}\left(\text{OneZero} \cdot \mathcal{F}^{-1}(\mu_l[k]X^*[k-lm]E[k])\right) \quad (2.13)$$

$$\hat{H}_l[k+1] = \hat{H}_l[k] + d\hat{H}_l[k] \quad (2.14)$$

Das Lautsprechersignal wird mit dem Fehlervektor multipliziert. Dadurch werden die gemeinsamen Frequenzen gefunden. Von  $X$  muss der konjugiert-komplexe Wert genommen werden, um die Phase des Koeffizienten der Impulsantwort zu bestimmen. Die Maske *OneZero* setzt die Koeffizienten der Impulsantwort in der zweiten Hälfte auf 0.

$\mu_l[k]$  ist der Schrittgrössenvektor der sowohl frequenz- als auch segmentabhängig ist. Er ist folgendermassen definiert:

$$\mu_l[k] = \begin{cases} \eta_l[k]s_l[k] \cdot P_{X,j}^{-1}[k-lm] & \text{für } P_{X,j}[k-lm] > c_1\bar{P}_{X,j}[k-lm] \\ 0 & \text{sonst} \end{cases} \quad (2.15)$$

Der Faktor  $\eta_l[k]$  ist die segmentabhängige Schrittgrösse, der Schaltfaktor  $s_l[k]$  wird in Gleichung 2.18 erklärt. Die Leistungsmittelungen  $P_{X,j}[k-lm]$  und  $\bar{P}_{X,j}[k-lm]$  berechnen sich wie folgt:

$$P_{X,j}[k] = (1 - \gamma_1)|X_j[k]|^2 + \gamma_1 P_{X,j}[k-1] \quad , \quad 0 \leq j < C \quad (2.16)$$

Die Leistung  $P_X$  wird pro Frequenzband mit einem IIR-Filter 1. Ordnung gemittelt.

$$\bar{P}_{X,j}[k] = \begin{cases} (1 - \gamma_2)|X_j[k]|^2 + \gamma_2 \bar{P}_{X,j}[k-1] & \text{für } \bar{P}_{X,j}[k] > P_{X,j}^{(min)} \\ P_{X,j}^{(min)} & \text{sonst} \end{cases} \quad (2.17)$$

Die Leistung  $\bar{P}_X$  wird auch pro Frequenzband gemittelt, jedoch gilt:  $0 < \gamma_1 < \gamma_2 < 1$ . Die Leistung  $\bar{P}_{X,j}[k]$  ist somit eine Mittelung über eine längere Zeitperiode, während  $P_{X,j}[k]$  einer Kurzzeitmittelung entspricht.

Die Adaption eines Frequenzbandes wird somit ausgeschaltet, wenn die Kurzzeitmittelung unter den Wert der Langzeitmittelung multipliziert mit  $c_1$  sinkt. Der Wert der Langzeitmittelung ist nach unten auf  $P_{X,j}^{(min)}$  beschränkt. Dies stoppt die Adaption wenn eine Frequenz länger nicht vorhanden war.

Um bei Sprechpausen und Double-Talk die Adaption zu stoppen, wird ein segmentabhängiger Schalter  $s_l[k]$  eingebaut.

$$s_l[k] = \begin{cases} 1 & \text{für } P_x[k-lm] > c_2 \bar{P}_X[k-lm] \cap P_x[k-lm] > r_l P_d[k] \\ 0 & \text{sonst} \end{cases} \quad (2.18)$$

$s_l[k]$  führt dazu, dass  $\eta_l[k]$  auf Null gesetzt wird. Dies stoppt die Adaptierung des ganzen Segments  $l$ .

$P_x[k]$  und  $P_d[k]$  den Leistungen des aktuellen Lautsprecher- bzw. Mikrofonsignals entsprechen.

$$P_x[k] = \sum_{i=N+1}^C x[i]^2 \quad (2.19)$$

$$P_d[k] = \sum_{i=N+1}^C d[i]^2 \quad (2.20)$$

Der Koeffizient  $c_2$  bestimmt wieviel grösser die momentane Leistung  $P_x$  sein muss als die gemittelte Leistung  $\bar{P}_x$ , damit die Adaption eingeschaltet wird.

$$\bar{P}_x[k] = \begin{cases} (1 - \gamma_3)P_x[k] + \gamma_3 \bar{P}_x[k-1] & \text{für } \bar{P}_x[k] > P_x^{(min)} \\ P_x^{(min)} & \text{sonst} \end{cases} \quad (2.21)$$

$\bar{P}_x[k]$  entsteht wiederum durch eine Mittelung mit einem IIR-Filter 1. Ordnung und ist nach unten auf einen Schwellwert begrenzt. Die erste Bedingung der Gleichung 2.18 dient der Erkennung der Sprechpausen; die zweite Bedingung detektiert Double-Talk. Hierfür ist noch der Koeffizient  $r_l$  notwendig.

$$r_l = \frac{c_4}{P_{h_l}[0]} \quad , \quad 0 \leq l < L \quad (2.22)$$

$P_{h_l}[0]$  entspricht der Leistung die in diesem Segment der Impulsantwort zu erwarten ist. Wenn mehr Leistung zurückkommt als erwartet, schaltet der Algorithmus das entsprechende Segment aufgrund von Double-Talk ab. Da am Anfang nicht genau bekannt ist, wieviel Leistung im entsprechenden Segment zurückkommt, wurde der Koeffizient  $c_4$  eingeführt. Dieser bestimmt das Toleranzband, in dem die Leistung des Mikrofonsignals sich befinden kann, ohne dass Double-Talk detektiert wird. Die Veränderung der Schrittweite aufgrund der Korrelation des Eingangssignals mit dem Mikrofonsignal wurde weggelassen, da sie in [EST96] zu ungenau beschrieben ist und wahrscheinlich zu einem erheblich grösseren Rechenaufwand führt.

## Bemerkungen

Viele der Koeffizienten müssen experimentell gewählt werden, und oft kann man nicht eindeutig eine beste Variante finden. In Tabelle 2.4 sind einige Kriterien zur Bestimmung der Koeffizienten aufgelistet.

Koeffizient	Bereich	Zweck	Folgen wenn: zu klein	... zu gross
c1	[0, 1]	minimale Leistung einer Frequenz	Adaption von zu leisen Frequenzen	zu seltene Adaption
c2	[0, 1]	minimale Leistung eines Segments	Adaption bei zu leisem Signal	zu seltene Adaption
c4	[0, 1]	Double-Talk Detektion	keine Detektion, da Sicherheitsabstand zu gross	zu oft Double-Talk, da Sicherheitsabstand zu klein
$\gamma_1$	[0, $\gamma_2$ ]	Leistungsmittelung	kein Mittelungseffekt	System reagiert zu langsam
$\gamma_2$	[ $\gamma_1$ , 1]	Leistungsmittelung	kein Mittelungseffekt	System reagiert zu langsam
$\gamma_3$	[0, 1]	Leistungsmittelung	kein Mittelungseffekt	System reagiert zu langsam
$P_{X,j}^{(min)}$	Signalabhängig	minimale Leistung einer Frequenz	Adaption von zu leisen Frequenzen	Leistungsschwelle wird nicht erreicht
$P_x^{(min)}$	Signalabhängig	minimale Leistung eines Segments	Adaption bei zu leisem Signal	Leistungsschwelle wird nicht erreicht
$P_{h_l}[0]$	[0, 1]	erwartete Leistung eines Segments der Impulsantwort	zu oft Double-Talk	zu selten Double-Talk

Tabelle 2.4: Kriterien zur Wahl der Koeffizienten

## 2.4.4 Rechenaufwand

Der Rechenaufwand um  $N$  Samples zu Filtern und die  $L$  Segmente des Filters zu adaptieren wird in Tabelle 2.6 analysiert. Die benützten ‘Wechselkurse’ sind in Tabelle 2.5 aufgelistet.

	entspricht	plus
FFT = IFFT	$(C/2) \log_2 C M_{komplex}$	$C \log_2 C A_{komplex}$
komplexe Multiplikation	$4 M_{reell}$	$2 A_{reell}$
komplexe Addition	$2 A_{reell}$	
reelle Zahl · komplexe Zahl	$2 M_{reell}$	
Vergleich, Subtraktion	$1 A_{reell}$	

Tabelle 2.5: Wechselkurse für Rechenaufwand-Analyse

Die verwendeten Variablen sind:

- $C$ : Blockgrösse
- $N$ : Anzahl Samples pro Block ( $C = 2N$ )
- $L$ : Anzahl Segmente

Art der Operation	Anzahl	Rechenschritt / Kommentar
<b>Filtern:</b>		
FFT	1	$x \rightarrow X$
$M_{komplex}$	$LN$	$Y = X \cdot H$
$A_{komplex}$	$(L - 1)N$	Aufsummierung
IFFT	1	$Y \rightarrow y$
$A_{reell}$	$N$	$e = d - y$
<b>Adaptieren:</b>		
FFT	1	$e \rightarrow E$
$M_{komplex}$	$LN$	$X \cdot E$
$M_{reell}$	$2LN$	$\mu \cdot (XE)$
FFT	$2L$	$dH$ im Zeitbereich Maskieren
$A_{komplex}$	$LN$	$H = H + dH$
<b>Steuerung:</b>		
$D_{reell}$	$LN$	$\mu = \eta / P_X$
$M_{reell}$	$LN$	$c_1 \bar{P}_X$
$A_{reell}$	$LN$	$P_X > c_1 \bar{P}_X$
$A_{reell}$	$LN$	$\bar{P}_X < P_X^{min}$
$P_X, \bar{P}_X$ berechnen:		
$M_{reell}$	$2LN$	$X^2 \quad Re(X) \cdot Re(X) + Im(X) \cdot Im(X)$
$A_{reell}$	$LN$	
$M_{reell}$	$2LN$	$\gamma_1 \cdot P_{X,alt} + (1 - \gamma_1) \cdot P_{X,neu}$
$A_{reell}$	$LN$	
$M_{reell}$	$2LN$	$\gamma_2 \cdot \bar{P}_{X,alt} + (1 - \gamma_2) \cdot P_{X,neu}$
$A_{reell}$	$LN$	
$M_{reell}$	$2N$	$P_x, P_d$ berechnen
$A_{reell}$	$2N$	Aufsummierung
<b>Total:</b>		
FFT	$2L + 3$	$\rightarrow (C/2) \log_2 C \quad M_{komplex} + C \log_2 C \quad A_{komplex}$
$M_{komplex}$	$2LN$	$\rightarrow 4M_{reell} + 2A_{reell}$
$M_{reell}$	$9LN + 2N$	
$A_{komplex}$	$2LN - N$	$\rightarrow 2A_{reell}$
$A_{reell}$	$5LN + 3N$	
$D_{reell}$	$LN$	

Tabelle 2.6: Rechenaufwand-Analyse

$M$ ,  $A$  und  $D$  stehen für Multiplikation, Addition und Division. Umgerechnet in reelle Operationen ergibt dies:

$$M_{reell} = 4(2LN + (2L + 3) \frac{C}{2} \log_2 C) + 9LN + 2N = (4L + 6)C \log_2 C + 17LN + 2N$$

mit  $C = 2N$

$$M_{reell} = (8LN + 12N)(\log_2 N + 1) + 17LN + 2N = (8LN + 12N) \log_2 N + 25LN + 14N$$

$$A_{reell} = 2(2LN - N + \frac{3}{2}(2L + 3)C \log_2 C) + 5LN + 3N + 2(2LN) = (6L + 9)C \log_2 C + 13LN + N$$

mit  $C = 2N$

$$A_{reell} = (12LN + 18N)(\log_2 N + 1) + 13LN + N = (12LN + 18N)\log_2 N + 25LN + 19N$$

$$D_{reell} = LN$$

Dies führt zu einem Aufwand pro Sample von:

$$M_{reell} = (8L + 12)\log_2 N + 25L + 14 \quad (2.23)$$

$$A_{reell} = (12L + 18)\log_2 N + 25L + 19 \quad (2.24)$$

$$D_{reell} = L \quad (2.25)$$

$$(2.26)$$

### Vergleich BNLMS - PFLMS: Aufwand pro Sample

$N = 1024$ ,  $L = 1 \rightarrow$  Länge der Impulsantwort: 1024

Art der Operation	BNLMS	PFLMS
$M_{reell}$	2048	239
$A_{reell}$	2048	344
$D_{reell}$	1	1

Tabelle 2.7: Vergleich des Rechenaufwands

Verglichen mit dem BNLMS-Algorithmus (Gleichung: 2.8 bis 2.10) ist der PFLMS-Algorithmus rund 8 mal schneller; dabei ist die Steuerung der Adaption bereits inbegriffen.

### 2.4.5 Speicheraufwand für Variablen

Der benötigte Speicher ist in Tabelle 2.8 nur grob abgeschätzt da er stark von der Implementation abhängt.

Variable	$d$	$e$	$h$	$x$	$y$	$E$	$H$	$X$	$Y$	$P_X$	$\bar{P}_X$	$\mu_t$
Anzahl	$2N$	$2N$	$2N$	$2N$	$2N$	$2N$	$2LN$	$2LN$	$2N$	$2LN$	$2LN$	$2LN$

Tabelle 2.8: Speicheraufwand

Dies ergibt total rund  $10LN + 14N$  Werte die gespeichert werden müssen. Bei  $N = 1024$ ,  $L = 2$  und dem Double-Datentyp à 10 Byte ergibt dies  $(10LN + 14N) \cdot 10\text{Byte} = 340\text{kByte}$

# Kapitel 3

## Implementation in Matlab

Die Matlab-Programme haben die Aufgabe eine Sound-Datei abzuspielen und gleichzeitig aufzunehmen oder die Aufnahme mittels einer simulierter Impulsantwort zu berechnen. Bei der Aufnahme wird dann mit dem entsprechenden Algorithmus das Echo so gut wie möglich ausgelöscht. Bei der Simulation besteht der Vorteil, dass weniger Störeinflüsse vorhanden sind; zudem kann man die Fehlanpassung besser messen. Sämtliche Programme sind auf der CD vorhanden.

### 3.1 PFLMS

Nachfolgend sind die wichtigsten Programmteile der PFLMS-Implementation kurz beschrieben:

- Abspielen / Aufnehmen: (nur bei PFLMSreal.m)  

```
soundsc(x,Fs);  
z1 = wavrecord(len,Fs,'double');  
z(450:len)=z1(1:len-449);
```

Hierbei ist zu beachten, dass der Zeitpunkt des Abspielens nicht mit dem Zeitpunkt der Aufnahme synchronisiert ist. Um dies zu erreichen, wird nachfolgend die Aufnahme noch um einen empirisch ermittelten Wert nach hinten geschoben. Bei erstmaligem Abspielen ist die Verzögerung oft anders, deshalb lohnt es sich, zuerst ein zweites Mal abzuspielen und erst dann den Wert zu ermitteln.

**Wichtig:**

In der Matlab-Datei playsnd.m müssen die beiden Zeilen:

```
fwrite(fp,lin2mu(y),'uchar');
```

durch die Zeile:

```
fwrite(fp,y,'int16');
```

ersetzt werden. Ansonsten ist die Qualität beim Abspielen nicht gut genug und der PFLMS-Algorithmus funktioniert nicht.

- Filtern:  

```
for i2 = 1:L  
    Yd = Yd + X(i2,:) .* H(i2,:);  
end
```

Aus den Eingangssignalen X und den Teil-Impulsantworten H wird das Ausgangssignal Yd berechnet. Dies entspricht der Schätzung des Mikrofonsignals.

- Berechnung des Fehlers:  

```
Yi = Zi - Yd;  
yi = real(iff(Yi));  
E = fft(Z0 .* yi);
```

Um den Fehler zu berechnen wird die Schätzung Yd vom Mikrofonsignal Zi abgezogen. Da

der Fehler nur in der zweiten Hälfte des Blocks relevant ist, muss dieser noch im Zeitbereich maskiert werden. Dies geschieht mit dem Vektor Z0, der in der ersten Hälfte Nullen und in der Zweiten Einer enthält.

- Adaption:
 

```
H(i2,:) = H(i2,:) + fft(OZ.*(ifft(conj(X(i2,:)).*E.*mu(i2,:))));
```

 Diese Zeile berechnet aus dem Eingangssignal X, dem Fehler E und dem Schrittgrößenvektor mu die Änderung der Impulsantwort. Da die Impulsantwort in der zweiten Hälfte 0 sein muss, wird diese im Zeitbereich mit dem Vektor OZ maskiert.
- Ausgabewerte: Nach dem Ablauf des Programms erscheinen folgende Ausgabewerte:
 

```
elapsed_time = ...
```

 Benötigte Zeit in Sekunden um das Signal zu Filtern und die Adaption zu berechnen.
 

```
mD = 10*log10(mean(z.^2) / mean(y.^2))
```

 Mittlere Dämpfung des ganzen Signals.
 

```
mD1 = 10*log10(mean(z(1:len/2).^2) / mean(y(1:len/2).^2))
```

```
mD2 = 10*log10(mean(z(len/2:len).^2) / mean(y(len/2:len).^2))
```

 Mittlere Dämpfung in der ersten bzw. zweiten Hälfte des Signals.
 

```
Dekorr = max(xcorr(x,z)) / max(xcorr(x,y))
```

 Dieser Wert entspricht dem Verhältnis der maximalen Kreuzkorrelationen vor und nach dem Filtern.
- Diagramme: In diversen Diagrammen werden folgende Signale angezeigt:
  - Signal vor dem Abspielen, nach dem Abspielen und nach dem Filtern
  - Dämpfung und Anpassung in jedem Block
  - geschätzte Impulsantwort
  - Segmentschalter
  - 3D-Analyse (kann an/abgeschaltet werden)

## 3.2 BLMS

Die wichtigsten Programmteile der BLMS-Implementation sind:

- Abspielen / Aufnehmen: (nur bei BLMSreal.m) Das Abspielen und Aufnehmen ist gleich gelöst wie bei der PFLMS-Implementation (siehe Kapitel 3.1).
- Filterung und Berechnung des Fehlers:
 

```
for i= 1: M
    y(i)=w*flipud(u1((k+1)*L+i:(k+1)*L+M-1+i));
end
e=di-y;
```

 Die Filterung wird mit einer Faltung im Zeitbereich vorgenommen. Die Funktion flipud() kehrt ein Signal, sodass der letzte Wert zuerst kommt.
- Adaption:
 

```
w = w+(2*mu*(e1((k+1)*L+i))/p)*ui';
```

 Die Koeffizienten werden nach dem LMS-Verfahren adaptiert. Dazu wird eine Kreuzkorrelation zwischen dem Fehler und dem Eingangssignal gemacht. Der Faktor p beinhaltet die Leistungsnormierung.
- Ausgabewerte und Diagramme:
 Die Ausgabewerte und die Diagramme sind weitgehend die Gleichen wie bei der PFLMS-Implementation (siehe Kapitel 3.1).

# Kapitel 4

## Implementation in C/C++

### 4.1 Benötigte Software-Bibliotheken

#### 4.1.1 DirectX/DirectSound

DirectX[1] ist eine Software-Schnittstelle zur Programmierung von Multimedia-Geräten mit automatischer Verwendung der Hardware-Beschleunigung, sofern der Treiber dies unterstützt. DirectSound ist der Teil von DirectX, welcher sich um die Wiedergabe und Aufnahme von Audiosignalen kümmert. Die API<sup>1</sup> erlaubt es Daten über Buffer mit der Soundkarte auszutauschen.

In folgenden Abschnitten werden die nötigen Grundlagen für das Abspielen und Aufnehmen von Audiosignalen mit DirectSound erläutert.

#### Aufbau von DirectSound für die Wiedergabe

Die Wiedergabe von Audiosignalen erfolgt bei DirectSound über zwei verschiedene Arten von Buffern (vgl. Abbildung 4.1). Der primäre Buffer repräsentiert den Hardware-Buffer der Sound-

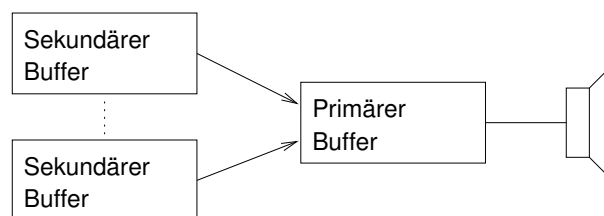


Abbildung 4.1: DirectSound Bufferstruktur für die Wiedergabe

karte. Eine Applikation muss genau einen primären Buffer verwenden.

Von den sekundären Buffern darf es beliebig viele geben. Sie repräsentieren die Sounddaten, die der Benutzer abspielen will. Beim Abspielen der sekundären Buffer werden deren Daten zuerst in den primären Buffer gemischt, danach wird der primäre Buffer von der Soundkarte abgespielt.

#### Initialisierung des DirectSound-Interface

Für die Programmierung mit DirectSound werden mindestens die Header Dateien

`dsound.h`, `dxerr9.h`, `mmreg.h`

benötigt und die Applikation muss mit folgenden Bibliotheken gelinkt werden:

---

<sup>1</sup>Application Programming Interface

`dsound.lib, dxerr.lib, dxguid.lib`

Die Initialisierung des DirectSound-Interface erfolgt in 3 Schritten:

1. Instanziierung eines DirectSound Objektes
2. Konfiguration des primären Buffers
3. Erstellen der sekundären Buffer

## 1. Instanziierung eines DirectSound Objektes

Als erstes sollte die COM Bibliothek mit dem Befehl

```
CoInitialize(NULL)
```

initialisiert werden. Dann kann ein DirectSound Objekt erstellt werden:

```
DirectSoundCreate(LPCGUID lpcGuidDevice,  
                 LPDIRECTSOUND* ppDS,  
                 LPUNKNOWN pUnkOuter)
```

Parameterbeschreibung:

**lpcGuidDevice:** Adresse der GUID der Soundkarte. Setzt man den Parameter NULL, wird die Standard-Soundkarte verwendet.

**ppDS:** Pointer auf eine LPDIRECTSOUND Variable, über welche auf das DirectSound-Interface zugegriffen werden soll.

**pUnkOuter:** Muss NULL sein.

Im Folgenden wird `lpDS` eine LPDIRECTSOUND Variable bezeichnen, über die die DirectSound Methoden aufgerufen werden.

Direkt nach der Objekterzeugung muss die Kooperationsstufe gesetzt werden:

```
lpDS->SetCooperativeLevel(HWND hWnd, DWORD dwLevel)
```

Parameterbeschreibung:

**hWnd:** Handle auf das Applikationsfenster. Bei MFC Klassen gibt die Methode `GetSafeHwnd()` den entsprechenden Handle zurück.

**dwLevel:** Bestimmt den Grad der Kooperation. Um den primären Buffer konfigurieren zu können, muss dieser Parameter auf `DSSCL_PRIORITY` gesetzt werden.

## 2. Konfiguration des primären Buffers

Um einen Buffer konfigurieren zu können, braucht man zuerst ein Objekt, das einen Buffer repräsentiert. Dazu muss die Struktur `DSBUFFERDESC` ausgefüllt werden. Für den primären Buffer müssen folgende Felder dieser Struktur bestimmte Werte haben:

- `dwFlags = DSCAPS_PRIMARYBUFFER`
- `dwBufferBytes = 0`
- `lpwfxFormat = NULL`

Nun kann mit dieser Struktur ein Objekt erzeugt werden, über welches man auf den primären Buffer zugreifen und diesen so konfigurieren kann:

```
lpDS->CreateSoundBuffer(LPCDSBUFFERDESC pcDSBufferDesc,  
                        LPDIRECTSOUNDBUFFER* ppDSBuffer,  
                        LPUNKNOWN pUnkOuter)
```

#### Parameterbeschreibung:

**pcDSBufferDesc:** Pointer auf die DSBUFFERDESC-Struktur

**ppDSBuffer:** Pointer auf eine LPDIRECTSOUNDBUFFER-Variable über welche auf den Buffer zugegriffen werden soll. Im Folgenden mit `lpDSBPr` bezeichnet.

**pUnkOuter:** Muss NULL sein.

Über die eben erzeugte Variable kann nun der primäre Buffer konfiguriert werden. Da dieser Buffer den Buffer der Soundkarte repräsentiert, werden so die Parameter (Samplingrate, Auflösung, usw.) der Soundkarte gesetzt. Dazu wird die Struktur `WAVEFORMATEX` benutzt, die folgende Felder besitzt:

**wFormatTag:** Legt das Format der Sounddaten fest, z.B. `WAVE_FORMAT_PCM`.

**nChannels:** Legt die Anzahl der Kanäle fest (1 = Mono, 2 = Stereo).

**nSamplesPerSec:** Legt die Samplerate fest.

**wBitsPerSample:** Legt die Auflösung der Samples fest.

**nBlockAlign:** Gibt die Grösse eines einzelnen Samples in Byte an. Berechnet sich aus den Feldern  $nChannels * wBitsPerSample / 8$ .

**nAvgBytesPerSec:** Gibt die Grösse von 1 Sekunde Audiodaten an. Berechnet sich aus den Feldern  $dwSamplerate * nBlockAlign$ .

Mit dieser Struktur kann der primäre Buffer nun konfiguriert werden:

```
lpDSBPr->SetFormat(LPCWAVEFORMATEX pcwfxFormat)
```

#### Parameterbeschreibung:

**pcwfxFormat:** Pointer auf die `WAVEFORMATEX` Struktur.

### **3. Erstellen der sekundären Buffer**

Für die sekundären Buffer wird auch eine `DSBUFFERDESC` Struktur benötigt, jedoch mit anderen Feldwerten. Die Wichtigsten davon sind:

**dwFlags:** Legt die Fähigkeiten des Buffers fest. Zur Buffersynchronisation wird hier der Wert `DSBCAPS_CTRLPOSITIONNOTIFY` benötigt.

**dwBufferBytes:** Gibt die Grösse des Buffers in Byte an.

**lpwfxFormat:** Muss ein Pointer auf eine ausgefüllte `WAVEFORMATEX` Struktur sein um das Format der Bufferdaten zu setzen. Hat der sekundäre Buffer ein anderes Format wie der primäre Buffer, wird beim Mischen der sekundären Buffer in den primären Buffer automatisch eine Konversion durchgeführt. Durch Benutzung der gleichen `WAVEFORMATEX` Struktur, kann diese Konversion verhindert werden.

Wie beim primären Buffer kann ein sekundärer Buffer mit der Methode

```
lpDS->CreateSoundBuffer(...)
```

erzeugt werden.

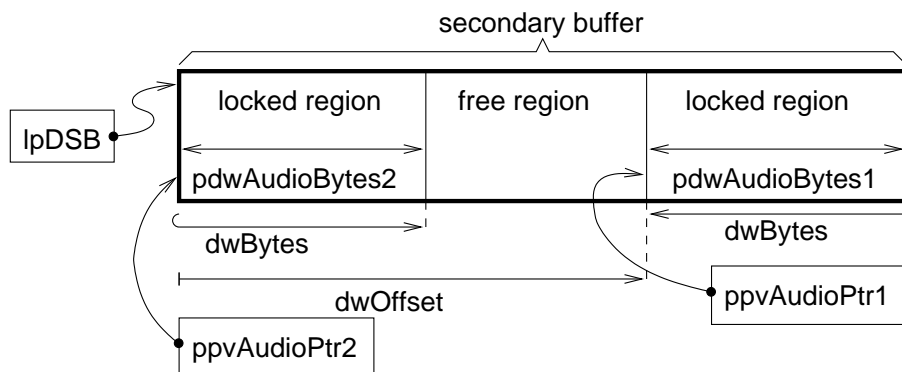


Abbildung 4.2: Locked DirectSound Buffer

## Bufferdaten manipulieren

Im Folgenden bezeichnet `lpDSB` eine `LPDIRECTSOUNDBUFFER` Variable für einen sekundären Buffer. Die Methode

```
lpDSB->Lock(DWORD dwOffset,
            DWORD dwBytes,
            LPVOID* ppvAudioPtr1,
            LPDWORD pdwAudioBytes1,
            LPVOID* ppvAudioPtr2,
            LPDWORD pdwAudioBytes2,
            DWORD dwFlags)
```

gibt einen Block der Grösse `dwBytes` ab einem Offset `dwOffset` zum Zugriff frei und setzt den Pointer `ppvAudioPtr1` an dessen Anfang (vgl. Abbildung 4.2). Ragt der Block über das Bufferende hinaus, wird der Rest am Bufferanfang freigegeben und `ppvAudioPtr2` zeigt auf den Bufferanfang. Entsprechend der Aufteilung des Blockes werden die Werte von `pdwAudioBytes1` und `pdwAudioBytes2` gesetzt. Wird der Block nicht aufgeteilt, ist `ppvAudioPtr` ein NULL Pointer und `pdwAudioBytes2` weist den Wert 0 auf.

Der Parameter `dwFlags` kann folgende Werte oder 0 aufweisen:

**DSBLOCK\_FROMWRITECURSOR:** Gibt den Block ab dem Write-Cursor frei. `dwOffset` wird dann ignoriert.

**DSBLOCK\_ENTIREBUFFER:** Gibt den gesamten Buffer frei. `dwBytes` wird dann ignoriert.

Über die beiden Pointer `ppvAudioPtr1` und `ppvAudioPtr2` kann der Bufferbereich mit Daten gefüllt werden.

Die Methode

```
lpDSB->Unlock(LPVOID pvAudioPtr1,
             DWORD dwAudioBytes1,
             LPVOID pvAudioPtr2,
             DWORD dwAudioBytes2)
```

sperrt den durch die 4 Parameter definierten Bereich wieder für die Bearbeitung.

## Abspielen eines Buffers

Jeder Buffer in DirectSound besitzt einen Play- und einen Write-Cursor (vgl. Abbildung 4.3). Der Play-Cursor zeigt auf die Stelle im Buffer, welche gerade von der Soundkarte bearbeitet wird. Der Write-Cursor folgt dem Play-Cursor in einem kleinen Abstand. Er zeigt auf die Stelle im Buffer, ab

der die Bufferdaten verändert werden können, ohne die gerade benötigten Daten des Play-Cursors zu zerstören.

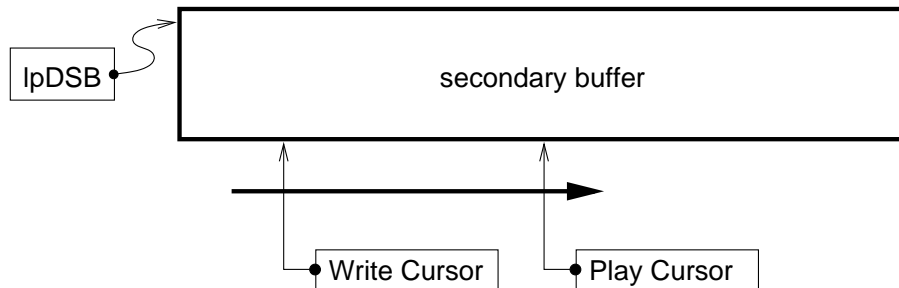


Abbildung 4.3: Abspielen eines DirectSound Buffer

Mit der Methode `Play(...)` kann die Wiedergabe eines Buffers gestartet werden:

```
lpDSB->Play(DWORD dwReserved1,
            DWORD dwPriority,
            DWORD dwFlags)
```

Parameterbeschreibung:

**dwReserved1:** Muss 0 sein.

**dwPriority:** Gibt die Priorität des Buffers beim Mischen in den primären Buffer. Dieser Parameter kann nur gesetzt werden, wenn der Buffer mit dem Wert `DSBCAPS_LOCFDEFER` erstellt wurde (in der `DSBUFFERDESC` Struktur).

**dwFlags:** Spezifiziert, wie der Buffer abgespielt wird. `DSBPLAY_LOOPING` spielt den Buffer zirkular ab.

**Anmerkung:** Beim Reservieren eines Bufferbereiches mit der Methode `Lock(...)` muss darauf geachtet werden, dass sich der Bereich *hinter* dem Write-Cursor befindet, da sonst evtl. die Daten des Play-Cursors zerstört werden. Ausserdem sollte der Bereich wieder freigeben werden *bevor* der Play-Cursor in den Bereich eintritt.

## Aufbau von DirectSound für die Aufnahme

Das Aufnehmen von Audiosignalen mit DirectSound ist ähnlich aufgebaut wie das Abspielen. Wie

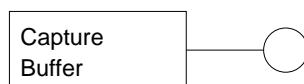


Abbildung 4.4: DirectSound Bufferstruktur für die Aufnahme

in Abbildung 4.4 dargestellt, benötigt die Aufnahme nur einen Capture-Buffer. Die Soundkarte legt die aufgenommenen Signale in diesem Buffer ab, der direkt von der Applikation ausgelesen werden kann.

## Erstellen des Capture-Buffers

Um einen Capture-Buffer erstellen zu können, muss zuerst ein `DirectSoundCapture`-Objekt instanziiert werden.

```
DirectSoundCaptureCreate(LPCGUID lpGUID,  
                          LPDIRECTSOUNDCAPTURE* lpDSC,  
                          LPUNKNOWN pUnkOuter)
```

#### Parameterbeschreibung:

**lpGUID:** Adresse der GUID des Aufnahmeegerätes. Der Wert NULL wählt das Standard Aufnahmeegerät.

**lpDSC:** Pointer auf eine LPDIRECTSOUNDCAPTURE-Variable, über welche auf das DirectSound-Capture-Interface zugegriffen werden soll.

**pUnkOuter:** Muss NULL sein.

Für das Erstellen des Capture-Buffers wird eine DSCBUFFERDESC-Struktur benötigt, welche gleich aufgebaut ist, wie die DSBUFFERDESC-Struktur der Abspiel-Buffer. Damit kann ein Capture-Buffer erstellt werden:

```
lpDSC->CreateCaptureBuffer(LPDSBUFFERDESC pDSCBDesc,  
                           LPDIRECTSOUNDCAPTUREBUFFER* ppDSCB,  
                           LPUNKNOWN pUnkOuter)
```

#### Parameterbeschreibung:

**pDSCBDesc:** Pointer auf eine DSCBUFFERDESC-Struktur.

**ppDSCB:** Pointer auf eine LPDIRECTSOUNDCAPTUREBUFFER-Variable, über welche auf den Buffer zugegriffen werden soll.

**pUnkOuter:** Muss NULL sein.

## **Aufnahme starten und stoppen**

Gleich wie die Playback-Buffer, besitzt auch der Capture-Buffer einen Play- und einen Write-Cursor, welche die Position im Buffer angeben, die gerade von der Soundkarte bearbeitet wird. Mit der folgenden Methode wird die Aufnahme gestartet (analog zur Play(...) Methode der Playback-Buffer):

```
ppDSCB->Start(DWORD dwFlags)
```

#### Parameterbeschreibung:

**dwFlags:** Mit dem Flag wird angegeben, wie die Daten im Buffer gespeichert werden. Mit dem Wert DSCBSTART\_LOOPING werden die Daten wie in einen Ringbuffer geschrieben.

Mit der Methode

```
ppDSCB->Stop()
```

kann die Aufnahme gestoppt werden.

## **Den Capture-Buffer auslesen**

Wie bei den Playback-Buffer, muss vor dem Zugriff ein Bereich des Buffers gesperrt werden. Dies geschieht ebenfalls mit den Methoden ppSDCB->Lock(...) und ppSDCB->Unlock(...). Die Parameter und das Verhalten der Methoden sind dieselben wie bei den Playback-Buffer.

## Buffersynchronisation

Ein Buffer kann einen Event auslösen, wenn der Play-Cursor eine bestimmte Position im Buffer erreicht hat. Dies geschieht über das `IDirectSoundNotify`-Interface und der `DSBPOSITIONNOTIFY`-Struktur. Mit

```
LPDIRECTSOUNDNOTIFY lpDSPN;  
lpDSPN->QueryInterface(IID_IDirectSoundNotify, (LPVOID*) &lpDSPN);
```

wird das Interface geladen und über `lpDSPN` ansprechbar. Bei der `DSBPOSITIONNOTIFY`-Struktur müssen zwei Felder gesetzt werden:

**dwOffset:** Gibt an, mit welchem Offset (in Bytes) vom Bufferanfang die Positionnotify gesetzt wird.

**hEventNotify:** Dies ist der Handle des Events, der ausgelöst wird, wenn der Play-Cursor die Positionnotify erreicht hat. Ein solcher Event kann mit der Win32-Methode `CreateEvent(...)` erzeugt werden, welche den Event-Handle zurückliefert.

Mit der folgenden Methode werden die Positionnotifies einem Buffer zugewiesen:

```
lpDSPN->SetNotificationPositions(DWORD dwPositionNotifies,  
                                LPCDSBPOSITIONNOTIFY pcPositionNotifies)
```

### Parameterbeschreibung:

**dwPositionNotifies:** Anzahl der Positionnotifies.

**pcPositionNotifies:** Pointer auf ein Array von `DSBPOSITIONNOTIFY`-Variablen.

In der Applikation können diese Events z.B. mit `MsgWaitForMultipleObjects(...)` abgefangen und entsprechend behandelt werden.

### **Anmerkungen:**

1. Beim Aufruf der `SetNotificationPositions(...)` für den Playback Buffer werden die Events gleich ausgelöst, aber nicht beim Capture Buffer. Dieses Fehlverhalten sollte abgefangen werden, um eine saubere Buffersynchronisation zu gewährleisten.
2. In Windows XP werden die Events im *10ms* Raster abgearbeitet. Früher ausgelöste Event werden entsprechend verzögert. Man sollte daher darauf achten, dass die Events in dieses Raster passen, da sich sonst die Wartezeiten akkumulieren und zu einem Fehlverhalten der Buffersynchronisation führen.

### 4.1.2 Die FFTW Bibliothek

Die FFTW[2] Bibliothek ist eine C-Bibliothek zur schnellen Berechnung aller möglichen Arten diskreter Fouriertransformationen (DFT) — von 1 bis  $n$ -dimensionalen FFTs bis zu Hartley Transformationen. Grundsätzlich benutzt die FFTW Bibliothek ein zweistufiges Verfahren. Zuerst wird ein Plan für die gewünschte Transformation erstellt, welcher die besten Parameter für den Transformationsalgorithmus (abhängig von den Prozessorfähigkeiten und der Datengrösse) ermittelt. Dieser Plan kann dann beliebig oft ausgeführt werden. Unabhängig von der Datengrösse garantiert die FFTW Bibliothek einen Algorithmus der Ordnung  $N \log_2(N)$ .

Im Folgenden wird die Benutzung der FFTW Bibliothek für die FFT von reellen, eindimensionalen Signalen erläutert. Eine Beschreibung der restlichen Funktionen kann in der Dokumentation[3] der Bibliothek gefunden werden.

#### Die FFT reeller Signale

Für die FFT mit reellen, eindimensionalen Eingangsdaten wird der Plan mit folgender Funktion erstellt:

```
fftw_plan fftw_plan_dft_r2c_1d(int n,
                                double* in,
                                fftw_complex* out,
                                unsigned int flags)
```

Parameterbeschreibung:

**n:** Datengrösse (Anzahl Samples).

**in:** Array der Grösse  $n$  mit den Eingangsdaten.

**out:** Array der Grösse  $\frac{n}{2} + 1$  für die transformierten Daten.

**flags:** Bestimmt die Vorgehensweise bei der Algorithmenermittlung. Mit `FFTW_MEASURE` wird die Berechnungszeit des Algorithmus optimiert. Mit diesem Flag kann die Planerstellung eine gewisse Zeit dauern, da verschiedene Algorithmen getestet werden. Mit `FFTW_ESTIMATE` wird ein heuristischer Algorithmus verwendet. Mit diesem Flag ist die Planerstellung am schnellsten, dafür ist wahrscheinlich der FFT-Algorithmus nicht optimal.

#### Die inverse FFT

Für die Rücktransformation des oben beschriebenen Signals, wird der Plan mit folgender Funktion erstellt:

```
fftw_plan fftw_plan_dft_c2r_1d(int n,
                                fftw_complex* in,
                                double* out,
                                unsigned int flags)
```

Parameterbeschreibung:

**n:** Datengrösse (Anzahl Samples).

**in:** Array der Grösse  $\frac{n}{2} + 1$  mit den komplexen Daten.

**out:** Array der Grösse  $n$  für die rücktransformierten Daten.

**flags:** Bestimmt die Vorgehensweise bei der Algorithmenermittlung. Die Flags sind die gleichen wie oben beschrieben.

Bei der Rücktransformation ist zu beachten, dass sie nicht normalisiert ist. D.h. die rücktransformierten Daten sind mit dem Faktor  $n$  gestreckt. Transformiert man z.B. das reelle Signal ( $n = 8$ )

$$x(k) = 1, 0, 0, 0, 0, 0, 0, 0$$

in den Frequenzbereich, ergibt das

$$X(k) = 1 + j0, 1 + j0, 1 + j0, 1 + j0, 1 + j0.$$

Führt man die entsprechende Rücktransformation aus, resultiert das in dem reellen Signal

$$\tilde{x}(k) = 8, 0, 0, 0, 0, 0, 0, 0.$$

### Die Transformation ausführen

Mit der Funktion

```
void fftw_execute(const fftw_plan)
```

wird die Transformation gemäss dem Plan berechnet.

Die FFTW Bibliothek hat für komplexe Zahlen selbst einen komplexen Datentyp `fftw_complex` definiert, als ein `double` Array mit zwei Elementen. Der Realteil ist im ersten und der Imaginärteil ist im zweiten Element gespeichert.

Mit einem ANSI C99 kompatiblen Compiler kann alternativ auch der C Datentyp `complex` verwendet werden. Die Funktionen der FFTW Bibliothek verwenden dann intern diesen Datentyp. Dazu muss die Headerdatei `complex.h` vor `fftw3.h` geladen werden.

Damit die Funktionen die Prozesseigenschaften (SIMD, MMX, usw.) optimal ausnutzen können, sollte zur Speicherallozierung die Funktion

```
void fftw_malloc(size_t n)
```

verwendet werden, um die für die Prozessorinstruktionen benötigte Byte-Anordnung zu gewährleisten. Die so allozierten Speicherbereiche sollten dann konsequenterweise mit der Funktion

```
void fftw_free(void* p)
```

freigegeben werden.

## 4.2 IP Telefon

Um den Algorithmus in der Realität zu testen, musste ein einfaches IP Telefon implementiert werden. Die Implementation beschränkt sich auf die grundlegenden Eigenschaften eines IP Telefons, so wird z.B. auf eine Kodierung des Audiosignals verzichtet.

### 4.2.1 Aufbau der Software

Die Applikation ist modular aufgebaut, um die einzelnen Programmbereiche im Sinne der Objektorientierung in Modulen zu kapseln, wie in Abbildung 4.5 dargestellt ist. Damit die Module

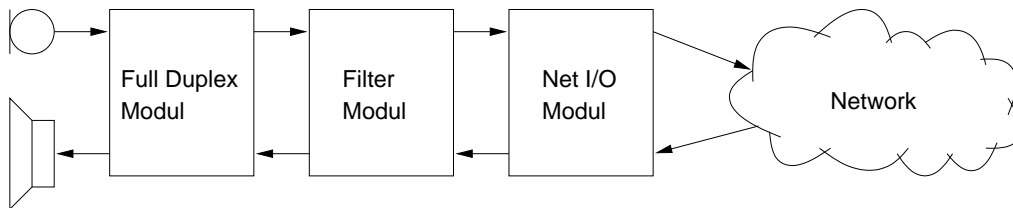


Abbildung 4.5: Module des IP-Telefons

miteinander kommunizieren können, wurde in UML<sup>2</sup> ein sog. Design Pattern entworfen, das eine solche Kommunikation ermöglicht und vereinheitlicht.

### 4.2.2 Das Controller-Actor Design Pattern

Das Controller-Actor Pattern beschreibt eine bidirektionale Datenverbindung zwischen zwei Modulen, wobei der Daten-Verkehr von dem einen Modul gesteuert wird (**Controller**) und das andere die Daten empfängt bzw. sendet (**Actor**). Es liegt also ein unidirektionaler Kontrollfluss vor.

#### UML Darstellung

Das Pattern lässt sich als UML-Diagramm gemäss Abbildung 4.6 darstellen. Dadurch, dass ein Actor auch ein Controller ist, können mit diesem Pattern mehrstufige Verbindungen realisiert werden, die von der Klasse, die nur von Controller abgeleitet ist, gesteuert wird.

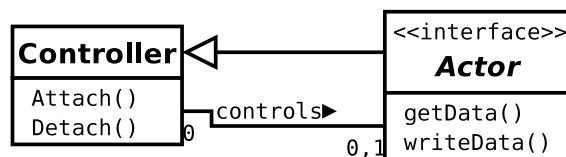


Abbildung 4.6: Controller-Actor Pattern

#### Controller Methoden:

**Attach():** Baut eine Kommunikationsverbindung zu einem Actor auf.

**Detach():** Löst die Kommunikationsverbindung zu einem Actor wieder.

#### Actor Methoden:

**GetData():** Der Controller fordert Daten vom Actor an.

**WriteData():** Der Controller gibt über diese Methode Daten an den Actor weiter.

Mit dem Controller-Actor Pattern sieht das UML-Diagramm der gesamten Applikation wie in Abbildung 4.7 aus. Die Datenkommunikation der ganzen Modulkette wird vom Full Duplex Modul

<sup>2</sup> Unified Modeling Language

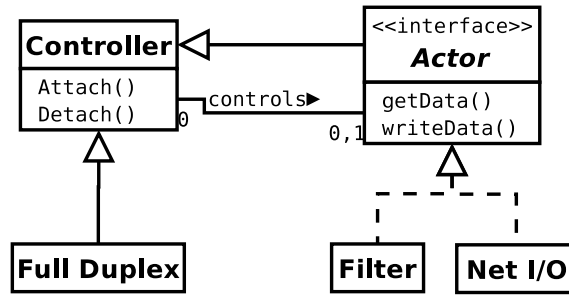


Abbildung 4.7: UML-Diagramm der Applikation

gesteuert, da dieses Modul die Echtzeitbedingungen vorgibt.

### 4.2.3 Das Full Duplex Modul

Dieses Modul kümmert sich um die Aufnahme und Wiedergabe der Audiosignale. Der Aufbau des Moduls ist in Abbildung 4.8 dargestellt. Die beiden Buffer sind durch Positionnotifies in zwei

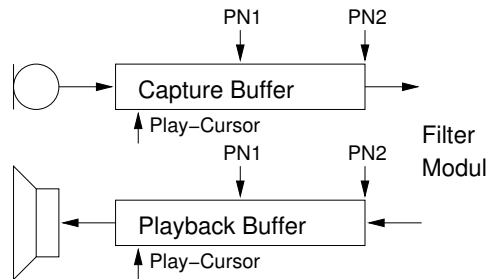


Abbildung 4.8: Aufbau des Full Duplex Modules

Blöcke unterteilt, wobei jeder Block  $20ms$  lang ist. Die Buffergrösse (in Bytes) berechnet sich folglich aus:

$$S_{Bytes} = AnzPositionotifies \cdot 20ms \cdot Samplingrate \cdot \frac{BitPerSample}{8}$$

Ein Thread fängt die Events ab und ruft eine Methode auf, die beim entsprechenden Buffer den gerade bearbeiteten Block abspielt (Playback-Buffer) bzw. an das Filter Modul weitergibt (Capture-Buffer). Dadurch entsteht eine Systemverzögerung von zwei Blöcken:

**Playback-Buffer:** Der Block der gerade abgespielt wird, wurde einen Block früher vom Filter Modul geliefert.

**Capture-Buffer:** Der Block der gerade an das Filter Modul gesendet wird, wurde einen Block früher aufgenommen.

Das Modul (und damit die gesamte Applikation) arbeitet somit blockbasiert mit einer Blockgrösse von

$$N = 20ms \cdot Samplingrate$$

Samples. Damit sind auch die Echtzeitbedingungen der Applikation gegeben, denn ein Block muss von der gesamten Modulkette innerhalb von  $20ms$  bearbeitet werden.

Bei Versuchen wurde festgestellt, dass die beiden Buffer nicht gleich schnell arbeiten. Der Play-Cursor des Playback-Buffers läuft schneller als der Play-Cursor des Capture-Buffers. Mit dem  $10ms$ -Raster der Event-Bearbeitung führt dies dazu, dass Blöcke verloren gehen, wenn sich

der Play-Cursor im gesperrten Bereich des Buffers befindet. Mit einer einfachen 2-Punktregelung für die Samplerate des Playback-Buffers lässt sich das Problem beheben: Überschreitet die Differenz der beiden Play-Cursor einen Grenzwert, wird die Samplerate des Playback-Buffer erhöht bzw. verringert. Die Applikation verwendet einen Grenzwert von  $\pm 5$  Byte und eine Samplerate-Korrekturgröße von  $5Hz$ . Eine genauere Regelung ist nicht empfehlenswert, da:

1. die Samplerate nicht beliebig genau einstellen lässt.  $5Hz$  ist ein Korrekturwert, den jede moderne Soundkarte beherrscht.
2. die Samplerate nicht zu oft verändert werden sollte.

#### 4.2.4 Das Filter Modul

Das Filter Modul ermöglicht eine Filterung der Daten die mit dem Full Duplex Modul ausgetauscht werden. Der Applikation stehen zwei Filter zur Verfügung: Ein Tiefpass-Filter und das PFLMS-Filter.

Das Tiefpass-Filter ist eine direkte Implementation des 'Overlap-Save' Verfahrens und wird hier nicht näher beschrieben.

Für das PFLMS-Filter ist das Modul wie in Abbildung 4.9 aufgebaut. Da die FFTW-Bibliothek

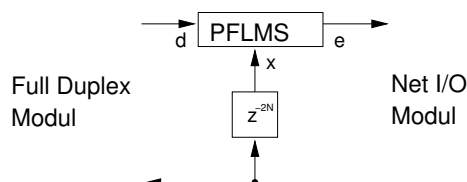


Abbildung 4.9: Aufbau des Filter Modules

nur mit `double`-Werten operieren kann, muss bei jedem Ein- und Ausgang des Moduls eine Datenkonversion durchgeführt werden.

Weil das Full Duplex Modul eine feste Systemverzögerung von 2 Blöcken aufweist, wird das Eingangssignal  $x$  des PFLMS-Filters vorher um 2 Blöcke verzögert. Dadurch kann die Impulsantwort des PFLMS-Filters um diese 2 Blöcke verkürzt werden.

In Abbildung 4.10 ist der detaillierte Aufbau der PFLMS-Implementation dargestellt, jedoch fehlt der Steuerungsteil. Zur Absicherung gegenüber Fehladaptationen wurde eine Dämpfungüber-

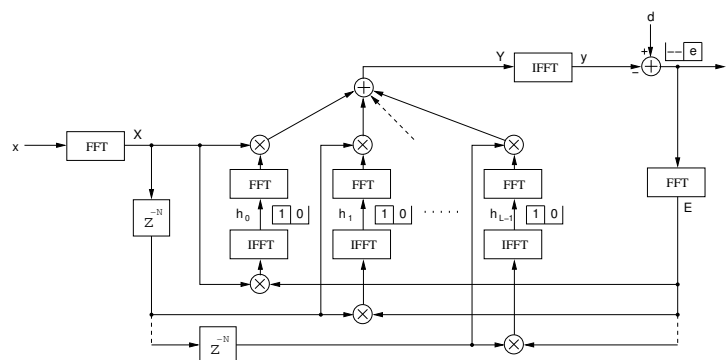


Abbildung 4.10: Detaillierter Aufbau des PFLMS Filters

prüfung eingebaut. Diese vergleicht die Leistung des Mikrofonsignals  $d$  mit der Leistung des Fehlersignals  $e$ . Nur wenn die Leistung im Fehlersignal kleiner ist, wird dieses an das Net I/O Modul weitergegeben, im anderen Fall das Mikrofon signal selbst.

## 4.2.5 Das Net I/O Modul

Dieses Modul stellt der Applikation die Netzwerkanbindung zu Verfügung. Die Implementation ist nur auf das Wesentliche beschränkt. Wie in Abbildung 4.11 zu sehen ist, werden die Blöcke versendet, wie sie vom Filter Modul geliefert werden. Sie werden weder gepuffert, noch auf irgendeine Art komprimiert.

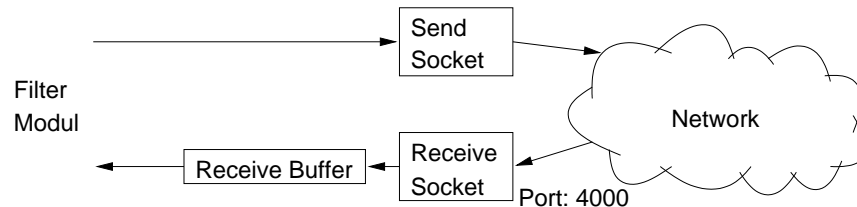


Abbildung 4.11: Aufbau des NetI/O Modules

Zum Empfangen von Blöcken wartet der Receive-Socket auf dem Port 4000 in einem Thread. Empfängt er einen Block, legt er ihn in den Receive-Buffer, von welchem auch das Filter Modul seine Daten beziehen kann.

Dadurch, dass die Daten nicht komprimiert werden, ist die Netzbelastung ziemlich hoch, so dass das IP Telefon nur im LAN<sup>3</sup> verwendet werden kann. Bei einer Samplerate von  $44.1kHz$  mit einer Sampleauflösung von 16 Bit, entstehen pro Block

$$44100Hz \cdot 16\text{Bit} \cdot 20 \frac{ms}{\text{Block}} = 1764 \frac{\text{Byte}}{\text{Block}}$$

Nutzdaten, die verschickt bzw. empfangen werden. Ein UDP/IP Packet kann maximal

$$\underbrace{65535\text{Byte}}_{\text{IP Packetgrösse}} - \underbrace{20\text{Byte}}_{\text{IP Headergrösse}} - \underbrace{8\text{Byte}}_{\text{UDP Headergrösse}} = 65507\text{Byte}$$

Daten speichern. Pro Block wird also ein UDP/IP Packet benötigt. Ein Ethernet Packet kann aber nur

$$\underbrace{1500\text{Byte}}_{\text{Ethernet Packetgrösse}} - \underbrace{20\text{Byte}}_{\text{Ethernet Headergrösse}} = 1480\text{Byte}$$

Daten aufnehmen. Pro Block werden also zwei Ethernet Pakete benötigt. Insgesamt werden folglich für einen Block

$$\underbrace{1764\text{Byte}}_{\text{Blockgrösse}} + \underbrace{8\text{Byte}}_{\text{UDP Headergrösse}} + \underbrace{20\text{Byte}}_{\text{IP Headergrösse}} + 2 \cdot \underbrace{20\text{Byte}}_{\text{Ethernet Headergrösse}} = 1832\text{Byte}$$

Daten über das Netz versendet.

Für eine vollständige Kommunikation müssen von der Applikation 50 Blöcke pro Sekunde versendet und empfangen werden. Das Netz wird demnach mit

$$2 \cdot 50 \frac{\text{Blöcke}}{s} \cdot 1832 \frac{\text{Byte}}{\text{Block}} = 183200 \frac{\text{Byte}}{s}$$

belastet.

---

<sup>3</sup>Local Area Network

## 4.2.6 Das GUI des IP Telefons

Das Hauptfenster der Applikation (vgl. Abbildung 4.12) ist in drei Sektionen unterteilt, die den einzelnen Modulen der Applikation repräsentieren. In der Sektion 'Audio' können die Parameter

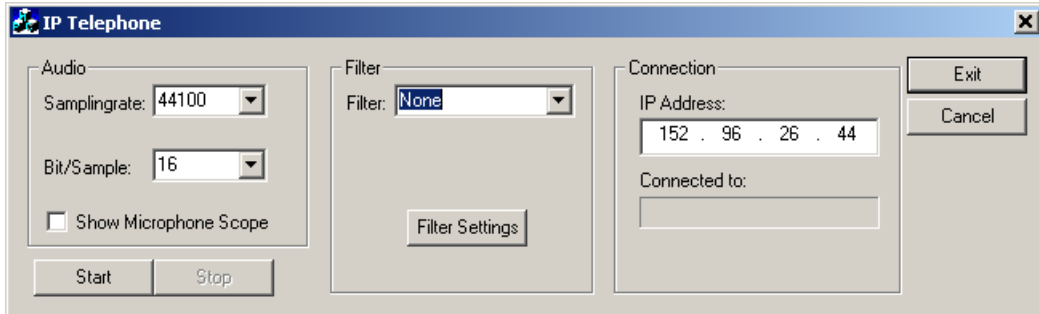


Abbildung 4.12: Hauptfenster

des Full Duplex Moduls eingestellt werden. Für die Samplerate stehen die Werte 8, 11.025, 22.05 und 44.1kHz zu Verfügung. Die Sampleauflösung (Combo Box: 'Bit/Sample') kann auf 8 oder 16 Bit eingestellt werden. Wird die Check-Box 'Show Microphone Scope' aktiviert, wird während dem Gespräch das Mikrofonsignal in Echtzeit dargestellt (vgl. Abbildung 4.13).

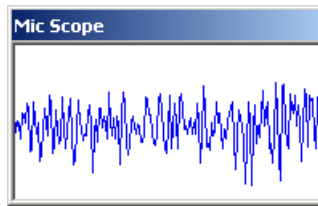


Abbildung 4.13: Darstellung des Mikrofonsignals

In der Sektion 'Filter' kann der Filter ausgewählt werden, der während dem Telefonat benutzt werden soll. Zur Auswahl stehen:

- Kein Filter
- Tiefpass Filter
- PFLMS Filter zur Unterdrückung des lokalen Echos

Über den Button 'Filter Settings' lassen sich die jeweiligen Filter-Parameter einstellen.

Im Settings-Dialog des Tiefpass Filter Moduls (vgl. Abbildung 4.14) lässt sich die Grenzfrequenz des Filters in 100Hz-Schritten in einem Bereich von 100Hz bis 4000Hz einstellen.

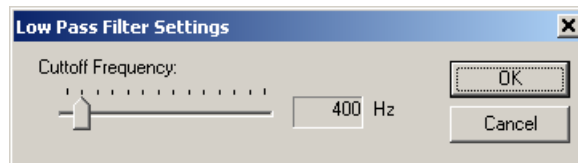


Abbildung 4.14: Einstellungen des Tiefpass Filters

Beim PFLMS Filter lassen sich im Settings-Dialog (Abbildung 4.15) die vier Parameter  $c_2$ ,  $c_4$ ,  $P_X^{min}$  und  $P_{X,k}^{min}$  einstellen. Die vorgegeben Werte haben sich aus verschiedenen Versuchen ergeben. Durch Aktivierung der Check-Box 'Log' werden während dem Telefonat die Parameter

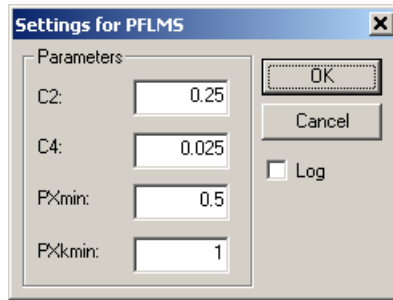


Abbildung 4.15: Einstellungen des PFLMS Filters

$P_{Xmomentan}$ ,  $P_H$  und die Dämpfung (in dB) in die Datei `PFLMS.log` und jede fünfte Impulsantwort des Filters in die Datei `h.log` geschrieben.

Während dem Telefonat wird in einem Fenster (vgl. Abbildung 4.16) zu jedem Zeitpunkt die Impulsantwort des Filters dargestellt.

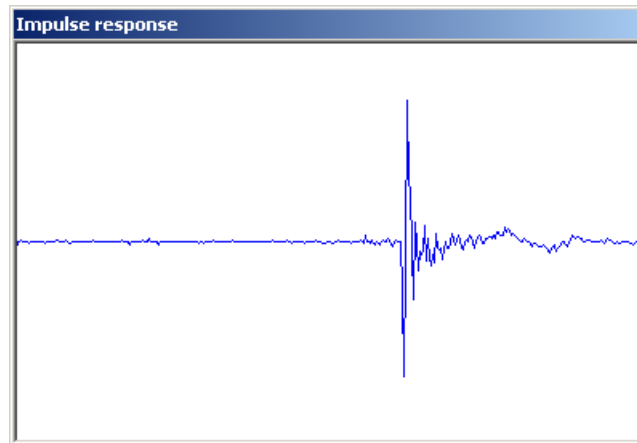


Abbildung 4.16: Darstellung der Impulsantwort

# Kapitel 5

## Messungen und Resultate

### 5.1 Erzielte Resultate

Die Messungen fanden auf dem Computer 'pel6003003' mit den Einstellungen  $F_s = 44100Hz$  und 16 Bit statt. Die übrigen Einstellungen sind in Tabelle 5.1 zu finden. Zu beachten ist, dass die Werte für die Leistung normiert sind;  $P_x$  und  $P_{X,j}$  können maximal 882 werden.

$N$	$C$	$L$	$c1$	$c2$	$c4$	$\gamma_1$	$\gamma_2$	$\gamma_3$
882	1764	2	0.25	0.25	0.05	0.9	0.95	0.85
$P_x^{min}$		$P_{X,j}^{min}$		$\eta_0$	$\eta_1$	$P_{h_0}[0]$	$P_{h_1}[0]$	
1		0.5		0.05	0.15	0.07	0.03	

Tabelle 5.1: Testeinstellungen

Die Versuchsanordnungen sind in Abbildung 5.1 aufgezeichnet. Die verwendeten Lautsprecher sind vom Typ 'SC-C47' von 'aiwa', als Mikrofon wurde das vom Headset 'm@b 40' von 'Sennheiser' verwendet.

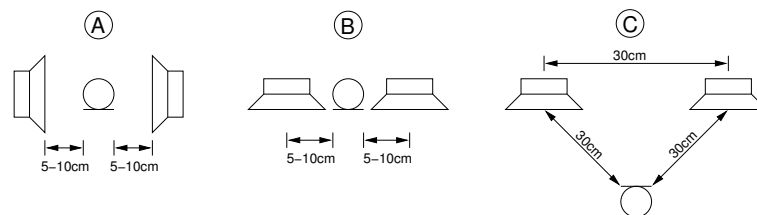


Abbildung 5.1: Verschiedene Versuchsanordnungen

Bei der Anordnung A wurden folgende Messungen gemacht:

- Verlauf der Leistung und Dämpfung:

In Abbildung 5.2 ist ersichtlich dass nur dann eine Dämpfung erzielt wird, wenn auch Leistung vorhanden ist. Bei starker Einkopplung (Anordnung A) erreicht die Dämpfung bis 25dB. Bei Anordnung B beträgt die maximale Dämpfung noch rund 15-20dB; bei Anordnung C steigt die Dämpfung nicht mehr über 15dB. Die Einstellung der Lautstärke am Lautsprecher hat einen direkten Einfluss auf die erreichbare Dämpfung.

- Gefundene Impulsantwort:

Abbildung 5.3 zeigt eine typische Impulsantwort bei Anordnung A. Die grosse Verzögerung kommt nicht nur durch die akustische Laufzeit zustande, sondern auch durch die Verzögerungen von der Wiedergabe und Aufnahme.

- Verlauf der Impulsantwort:

Ein möglicher Verlauf der Impulsantwort ist in Abbildung 5.4 dargestellt. Aufgezeichnet wurde jede 5te Impulsantwort (Achse: Blocknummer[5]).

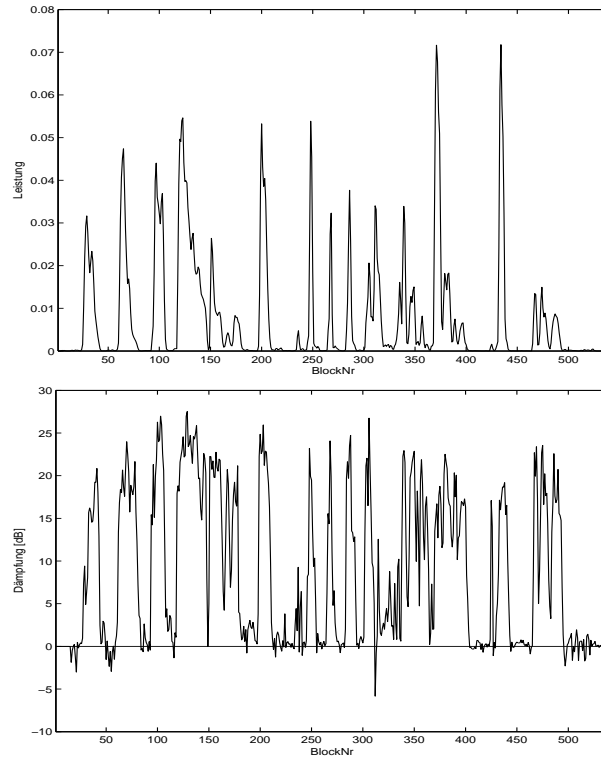


Abbildung 5.2: Leistungs- und Dämpfungsverlauf

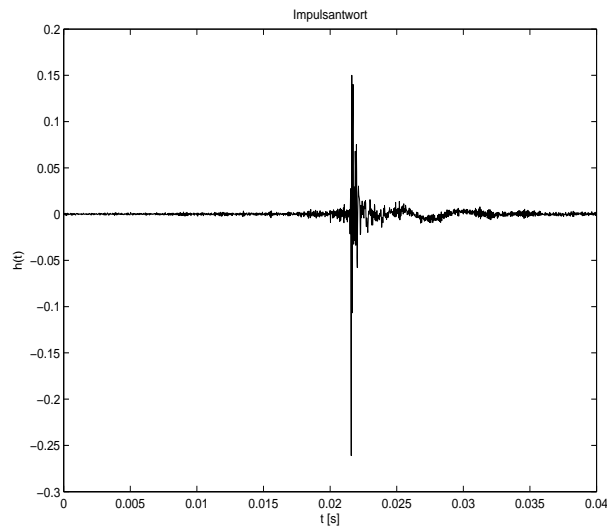


Abbildung 5.3: Impulsantwort

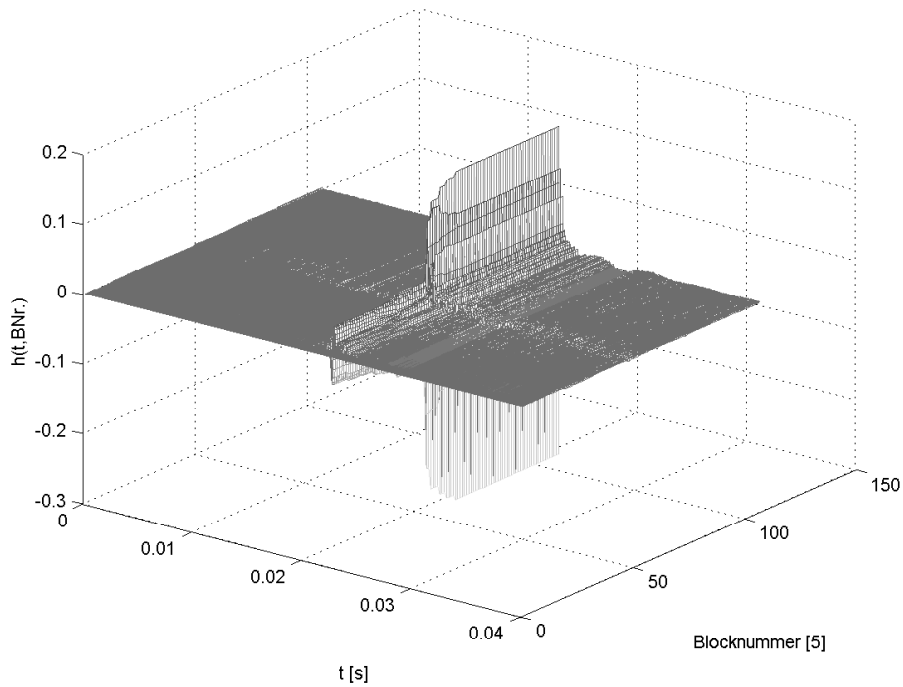


Abbildung 5.4: Impulsantwortverlauf

## 5.2 Benötigte Ressourcen

Da die Blockgröße  $20ms$  gewählt wurde, ist die Anzahl Samples nicht der Form  $2^a$ . Die Blockgröße ist somit:

$$N = F_s \cdot 20ms = 44100 \cdot 20ms = 882 \text{ Samples}$$

Die FFT-Größe ist somit:

$$C = 2N = 1764$$

Die FFTW-Bibliothek stellt trotzdem einen sehr effizienten Algorithmus zur Verfügung, deshalb wird der Rechenaufwand nicht stark beeinflusst. Die benötigten Operationen können mit der Funktion `fftw_flops()` ausgelesen werden. In Tabelle 5.2 sind die Anzahl der Operationen aufgelistet. *fma* bedeutet ‘fused multiply add’; falls der Prozessor über eine solche Multiplizier-Addier-Operation verfügt, entspricht dies einer Operation, ansonsten 2; vom zweiten Fall wird in der Berechnung ausgegangen. Die Anzahl ist dennoch geringer als aus Tabelle 2.5 für  $C=1764$  hervorgeht ( $\approx 95000$ ).

FFT-Größe	1764
$A_{reell}$	14308
$M_{reell}$	9100
<i>fma</i>	4592
$FLOP_{total}$	32592

Tabelle 5.2: Benötigte Operationen der FFTW-Library

Mit Tabelle 2.6 beträgt die totale Anzahl an FLOPs<sup>1</sup> pro Block:

$$FLOPs/Block = 7 \cdot 32592 + 6 \cdot 2LN + (9LN + 2N) + 2(2LN - N) + (5LN + 3N) + LN \approx 285500$$

<sup>1</sup>FLOPs = Floating point operations

Pro Sekunde müssen 50 Blöcke verarbeitet werden. Dies führt zu einer Anzahl an Operationen pro Sekunde von

$$FLOPs/s = 50 \cdot 285500 = 14.3M$$

Der Computer wurde mit dem Testprogramm TestCPU [4] mit rund 410MFLOPs/s gemessen. Die theoretische Prozessorauslastung durch den Algorithmus beträgt somit:

$$\frac{14.3M}{410M} = 3.5\%$$

In Abbildung 5.5 ist die benötigte Rechenzeit für Filterung und Adaption aufgezeichnet. Wenn der Filter die Segmente adaptiert, benötigt die Berechnung rund 0.8ms.

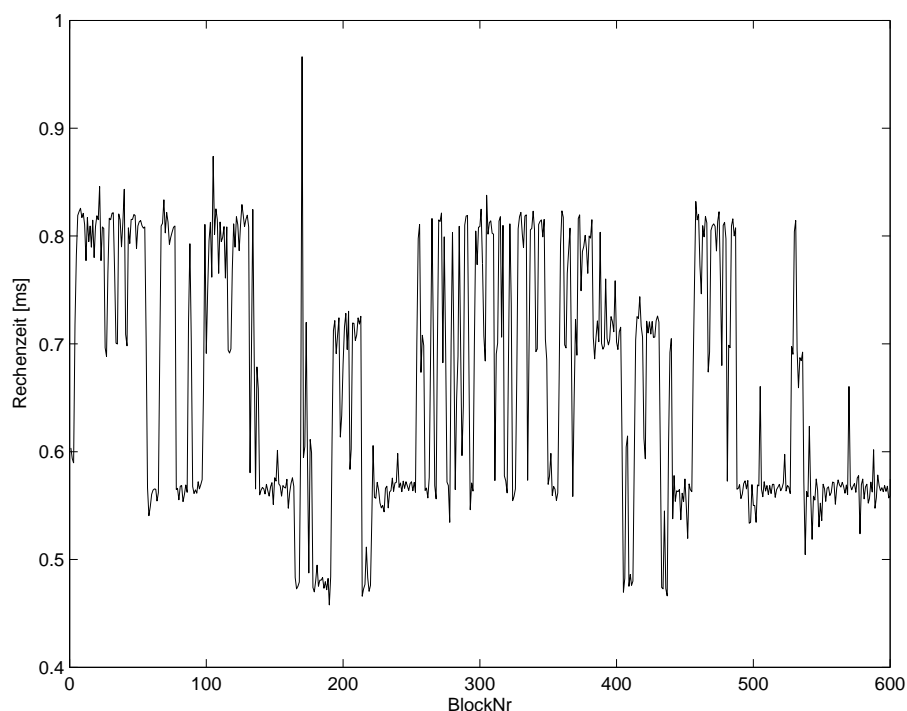


Abbildung 5.5: Benötigte Rechenzeit für Filterung und Adaption

Da alle 20ms ein neuer Block kommt beträgt die reale Prozessorauslastung durch den Algorithmus rund:

$$\frac{0.8ms}{20ms} = 4\%$$

### 5.3 Probleme

Folgende Probleme sind bei den Versuchen aufgetaucht:

- Soundkartentreiber  
Auf manchen PCs ist ein spezieller Soundkartentreiber (SoundMAX) installiert. Da dieser Treiber die Signale selber nochmals buffert und verarbeitet, erreichte das Programm keine Dämpfung und fand auch keine Impulsantwort.
- Double-Talk  
Bei leisem Double-Talk kann es vorkommen, dass die Impulsantwort sich sprunghaft verändert. Dies ist darauf zurückzuführen, dass der Koeffizient  $c_4$  jeweils nicht optimal eingestellt war, und die Leistungsschwelle für Double-Talk somit nicht überschritten wurde.

# Kapitel 6

## Fazit

### 6.1 Aktueller Stand der C++ Implementation

- Effiziente Implementierung des PFLMS-Algorithmus dank der FFTW-Bibliothek
- Schnelles Finden und gute Stabilität der Impulsantwort
- Dämpfung im Bereich von 10 bis 25dB (Abhängig von der Anordnung)
- Gute Erweiterbarkeit dank modularem Aufbau der Applikation

### 6.2 Erweiterungen

- Koeffizienten  
Eine wichtige Erweiterung wäre, die Koeffizienten automatisch einzustellen, um die Software somit automatisch an die Umgebungssituation anzupassen. Dies könnte entweder durch eine Initialisierungssequenz am Anfang oder durch entsprechende Steuerung während dem Telefonat geschehen. Besonders die Double-Talk Detektion könnte dadurch optimiert werden.
- Kompression  
Um die Netzwerkbelastung zu senken, sollten die Audiodaten im Net I/O Modul komprimiert werden. Zudem sollte ein Jitter-Buffer eingebaut werden, um die Funktionsfähigkeit auch bei schlechten Netzwerkparametern zu gewährleisten.

### 6.3 Schlusswort

Die Arbeit ermöglichte uns einen tieferen Einblick in die Theorie der adaptiven Filter und die Echtzeit-Programmierung. Zudem konnten wir unsere Kenntnisse in Digitaler Signalverarbeitung anwenden und vertiefen.

### 6.4 Danksagung

Wir danken den beiden Laborassistenten Matthias Engelhardt und Ursin Tuor für ihre Unterstützung bei der DirectSound- und MFC-Programmierung.

Rapperswil, 5. Juli 2004

Martin Rösch

Berni Imfeld

# Literaturverzeichnis

[SHY92] J. J. Shynk: *Frequency-Domain and Multirate Adaptive Filtering*, IEEE SP Magazine, Januar 1992

[EST96] P. Estermann, A. Kaelin: *A Hands-Free Phone System Based on a Partitioned Frequency-Domain Adaptive Echo Canceler*, 1996

[ZT94] G. Zelniker, F. J. Taylor: *Advanced Digital Signal Processing*, MARCEL DEKKER INC., 1994

[SCH02] A. Schüeli: *Digitale Signalverarbeitung*, HSR, Oktober 2002

[1] <http://www.microsoft.com/windows/directx/default.aspx>, Stand: 5. Juli 2004

[2] <http://www.fft.w.org>, Stand: 5. Juli 2004

[3] <http://www.fft.w.org/fft3.pdf>, Stand: 5. Juli 2004

[4] <http://testcpu.webz.cz>, Stand: 5. Juli 2004

# Anhang A

## Inhalt der CD

- **Matlab-Programme:**  
Im Verzeichnis `Matlab` befinden sich die Programme zur PFLMS- und BLMS Implementation. Zur Analyse der Log-Dateien des IP Telefons befindet sich ein zusätzliches Programm (`analyse.m`) auf der CD.
- **C++ Implementation:**  
Das IP Telefon befindet im Verzeichnis `IPTel` als Visual C++ Projekt.
- **FFTW Bibliothek:**  
Im Verzeichnis `FFTW` befinden sich die für diese Arbeit verwendete Version der FFTW Bibliothek mit der entsprechenden Dokumentation im PDF Format.
- **Dokumentation:**  
Die Dokumentation im PDF Format befindet sich im Grundverzeichnis der CD.
- **Publikationen:**  
Die im Text erwähnten Publikationen befinden sich im Verzeichnis `Publikationen`.
- **Tools:**  
Das TestCPU Programm befindet sich im Verzeichnis `Tools`.