Realisierung eines Anti-Schall-Systems (Active Noise Control System) mit integrierter Sekundärstreckenmessung auf einem digitalen Signalprozessor (DSP)

Wissenschaftliche Arbeit
zur Erlangung des akademischen Grades eines Magister Artium
im Fach Kommunikationswissenschaft
an der TU Berlin

15.03.2011

vorgelegt von
Danny Jaeger
[redacted]

Gutachter:

Prof. Dr. Stefan Weinzierl
Prof. Dr.-Ing. Michael Möser

Betreuer:

Dr. André Jakob

Eidesstattliche Erklärung

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt

Berlin, den 15.03.2011 _____

Danny Jaeger

Inhaltsverzeichnis

Eidesstattliche Erklärung ..... **i**

Inhaltsverzeichnis ..... **ii**

1 Einleitung **1**

1.1 Fragestellung und Ziel 1

1.2 Methode 1

2 Theoretische Grundlagen **3**

2.1 Identifikation akustischer Systeme 3

2.1.1 Überblick 3

2.1.2 Methoden der Systemidentifikation 4

2.2 Systemidentifikation mit MLS-Technik (Korrelationsmesstechnik) 7

2.2.1 Einleitung 7

2.2.2 MLS-Signale 8

2.2.3 Ermittlung der Impulsantwort 11

2.2.4 Anpassung an die Erfordernisse der Messung 12

2.3 Aktive Lärmbekämpfung 13

2.3.1 Grundlagen 13

2.3.2 Der LMS-Algorithmus (filtered-x-LMS-Algorithmus) 14

2.4 Das Anti-Schall-Fenster 17

2.4.1 Einleitung 17

2.4.2 Theorie 17

2.4.3 Aufbau und Eigenschaften 18

2.5 Digitale Signalprozessoren 20

2.5.1 Einführung 20

2.5.2 Aufbau und Wirkungsweise von DSPs 20

2.5.3 Anwendungen von DSPs 23

2.6 Das ADSP-21161N EZ-KIT Lite® Evaluation Kit for SHARC® processors 24

2.6.1 Überblick 24

2.6.2 ADSP-21161N EZ-KIT Lite® – Die Hardware 24

2.6.3 ADSP-21161N EZ-KIT Lite® – Die Software 28

2.6.4 Installation und Einrichtung 29

2.6.5 Die Visual-DSP Oberfläche und Bedienung 29

2.6.6 DSP-Programmierung 33

3 Programmierung/Programmbeschreibung ... 36
3.1 Einleitung ... 36
3.2 Erläuterung der zum Projekt gehörenden Dateien ... 36
3.3 Das Memory-Mapping ... 38
3.4 Anpassung der Abtastrate ... 41
3.5 Parameter und Einstellungen im Header-File „mls.h“ ... 44
3.5.1 Header-Datei für das ANC-System Feedforward ... 44
3.5.2 Header-Datei für das ANC-System Feedback ... 46
3.6 Grenzen des ANC-Systems ... 47
3.7 MLS-Generator ... 50
3.7.1 Einleitung ... 50
3.7.2 Variablen- und Vektorendeklaration ... 51
3.7.3 Programmbeschreibung ... 52
3.8 Ausgabe/Aufnahme der Messsignale ... 53
3.8.1 Einleitung ... 53
3.8.2 Variablen- und Vektorendeklaration ... 53
3.8.3 Vorbereitende Berechnungen für die Ausgabe der Messsignale ... 57
3.8.4 Programmbeschreibung Ausgabe und Einlesen der Signale ... 59
3.8.5 Ein- und Ausgabe über den AD1836 96 kHz Audio Codec ... 63
3.9 Berechnung der Impulsantwort ... 64
3.9.1 Einleitung ... 64
3.9.2 Programmbeschreibung Impulsantwortberechnung ... 64
3.9.3 Speicherphilosophie der Samples im Impulsantwortvektor ... 67
3.10 Programmierung der LMS-Algorithmen (filtered-x-LMS-Algorithmus) ... 70
3.10.1 Einleitung ... 70
3.10.2 Programmbeschreibung Feedforward –Variablen- und Vektorendeklaration ... 70
3.10.3 Programmbeschreibung Feedforward – Einlesen der Samples ... 71
3.10.4 Programmbeschreibung Feedforward – Berechnung des Ausgangssamples ... 72
3.10.5 Programmbeschreibung Feedforward – Berechnung der gefilterten Impulsantworten ... 74
3.10.6 Programmbeschreibung Feedforward – Adaptionsfilterkoeffizienten ... 75
3.10.7 Programmbeschreibung Feedback – Variablen- und Vektorendeklaration ... 78
3.10.8 Programmbeschreibung Feedback – Berechnung der Ausgangssamples ... 79
3.10.9 Programmbeschreibung Feedback – Schätzung der Referenzsignale ... 80
3.11 Bedienung des ANC-Systems ... 83
3.11.1 Ausführliche Schritt-für-Schritt-Anleitung ... 83

4 Test des Verfahrens am Antischallfenster ... 87
4.1 Versuchsaufbau ... 87
4.2 ANC-Versuche ... 89
4.2.1 Vorbereitende Messungen und Einstellungen ... 89
4.2.2 Test der ANC-Algorithmen ... 89
4.2.3 Test unter realen Bedingungen ... 90
5 Zusammenfassung und Ausblick ... 93
5.1 Zusammenfassung ... 93
5.2 Schlussbemerkungen ... 94
6 Literaturverzeichnis ... 95
7 Abbildungsverzeichnis ... 98
8 Tabellenverzeichnis ... 99
9 Anhang ... 100
9.1 Dateitypen im ADSP-21161N EZ-KIT Lite® Evaluation Kit for SHARC® processors ... 100
9.2 Kurzanleitung ... 102

1 Einleitung

1.1 Fragestellung und Ziel

Gegenstand dieser Arbeit ist die vollständige Realisierung eines mehrkanaligen Anti-Schall-Systems (Active Noise Control System) auf einem im Institut für Strömungsmechanik und Technische Akustik (ISTA) vorhandenen DSP-Entwicklungssystems (ADSP-21161N EZ-KIT Lite® Evaluation Kit).

Active Noise Control Systeme arbeiten mit Anti-Schall, der unter anderem durch Least-Mean-Square-Algorithmen (LMS) berechnet werden kann. Im vorliegenden Fall sollen x-filtered-LMS-Algorithmen verwendet werden, die zusätzlich die Kenntnis der Impulsantworten aller Sekundärstrecken erfordern.

Für die Umsetzung sollen eine Sekundärstreckenmessung mittels Maximalfolgen-Technik (MLS-Technik) sowie Least-Mean-Square-Algorithmen (LMS) vollständig in der VisualDSP++® Entwicklungsumgebung in Assembler implementiert werden. Dieses ANC-System soll dann an einem am ISTA entwickelten Schallschutzfenster getestet werden.

Damit dieses System auch erweiterbar und anpassbar an verschiedene Versuchsumgebungen ist, wird besonderer Wert auf eine umfassende Dokumentation der Entwicklungsumgebung, der entwickelten Programme und des Versuchsaufbaues gelegt.

1.2 Methode

Aktive Lärmbekämpfung beruht auf dem Prinzip des Anti-Schalls, bei dem Mikrofone den Schall der Störquelle aufnehmen und Lautsprecher ein mittels Computer oder DSP berechnetes Anti-Schall-Feld ausstrahlen, so dass das resultierende Signal möglichst minimal wird [1]. Bei der Berechnung des Gegensignals können adaptive Filter und der sogenannte Least-Mean-Square-Algorithmus (LMS) zur Ermittlung geeigneter Filterkoeffizienten eingesetzt werden. Speziell sollen in dieser Arbeit filtered-x-LMS-Algorithmen (Feedforward und Feedback) programmiert werden [2].

Um diese filtered-x-LMS-Algorithmen umsetzen zu können, ist die Kenntnis der Impulsantworten der Sekundärstrecken notwendig. Eine gut geeignete Möglichkeit der Messung von Impulsantworten, die mit kurzen Messzeiten auskommt und gering gegen Störungen anfällig ist, besteht in der Verwendung sogenannter Maximum Length Sequences (MLS). Maximalfolgen („maximum length sequences“ oder „maximum length binary sequences“) sind periodische, binäre und pseudo-stochastische Rauschsignale mit L=2m-1 Werten die 1 oder -1 betragen können [3], [4], [5], [6]. Da die Energiedichte von Maximalfolgen über alle Frequenzen gleich ist, erlauben

Sie einen hohen Störabstand (mehr als 30 dB gegenüber einer Einzelimpulsmessung) [7], so dass das Messobjekt nicht kurzzeitig mit hohen Pegeln strapaziert werden muss.

Maximalfolgen haben eine Autokorrelationsfunktion, die dem Ideal eines einzelnen Dirac-Impulses möglichst nahe kommt. Durch die Berechnung der Kreuzkorrelation zwischen ausgesendeter MLS und Systemantwort kann somit die Impulsantwort des zu untersuchenden Systems ermittelt werden.

Systemidentifikation und die ANC-Routinen werden zweikanalig vollständig auf dem DSP-Entwicklungssystem ADSP-21161N EZ-KIT Lite® Evaluation Kit in Assembler implementiert. Dazu ist es nötig, sich in die Programmierung des digitalen Signalprozessors auf Registerebene einzuarbeiten. Mit diesem System wird es dann möglich sein, Lärmminderungsversuche unter unterschiedlichsten Bedingungen durchzuführen, ohne vorher extra die für die ANC-Routinen benötigten Impulsantworten der Sekundärstrecken ermitteln und einlesen zu müssen.

Getestet wird das beschriebene ANC-System abschließend an einem Anti-Schall-Fenster [8], [9], [10], [11], [12]. Dieses flexible System soll dazu dienen, weitere Forschungen auf dem Gebiet der aktiven Lärmbekämpfung in verschiedensten Umgebungen und unter unterschiedlichsten Bedingungen zu ermöglichen.

2 Theoretische Grundlagen

2.1 Identifikation akustischer Systeme

2.1.1 Überblick

Die Ermittlung von Transferfunktionen und der zugehörigen Impulsantworten ist eine zentrale Aufgabe in praktisch allen Bereichen der Akustik. Beispiele hierfür sind die Aufnahme von Frequenz- und Phasengang von Lautsprechern oder Mikrofonen, die Raumakustik oder wie im Fall dieser Arbeit die Systemidentifikation für Active-Noise-Control-Systeme.

Prinzipiell ist der Gegenstand der Systemidentifikation, das stationäre und dynamische Verhalten eines realen Objektes zu bestimmen und in einer geeigneten Modellstruktur abzubilden. Es existieren verschiedene Ansätze für Systembeschreibungen, die an dieser Stelle kurz vorgestellt werden [13], [14].

In nichtparametrischen Modellen wird der Zusammenhang von Eingangs- und Ausgangsgrößen des Systems mit Wertefolgen bzw. nichtanalytischen Funktionen in Form von Messkurven ausgedrückt. Das können z.B. der Frequenzgang (Frequenzbereich) oder die Impulsantwort (Zeitbereich) sein.

Parametrische Systembeschreibungen haben zum Ziel, analytische Ausdrücke, d.h. Gleichungen zu finden, um tiefere Einblicke in das Systemverhalten zu gewinnen. Wichtig ist, dass Original und Modell die gleiche Abbildung ausführen. Parametrische Verfahren sind meist an eine konkrete Modellstruktur gebunden, um Modelle mit wenigen physikalisch interpretierbaren Parametern zu erhalten. Nach der Modellierung durch die Systemidentifikation werden den Parametern Zahlenwerte zugewiesen.

In Zustandsraummodellen werden die Beziehungen zwischen Eingangssignalen, Zustandsgrößen des Systems und Ausgangsgrößen abgebildet. Das System wird durch eine Zusammenfassung der Einzelzustände beschrieben.

Weitere Modelle zur Systembeschreibung sind u.a. das rationale Systemmodell, das Eigenschwingungsmodell und das Modalmodell.

Bei der Aufgabe der Systemidentifikation ist zu beachten, ob das zu untersuchende System „nett“, also linear, stabil und zeitinvariant (LTI-System) oder „garstig“, also nichtlinear und/oder zeitvariant ist. Unter realen Bedingungen kann es keine exakt linearen, zeitinvarianten Systeme geben. Allerdings kann man ein lineares Modell ansetzen, wenn der Arbeitsbereich linearisierbar ist und sich das System während der Messung nur unwesentlich ändert. Im Folgenden wird immer von LTI-Systemen ausgegangen.

2.1.2 Methoden der Systemidentifikation

Es existieren verschiedene nichtparametrische Messverfahren zur Identifikation akustischer Systeme. Es werden nichtparametrische Verfahren betrachtet, da als Ergebnis der Systemidentifikation eine Impulsantwort zur Weiterverarbeitung in den ANC-Algorithmen vorliegen soll. Die wichtigsten Verfahren werden nun kurz vorgestellt. Grundlage aller Techniken ist eine deterministische oder stochastische Auswertung der Systemantwort y(n) nach Anregung durch ein geeignetes Testsignal x(n). Voraussetzung ist lineares, zeitinvariantes Verhalten des Testsystems – auftretende Nichtlinearitäten werden als vernachlässigbar betrachtet.

Die einfachste und älteste Methode zur Systemidentifikation ist die Anregung des Messsystems mit einer Impuls- oder Sprungfunktion. Diese Technik wird auch Transientenanalyse genannt. Die aufgenommen Antwort des Systems auf eine Anregung durch einen Dirac-Impuls δ(n), z.B. mittels Knallfunkensender, durch Pistolenschuss oder durch einen platzenden Ballon in der Raumakustik, resultiert unmittelbar in der gewünschten Impulsantwort. Um daraus die Übertragungsfunktion zu bestimmen, ist nur eine einzige FFT nötig – der Rechenaufwand ist also vergleichsweise sehr gering. In Formeln ausgedrückt, stellt sich dieses Verfahren folgendermaßen dar:

$$y(n) = h(n) + z(n) = \hat{h}(n)$$

$$x(n) = \delta(n) = \begin{cases} 1 & n = 0 \\ 0 & sonst \end{cases}$$

$$y(n) = h(n) * x(n) + z(n)$$

Ein Dirac-Impuls, auch Delta-Funktion genannt, ist unendlich schmal und hoch, hat aber den Flächen- bzw. Energieinhalt 1. In der Realität ist dieser ideale Dirac-Impuls natürlich nicht zu erzeugen. Für die Audiomesstechnik genügen jedoch Impulse mit einem ausreichend frequenzkonstanten Amplitudenverlauf. Störungen z(n) bei der Messung gehen direkt in das Ergebnis ein. Durch Mittelung mehrerer Messungen ist es aber möglich, diese zu minimieren, wenn das Störsignal mittelwertfrei und unabhängig (unkorreliert) vom System ist. Der größte Nachteil der Messung mit einem Impuls als Anregungssignal ist dessen geringer Energiegehalt. Vor allem bei Messungen im tieffrequenten Bereich reicht die Dynamik daher oft nicht aus, was aber durch Mittelungen verbessert werden kann. [14], [15]

Die Fourier-Messtechnik ermöglicht eine höhere Messdynamik, da hierbei Anregungssignale verwendet werden können, deren Energie über die gesamte Messdauer

gleichmäßig verteilt ist. Zur Anregung werden Pseudozufallsrauschen, Gleitsinus oder andere definierte Signale verwendet. Es können auch a-priori unbekannte Signale eingesetzt werden, solange dieses Signal zum Zeitpunkt der Entfaltung (in Echtzeit) vorliegt. Diese sogenannte „quellenunabhängige“ Messung wird vor allem bei Konzerten oder Proben eingesetzt, um die Messung mit speziellen, aber störenden Anregungssignalen verkürzen oder vermeiden zu können. Einfache Messsysteme können nur das Betragsspektrum darstellen. Beispiele hierfür sind Handschallpegelmesser und tragbare Analysatoren. Mit modernerer Messtechnik können auch die komplexe Übertragungsfunktion oder die Impulsantwort eines LTI-Systems ermittelt werden. Dabei wird die Antwortfunktion y(t) als Faltungsprodukt des Anregungssignals x(t) und der Übertragungsfunktion h(t) dargestellt:

$$y(t) = h(t) * x(t)$$

Im Frequenzbereich wird diese Faltung zum einfachen Produkt und kann nach der Übertragungsfunktion H(ω) aufgelöst werden:

$$H(t) = \frac{Y(\omega)}{X(\omega)}$$

Dieses Verfahren wird auch als inverse Filterung bezeichnet. In Abhängigkeit von der Art der zu erwartenden Störsignale können geeignete Messsignale verwendet werden. Grundsätzlich bestimmt die Länge der Messsignale und die Anzahl der Zeitmittelungen, wie stark Störungen verringert werden können. Für kurze Messdauern sollten Anregungssignale mit hoher Energiedichte (z.B. (Pseudo-)Rauschsignale) gewählt werden, um einen hohen Signal-Rauschabstand zu ermöglichen. In letzter Zeit hat sich aufgrund messtechnischer Vorteile der Gleitsinus (Sweep) als wichtigstes Anregungssignal durchgesetzt. In seiner einfachsten Form ist der Gleitsinus ein fortlaufendes Sinussignal, dessen Frequenz sich mit der Zeit ändert:

$$s(t) \propto \sin(\varphi(t))$$

Die Verfügbarkeit immer leistungsfähiger Computer- und DSP-Technik ermöglichte erst diese Entwicklung. Der Gleitsinus stellt im Gegensatz zu anderen Signalformen eine stetige Funktion der Frequenz dar, was bei der D/A-Wandlung Verzerrungen

vermeidet. Außerdem sind harmonische Verzerrungsanteile bei Störungen im Messweg gut von der Impulsantwort separierbar [16].

Das Time Delay Spectrometry-Verfahren (TDS) wurde Ende der 60er Jahre von Richard Heyser für Anwendungen im Audiobereich entwickelt. Es ist ein 2-Kanal-Messsystem und arbeitet mit frequenzlinearen Sweeps konstanter Amplitude und kommt bei der Ermittlung der Übertragungsfunktion ohne Fouriertransformation aus. Ein TDS-System besteht aus einem Sweep-Generator, der ein Sinussignal und ein um 90° versetztes phasenstarres Cosinussignal steigender Frequenz erzeugen kann. Das zu testende Gerät wird mit dem Sinus angeregt und das aufgenommen Signal wird dividiert durch den Original-Sinus, um den Realteil der Übertragungsfunktion zu erhalten. Der Imaginärteil wird durch Division des Ausgangssignals durch das 90° phasenverschobene Cosinussignal berechnet. Diese Postprocessing-Signale werden dann tiefpassgefiltert. Nachdem also der Gleitsinuston in das System eingespeist wurde, wird das Ausgangssignal mit Bandpassfiltern gefiltert, deren Mittenfrequenz zeitverzögert (time-delayed) mitläuft. Durch Veränderung der Offset-Zeit zwischen Anregungssignal (Sweep) und Mitlauffilter einschließlich variierbarer Bandbreite wird eine frequenzabhängige Darstellung bestimmter Zeitabschnitte der Zeitfunktion möglich. Ein Vorteil dieser Messmethode ist der hohe Signal-Rausch-Abstand, der echofreie Messungen in nichtechofreien Messumgebungen unabhängig von Störgeräuschen ermöglicht.

Zur Identifikation von nichtlinearen Systemen gibt es inzwischen Ansätze unter Verwendung von neuronalen Netzen („time-delay neural networks“). Diese können auch als nichtlineare Filter aufgefasst werden. Prinzipiell wird bei der Modellbildung mit neuronalen Netzwerken das zu modellierende System als „Black Box“ verstanden und die Übertragungseigenschaften durch Kenntnis von Ein- und Ausgangssignal ermittelt [17]

Für die verschiedensten Anwendungen können unterschiedliche Messsignale zum Einsatz kommen, doch auch unabhängig von der Art des Messverfahrens müssen Messsignale einige Bedingungen erfüllen. Natürlich müssen alle interessierenden Frequenzen mit ausreichendem Pegel vorhanden sein, um die Messung weniger anfällig gegen Störungen zu machen. Auch sollten die Amplitudenunterschiede benachbarter Frequenzkomponenten nicht zu hoch ausfallen, da sonst Nichtlinearitäten zustande kommen würden. Die spektrale Verteilung des Anregungssignals sollte weiterhin sowohl den Belastungsgrenzen der verwendeten Messtechnik als auch der anwesender Personen angepasst sein. Tabelle 1 stellt eine Reihe möglicher Testsignalformen übersichtlich dar [14].

Tabelle 1: Testsignale

Testsignal			
deterministisch		stochastisch	
periodisch	transient	natürlich	künstlich
Monofrequente Signale (Sinus) Multifrequente Signale (Rechtecksign., Multitonsign.) Pseudo-stochastische Signale (MLS-Folgen)	Impulsfunktion Sprungfunktion Gleitsinus	Rauschen	Parametrische Rauschsignale Farbiges Rauschen

2.2 Systemidentifikation mit MLS-Technik (Korrelationsmesstechnik)

2.2.1 Einleitung

Die Maximalfolgentechnik ist ein Verfahren zur nichtparametrischen Systemidentifikation und eine besondere Form der digitalen Impuls- und Korrelationsmesstechnik. Sie kann überall dort eingesetzt werden, wo Impulsantworten oder stationäre Übertragungsfunktionen von akustischen oder elektroakustischen Systemen bestimmt werden sollen. Voraussetzung ist dabei jedoch, dass es sich um lineare, zeitinvariante Systeme (LTI) handelt. Als Anregungssignale werden Maximalfolgen verwendet. Diese MLS-Folgen sind keine impulsförmigen Signale, sondern pseudo-stochastische, dem weißen Rauschen ähnliche Signale, deren Signaleigenschaften gut für die Systemidentifikation geeignet sind. Hierbei wird vor allem ausgenutzt, dass die Autokorrelationsfunktion einer Maximalfolge dem Ideal eines einzelnen Dirac-Impulses möglichst nahe kommt. Bei der Korrelationsmesstechnik mit Maximalfolgen ist es aufgrund schneller Signalverarbeitungsalgorithmen wie der Hadamard-Transformation möglich, Impulsantworten für Echtzeit-Anwendungen zu ermitteln. Außerdem ist es mit dieser Technik durch Verwendung langer Folgen oder durch Mittelung möglich, trotz widriger Messbedingungen korrekte Ergebnisse zu erhalten. Der folgende Abschnitt beschäftigt sich eingehend mit der Maximalfolgenmesstechnik [14], [18].

2.2.2 MLS-Signale

Maximalfolgen („maximum length sequences“ oder „maximum length binary sequences“) sind periodische, binäre und pseudo-stochastische Rauschsignale mit $L=2^m-1$ Werten die 1 oder -1 betragen können. In Abbildung 1 ist eine MLS-Folge der 7. Ordnung mit 127 Samples dargestellt. Da die Energiedichte von Maximalfolgen über alle Frequenzen gleich ist, erlauben Sie einen hohen Störabstand (mehr als 30 dB gegenüber einer Einzelimpulsmessung [7]), so dass das Messobjekt nicht kurzzeitig mit hohen Pegeln strapaziert werden muss. Sie haben einen sehr niedrigen Formfaktor (Crest-Faktor, Verhältnis von Peak zu RMS) und können so eine große Menge breitbandiger Energie in ein System einbringen, ohne es damit in Verzerrung oder Sättigung zu treiben.

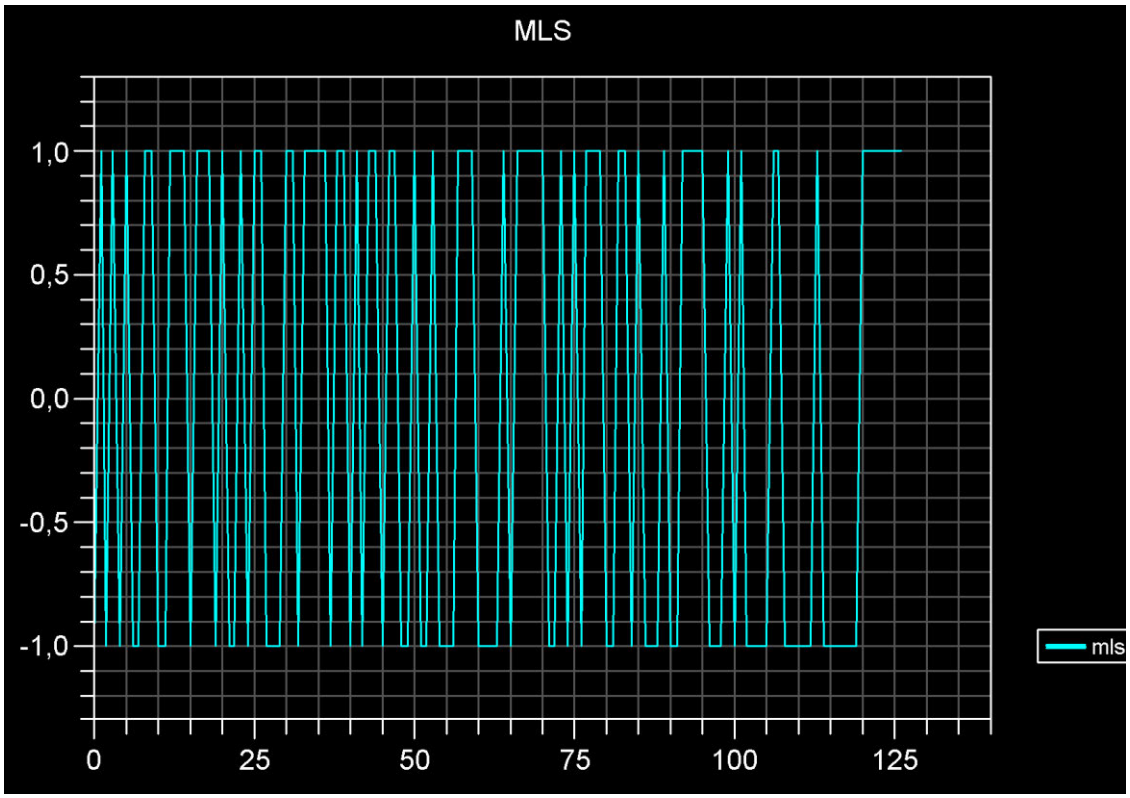


Abbildung 1: MLS-Folge der Ordnung 7

MLS-Folgen lassen sich relativ einfach in nahezu beliebiger Länge mittels eines rückgekoppelten binären Schieberegisters oder per Software generieren. Eine Maximalfolge hat folgende Form:

$$x[n] = X \cdot (-1)^{a[n]}; \qquad X \in \mathbb{R}$$

a[n] im Exponenten kann entweder 0 oder 1 werden. Der Effektivwert (RMS) dieses Signals hat den Wert X und der Peak-Wert liegt bei 1. Somit hat der Formfaktor (Peak/RMS) den möglichst niedrigen Wert.

Zur Erzeugung kann ein rückgekoppeltes Schieberegister mit m Stellen verwendet werden, bei dem alle möglichen Zustände durchlaufen werden außer der Kombination, bei der alle Stellen mit Nullen besetzt sind. Nur mit einer geeigneten Rückkopplungsvorschrift, die durch bestimmte „primitive Polynome“ definiert wird, lässt sich die „maximale Periode“ der Folge erzeugen. Jedes Element der Maximalfolge erhält man durch XOR-Verknüpfung mehrerer Ausgänge des Schieberegisters. Dieses wird auf den Eingang zurückgegeben. Am Ausgang entsteht damit eine deterministische Rechteckimpulsfolge, deren Elemente x[n] sich nach $L=2^m-1$ Perioden wiederholen. Die Art der Rückführung auf den Eingang wird durch einen Koeffizientenvektor b[m] definiert, der die Werte 0 oder 1 annehmen kann. Die Koeffizientenvektoren für die verschiedenen MLS-Längen entsprechen primitiven Polynomen und wurden [19] entnommen. Damit ist die Generierung von Maximalfolgen mit theoretisch bis zu $2^{30}-1=1.073.741.823$ Samples und einer Länge von mehr als 405 Minuten bei einer Abtastrate von 44,1 kHz möglich. Durch diese Polynome wird auch der erste Zustand des Schieberegisters definiert:

$$u_1 = z^{-1}(b_1 u_1 \oplus b_2 u_2 \oplus ... \oplus b_M u_M)$$

Die weiteren Zustände werden durch zeitliche Verschiebung aus den vorangegangenen Registerstufen gebildet:

$$u_m = z^{-1} u_{m-1};$$

Die Abbildung 2 zeigt eine Schieberegisterkette der Ordnung m=7. Das zur Erzeugung verwendete Polynom ist $b[m]=x^6+1$, in Binärform 1000001. Bei jedem Takt wird nun der Inhalt des Schieberegisters um eine Bitstelle nach links verschoben, wobei das höchstwertige Bit verloren geht. Das niederwertigste Bit ergibt sich aus Modulo-2-Addition (XOR-Verknüpfung) einer durch das Bildungspolynom bestimmten Menge von Bits des vorhergehenden Zustandes. In diesem Beispiel sind das die Bits 0 und 6. Modulo-2-Addition bedeutet, dass wenn diese Bits 0 und 6 eine gerade Anzahl von Einsen enthalten, eine Null in das niedrigstwertige Bit (Bit 0) geschrieben wird – bei ungerader Anzahl entsprechend eine Eins. Der Generator durchläuft hierbei also 127 Zustände, wobei eine Maximalfolge mit einer Periode von 127 Samples generiert wird. Noch handelt es sich hierbei um eine Folge von

„Nullen“ und „Einsen“. Die „Nullen“ müssen nun noch durch „-1“ ersetzt werden. Dies geschieht durch den abschließenden Berechnungsschritt x[n]=x[n]*2-1. Für das Verständnis der Programmierung eines MLS-Generators kann [20] hilfreich sein.

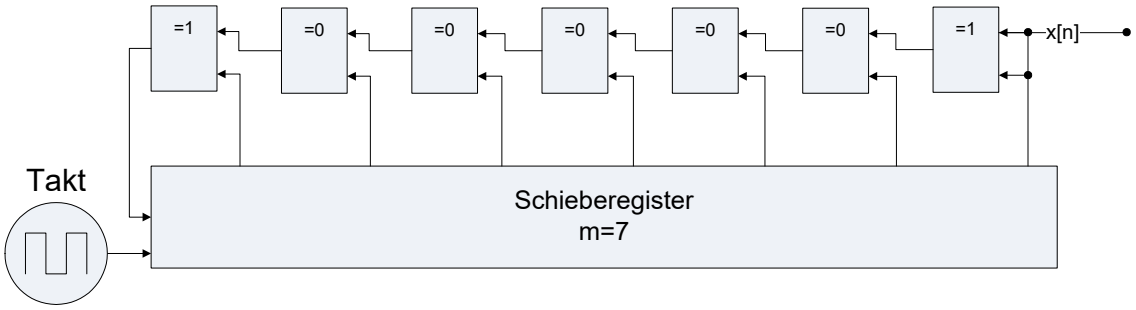


Abbildung 2: Schieberegisterkette der Ordnung m=7

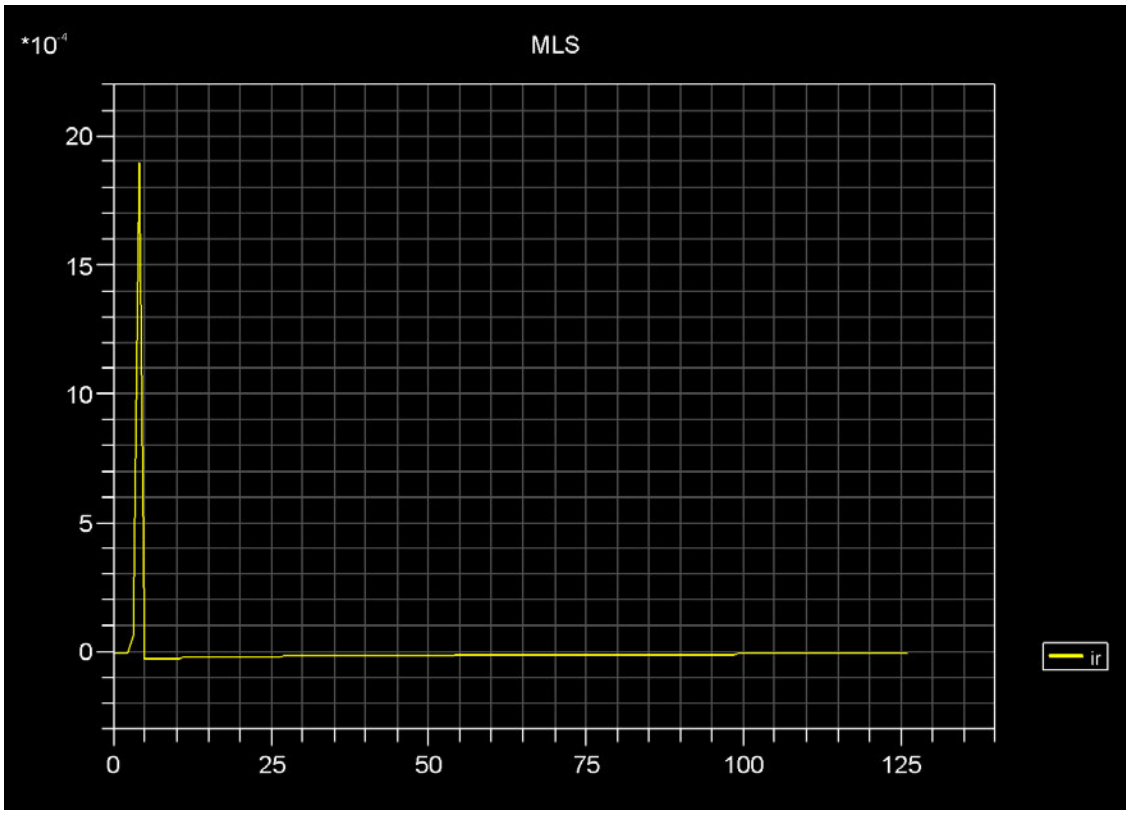


Abbildung 3: Kreuzkorrelation zwischen MLS und Systemantwort bei kurzgeschlossenen Anschlüssen

Abbildung 3 zeigt die Kreuzkorrelation zwischen MLS und Antwort bei kurzgeschlossenem Ein- und Ausgang am DSP-Board. Der ideale Dirac-Impuls wird dabei natürlich aufgrund der Verzerrungen, verursacht unter anderem durch Verzerrungen der AD/DA-Wandler und Verstärker, nicht erreicht.

Abschließend seien hier noch sogenannte Gold-Sequenzen (gold sequences) erwähnt. Diese haben bessere Kreuzkorrelationseigenschaften und kommen unter anderem bei Codemultiplexverfahren im Mobilfunkbereich zum Einsatz. Diese Gold-Sequenzen können durch Modulo-2-Addition zweier gleichlanger Maximalfolgen generiert werden [21].

2.2.3 Ermittlung der Impulsantwort

Maximalfolgen eignen sich besonders für die Impulsmesstechnik, da sich Kreuzkorrelationsalgorithmen gut auf Rechnern oder DSPs implementieren lassen. Zur Berechnung der Kreuzkorrelation eignen sich drei Algorithmen: die diskrete Faltung, FFT-Faltung und Hadamard-Transformation (FHT). In dieser Arbeit wurde zur Berechnung der Impulsantworten die diskrete Faltung implementiert. Dieser Berechnungsalgorithmus benötigt zwar die meisten Operationen (N^2), lässt sich aber am einfachsten umsetzen [16]. Da die Ermittlung der Impulsantworten im vorliegenden ANC-System vor den eigentlichen Echtzeit-Lärmbekämpfungsalgorithmen abläuft, sind hierbei keine Echtzeit-Anforderungen zu erfüllen und die Berechnung darf einige Sekunden in Anspruch nehmen.

Die Impulsantwort unseres Systems (Schallschutzfenster) kann also mittels periodischer Kreuzkorrelation von Messfolge x[n] und Systemantwort y[n] berechnet werden. Dies entspricht einer Faltung der Signale wobei noch auf die Länge der Maximalfolge L normiert wird. Die diskrete Faltung ist definiert als:

$$(x * y)[n] = \frac{1}{L} \sum x[k] * y[n-k]$$

Die FFT-Faltung benötigt nur $N(4\log_2 N+1)$ Berechnungen. Allerdings passen die Periodenlängen von Maximalfolgen ($L=2^m-1$) und die FFT-Blocklängen ($L=2^m$) nicht zusammen und es müssten zur Vermeidung von Fehlern aufwändige Abtastratenanpassungen vorgenommen werden [16].

Die eleganteste und schnellste Lösung zur Faltung und Korrelation im Zeitbereich bietet die schnelle Hadamard-Transformation (FHT). Grundlage der FHT bildet ein FFT-ähnlicher Butterflyalgorithmus, der aber nicht komplexwertig arbeitet. Hierbei wird das Korrelationsintegral als Matrizenoperation der Multiplikation eines Vektors mit einer Hadamard-Matrix dargestellt. Eine Hadamard-Matrix H ist eng mit Maximalfolgen verwandt. Weiterführende Erläuterungen zur FHT finden sich unter anderem in [16], [18] und [22].

Zur Bestimmung der benötigten Impulsantworten werden also Maximalfolgen generiert, die als Messsignale fungieren. Diese werden über Lautsprecher im Schallschutzfenster abgestrahlt, wobei vorher eine Tiefpassfilterung stattfindet, um Aliasing-Effekte zu vermeiden. Die Antwort des Systems wird mit Messmikrofonen aufgenommen und nach Tiefpassfilterung dem DSP-Board zugeführt. Dort wird die Berechnung der Impulsantworten mittels Kreuzkorrelation von ausgesendeter Maximalfolge und Systemantwort berechnet. Die Störquelle muss hierbei natürlich noch stumm sein. In Abbildung 4 ist der gesamte Aufbau schematisch dargestellt.

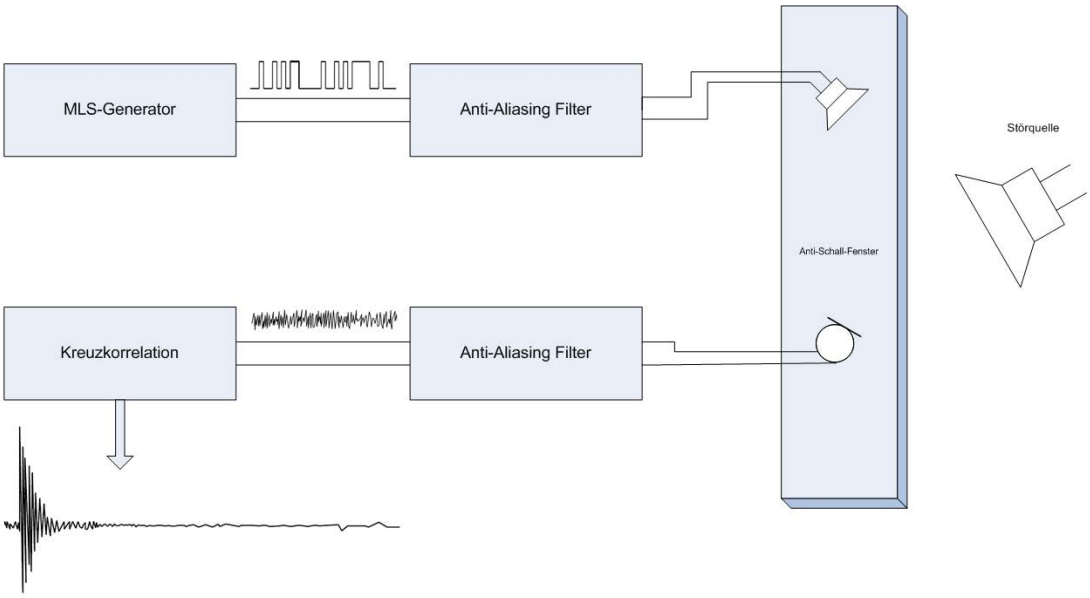


Abbildung 4: Prinzip der Ermittlung der Impulsantwort

2.2.4 Anpassung an die Erfordernisse der Messung

Das System bietet die Möglichkeit, mehrere Mess-Maximalfolgen hintereinander auszugeben und die Systemantwort zu mitteln. Dadurch können Störeinflüsse herausgemittelt werden.

Um Messfehler durch harte Clipverzerrungen zu vermeiden, ist es nötig, die Maximalfolge deutlich unterhalb der Vollaussteuerung in das System einzuspeisen. Der Pegel sollte in der Praxis unterhalb von -8 dBFS (Dezibel full scale) gehalten werden, um die Verzerrungen, die sich durch Phantompeaks in der Impulsantwort äußern, zu verhindern [15].

Desweiteren gibt es weitere Forderungen an das Messsystem. Natürlich muss, wie bei diskreten abgetasteten Signalen immer gefordert, das Abtasttheorem eingehalten werden, d.h. die Abtastfrequenz muss mindestens der doppelten Grenzfrequenz des

Signals entsprechen. Um „zeitliches Aliasing“ zu vermeiden, muss sichergestellt werden, dass die zu messende Impulsantwort kürzer als die Periode der ausgegebenen Maximalfolge ist. Grundsätzlich sollte die Maximalfolge so lang wie möglich sein, um die größtmögliche Messdynamik zu erreichen [22].

2.3 Aktive Lärmbekämpfung

2.3.1 Grundlagen

Aktive Lärmbekämpfung beruht auf dem Prinzip des Anti-Schalls, bei dem Mikrofone den Schall der Störquelle aufnehmen und Lautsprecher ein mittels Computer oder DSP berechnetes Anti-Schall-Feld ausstrahlen, so dass das resultierende Signal möglichst minimal wird. Bereits in den 70er Jahren des 19. Jahrhunderts wurden von Lord Rayleigh erste Versuche mit sich überlagernden Schallfeldern durchgeführt. Patentanmeldungen in den 1930ern durch Henri Marie Coanda und Paul Lueg zielten dann bereits auf aktive Lärmminderung ab, wobei Lueg als erster die Idee der aktiven Lärmminderung formuliert hat. In den 50er Jahren des 20. Jahrhunderts führten dann H. F. Olson und Everet May erste Laborexperimente mit elektroakustischen Komponenten durch und zeigten auch erste potentielle Anwendungen auf. Erst in jüngerer Zeit ist man jedoch mit Hilfe adaptiver Filter in der Lage, funktionierende Antischallsysteme zu entwickeln [1], [2].

Der Antischall-Technik sind jedoch Grenzen gesetzt. Im Gegensatz zu passiven Absorberlösungen, die höhere Frequenzen gut unterdrücken können, sind aktive Systeme vor allem für niederfrequente Anwendungen geeignet. Auch sollten die Störgeräusche nicht allzu große Schwankungen aufweisen, also stationär sein.

Die grundlegenden Ziele der aktiven Lärmbekämpfung sind Minimierung der Schallabstrahlung, Absorption von Schallleistung oder das Herstellen einer „Zone der Ruhe“ („zone of quiet“). Das Störsignal wird auch als Primärquelle bezeichnet, die Lautsprecher, die das Gegensignal abstrahlen, nennt man Sekundärquellen.

Es gibt zwei Strategien in der aktiven Lärmbekämpfung – Feedback und Feedforward. Der Feedback-Ansatz wurde zuerst von Olson und May beschrieben und ist in Abbildung 5 dargestellt. Hier stellt e das entsprechend der Kombination aus Störsignal und Rückkopplung vom Fehlermikrofon aufgenommene Signal dar. Die elektrische Transferfunktion H repräsentiert eine einfache Verstärkungs- und Phasenanpassung bzw. –inversion. Die Transferfunktion C zwischen Sekundärquelle und Ausgang am Mikrofon wird als Sekundärstrecke oder Fehlerstrecke bezeichnet. Das Ziel ist nun entsprechend der ANC-Thematik, den Ausgang e(n) klein im Vergleich zu d(n) werden zu lassen und so den Schalldruck am Fehlermikrofon zu minimieren [2].

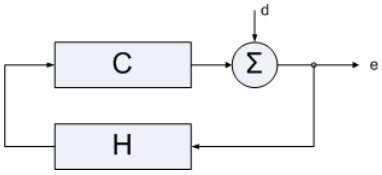


Abbildung 5: Blockdiagramm FB

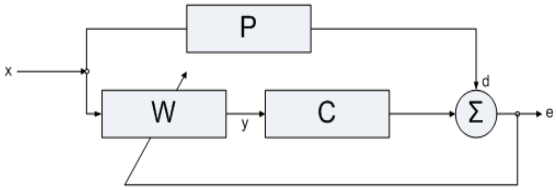


Abbildung 6: Blockdiagramm FF

Im Gegensatz zur Feedback-Variante kommt bei dem Feedforward-Ansatz zusätzlich ein separates Referenzsignal x zum Einsatz. Es wird durch einen Regler (W) angepasst und dann über die Sekundärquelle ausgegeben. Dieses Referenzsignal muss gut mit dem Störsignal korreliert sein, denn es soll dem System mehr Informationen über das Störsignal geben, und zwar bevor dieses das Fehlermikrofon erreicht Abbildung 6 das Blockdiagramm Feedforward ANC-Systems. P repräsentiert die Primärstrecke, C wieder die Sekundärstrecke.

2.3.2 Der LMS-Algorithmus (filtered-x-LMS-Algorithmus)

Generell beruhen alle ANC-Strategien auf dem Prinzip der Superposition (Überlagerung). Zur Berechnung eines geeigneten Gegensignals können adaptive Filter und der sogenannte Least-Mean-Square-Algorithmus (LMS) zur Ermittlung geeigneter Filterkoeffizienten eingesetzt werden. Weiterführende Informationen zu LMS-Algorithmen finden sich z.B. in [23]. Dieser „normale“ LMS-Algorithmus ist jedoch ohne Modifikation nicht für ANC-Zwecke zu gebrauchen, da er zu Instabilitäten aufgrund von Phasenverschiebungen zwischen dem Signal des adaptiven Filters und der Sekundärstrecke neigt. Um dieses Problem zu lösen, wurde ein Filter eingeführt, der ein mit der Impulsantwort des Systems gefiltertes Referenzsignal errechnet. Dieser modifizierte LMS-Algorithmus heißt filtered-x-LMS-Algorithmus und wurde in dieser Arbeit implementiert. Das Block-Diagramm in Abbildung 7 zeigt die prinzipielle Funktionsweise des filtered-x-LMS-Algorithmus„ (Feedforward).

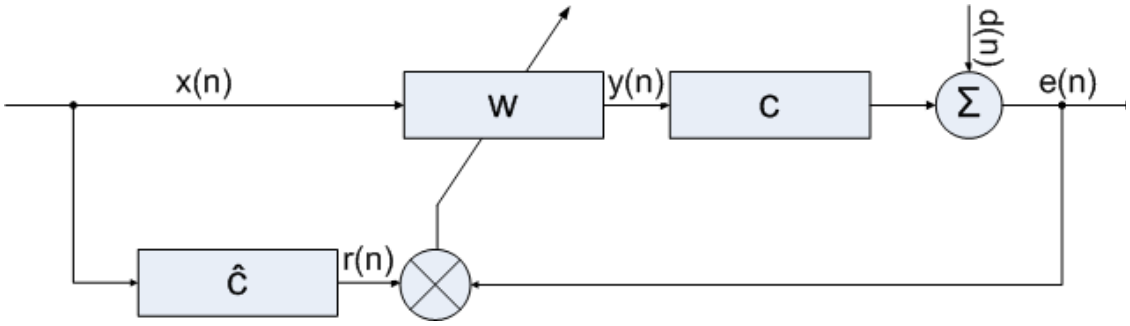


Abbildung 7: Prinzipschaltbild des filtered-x-LMS-Algorithmus,, (Feedforward)

(x(n) - Referenzsignal; y(n) - Ausgangssignal; r(n) - gefiltertes Referenzsignal; e(n) - Fehlersignal; d(n) - Störsignal; w - Adaptionsfilter; c - Impulsantwort; - berechnete Impulsantwort)

Das Ausgangssignal y(n) wird durch Filterung mit den Adaptionsfilterkoeffizienten w(n) erzeugt. Im Gegensatz zum herkömmlichen LMS-Algorithmus wird das Referenzsignal x(n) mit der Impulsantwort c(n) gefiltert, bevor die Adaptionsfilterkoeffizienten durch Multiplikation des gefilterten Referenzsignals r(n) mit dem Fehlersignal generiert werden. Die zugrunde liegenden Formeln, die in der Programmierung umgesetzt werden müssen, lauten:

$$\vec{y}_m[n] = (\vec{x}_k * \vec{w}_{mk})[n]$$

$$\vec{r}_{lmk}[n] = (\vec{x}_k * \vec{c}_{lm})[n]$$

$$\vec{w}_{mk}[n+1] = \vec{w}_{mk}[n] - \alpha \sum_{l=1}^{L} \vec{r}_{lmk}[n] e_l[n]$$

Die Indizes symbolisieren die verschiedenen Vektoren für mehrere Sekundärstrecken (l - Fehlersensoren; m - Sekundärquellen; k - Referenzsignale).

In der Feedback-Variante muss zusätzlich das Referenzsignal x(n) mittels Filterung des Ausgangssignals y(n) mit der Impulsantwort c(n) und Addition des Fehlersignals e(n) geschätzt werden. Es kommt daher eine weitere Formel hinzu:

$$\vec{x}_k[n] = e_l + \sum_{m=1}^{M} (\vec{y}_m * \vec{c}_{lm})[n]$$

Die Abbildung 8 zeigt das Block-Diagramm des filtered-x-LMS-Algorithmus,, (Feedback):

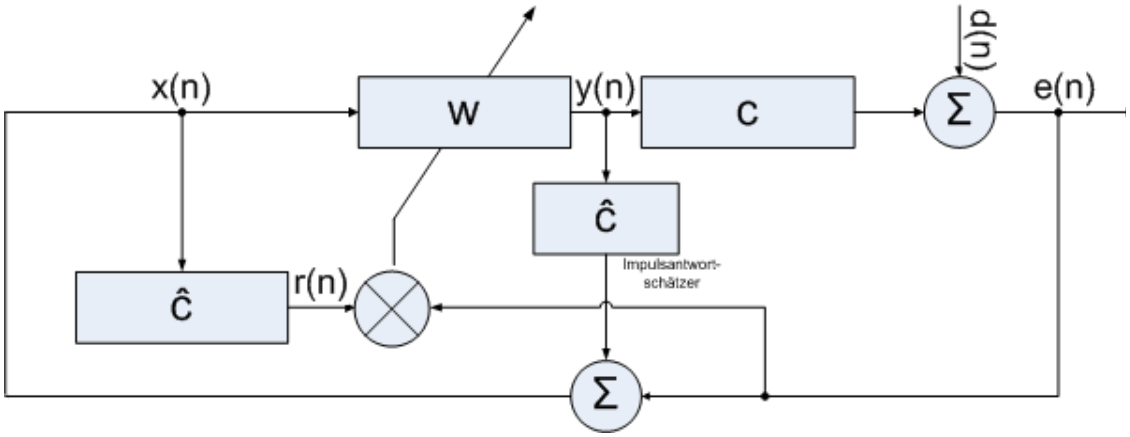


Abbildung 8: Prinzipschaltbild des filtered-x-LMS-Algorithmus,, (Feedback)

(x(n) - Referenzsignal; y(n) - Ausgangssignal; r(n) - gefiltertes Referenzsignal; e(n) - Fehlersignal; d(n) - Störsignal; w - Adaptionsfilter; c - Impulsantwort(-schätzung); ĉ - berechnete Impulsantwort)

Wie bereits aus den Formeln ersichtlich wurde, können mit diesen Algorithmen auch mehrkanalige Systeme realisiert werden. Ein einkanaliges System ist nur dazu geeignet, eine Zone der Ruhe um das Fehlermikrofon zu schaffen. Um eine globale Lärmminderung zu erreichen, muss mit mehreren Fehlermikrofonen gemessen werden, um die gesamte Schallenergie im System besser abzuschätzen zu können [2]. Dieser Multiple Error LMS-Algorithmus wurde 1985 von S. J. Elliot und P. A. Nelson entwickelt und kann hier nachgelesen werden [24].

2.4 Das Anti-Schall-Fenster

2.4.1 Einleitung

Vor einigen Jahren wurde am Institut für Technische Akustik (heute Institut für Strömungsmechanik und Technische Akustik (ISTA)) der TU Berlin ein Doppelglasfenster entwickelt, dessen Schalldämmung durch aktive Maßnahmen verbessert werden sollte. In diesem Doppelglasfenster mit geringem Scheibenabstand (4mm) wurden Lautsprecher und Mikrofone eingebaut, die den Einsatz aktiver Lärmbekämpfungsalgorithmen ermöglichen sollen. Durch geeignete Ansteuerung der Lautsprecher mittels eines ANC-Systems soll der Schalldruck im Innern des Fensters minimiert werden [10].

2.4.2 Theorie

Physikalisch handelt es sich bei dem Doppelglasfenster um ein Masse-Feder-System mit einer bestimmten Resonanzfrequenz (Tonpilzresonanz), in deren Bereich die Schalldämmung des Fensters besonders gering ist. Messungen haben ergeben, dass die Tonpilzresonanz dieses Fensters bei 125 Hz liegt. Für theoretische Untersuchungen wurde ein analytisches modales Modell erstellt, das in [25] ausführlich dargestellt wird. Das Doppelglasfenster kann als System angesehen werden, bei dem die beiden Scheiben durch jeweils eine Biegewellendifferentialgleichung und der Zwischenraum durch eine Luftschallwellendifferentialgleichung beschrieben werden können. Anregungen für die jeweiligen Teilsysteme sind die einfallende Schallwelle, die Lautsprecher und die Einflüsse der jeweiligen benachbarten Teilsysteme, wodurch die drei Differentialgleichungen gekoppelt sind. Aus dieser Arbeit ergeben sich folgende mögliche zu minimierende Größen:

- abgestrahlte Schallleistung empfangsseitig
- mittleres Schnellequadrat einer der beiden Scheiben
- mittleres Schalldruckquadrat im Fensterhohlraum
- lokales Schnellequadrat einer der beiden Scheiben
- lokales Schalldruckquadrat im Fensterhohlraum

Vor allem die letzten beiden Größen sind für praktische Untersuchungen von Bedeutung, da sie sich durch Mikrofone als Fehlersensoren oder Beschleunigungsaufnehmer auf den Scheiben realisieren lassen. In einer umfangreichen Parameterstudie wurden Ansatzpunkte für Minimierungsmöglichkeiten untersucht [26]. Hier die wichtigsten Ergebnisse und Ansatzpunkte für die aktive Lärmminderung mittels Lautsprechern im Doppelglasfenster [11]:

- Eine Minimierung der abgestrahlten Schallleistung ist besser als eine Minimierung der inneren Scheibenschwingung, diese ist besser als eine Minimierung des Hohlraumschallfeldes und diese ist besser als eine Minimierung der äußeren Scheibenschwingung
- Bei Lautsprecheranregung entfaltet sich die Wirkung vor allem im Bereich der Tonpilzresonanz des Masse-Feder-Masse-Schwingers, der durch das Scheibe-Hohlraum-Scheibe-System gebildet wird
- Lautsprecher im Hohlraum sind besser als Kräfte auf den Scheiben
- Gemischte Varianten mit Lautsprechern und Kräften bringen keine Verbesserungen gegenüber der gleichen Anzahl an Lautsprechern

2.4.3 Aufbau und Eigenschaften

Es wurde ein Doppelglasfenster mit zwei 4 mm Glasscheiben und einem Scheibenabstand von 4 cm aufgebaut. Die Abmessungen der Scheiben betragen 100 x 125 cm. Die Abbildung 9 zeigt das in den Fensterprüfstand eingebaute Fenster. In das Fenster sind insgesamt 12 Lautsprecher und vier Mikrofone eingebaut, die sich einzeln oder auch in beliebiger Kombination ansteuern lassen.

Abbildung 9: Antischallfenster

Die verwendeten Lautsprecher sollten in den 4cm Zwischenraum passen und möglichst lang sein, um die schallabstrahlende Membran so groß wie möglich zu machen. Die Abmessungen betragen 130 x 33 x 33 mm, der Hersteller gibt eine Leistung von 2W an 16 Ω an und der Frequenzbereich liegt bei 180-17000 Hz. Dieser Frequenzbereich liegt über der Tonpilzfrequenz von 125 Hz, aber die Tatsache, dass die Lautsprecher nur in ein recht kleines Volumen abstrahlen, macht dieses Manko

teilweise wett. Auch mit diesen preiswerten Lautsprechern (Stückpreis 1€) lassen sich in kleinen, relativ schallhart abgeschlossenen Räumen bei tiefen Frequenzen recht hohe Schalldrücke erzeugen. Abbildung 10 zeigt die verwendeten Lautsprecher vor dem Einbau [8].

Abbildung 10: Lautsprecher vor dem Einbau

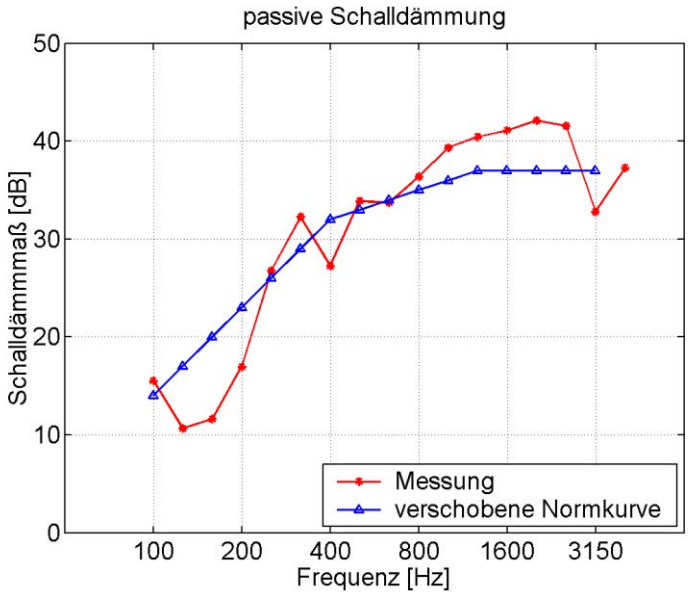


Abbildung 11: Schalldämmmaßkurve des Fensters

Die Tonpilzfrequenz des Fensters liegt bei 125 Hz. Das Schalldämmmaß des Fensters ohne aktive Maßnahmen beträgt in dieser 125 Hz-Terz etwa 11 dB und erreicht bei höheren Frequenzen etwa 43 dB (siehe Abbildung 11). Das bewertete Schalldämmmaß beträgt R_w'=33 dB [Quelle: Dr. André Jakob].

2.5 Digitale Signalprozessoren

2.5.1 Einführung

Digitale Signalverarbeitung ist eine der leistungsfähigsten Technologien des 21. Jahrhunderts und hat einen starken Einfluss auf die heutige Wissenschafts- und Techniklandschaft. In vielen Bereichen wurden revolutionäre Innovationen eingeführt, von denen viele auf eigens dafür entwickelten DSP-Technologien beruhen – mit einer speziellen Mathematik und entsprechenden Algorithmen [27].

Unter digitalen Signalprozessoren (DSPs) versteht man im Allgemeinen Rechnerstrukturen, die eine effiziente Implementierung von Algorithmen der digitalen Signalverarbeitung ermöglichen. Neben den weit verbreiteten universellen DSPs (General Purpose DSP), die frei programmierbar sind, gibt es inzwischen eine Vielzahl von Prozessoren, die nur für bestimmte Zwecke entworfen wurden – sogenannte fest verdrahtete DSPs. Hierbei wird meist so vorgegangen, dass ausgehend von den universellen DSPs die Architektur auf die speziellen Anforderungen wie Durchsatzrate, Leistungsverbrauch und Kosten optimiert und für den Massenmarkt angepasst wird.

Die wichtigsten zentralen Anforderungen an Digitale Signalprozessoren sind [28], [29], [30]:

- schnelle, flexible arithmetische Rechenwerke
- ungehinderter Datenfluss zu und von den Rechenwerken mit hoher Datendurchsatzrate
- Echtzeitverarbeitung (Real-Time)
- erweiterte Präzision und großer Dynamikbereich in den Rechenwerken
- zwei Adressgeneratoren mit Unterstützung für Ringpuffer
- effiziente Programm- und Befehlsausführung
- kostengünstige Single-Chip-Lösungen

2.5.2 Aufbau und Wirkungsweise von DSPs

Mikroprozessoren beruhen heutzutage entweder auf der Von-Neumann-Architektur (Personal Computer) oder der Harvard-Architektur (DSPs). Im Gegensatz zu Von-Neumann-Rechnern, die so konzipiert sind, dass der Speicherzugriff von der CPU auf Befehle und Daten nur über einen Bus erfolgt, besitzt die Harvard-Architektur getrennte Busse für Programm- und Datenspeicher. Befehls- und Datentransporte laufen darum gleichzeitig und ohne gegenseitige Beeinflussung ab, so dass in einem einzigen Befehlszyklus ein simultaner Zugriff auf zwei Speicherbereiche möglich ist. Dadurch werden komplexe Berechnungen in Echtzeit auch bei hohen Abtastraten möglich. Der für dieses Projekt benutzte ADSP-21161N arbeitet mit einer Taktfre-

quenz von 100 MHz. Damit sind bei einer Abtastrate von 44,1 kHz theoretisch mehr als 2000 Befehlszyklen in einem Abtastschritt möglich. Inzwischen gibt es verschiedene modifizierte Harvard-Architekturen. Im ADSP-21161N kommt z.B. eine Architektur zum Einsatz, bei der ein zusätzlicher Cache zur Zwischenspeicherung von Befehlen verwendet wird. Damit ist der Programmspeicher frei zum Auslesen des zweiten Operanden und die Befehlsausführungszeit wird verkürzt.

Grundsätzlich besteht ein DSP-Kern aus einem Datenrechenwerk (ALU), einem Adressrechenwerk und einem Programmcontroller, die parallel arbeiten. Das Datenrechenwerk besteht aus einem Hardware-Multiplizierer/Addierer (MAC), der eine Multiplikation mit Addition in einem Befehlszyklus ausführen kann und einem Barrelshifter für Schiebeoperationen. Das Datenrechenwerk (Datenaddressierungsgenerator) besteht meist aus zwei getrennten Einheiten und ist mit den Datenspeichern verbunden. Es ist mit einem eigenen Registersatz ausgestattet und kann Adressen unabhängig vom Programmcontroller berechnen und damit dem Datenrechenwerk Daten bzw. Operanden parallel zur aktuellen Befehlsausführung zur Verfügung stellen. Der Programmcontroller dient zur Ablaufsteuerung des Programms. In ihm sind Schleifenbefehle hardwaremäßig implementiert, um Verarbeitungszeit zu sparen. Wenn die Schleife erst einmal läuft, werden keine Prozessorzyklen mehr für die Schleifenabfrage verbraucht.

Digitale Signalprozessoren können verschiedene Zahlenformate verarbeiten - sie werden deshalb auch in Festkomma- und Gleitkomma-Prozessoren unterteilt. Festkomma-DSPs haben den Vorteil einfacherer Arithmetikeinheiten und kleinerer Wortlängen, womit ein geringerer Hardwareaufwand nötig ist. Daraus ergeben sich geringere Kosten und ein geringerer Energieverbrauch als bei Gleitkomma-DSPs. Hauptnachteil ist der relativ kleine nutzbare Zahlenbereich, wodurch beim Entwurf von Signalverarbeitungsalgorithmen häufig Über- und Unterläufe berücksichtigt werden müssen. Die daraus resultierende Rechenungenauigkeit führt zu parasitären Effekten wie Rundungsrauschen, die das Systemverhalten massiv beeinträchtigen können. Bei der Festkommazahlendarstellung wird die Zweierkomplementdarstellung verwendet. Dieses Zahlenformat wird auch Fraktaldarstellung genannt und hat Vorteile wie die Eindeutigkeit der Darstellung der Null, einfache Subtraktion und effiziente Algorithmen zur Multiplikation. Der Zahlenbereich von Festkommazahlen liegt in der Regel zwischen -1,0 und +1,0. Der Dezimalpunkt wird hierbei immer hinter das Vorzeichenbit gesetzt, welches bei einer negativen Zahl auf „eins“ gesetzt ist. Der Betrag der Festkommazahl wird in den verbleibenden Bits kodiert, bei der gebräuchlichen Datenbreite von 16 Bit also mit 15 Bit. Festkomma-DSPs können auch Integerzahlen (Ganzzahlen) verarbeiten.

Eine Gleitkommazahl besteht aus einer Mantisse, die in der Regel ein Bruch zwischen -1,0 und +1,0 ist, und einem Exponenten, der die Größe der Zahl in Zehner- oder wie bei DSPs in Zweierpotenzen angibt - also die Verschiebung des Kommas nach links oder rechts. Die Vorteile liegen hier im großen Dynamikbereich und der einfachen Implementierung von DSP-Algorithmen. Der Nachteil ist entsprechend, dass die Chip-internen Operationen komplexer sind und damit auch die Kosten steigen. Der ADSP-21161N ist ein Gleitkomma-DSP, kann aber auch mit Festkomma-Zahlen umgehen.

Digitale Signalprozessoren werden möglichst so entworfen, dass ein Maximum an anwendungsspezifischer Peripherie auf dem Chip untergebracht ist und die Komponenten somit direkt und ohne externe Busleitungen miteinander kommunizieren können. Je nach Ausführung befinden sich auf dem DSP-Chip auch Einheiten wie DMA-Controller, serielle Schnittstellen, programmierbare Ports für Ein- und Ausgabe-Schnittstellen und weitere Schnittstellen für Speichererweiterungen und verschiedene andere externe Geräte [28], [29], [30].

2.5.3 Anwendungen von DSPs

Die Einsatzgebiete für digitale Signalprozessoren sind mit der fortschreitenden Entwicklung immer vielfältiger geworden. Die folgende Tabelle 2 versucht einen Überblick zu geben:

Tabelle 2: Anwendungen von DSPs

Kommunikation	• Mobiltelefonie (GSM, UMTS etc.) • Satellitentelefonie • Schnelle Datenübertragung (Modems, ADSL, VDSL) • Global Positioning Systems (GPS) • Videokonferenzen • Digitale Anrufbeantworter • Faxgeräte
Multimedia	• Surround-Systeme • Digital-TV • Kompression von Sprache, Audio, Bildern und Video (MP3, WMA, MPEG2/4 etc.) • Video-Spezialeffekte
Industrieller Bereich	• Roboter • Bildverarbeitung • Motorsteuerungen
Messtechnik	• Signalanalyse • Funktions-/Signalgeneratoren
Medizintechnik	• Diagnostische Bildverarbeitung (CT, Ultraschall) • Elektrokardiogrammanalyse • Prothesen für Sinnesorgane und Gliedmaßen
Sonstige Anwendungen	• Aktive Lärmbekämpfungsalgorithmen in der Akustik • Erdbebenanalyse

2.6 Das ADSP-21161N EZ-KIT Lite® Evaluation Kit for SHARC® processors

2.6.1 Überblick

Das ADSP-21161N EZ-KIT Lite® Evaluation Kit besteht aus dem EZ-KIT Lite Board mit dem ADSP-21161N SHARC® Processor (Abbildung 12) und der VisualDSP++® Entwicklungsumgebung. Dieser High-Performance 32-Bit DSP wurde entwickelt, um verschiedene Anwendungen der digitalen Signalverarbeitung zu implementieren. Einige Beispiele dafür sind Bildverarbeitung, Kommunikationstechnik, militärische Anwendungen, Testverfahren, 3d-Grafik, Spracherkennung und Motorenkontrolle. Im Zusammenhang mit dieser Arbeit wird das System zur Audio-Verarbeitung, speziell zur Umsetzung von aktiven Lärmbekämpfungsalgorithmen verwendet.

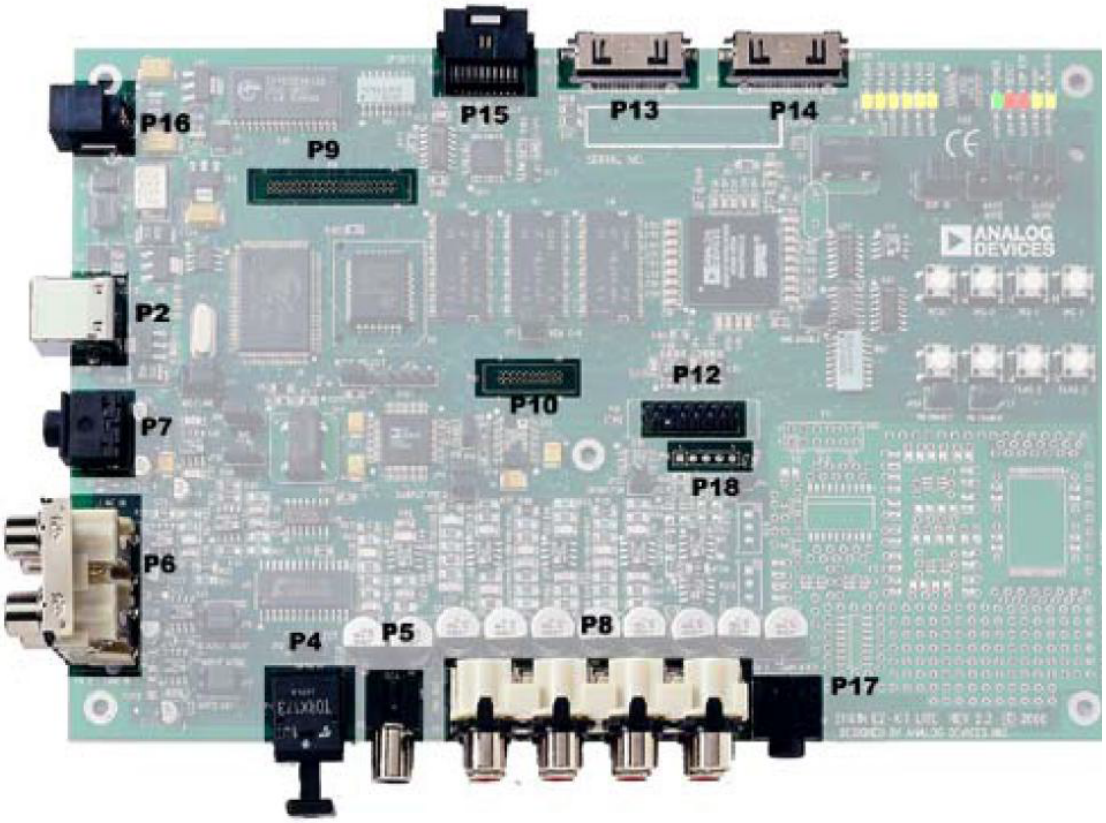


Abbildung 12: ADSP-21161N EZ-KIT Lite® Evaluation Kit Board

2.6.2 ADSP-21161N EZ-KIT Lite® – Die Hardware

Das ADSP-21161N EZ-KIT Lite® baut auf einem Kern der ADSP-21000 Familie auf und ist ein komplettes „System-On-a-Chip“ mit zweikanaligem SRAM, integrierten I/O-Schnittstellen und einem zusätzlichen Prozesselement zur gleichzeitigen Verar-

beitung von mehreren Datenelementen in einem Berechnungsschritt, genannt Single-Instruction-Multiple-Data (SIMD). Auf der Leiterplatte befinden sich außerdem AD/DA-Wandler, ein SPDIF Receiver, Flash-Speicher, SDRAM, diverse Anschlüsse, Jumper, Taster, Status-LEDs und Spannungswandler.

Die SHARC-Prozessoren basieren auf einer 32-Bit Super Harvard Architektur mit einem Hochleistungskern und leistungsfähigen Bussen zum Programmspeicher, Datenspeicher und zu den Input-/Outputschnittstellen. Der ADSP-21161N arbeitet mit einer Taktfrequenz von 100 MHz, die sich per Jumper jedoch auch verändern lässt. Der Prozessor verfügt über folgende Ausstattungsmerkmale (Abbildung 13):

- Zwei Prozesselemente (PEx und PEy), die jeweils ein 32-Bit Gleitkomma-Rechenwerk mit Multiplizierer, ALU, Schiebeeinheit und Datenregister besitzen
- Programmablaufsteuerung mit Befehlszwischenspeicher, Taktgeber und zwei Datenaddressierungsgeneratoren (DAG1 und DAG2)
- Zweikanaliges SRAM
- Einen externen Port, um externen Speicher (SDRAM), periphere Baugruppen wie AD/DA-Wandler oder Multiprozessorsysteme ansprechen zu können
- I/O-Prozessor mit integriertem DMA-Controller und weiteren Ports
- JTAG-Port für Emulations-/Testzwecke

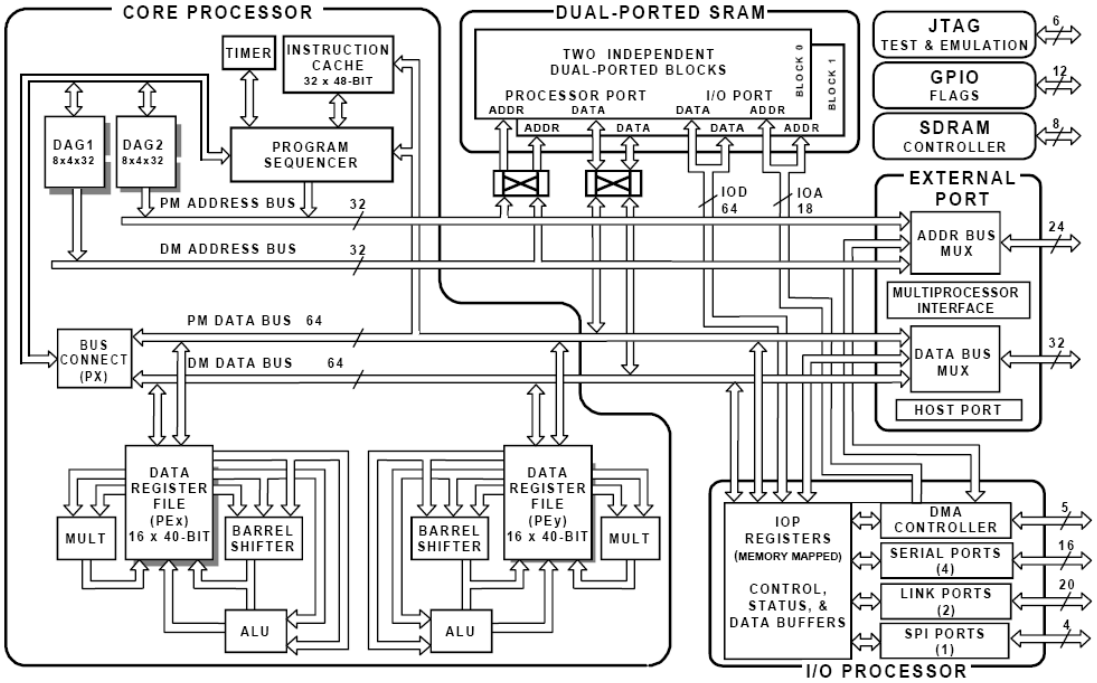


Abbildung 13: ADSP-21161 SHARC Blockschaltbild

Zur Umwandlung der Signale für die Ein- und Ausgabe kann der Kernprozessor mehrere Audioverarbeitungschips nutzen. Der Mächtigste ist der AD1836 96 kHz

Audio Codec mit drei Stereo-DA-Wandlern und zwei Stereo-AD-Wandlern, die auf dem Prinzip der von Analog Devices patentierten Multibit-Sigma-Delta-Wandlung beruhen [31]. Abbildung 14 zeigt das Blockschaltbild des AD1836 96 kHz Audio Codecs. Es ist ein SPI-Port enthalten, über den der Kernprozessor verschiedene Parameter wie z.B. Lautstärke oder Filtereinstellungen festlegen kann. Diese Einstellungen und Parameter werden in der Datei „AD1836_reg_init_viaSPORTs.asm“ definiert.

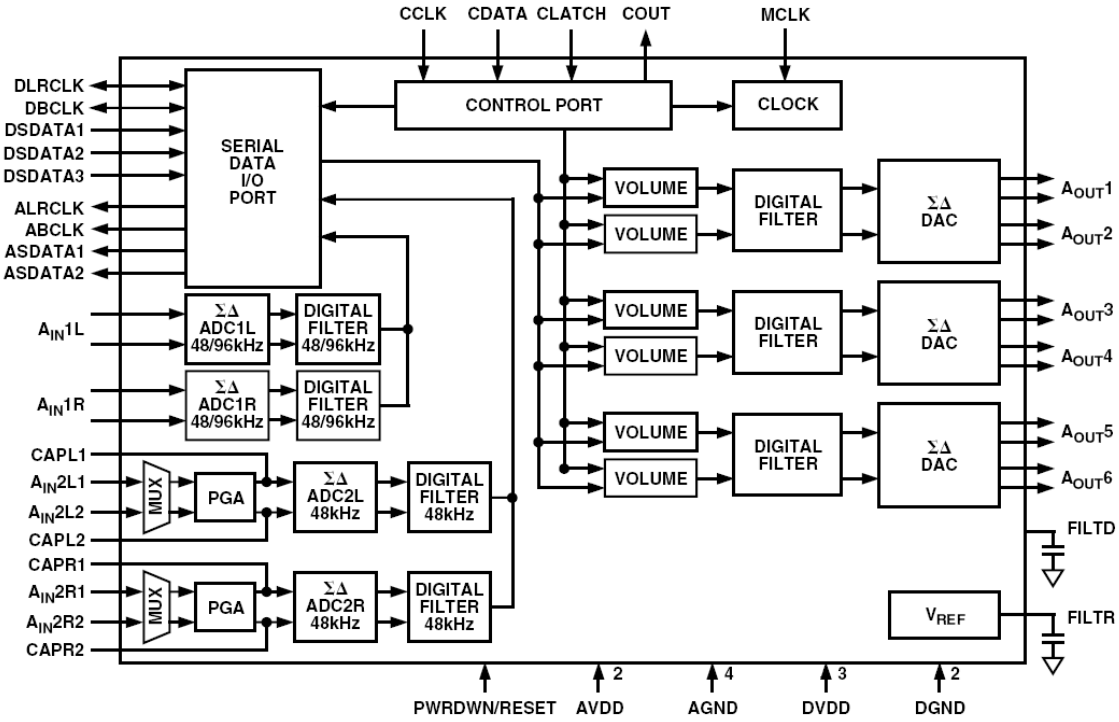


Abbildung 14: AD1836 Blockschaltbild

Auf dem DSP-Board ist noch ein weiterer Soundchip integriert. Der Analog Devices AD1852 192 kHz Auxiliary DAC ist ein hochwertiges digitales Audio Stereo Wiedergabesystem [32]. Die Abbildung 15 zeigt das Blockschaltbild. Es enthält einen Multibit-Sigma-Delta-Modulator, digitale Interpolationsfilter, analoge Wiedergabetechnik, einen Stereo-Lautstärkeregler und wie der AD1836 eine SPI-Schnittstelle, über den er per DSP gesteuert werden kann. Der AD1852 ist kompatibel zu allen DVD-Audioformaten mit Abtastraten bis 192 kHz und auch abwärtskompatibel zum „Redbook“-Standard für Compact Discs.

Die beiden Codec-Chips AD1836 und AD1852 steuern insgesamt acht Cinch-Anschlüsse und eine Stereo-Klinkenbuchse zur Ausgabe von Signalen und vier

Cinch-Anschlüsse für Eingangssignale an, von denen zwei mittels Jumper gegen eine Stereo-Klinkenbuchse zum Anschluss eines Mikrofons ausgetauscht werden können.

Der Crystal Semiconductor CS8414 96 kHz SPDIF Receiver ermöglicht den Anschluss von digitalen Audio-Quellen. Auf dem Board gibt es dafür jeweils einen optischen und koaxialen (Cinch) Eingang zur Eingabe von SPDIF-Signalen.

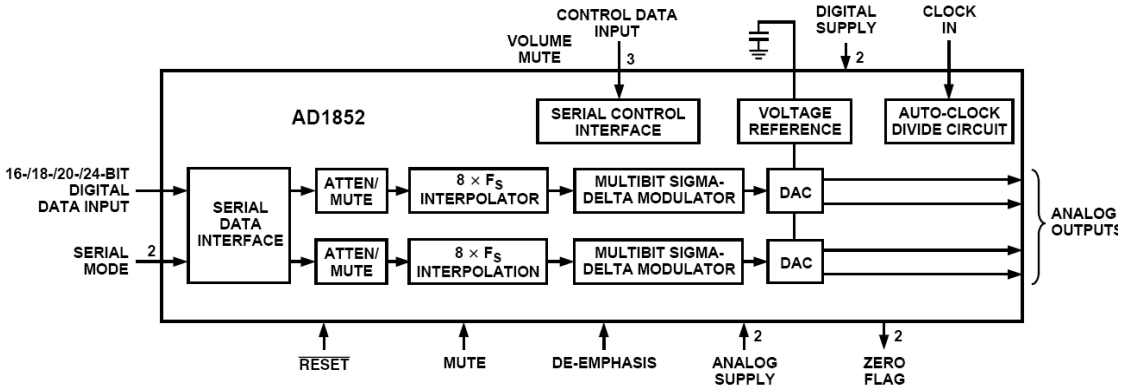


Abbildung 15: AD1852 Blockschaltbild

Auf der Leiterplatte befindet sich weiterhin 512K x 8-Bits Flash Speicher, der den Boot-Code enthält und über das im ADSP-21161N EZ-KIT Lite Evaluation System enthaltene Flash Programmer Utility mit optionalem Boot-Code programmiert werden kann.

Neben einem USB-Port zum Anschluss an einen PC für Programmier- und Debugzwecke gibt es auch einen „14-Pin Emulator Connector for JTAG“-Anschluss. Dieser JTAG-Anschluss ermöglicht eine schnellere Verbindung zwischen DSP-Board und Rechner sowie direkten Zugriff auf den internen und externen Speicher des DSPs sowie auf spezielle Funktionsregister. Weitere Anschlussmöglichkeiten sind SPORT-Anschlüsse und Link Ports. Zur rudimentären Eingabe befinden sich auf dem Board ein Reset-Schalter, vier frei belegbare Tasten und 3 weitere Interrupt-Tasten. Außerdem gibt es insgesamt elf Status-LEDs. Diverse Jumper ermöglichen verschiedene Einstellungen wie Änderung der Abtastrate oder Konfiguration der Audio-Ein-und Ausgänge [30], [33].

2.6.3 ADSP-21161N EZ-KIT Lite® – Die Software

Das ADSP-21161N EZ-KIT Lite® enthält die (easy-to-use) VisualDSP++® Entwicklungsumgebung (Integrated Development Environment (IDE)) mit Debugger. Aufgrund der integrierten Projektentwicklung und Debug-Umgebung können komplette DSP-Projekte realisiert werden, bei denen einfach zwischen Quellcode-Bearbeitung, Assembler, Compiler, Linker, Debugger usw. umgeschaltet werden kann [34].

Zur Entwicklungsumgebung (IDE) gehört ein mächtiger Editor mit Syntax-Hervorhebung für verschiedene Programmiersprachen. Weitere Features des Editors sind OLE drag & drop (OLE=Object Linking and Embedding, dt. Objekt-Verknüpfung und –Einbettung), Bookmarks und Standardfunktionen wie rückgängig machen, kopieren usw.

Weitere Bestandteile sind ein Debugger und die SHARC Software Development Tools mit C/C++-Compiler, Assembler, Linker, Loader, Simulator und Splitter. Der Simulator ermöglicht das Arbeiten am Quellcode und Tests ohne angeschlossenes DSP-Board.

In einem DSP-Projekt kommen verschiedene Dateitypen zum Einsatz, von denen die wichtigsten im Folgenden kurz erläutert werden. Projektdateien heißen „*.dpj“ und enthalten die globalen Informationen eines Projekts wie Projekteigenschaften, Einstellungen und eine Liste der zum Projekt gehörenden Dateien. Programmdateien mit Assembler-Code besitzen die Dateiendung „*.asm“, in C programmierte Programmteile die Endung „*.c“. Header- oder Include-Files, in denen Variablen, Parameter und Werte definiert werden können, heißen „*.h“. Das wichtigste Header-File ist die Register- und Bit-Definitionsdatei „def21161.h“, in welcher Makros definiert sind, die es dem Programmierer ermöglichen, vorgefertigte Variablen und Symbole zu verwenden. Diese Datei ist Bestandteil der Programmierumgebung und muss in jedes Programmteil, d.h. in jede Programmdatei per ‚#include‘-Befehl eingebunden werden. In der Datei „21161_EZKIT_Lite.ldf“ stehen die Parameter zum Memory-Mapping. Diese Datei kann im Editor, oder komfortabler, im Expert-Linker bearbeitet werden. Näheres dazu ist im Abschnitt 3.3 beschrieben, der sich mit dem Memory-Mapping befasst. Datendateien können beliebige Endungen haben, da sie keiner Konvention unterliegen und nur im Variablendefinitionsbereich eingelesen werden. Der Übersichtlichkeit halber habe ich diese Datendateien „*.dat“ genannt, da diese Endung auch für zu exportierende Daten vorgesehen ist. In dem vorliegenden Active-Noise-Control-Projekt enthält zum Beispiel die Datei „taps.dat“ die Koeffizienten für die Generierung der Maximalfolgen verschiedener Länge.

2.6.4 Installation und Einrichtung

Im Rahmen der vorliegenden Arbeit wurde mit der Version VisualDSP++® 4.0 programmiert. Ein Upgrade auf Version 4.5 ist nicht zu empfehlen, da die Programme ohne Änderungen nicht auf dieser Plattform laufen. Projekte, die mit der Version 3.5 erstellt wurden, können problemlos in ein mit Version 4.0 benutzbares Format umgewandelt werden. Das wird schon beim Öffnen in Version 4.0 angeboten und zuverlässig erledigt. Allerdings ist diese Umwandlung nicht ohne Weiteres rückgängig zu machen – also unbedingt (und nicht nur hier) Backups anlegen.

Die Installation gestaltet sich Windows-typisch. Nach Anschluss des Boards an einen USB-Anschluss wird dieses erkannt und der Treiber muss installiert werden. Das Board sollte immer am gleichen USB-Port betrieben werden, da ansonsten der Treiber neu installiert werden muss. Der zusätzlich verfügbare JTAG-Anschluss wurde in diesem Projekt nicht verwendet.

2.6.5 Die Visual-DSP Oberfläche und Bedienung

Beim ersten Start der Programmierumgebung muss eine neue Session eingerichtet werden. Es gibt die Möglichkeit einer Simulationssession, bei der kein Board angeschlossen sein muss. Hier ist es z.B. möglich, Routinen zu programmieren und zu testen. Allerdings sind Funktionen ausgeschlossen, die nur das Board erledigen kann. Dazu gehören vor allem mit der Ein- und Ausgabe von Daten verbundene Operationen. Es sollte also bei angeschlossenem Board gleich eine EZ-KIT Lite Session eingerichtet werden. Hierbei ist darauf zu achten, das passende Debug-Target und die richtige Plattform auszuwählen. Die Abbildung 16 zeigt die für das in diesem Projekt verwendete DSP-Board zu wählenden Einstellungen. Im Configurator kann getestet werden, welche Plattformen zur Verfügung stehen.

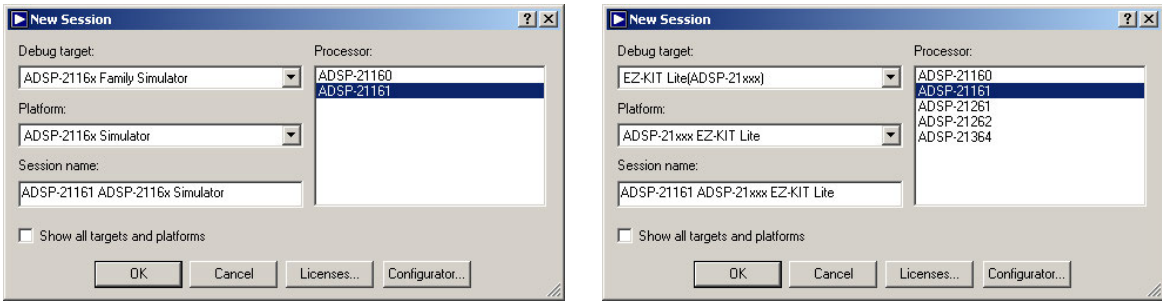


Abbildung 16: Screenshots „New Session“

Erste Schritte kann man mit Hilfe zweier Tutorials machen, die sich in der Hilfe unter „Getting Started“ finden. Auch für später auftauchende Probleme ist die Hilfe gut zu gebrauchen.

Um sich mit der Bedienungsoberfläche und vor allem dem Aufbau von Projekten vertraut zu machen, ist zu empfehlen, sich für den Anfang ein vorhandenes Beispielprojekt auszuwählen und damit erste Schritte zu unternehmen. Hierfür bietet es sich an, gleich ein Beispielprojekt auszuwählen, dass man als Grundgerüst für das spätere eigene Projekt verwenden kann. In dieser Arbeit wurde „Talkthru“ benutzt. Hierbei handelt es sich um ein Programmgerüst zum Einbau eigener Audio-Verarbeitungsroutinen, in dem die nötigen Initialisierungsroutinen für die AD-/DA-Wandler, für das RAM usw. schon vorhanden sind. Gerade für Neueinsteiger in die DSP-Programmierung von Audio-Verarbeitungsalgorithmen ist dieses Beispielprojekt gut geeignet. Beispielroutinen und –projekte finden sich nach der Installation von VisualDSP++® 4.0 in den Verzeichnissen

„\Analog Devices\VisualDSP 4.0\211xx\Examples“

„\Analog Devices\VisualDSP 4.0\211xx\EZ-KITs\ADSP-21161N\Examples\“.

Wichtig ist es, die Beispiele für die richtige Plattform zu benutzen (211xx-Serie), da ansonsten Inkompatibilitäten auftreten können. Es gibt Assembler-Beispiele (ASM), C-Beispiele und auch gemischte Beispielprojekte. Das hier verwendete „Talkthru“ ist ein Assembler-Projekt und findet sich im Verzeichnis

„\Analog Devices\VisualDSP 4.0\211xx\EZ-KITs\ADSP-21161N\Examples\ASM\Talkthru\“.

Es empfiehlt sich, dieses Verzeichnis vor dem Testen und Ändern in ein Verzeichnis für eigene Projekte zu kopieren, um die ursprüngliche Version nicht zu verändern.

Abbildung 17 zeigt die Visual-DSP-Oberfläche mit geöffnetem „Talkthru“-Projekt. Die Funktionen Neues Projekt, Öffnen, Schließen, Speichern usw. finden sich Windows-typisch im Menü „File“. Natürlich gibt es für die wichtigsten Funktionen auch Buttons; Tooltipps helfen beim Finden der Richtigen. Neu geöffnete Fenster wie z.B. Register-Fenster oder Plot-Fenster werden immer in der Oberfläche angedockt angezeigt. Um sie vergrößern zu können, müssen Sie durch Rechtsklick und Häkchen bei Allow Docking erst einmal abgedockt werden. Dann kann das Fenster verschoben und vergrößert werden.

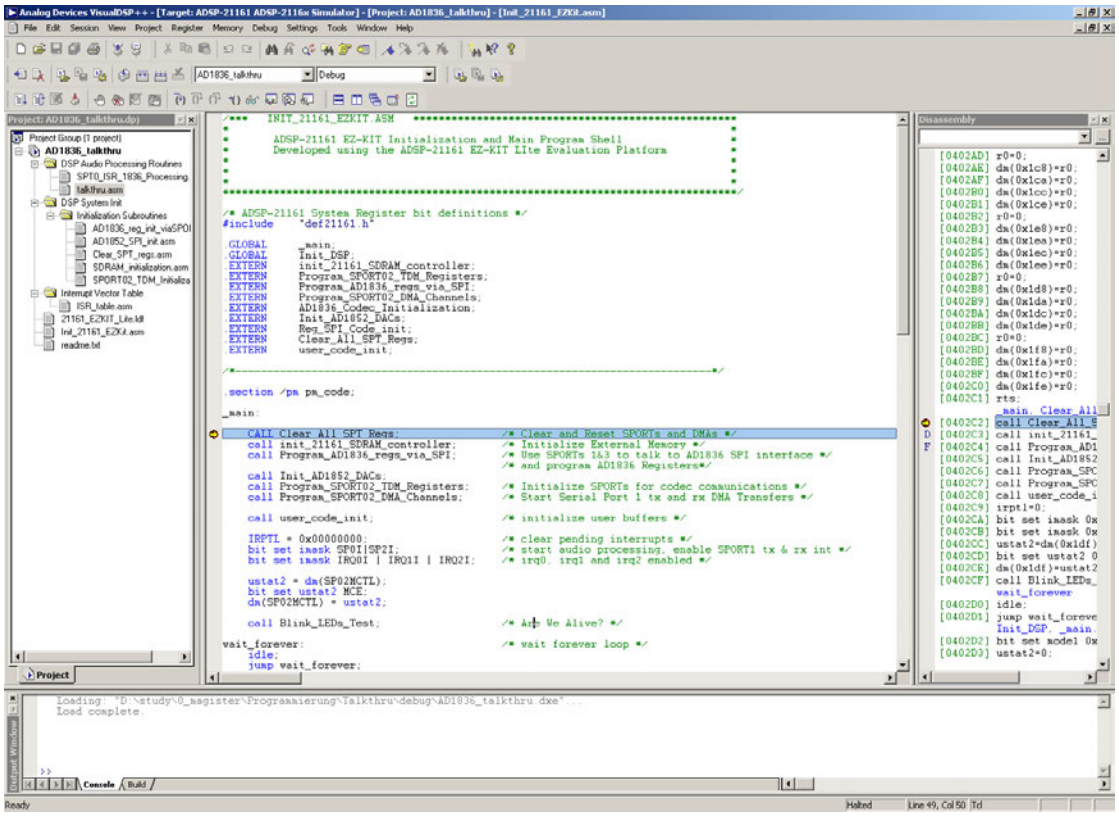


Abbildung 17: Screenshot der Visual-DSP-Oberfläche

Ist nun ein Projekt durch Öffnen der entsprechenden dpj-Datei (beim Beispiel Talkthru „AD1836_Talkthru.dpj“) geladen, kann man entweder gleich die Quellcode-Dateien durch Doppelklick im Projekt-Browser öffnen und editieren oder erst einmal das gesamte Projekt bauen (Build Project (F7)). Unter Build wird verstanden, dass die Quelltexte übersetzt, d.h. compiliert werden wobei auch Header-Files in den Programmcode eingebunden werden und die definierten Variablen und Puffer dem physikalischen Speicher zugeordnet werden (Link). Nach dem Build wird automatisch die Quelldatei mit der Hauptroutine (main) im Editorfenster geöffnet und das Ergebnis, also im besten Fall „Build completed successfully“ in der Build-Ansicht und „Load complete“ in der Consolen-Ansicht des Outputfensters (unten) ausgegeben. Der Program Counter (PC), also die Adresse bei der sich die aktuelle Programmausführung gerade befindet, steht nun am Beginn des auszuführenden Projekts. Die aktuelle Position des Program Counters ist durch einen gelben Pfeil gekennzeichnet. Zu diesem Zeitpunkt befindet sich an dieser Stelle auch ein rot-brauner Punkt. Dabei handelt es sich um einen Breakpoint (Haltepunkt oder Unterbrechungspunkt). Dieser wird automatisch nach dem Builden gesetzt. Haltepunkte können aber

natürlich auch manuell durch Doppelklicken oder Rechtsklick (Menu) an beliebige Stellen des Programms gesetzt werden, z.B. zu Debug-Zwecken.

Jetzt kann das Projekt durch Run (F5) gestartet werden. Oft ist es vorteilhaft, die Programmausführung nur bis zu einer bestimmten Stelle des Programms zuzulassen, um etwa nur einen bestimmten Programmteil zu testen. Hierfür gibt es neben dem schon erwähnten Haltepunkt noch die Möglichkeiten Step Into (F11) und Run to Cursor (STRG+F10). Step Into bewirkt einfach, dass jeweils nur eine Anweisung ausgeführt und der PC dann wieder angehalten wird. Es ist dabei gut nachzuvollziehen, ob jede Anweisung auch das bewirkt, was sie soll. Bei auftretenden Interrupts kann das aber zu verwirrenden Ergebnissen führen, da unter Umständen in eine Interrupt Service Routine gesprungen wird und man dadurch in einem anderen Programmteil landet. Nach meiner Erfahrung lässt sich dieser Effekt mit dem Befehl Run to Cursor vermeiden. Dazu kann man den Cursor an eine beliebige Position im Programmcode der gewünschten Quelldatei setzen und die Programmausführung wird dort gestoppt. Das funktioniert im Gegensatz zu Step Into auch bei eventuell auftretenden Interrupts zuverlässig. Nachdem das Programm gestartet wurde, kann es durch den Befehl Halt (UMSCHALT+F5) gestoppt werden. Erneutes Run veranlasst den Prozessor zu Fortsetzung des Programms an der zuvor gestoppten Stelle.

Nach Änderungen im Quelltext oder auch nur zum Neustart des Programms am Anfang ist ein erneutes Build nötig. Dazu ist der Befehl Build Project ausreichend, da hierbei geprüft wird, ob und welche Quellcode-Datei geändert wurde und nur diese erneut compiliert wird. Die langwierigere Variante ist der Befehl Rebuild All, der die komplette Neucompilierung aller Quelltext-Dateien und auch ein erneutes Linking (Memory Mapping) zur Folge hat.

Das Memory Mapping bezeichnet die Zuordnung der im Deklarationsteil der Quelldateien definierten Variablen, Puffer und Routinen oder Unterprogramme und wird bei Linken erledigt. Die Zuordnung der in den Deklarationsabschnitten angelegten Eingabe-Sektionen zu physikalischen Speicherbereichen kann am einfachsten mit Hilfe des Expert Linkers erfolgen. Dazu ist nur ein Doppelklick auf die ldf-Datei (im Beispiel „21161_EZKIT_Lite.ldf“) im Projektbrowser-Fenster nötig. Es stehen Programmspeicher, Datenspeicher und SD-RAM zur Verfügung. Eine genauere Erläuterung findet sich in später im Abschnitt „Das Memory Mapping“. Für eine übersichtlichere Darstellung kann man das Expert Linker-Fenster zuerst Abdocken und dann Vergrößern. Außerdem sollte man im rechten Bereich (Memory Map) durch Rechstklick den Ansichtsmodus auf Memory Map Tree umschalten. Die Eingabe-Sektionen können nun per Drag and Drop nach rechts auf das gewünschte Speichersegment gezogen werden. Da es sich bei der Mapping-Datei um eine normale Text-

datei handelt, kann man alternativ zur Benutzung des Expert Linkers die ldf-Datei auch in einem externen Editor öffnen und bearbeiten.

Um herauszufinden, ob Anweisungen auch das richtige Ergebnis liefern, ist das Auslesen der entsprechenden Register sehr hilfreich. Durch bewegen und halten des Cursor über ein Register im Quellcode wird bereits dieser Registerwert im Hex-Format angezeigt. Im Menü Register kann man sich zusätzlich ganze Registergruppen anzeigen lassen. Hier ist Abdocken wieder ratsam. Es ist auch möglich, sich ein eigenes Fenster mit ausgewählten Registern selbst zusammenzustellen.

Visual-DSP bietet weiterhin zur graphischen Auswertung von Vektoren die Möglichkeit, diese als Plot darzustellen. Dazu muss man über das Menü „View“ und dann „Debug Windows“ zu „Plot“ navigieren. Wichtig ist, den richtigen Speicherbereich (PM, DM) einzustellen. Dann kann man nach Klicken auf den Browse-Button den gewünschten Vektor auswählen. Es ist auch möglich, mehrere Vektoren in einem Plot-Fenster darzustellen. Hierbei sollte man aber im Hinterkopf behalten, dass die Dimensionen der Vektoren um einige Zehnerpotenzen auseinander liegen können und somit der Vektor mit den viel kleineren Werten nicht zu erkennen ist. Allerdings besteht die Möglichkeit, ohne große Umstände einen Vektor abzuschalten. Nachdem auch die Anzeigeeinstellungen zur Zufriedenheit erledigt sind, empfiehlt es sich, die Einstellungen des Plots in eine Datei zu speichern. So kann dieser jederzeit wieder mit den gleichen Einstellungen aufgerufen werden. Eine Exportfunktion erlaubt es, Plots in die Zwischenablage zu kopieren, sie als Bilddatei abzuspeichern, zu Drucken und die Vektoren über die Soundkarte auszugeben.

Ein weiteres wichtiges Feature besteht in darin, Vektoren/Puffer zur Verwendung außerhalb der Entwicklungsumgebung zu exportieren. Diese Funktion verbirgt sich im Menü „Memory“ unter „Dump…“. Nach Auswahl von „Dump to file“ und der Auswahl der geeigneten Parameter werden die Werte aus dem gewünschten Vektor in eine Textdatei geschrieben.

2.6.6 DSP-Programmierung

Die Entwicklungsphasen bei der konkreten Realisierung von DSP-Projekten von der Aufgabenstellung zur fertigen Applikation werden unter dem Begriff Top-Down Designschema zusammengefasst (Abbildung 18). In realen Projekten wird dieses Schema meist nicht nur einmal von oben nach unten, sondern in Schleifen durchlaufen, da z.B. auf Ergebnisse von Simulationen oder Programmierprobleme reagiert werden muss. Anhand dieses Schemas wird im Folgenden das Programmierprojekt dieser Arbeit beschrieben [29].

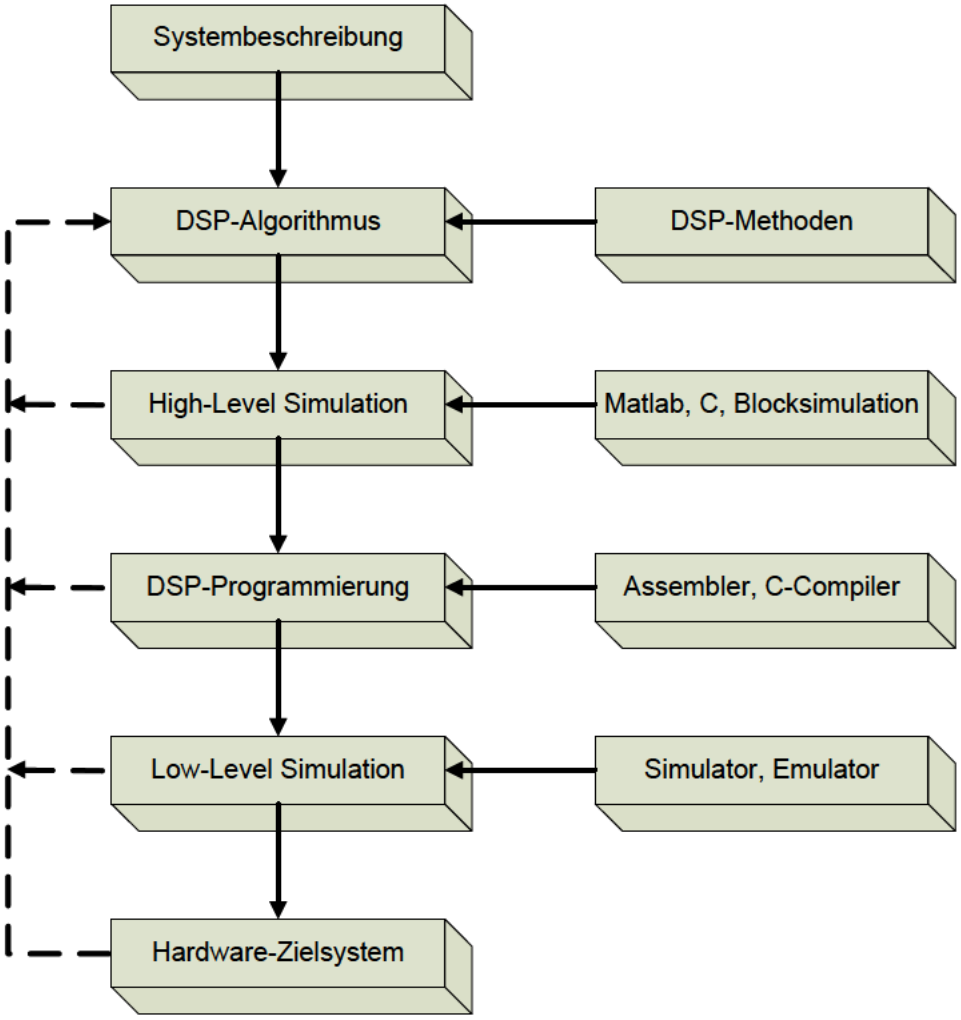


Abbildung 18: Top-Down Design von DSP-Anwendungen

An der Spitze dieses Entwicklungsweges steht eine Systembeschreibung mit den genauen Spezifikationen. Im Falle dieser Arbeit sollen Algorithmen zur Systemidentifikation und zur aktiven Lärmbekämpfung programmiert werden. Die Algorithmen sollen für bis zu drei Kanäle implementiert werden. Grundsätzlich soll es möglich sein, mit verschiedenen Abtastraten zu arbeiten.

Im zweiten Schritt des Top-Down Designs werden passende Algorithmen ausgewählt oder auch neu entwickelt. Zur Systemidentifikation ist es nötig, eine Maximalfolge zu generieren, deren Länge variabel sein soll. Während der eigentlichen Systemidentifikation, d.h. der Ermittlung der Impulsantwort des Systems (hier ein Anti-Schall-Fenster) wird die generierte MLS-Folge über die Lautsprecher im Anti-Schall-Fenster ausgegeben und die Systemantwort aufgenommen. Es soll hierbei möglich sein, die Folge mehrmals auszugeben, um dann durch Mittelung Störfaktoren auszu-

schließen und die Genauigkeit zu erhöhen. In den mehrkanaligen Versionen soll diese Maximalfolge zeitversetzt ausgegeben werden können. Anschließend werden mittels Kreuzkorrelation von MLS-Folge und Systemantworten die Impulsantworten für die verschiedenen Übertragungswege berechnet und zur späteren Verwendung in den eigentlichen ANC-Algorithmen in einen Vektor geschrieben. Zur aktiven Lärmbekämpfung sollen filtered-x-LMS-Algorithmen in den Versionen Feedforward und Feedback programmiert werden.

Es folgen in der Regel High-Level Simulationen mit komfortabler Software wie MATLAB, um die Funktionsweise der zu verwendenden Algorithmen zu überprüfen und gegebenenfalls Referenzergebnisse für spätere Low-Level Simulationen und abschließende Funktionstests zu gewinnen. Im Zuge dieser Arbeit wurden einige Vorabtests mit MATLAB durchgeführt. So wurde etwa der Algorithmus zur Generierung von Maximalfolgen ausführlich durchgespielt – ein von mir auch zu Verständniszwecken angepasstes MATLAB-Programm findet sich auf der beigelegten CD-ROM. In dieser Phase sollte auch entschieden werden, welches Hardware-Zielsystem den Anforderungen wie Rechengenauigkeit oder -geschwindigkeit, Kosten oder einfach nur Verfügbarkeit entspricht. In diesem Fall handelt es sich um das oben bereits beschriebene ADSP-21161N EZ-KIT Lite®.

Vor der ersten Low-Level Simulation muss natürlich der Algorithmus (Teile davon oder auch mehrere) programmiert werden. Möglich ist die Realisierung der Algorithmen in einer höheren Programmiersprache wie C/C++ oder wie in diesem Fall in Assembler. Gerade bei Echtzeit-Anwendungen, um die es sich auch bei den ANC-Algorithmen handelt, ist Assembler die erste Wahl. Allerdings geht die Entwicklung in den letzten Jahren in die Richtung optimierter C-Compiler, die auch zeitkritische Anwendungen möglich machen sollen. Die genaue Programmbeschreibung würde diesen Abschnitt sprengen und folgt darum im nächsten Kapitel.

Mit dem verwendeten Evaluationssystem sind auch Low-Level Simulationen möglich, d.h. die Hardware (das DSP-Board) muss nicht unbedingt angeschlossen sein, da es von der Programmierumgebung im PC simuliert wird. Allerdings gibt es natürlich Einschränkungen für Tests, bei denen Ein- bzw. Ausgabeoperationen beteiligt sind. Die bessere Variante ist, das Projekt auf einem Emulator wie dem hier verwendeten DSP-Board zu testen, da es möglich ist, das Programm anzuhalten und interne Prozessorzustände und Speicherinhalte zu überprüfen oder auch zu verändern.

Der letzte Schritt wäre dann, das fertig programmierte Projekt auf das Hardware-Zielsystem, einen Chip, MP3-Player oder ähnliches zu übertragen und finale Tests durchzuführen.

3 Programmierung/Programmbeschreibung

3.1 Einleitung

Als Grundlage für die Programmierung wurde das im EZ-Kit-Lite enthaltene Beispielprojekt „Talkthru“ verwendet. Es handelt sich hierbei um ein Programm-Gerüst zum Einbau eigener Audio-Verarbeitungsroutinen, in dem die nötigen Initialisierungsroutinen für die AD-/DA-Wandler, für das RAM usw. schon vorhanden sind. Dieses Projekt ist ein reines Assembler-Programm. Es nimmt Werte aus einem Puffer und kopiert diese in einen Ausgabepuffer. Während der Laufzeit des Programms wird das Audio-Signal einfach inklusive D/A-A/D-Wandlung durchgeschleift. Hinweise zur Benutzung und auch zum Anschluss von Mikrofon und Lautsprechern gibt es in der zum Projekt gehörenden „readme.txt“.

3.2 Erläuterung der zum Projekt gehörenden Dateien

Die zum Projekt „noise_control“ gehörenden Dateien können übersichtlich in Ordnern sortiert werden (Abbildung 19). Im Falle von „Talkthru“ sind es die Ordner „DSP Audio Processing Routines“, „DSP System Init“ mit dem Unterordner „Initialization Subroutines“ und „Interrupt Vector Table“. Außerdem die Dateien „21161_EZKIT_Lite.ldf“ (Linker Description File), „mls.h“ (Header-File mit Parametern) und „Init_21161_EZKit.asm“ (Hauptprogramm). Im Folgenden werden die einzelnen Programmteile in der Reihenfolge des Programmablaufs erläutert.

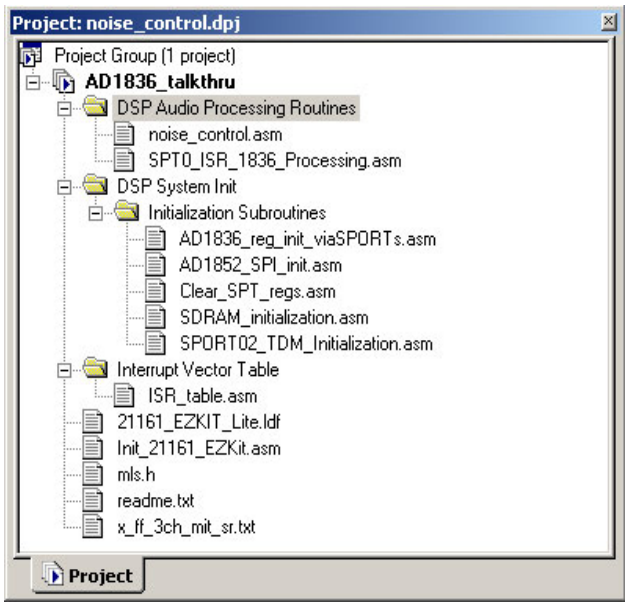


Abbildung 19: Dateien des Projekts

Im Hauptprogramm „Init_21161_EZKit.asm“ werden Initialisierungsroutinen aufgerufen, Interrupts zurückgesetzt und natürlich wird in die programmierten Routinen und Algorithmen, beginnend mit der Generierung der MLS-Sequenz, gesprungen.

Zuerst wird mit ‚CALL Clear_All_SPT_Regs;‘ die Routine ‚Clear_All_SPT_Regs‘ aufgerufen. Diese Routine befindet sich in der Datei bzw. im Programmteil „Clear_All_SPT_Regs.asm“. Hier werden Kontroll- und Konfigurationsregister (serielle Ports, SPORTs) für den AD1836 Audio Codec und die DMA-Register gelöscht bzw. auf Standardwerte gesetzt. In diesem Programmteil sind keine Veränderungen oder Anpassungen nötig.

Die Einrichtung des externen SD-RAMs findet in dem Unterprogramm ‚init_21161_SDRAM_controller‘ in der Datei „SDRAM_initialization.asm“ statt. Um dieses zusätzliche SD-RAM nutzen zu können, muss der Speicherbereich in der Memory-Map im Linker-File definiert werden.

Die Routine ‚Program_AD1836_regs_via_SPI” befindet sich in der Datei „AD1836_reg_init_viaSPORTs.asm”. Hier werden die Register des SPI Ports initialisiert. Der SPI Port ist mit dem AD1836 Audio Codec und dem zusätzlichen DA-Wandler AD1852 verbunden und ist für das Schreiben und Lesen der Kontroll-Register (SPORTs) dieser Audiogeräte zuständig. In diesen Kontroll- und Konfigurationsregistern können die Voreinstellungen für Lautstärke, Abtastrate und Filterparameter geändert werden. Außerdem befinden sich hier die Interrupt Service Routinen für die SPORTs. Diese Routinen wurden geändert, um die Abtastraten-Reduzierung zu bewerkstelligen. Diese Anpassung der Abtstrate wird an späterer Stelle genauer erläutert (Abschnitt 3.4).

Mit ‚call Init_AD1852_DACs;‘ wird in die Datei „AD1852_SPI_init.asm” gesprungen und der zusätzliche AD1852 DA-Wandler initialisiert. Dieser Chip wird in diesem Projekt nicht verwendet.

Die Routine ‚Program_SPORT02_TDM_Registers‘ befindet sich in der Datei „SPORT02_TDM_Initialization.asm”. Sie ist der Initialisierungstreiber für den AD1836 Audio Codec. Hier werden die Einstellungen für den In- und Output der SPORTs programmiert. In derselben Datei findet sich auch die Routine ‚Program_SPORT02_DMA_Channels‘, die für die Einstellungen des DMA-Controllers der SPORTs zuständig ist.

‚call user_code_init;‘ springt in den Teil des Projekts, in dem die Puffer und anderen Vektoren sowie Variablen definiert werden, die später für die Ausführung der eigentlichen MLS-Generator- und Audioverarbeitungsalgorithmen verwendet werden. Ab hier wird es also für das Verständnis des Projekts zum ersten Mal interessant. Dieser

Teil befindet sich in der Datei „noise_control.asm” und wird wie alle in dieser Arbeit programmierten Programmteile im Folgenden natürlich noch genauer beschrieben.

‚mls_gen‘ generiert eine Maximalfolge mit den in dem Header-File „mls.h” vorgegebenen Parametern. Die folgenden Routinen in „noise_control.asm” besorgen die Ausgabe der MLS-Samples und das Einlesen der Systemantwort (‚samples_out_in‘, ‚read_response_sample‘).

Zur Eingabe und Ausgabe der Samples wird in die Routine ‚Process_AD1836_Audio_Samples‘ in der Datei „SPT0_ISR_1836_Processing.asm”, gesprungen. Außerdem wird hier auch zwischen den Programmteilen Input/Output der MLS (‚input_output‘) und dem eigentlichen Lärmbekämpfungsalgorithmus (‚noise_control‘) umgeschaltet.

In der Interrupt-Vektor-Tabelle („ISR_table.asm“) wird definiert, was bei auftretenden Interrupts passieren soll, z.B. wohin gesprungen wird. Für die Abastraten-Anpassung wurden im Teil ‚SP0I_svc‘, der für die Interrupt-Behandlung der DMA-Transfers des SPORT0 zuständig ist, einige Veränderungen vorgenommen. Im Prinzip werden hier die Interrupts gezählt und es wird entsprechend der gewünschten Abtastrate entweder zur Ein- und Ausgabe der Samples oder zurück zur Berechnung des LMS-Algorithmus‘ gesprungen. Hier wird auch ersichtlich, dass die Hauptroutine des Projekts die für die Ein- und Ausgabe der Sound-Werte zuständige Routine ‚Process_AD1836_Audio_Samples‘ in der Datei „SPT0_ISR_1836_Processing.asm” ist, d.h. es wird nach auftretendem Interrupt oder dem ‚rts„-Befehl immer in diesen Programmteil zurückgesprungen.

3.3 Das Memory-Mapping

Zur Vorhaltung von Variablen, Puffern usw. verfügt das EZ-Kit-Lite über internen (SRAM) und externen Speicher (SDRAM) [33]. Der interne Speicher teilt sich Programmspeicher (PM) und Datenspeicher (DM). Damit eine Berechnung, z.B. eine MAC-Operation, in einem Taktzyklus verarbeitet werden kann, müssen die Operanden in jeweils unterschiedlichen Speicherbereichen abgelegt sein. Darum ist es nötig, schon im Variablendeklarationsteil der Programme auf eine sinnvolle Verteilung der verwendeten Vektoren auf PM und DM zu achten. Reicht das interne RAM nicht aus, muss auf den Erweiterungsspeicher (SDRAM) ausgewichen werden. Dort sollten aber möglichst nur Variablen/Vektoren abgelegt werden, die bei nicht zeitkritischen Berechnungen verwendet werden. So war es in dieser Arbeit nötig, die MLS-Folge und die aufgenommene Antwort des Systems im SDRAM zu speichern. Da die Systemantwort für die Berechnung der Impulsantwort gebraucht wird, verzögert sich

diese Berechnung erheblich. Das ist aber zu verschmerzen, da an diese Berechnung keine Echzeitanforderungen gestellt werden, weil sie vor dem eigentlichen Lärmbekämpfungsalgorithmus ablaufen. Um diesen Sachverhalt zu verdeutlichen, wurde eine Testschleife programmiert um festzustellen, wie viele Operationen in einem Abtastzyklus berechnet werden können – bei unterschiedlicher Anordnung der Operanden im Speicher. Theoretisch wären bei einer Abtastrate von 44,1 kHz und 100 MHz Takt des DSPs gut 2000 Schleifendurchläufe möglich. Einige Millisekunden gehen dabei noch durch DSP/Programm-bedingte Delays verloren, so dass bei einer MAC-Berechnung, bei der ein Operand (Px) im Datenspeicher und der andere (Py) im Programmspeicher gespeichert waren, die Schleife 1930-mal durchlaufen werden konnte. Die folgende Tabelle 3 zeigt das komplette Ergebnis.

Tabelle 3: Operationen pro Abtastzyklus

PM	DM	SDRAM	Schleifendurchläufe	Zyklen/MAC
Px	Py		1930	1
Px und Py			960	2
Px		Py	860	<2
		Px und Py	90	21

Es zeigt sich, dass vor allem auf Berechnungen mit Operanden, die beide im externen Speicher abgelegt sind, verzichtet werden sollte.

Praktisch wird das Speicher-Mapping in VisualDSP++® 4.0 im Expert Linker erledigt, der durch Doppelklick auf die Datei „21161_EZKIT_Lite.ldf“ aufgerufen wird und die komfortable Bearbeitung des Linker Description Files erlaubt. Jede im Variablendeklarationsteil definierte Eingabe-Sektion muss dort per Drag and Drop einem Segment zugeordnet werden (Abbildung 20). Die Sektion „mem_dmres“ wurde aus den ursprünglichen Sektionen „seg_stak“ und „seg_heap“ zusammengesetzt, um dort die Speicherung langer Vektoren zu ermöglichen. Achtung, bei Einsatz von C++ müssen diese Sektionen aber zwingend vorhanden sein! Außerdem wurde die Sektion „mem_pmda“ vergrößert, da unverständlicherweise nicht der volle zur Verfügung stehende Bereich ausgenutzt wurde. Um die Auswahl der Sektionen zur Speicherung von Daten zu erleichtern, geht aus der Abbildung 21 zusätzlich hervor, wie viele Datenworte, also Vektor-/Pufferelemente in jeder Sektion maximal abgelegt werden können.

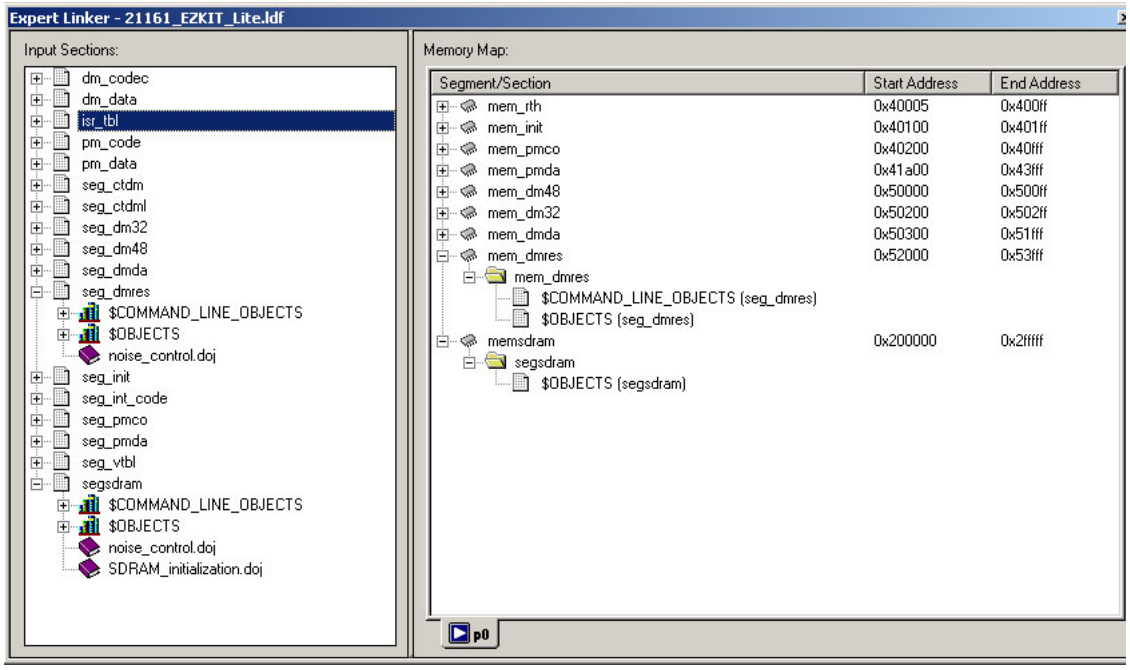


Abbildung 20: Memory-Mapping - Speichersektionen

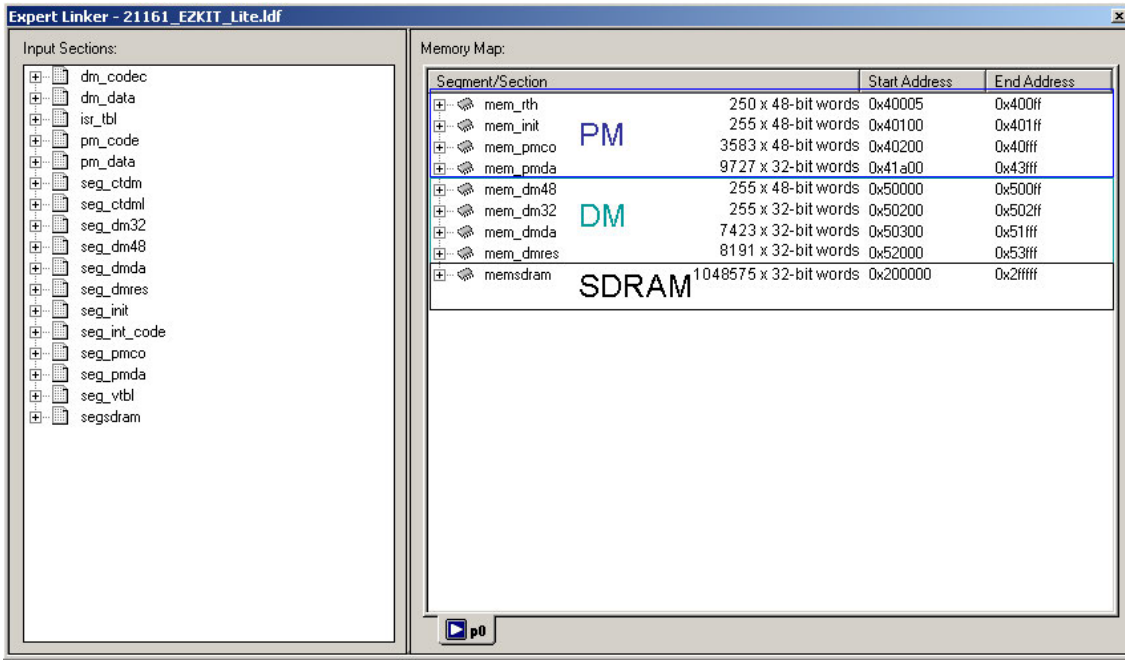


Abbildung 21: Memory- Mapping – speicherbare Datenworte

3.4 Anpassung der Abtastrate

Der AD1836 96 kHz Audio Codec mit seinen A/D-D/A-Wandlern lässt sich mit den Abtastraten von 96 kHz und 48 kHz betreiben. Hier wird er mit 48 kHz getaktet – das ist auch die Werksvoreinstellung. Die Taktfrequenz lässt sich mittels Sample Frequency Jumper (JP6) auf dem Board einstellen [30]. Für die Anwendung als Active Noise Control System sind diese 48 kHz nicht nötig und bei mehrkanaliger Ausführung vom Prozessor auch Performance-technisch nicht zu bewältigen, da pro Takt nur gut 2000 Berechnungszyklen möglich sind. Daher wurde programmiertechnisch eine künstliche Abtastraten-Reduktion implementiert. Der Grundgedanke hierbei ist, dass man zwar jedes Sample abliest, aber nur jedes N-te Sample bearbeitet. Dafür wurde eine neue Subroutine angelegt und die ISR-Tabelle (ISR=Interrupt Service Routine) verändert.

Im Allgemeinen funktioniert das so, dass mittels einer Zähler-Routine am Anfang jeder Interrupt-Behandlung (SP0I_svc) überprüft wird, ob seit der letzten Sample-Bearbeitung die ISR schon (N-1)-mal aufgerufen worden ist. Falls ja, wird eine weitere Sample-Bearbeitung gestartet, falls nicht, wird die ISR beendet und es wird auf einen weiteren Interrupt gewartet. Das Ergebnis ist, dass nun für die Berechnung des Algorithmus" auch die folgenden ausgelassenen Abtastzyklen zur Verfügung stehen und damit ein vielfaches der eigentlich pro Abtastzyklus möglichen Rechenschritte abgearbeitet werden können. Damit das funktioniert, darf die Rechenroutine aber nicht als ein Teil des Interrupts gestartet sein, da dadurch die Behandlung der nachfolgenden Interrupts gesperrt wird. Die Lösung ist die Verwendung der Direktive JUMP(CI) - Clear Interrupt. Diese funktioniert nur dann, wenn Nesting Interrupts deaktiviert sind, das heißt, dass Interrupts mit höherer Priorität solche mit niedrigerer Priorität unterbrechen können. Dadurch entsteht eine geschachtelte Interrupt-Struktur, die auch den Namen Nesting Interrupts erklärt. Wenn man dies vermeidet, kann man aus einer ISR eine Subroutine aufrufen, ohne dass diese vom DSP als ein Teil der ISR angesehen wird.

ISR_table.asm:

```
.EXTERN        SP0I_counter;
.EXTERN        Count_SPORT0_RX_IRQs;
…
// Vectors for Serial port DMA channels:
/* 0x28 - Vector address for serial port 0 primary A, secondary B RX/TX buffers (DMA Chan-
nels 0 & 1) */
SP0I_svc:      call Count_SPORT0_RX_IRQs;
                                r15=dm(SP0I_counter);
                                if eq jump (Process_AD1836_Audio_Samples)(CI);
// if r15=0, do I/O, else back to routine
                                RTI;
```

Wie man sieht, wird hier am Anfang der ISR eine Zähler-Routine aufgerufen. Diese befindet sich in der Datei „AD1836_reg_init_viaSPORTs.asm“. Es wird erst ein weiteres Sample eingelesen, wenn der Zähler auf Null ist, ansonsten wird zurück in die aktuell laufende Berechnungsroutine zurückgesprungen.

AD1836_reg_init_viaSPORTs.asm:

```
.GLOBAL        SP0I_counter;
.GLOBAL        Count_SPORT0_RX_IRQs;
…
.VAR           SP0I_counter = 0;
…
Count_SPORT0_RX_IRQs:
        r14=dm(SP0I_counter);        /* get last count */
        r15=S;                       // take every r15th sample, S defined in mls.h
        r15=r15-1;
        r15=r15-r14;
        if eq r14=r15;
        if ne r14=r14+1;             /* increment count */
        dm(SP0I_counter)=r14;        /* save updated count */
        RTS;
Count_SPORT0_RX_IRQs.end:
```

In dieser Routine wird in r15 gespeichert, wie viele Interrupts zwischen zwei Bearbeitungen liegen müssen. Die Variable S wird im Header-File „mls.h“ definiert. Bei S=24 ergibt sich die künstlich erzeugte Abtastrate von 48000 Hz/24=2000 Hz.

Die Funktion Process_AD1836_Audio_Samples, in der die Samples ein-/ausgegeben und bearbeitet werden, muss auch ein wenig verändert werden. Die Direktive „rti(db)„ wurde durch „rts(lr)„ ersetzt.

SPT0_ISR_1836_Processing.asm:

```
tx_done:
        r0=dm(audio_frame_timer);        /* get last count */
        r0=r0+1;                         /* increment count */
        dm(audio_frame_timer)=r0;        /* save updated count */
        rts(lr);

Process_AD1836_Audio_Samples.end:
```

„lr„ bedeutet Loop Return und garantiert ein problemloses Zurückspringen in die ISR und in den idle-Zyklus.

Sehr wichtig ist, dass man Nesting Interrupts deaktiviert. Dazu wurde in der Routine ‚Init_DSP„ folgende Zeile

Init_21161_EZKit.asm:

```
bit set mode1 IRPTEN | NESTM;                    /* enable global interrupts, nesting */
```

mit den Zeilen

```
        bit clr mode1 NESTM;                     /* disable nesting */
        bit set mode1 IRPTEN;                    /* enable global interrupts */
```

ersetzt.

Durch diese kleinen Erweiterungen des ursprünglichen Quellcodes aus Talkthru wurde ein sehr signifikanter Performance-Zuwachs ermöglicht – allerdings auf Kosten der Abtastpräzision. Die schematische Abbildung 22 verdeutlicht den geänderten Programmablauf:

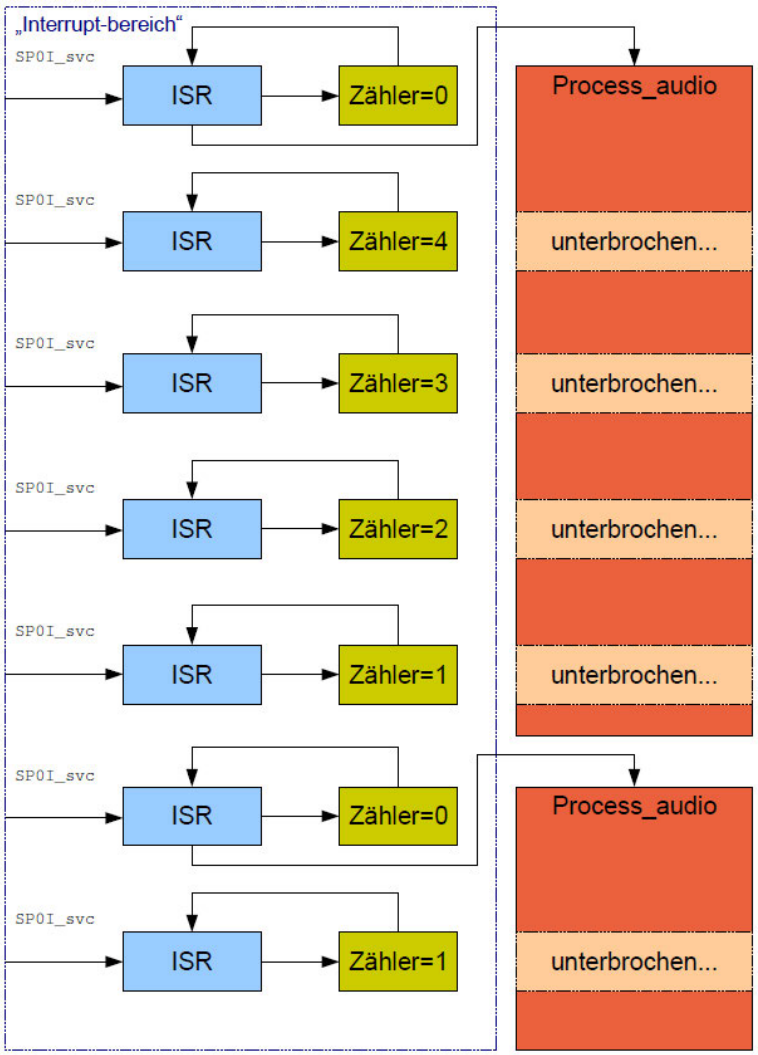


Abbildung 22: Programmablauf der Abtastratenanpassung

3.5 Parameter und Einstellungen im Header-File „mls.h“

3.5.1 Header-Datei für das ANC-System Feedforward

```
/*************************************************************************
 *
 * mls.h
 *
 *
 *************************************************************************/
#define M 11              // order of mls-sequence
#define repeat_mls 3      // iterations of mls (first and last mls is dummy, not included)
#define L 2047            // length of mls-sequence
#define LM 4094
#define S 24              // sample rate div (sample frequency =48kHz/S)
#define amp 0.05          // adjust MLS-amplitude
#define V 1.0             // adjust response-amplitude
#define VO 1.0            // adjust output-amplitude
#define CC 500            // number of samples of c and x / coefficiants used for filtering
#define CW 500            // number of samples of w and r / coefficiants used for adapting
#define CWM 1000
#define CWMM 2000
#define A 0.08            // alpha
```

Abbildung 23: Header-File „mls.h“ (Feedforward, zweikanalig)

Im Header-File „mls.h“ können die nötigen Parameter und Einstellungen zentral eingestellt werden. Es wird so vermieden, dass Änderungen an verschiedenen Stellen im Quelltext nötig sind. Neben „def21161.h“, das essentielle boardspezifische Parameter zur Verfügung stellt, muss dieses ebenfalls zu Beginn einer Quellcode-Datei per include-Befehl eingebunden werden. Die Abbildung 23 zeigt die „mls.h“ für das zweikanalige Feedforward System. Variablen werden in Header-Dateien mit dem ‚#define‚-Befehl deklariert.

```
#define M 11                    // order of mls-sequence
```

Die Variable ‚M‚ steht für die Ordnung der zur Systemidentifikation verwendeten MLS-Folge. Daraus ergibt sich mit $L=2^m-1$ auch die Länge der MLS (‚mls[L]‚).

```
#define L 2047                  // length of mls-sequence
```

Leider ist es in Header-Files nicht möglich, Berechnungen durchzuführen – es wäre natürlich komfortabler, wenn sich die Länge der MLS direkt aus der Ordnung berechnen würde.

Bei der Bestimmung der Impulsantwort (‚ir[LM]‚) eines Systems ist es vorteilhaft, mehrere Folgen auszusenden und die Antworten des Systems zu mitteln, wodurch die Messgenauigkeit erhöht wird und Störsignale heraus gerechnet werden. Die Anzahl der Wiederholungen wird in der Variable ‚repeat_mls‚ gespeichert. Durch Setzen dieser Variable auf „1“ wird die MLS dreimal ausgegeben, wobei die erste und letzte Folge zur Vermeidung von Störungen durch Einschwingen, Ausschaltknacks und andere Effekte dienen. Durch setzen von ‚repeat_mls‚=4 wie im Beispiel werden also zwei Maximalfolgen für die Messung verwendet und die Systemantwort gemittelt.

```
#define repeat_mls 4            // iterations of mls (first and last mls is dummy, not included)
```

Da die Länge der zu definierenden Vektoren gleich zu Beginn im Deklarationsteil festgelegt werden muss, ist es für die mehrkanaligen Anwendungen auch nötig, das entsprechende Vielfache der MLS-Länge zu definieren.

```
#define LM 4094
```

Da im vorliegenden Beispiel die zweikanalige Version verwendet wurde, ergeben sich für ‚LM„ also 2*L=4094 Samples. ‚LM„ bestimmt hier die Länge der Vektoren für Impulsantwort und Systemantwort. Die Art der Speicherung, d.h. Reihenfolge der Werte in den Vektoren wird später noch genauer erläutert (Abschnitt 3.9.3).

Ein wichtiger Parameter ist der Teiler für die Abtastratenreduktion, da sich aus der Abtastrate die z.B. die benötigte Anzahl der in den Vektoren zu speichernden Werte ergibt. Wie bereits zuvor erläutert, ist die Verringerung der Abtastrate auch aus Performance-Gründen sinnvoll. Bei ‚S„=24 ergibt sich für das obige Beispiel also eine Abtastrate von 2 kHz.

```
#define S 24                    // sample rate div (sample frequency =48kHz/S)
```

Die Variable amp ermöglicht es, die Amplitude der auszugebenden Maximalfolge zu regulieren. Jeder Wert der MLS wird vor der Ausgabe mit diesem Parameter multipliziert. Mittels ‚V„ und ‚VO„ können analog dazu die Amplituden der auszugebenden MLS-Samples und die eingelesenen Werte der Systemantwort (‚response[LM]„) angepasst werden.

```
#define amp 1.0                 // adjust MLS-amplitude
#define V 1.0                   // adjust response-amplitude
#define VO 1.0                  // adjust output-amplitude
```

Die folgenden Variablen ‚CC„ und ‚CW„ definieren die Anzahl der Werte und Filterkoeffizenten, die in den entsprechenden Filterroutinen verwendet werden. ‚CC„ bestimmt also die Filterung des Referenzsignals x (‚reference[CC]„) mit der Impulsantwort c (‚ir[LM]„) und ‚CW„ die Filterung des Referenzsignals x (‚reference[CC]„) mit den Adaptionsfilterkoeffizienten w (‚w_filter[CWM]„) zur Berechnung des Ausgangssignals y und entsprechend auch die Berechnung der Adaptionsfilter-Koeffizienten w (‚w_filter[CWM]„).

```
#define CC 500          // number of samples of c and x / coefficiants used for filtering
#define CW 500          // number of samples of w and r / coefficiants used for adapting
#define CWM 1000
#define CWMM 2000
```

Für die mehrkanaligen Algorithmen sind zudem die Vielfachen von ‚CW„ nötig. ‚CWM„ bestimmt die nötige Länge des Vektors mit den Adaptionsfilter-Koeffizenten (‚w_filter[CWM]„) – für zweikanalig eben 2*CW=1000 Samples, die zwischengespeichert werden müssen. ‚CWMM„ gibt die Länge des Vektors vor, der die Werte des gefilterten Referenzsignals r (‚filtered_reference[CWMM]„) vorhält.

Zu guter Letzt und eigentlich selbsterklärend wird die Schrittweite des Adaptionsalgorithmus“ ‚A„ bzw. α definiert.

```
#define A 0.08                    // alpha
```

3.5.2 Header-Datei für das ANC-System Feedback

```
/*************************************************************************
 *
 * mls.h
 *
 *
 *************************************************************************/
#define M 11              // order of mls-sequence
#define repeat_mls 4      // iterations of mls (first and last mls is dummy, not included)
#define L 2047            // length of mls-sequence
#define LM 4094
#define S 24              // sample rate div (sample frequency =48kHz/S)
#define amp 0.05          // adjust MLS-amplitude
#define V 1.0             // adjust response-amplitude
#define VO 1.0            // adjust output-amplitude
#define CC 600            // number of samples of c and x / coefficiants used for filtering
#define CCM 1200
#define CW 500            // number of samples of w and r / coefficiants used for adapting
#define CWM 1000
#define CWMM 2000
#define CWMMM 4000
#define A 0.2             // alpha
```

Abbildung 24: Header-File „mls.h“ (Feedback, zweikanalig)

Prinzipiell unterscheidet sich der Feedback-Algorithmus vom Feedforward-Algorithmus ja dadurch, dass kein Referenzsignal x zu Verfügung steht, sondern dieses mittels Filterung des Ausgangssignals y mit der Impulsantwort c und Addition des Fehlersignals geschätzt wird. Dadurch ergeben sich kleine Ergänzungen in der „mls.h“ für die mehrkanaligen Versionen. Die Parameter, die schon für den Feedforward-Algorithmus im Abschnitt zuvor erläutert wurden, werden hier nicht noch einmal behandelt. Die Abbildung 24 zeigt die modifizierte „mls.h“ für das zweikanalige Feedback System. Hinzugekommen sind die Variablen ‚CCM„ und ‚CWMMM„ ‚CCM„ wird nötig, da das Ausgangssignal y nun ebenfalls in einem Puffer vorgehalten werden muss (‚y_buffer[CCM]„). Die Länge dieses Vektors ergibt sich entsprechend der Anzahl der Kanäle aus dem Vielfachen von CC. Da bei dem Feedforward-Algorithmus pro Kanal ein geschätztes Referenzsignal vorhanden ist, erhöht sich dementsprechend auch die Anzahl der gefilterten Referenzsignale r (‚filtered_reference[CWMMM]„) um ein weiteres Vielfaches der Kanalanzahl – im zweikanaligen Fall also auf 8 gefilterte Referenzsignale (‚filtred_reference[CWMMM]„).

```
#define CCM 1200
…
#define CWMMM 4000
```

3.6 Grenzen des ANC-Systems

Dem ANC-System sind vor allem aufgrund der Performance und der technischen Daten des DSP-Boards Grenzen gesetzt. Damit diese nicht überschritten werden und somit das Funktionieren der Algorithmen gewährleistet ist, werden diese Grenzen im Folgenden genauer aufgezeigt.

Entsprechend den Anforderungen eines ANC-Systems müssen bei der Verarbeitung der Signale teilweise sehr viele Werte in Vektoren oder Puffern gespeichert werden. Vor allem die Maximalfolge zur Systemidentifikation, die (gefilterten) Systemantworten und die Impulsantworten können bei den mehrkanaligen Varianten sehr lang werden. Die Abtastrate ist hierbei natürlich auch ein Faktor. Die Abbildung 25 und die Tabelle 4 und Tabelle 5 zeigen die Speicherausstattung des Boards und welche Speicherbereiche für die verschiedenen Vektoren genutzt werden. Die Werte in Klammern in der Spalte Vektor/Variablen der Tabelle bezeichnen die Zeiger auf die Vektoren, wie sie auch in den Programmen verwendet werden. Da die Speicherbelegung für die Feedback-Algorithmen wegen des Hinzukommens des Puffers für das Ausgangssignal y verändert wurde, sind 2 Tabellen zur Dokumentation nötig. Die farbige Unterscheidung der Vektoren soll verdeutlichen, welche Vektoren in den zeitkritischen ANC-Algorithmen zusammen für Berechnungen benötigt werden und deshalb in unterschiedlichen Speicherbereichen abgelegt worden sind, um die Berechnung in nur einem Zyklus zu ermöglichen (siehe Abschnitt 3.3).

Memory Map:

Segment/Section			Start Address	End Address
mem_rth	PM	250 x 48-bit words	0x40005	0x400ff
mem_init	PM	255 x 48-bit words	0x40100	0x401ff
mem_pmco	PM	3583 x 48-bit words	0x40200	0x40fff
mem_pmda	PM	9727 x 32-bit words	0x41a00	0x43fff
mem_dm48	DM	255 x 48-bit words	0x50000	0x500ff
mem_dm32	DM	255 x 32-bit words	0x50200	0x502ff
mem_dmda	DM	7423 x 32-bit words	0x50300	0x51fff
mem_dmres	DM	8191 x 32-bit words	0x52000	0x53fff
memsdram	SDRAM	1048575 x 32-bit words	0x200000	0x2fffff

Abbildung 25: Speicherbelegung DSP-Board

Nicht nur die Vektoren und Variablen müssen im Speicher vorgehalten werden, sondern natürlich auch der Programmcode. Dieser wird in der Sektion mem_pmco gespeichert. Die Koeffizienten (‚taps„ (i15)) zur Berechnung der Maximalfolge für die Ermittlung der Impulsantworten und diverse Variablen, die später in der betreffenden

Programmbeschreibung genauer erläutert werden, wurden in der Sektion „mem_pmda“ untergebracht. In dieser Sektion werden auch die Adaptionsfilterkoeffizienten w (‚w_filter„ (i10, i11)) und die Werte der Impulsantworten c (‚ir„„ (i9)) gespeichert. Da die Impulsantwort der längste Vektor ist, der bei den zeitkritischen ANC-Berechnungen Verwendung findet, wurde hierfür dieser größte Speicherbereich gewählt. Es ist zu beachten, dass die Anzahl der Werte für die Impulsantworten nicht willkürlich gewählt werden kann, sondern abhängig von der Länge der Impulsantwort des Systems und der Abtastrate ist. Außerdem wurde bei der Systemidentifikation für die Impulsantwort die gleiche Länge definiert wie die Länge der Maximalfolge. Wenn also beim Linken die Fehlermeldung auftritt, dass in der Sektion mem_pmda nicht genügend Speicherplatz vorhanden ist, bestehen nur die Möglichkeiten, die Anzahl der Koeffizienten des Adaptionsfilters zu verringern und/oder die Abtastrate zu verringern.

Tabelle 4: Speicherbelegung Feedforward

Input-Sektion	Speichersegment	Vektor/Variablen	Max. Länge	Bereich
pm_code	mem_pmco	Programmcode, Sytemvariablen	3583	PM
pm_data	mem_pmda	w_filter (i10,i11) ir (i9) taps (i15) div. Variablen	9727	PM
dm_data	mem_dmda	filtered_reference (i3)	7423	DM
seg_dmres	mem_dmres	reference (i2)	8191	DM
seg_sdram	mem_sdram	mls (i1) response (i8)	1048575	SDRAM

In den Sektionen „mem_dmda“ und „mem_dmres“ des Datenspeichers werden die Samples der Referenzsignale x (‚reference„ (i2)) und der gefilterten Referenzsignale r (‚filtered_reference„ (i3)) sowie in den Feedback Varianten die Ausgangssignalwerte y (‚y_buffer„ (i14)) gespeichert.

Da Berechnungen mit Werten aus dem externen Speicher (SDRAM) mehrere Takt-Zyklen erfordern, wurden hier nur Vektoren gespeichert, die für nicht-zeitkritische Berechnungen und Operationen verwendet werden. Die MLS-Folge ‚mls„ (i1) wird nur einmal zur Ausgabe als Messsignal und dann noch zur Berechnung der Impulsantwort durch Kreuzkorrelation mit der ebenfalls in diesem Bereich gespeicherten Systemantwort ‚response„ (i8) vor den eigentlichen Active-Noise-Control-Algorithmen benötigt. Dabei spielt die Berechnungszeit nur eine untergeordnete Rolle. Da diese Vektoren sehr lang werden können, musste hier auf das externe SDRAM ausgewichen werden.

Tabelle 5: Speicherbelegung Feedback

	Speicherseg-ment	Vektor/Variablen	Max. Länge	Bereich
pm_code	mem_pmco	Programmcode, Sytemvariablen	3583	PM
pm_data	mem_pmda	w_filter (i10,i11) ir (i9), ir (i9) taps (i15) div. Variablen	9727	PM
dm_data	mem_dmda	y_buffer (i4)	7423	DM
seg_dmres	mem_dmres	reference (i2) filtered_reference (i3)	8191	DM
seg_sdram	mem_sdram	mls (i1) response (i8)	1048575	SDRAM

In Tabelle 3 wurde bereits gezeigt, welche theoretischen Grenzen dem System bei der Echtzeit-Berechnung gesetzt sind. Um mit dem System sinnvoll arbeiten zu können, wurden außerdem Tests gemacht, mit welcher maximalen Anzahl von Adaptionsfilterkoeffizienten (‚CW„) die Feedforward-Algorithmen ein und zweikanalig noch korrekt arbeiten. Bei der Anwendung am Anti-Schall-Fenster ist die Länge der Impulsantworten mit etwa 0,3 s zu veranschlagen. Für verschiedene Abtastfrequenzen ergeben sich daraus entsprechend viele Samples, die im Impulsantwortvektor

gespeichert und zur Berechnung herangezogen werden müssen. Bei f_S=2 kHz haben die Impulsantworten beispielsweise eine Länge von f_S*0,3s=600 Samples. Neben der Anzahl der möglichen Koeffizienten („CW„) wird in Tabelle 6 auch der für die Abtastratenanpassung einzustellende Teiler („S„) und die minimal benötigte Länge der MLS („L„) und deren Ordnung („M„) aufgeführt. Bei geringen Abtastfrequenzen wird die Grenze übrigens nicht durch die Echtzeitberechnung, sondern durch die Speicherkapazität gesetzt.

Tabelle 6: Echtzeit-Berechnungsgrenzen Feedforward

fs (Hz)	S	length of IR (samples)	L	M	CW
einkanalig					
2000	24	600	1023	10	6800
4000	12	1200	2047	11	5000
6000	8	1800	2047	11	2000
8000	6	2400	4095	12	450
zweikanalig					
1000	48	300	1023	10	1500 (Speicher)
2000	24	600	2047	11	1500 (Speicher)
4000	12	1200	4095	12	450

3.7 MLS-Generator

3.7.1 Einleitung

Grundlage für die Systemidentifikation des aktiven Lärmschutzaufbaus ist die Generierung des geeigneten Messsignals – hier eine Maximalfolge. Die theoretischen Hintergründe wurden bereits in einem früheren Abschnitt behandelt. Der MLS-Generator ist so ausgelegt, dass Maximalfolgen bis zur Ordnung M=31 und damit einer Länge von $L=2^M-1=2147483647$ Samples erzeugt werden können. Dieser Wert ist natürlich nur akademischer Natur. Im Zusammenhang mit diesem Projekt können aufgrund der Speicherausstattung des DSP-Boards nur maximal 1048575 Samples im SDRAM gespeichert werden. Zudem ist für die Berechnung der Impulsantwort die gleiche Anzahl Samples für die Systemantwort nötig, so dass man auf einen Maximalwert von 524287 Samples jeweils für die MLS und die Systemantwort zur Bestimmung der Impulsantwort kommt. Allerdings sind auch Anwendungen denkbar,

für die nur das Aussenden der Maximalfolge ohne Speicherung nötig ist. Zu diesem Zweck könnte das Programm einfach angepasst werden. Eine kostenlose Alternative dazu stellt das Programm Signalgenerator von Dr-Jordan-Design dar [35].

Im folgenden Abschnitt wird noch etwas genauer auf die Variablen und- Pufferdeklarationen eingegangen. In den späteren Beschreibungen der anderen Programmteile wird darauf dann jedoch weitgehend verzichtet.

3.7.2 Variablen- und Vektorendeklaration

Der MLS-Generator ist ein relativ einfaches Programm, das mit wenigen Variablen auskommt. Die Ordnung bzw. Länge der zu erzeugenden Maximalfolge wird im Header-File „mls.h“ mittels der Variablen M bzw. L festgelegt. Damit korrespondierend sind die Rückkopplungskoeffizenten für die jeweilige Ordnung der zu generierenden MLS nötig, die aus der Datei „taps.dat“ in den Vektor „taps“ eingelesen werden.

```
.section /pm        pm_data;
…
.VAR    taps[31]="taps.dat";
```

Weiterhin muss noch der Vektor definiert werden, in dem die einzelnen Werte der MLS-Folge gespeichert werden.

```
.section /dm      segsdram;
.VAR    mls[L];
```

Initialisiert wird dieser Vektor dann im Bereich des Codes, in dem die Vektoren und Puffer angelegt werden. b1 ist das Basisregister und gibt die Startadresse des Vektors an, l1 Längenregister) definiert die Länge des Vektors.

```
user_code_init:
…
b1=mls; l1=@mls;
…
m1=1;
```

m1 ist der Wert, um den der Zeiger des Vektors nach dem Schreiben oder Auslesen eines Werts erhöht bzw. verringert wird. Da in diesem Fall die Werte einer nach dem anderen in den MLS-Vektor geschrieben werden, ist das sogenannte Modifizierungsregister m1 hier mit der Schrittweite 1 definiert.

Jedes Unterprogramm muss ebenfalls im Deklarationsteil einer bestimmten Input-Sektion und damit einem bestimmten Speichersegment zugeordnet werden.

```
.section /pm pm_code;
…
.GLOBAL                    mls_gen;
```

3.7.3 Programmbeschreibung

Wie bereits an früherer Stelle ausführlicher beschrieben, wird zur Erzeugung einer Maximalfolge ein rückgekoppeltes Schieberegister (r3) verwendet. Es wird, da es einen Anfangszustand ungleich 0 haben muss, einfach mit der Länge der zu erzeugenden MLS initialisiert.

```
r3=L;
```

Als Rückkopplungsvorschrift werden die in die Variable „taps“ eingelesenen Werte verwendet. Diese werden in das Register r4 eingelesen.

```
r4=pm(M,i15);                                   // read coefficients for MLS
```

Durch Modulo-2-Addition (XOR-Verknüpfung) dieser beiden Register wird jeweils ein Wert der MLS erzeugt. Da diese logische Operation nicht zur Verfügung steht, musste sie durch logisches AND, anschließendem Zählen der „1“ und der Prüfung, ob diese Summe gerade oder ungerade ist, implementiert werden (bei ungerader Anzahl wird eine „1“ in das niederwertigste Bit 2^0 des Schieberegisters geschrieben). Deshalb besteht das Grundgerüst des MLS-Generators aus zwei Programmschleifen. In der äußeren Schleife „mls_calc“ findet mit

```
r10=r3 and r4;                                  // register AND taps
```

zuerst die logische AND-Verknüpfung statt. In der inneren Schleife ‚bit_count„ werden dann die „1“ im Ergebnisregister r10 gezählt, indem jeweils geprüft wird, ob an der Stelle 2^0 eine „1“ steht. Diese „Einsen“ werden dann im Register R14 aufaddiert:

```
r13=r10 and r1;                                 // "1" at 2^0 ? --> 1
r14=r14+r13;
```

Es handelt sich hierbei um Bit-Operationen. Am Ende der Schleife wird das Ergebnisregister r10 um eine Stelle nach rechts verschoben, um dann im nächsten Schleifendurchlauf die Stelle 2^1 (nun 2^0) auf „1“ zu prüfen.

```
bit_count: r10 = LSHIFT r10 BY -1;              // shift 2^1 --> 2^0
```

Die Summe in r14 wird nun mittels AND-Verknüpfung mit r1=1 darauf geprüft, ob sie gerade oder ungerade ist.

```
r14=r14 and r1;                                 // 1-odd, 0-even --> MLS
```

Ist das Ergebnis der AND-Operation „1“, d.h. es steht an der Stelle 2^0 eine „1“, ist die Anzahl der „1“ ungerade und damit der Wert für die MLS „1“. Auch nach den folgenden Operationen bleibt im Register r11 die „1“ erhalten. Die Operationen

```
r11=r11+r11;                                    // 2*r11-1
r11=r11-r1;                                     // Substitution 0 --> -1
```

dienen dazu, das Ergebnis „0“ in r11 in „-1“ umzuwandeln, da die Maximalfolge für Messzwecke aus „1“ und „-1“ bestehen soll.

Die nächsten drei Programmzeilen dienen der Anpassung der Amplitude der MLS. Dabei wird auch gleich die Umwandlung des Integer-Wertes in das Gleitkomma-Format vorgenommen. Die Variable ‚amp„ kann in der Header-Datei definiert werden.

```
f11=float r11;                    // convert to floating point
f12=amp;                          // read amplitude multiplicator
f11=f11*f12;                      // adjust amplitude
```

Dann wird der MLS-Wert in den Vektor „mls“ geschrieben:

```
dm(i1,m1)=f11;                    // write mls-result
```

Zu guter Letzt muss noch das Schieberegister r3 um eine Stelle nach links verschoben und in dessen neue Stelle 2^0 der ermittelte MLS-Wert geschrieben werden. Das höchstwertige Bit fällt dabei quasi links heraus.

```
r3=LSHIFT r3 BY 1;                // shift register
mls_calc: r3=r3+r14;              // "0" or "1" to 2^0
```

Damit ist dieser Generatorschritt abgeschlossen und es kann der nächste Maximalfolgenwert erzeugt werden. Die Programmschleife wird genau so oft ausgeführt, wie die zu generierende MLS Werte hat, d.h. ‚L„-mal. Danach liegt die komplette Folge im Vektor ‚mls„ (i1) vor und kann zur Messung ausgegeben werden.

3.8 Ausgabe/Aufnahme der Messsignale

3.8.1 Einleitung

Bei der Ein- und Ausgabe der Messsignale werden in diesem Projekt erstmals die Möglichkeiten des DSP-Boards genutzt, mit der analogen Welt zu kommunizieren. Der folgende Abschnitt erläutert die dazu nötigen System- und Uservariablen. Die digitale Welt ist in Abtastraten unterteilt und damit muss die Ein- und Ausgabe von Samples im gleichen Abtastschritt erfolgen. Darum wurden diese Prozeduren in einem Unterprogramm (‚input_output„) umgesetzt.

Es werden die zweikanaligen Varianten erläutert, damit keine Unklarheiten bezüglich der Kanalbelegungen entstehen können und eine korrekte Verkabelung gewährleistet ist. Die einkanaligen Versionen sind analog programmiert.

3.8.2 Variablen- und Vektorendeklaration

In jeder Datei, die Teil des Gesamtprojektes ist, müssen die verwendeten Variablen, Vektoren und Unterprogramme deklariert werden. Die konkrete Ein- und Ausgabe erfolgt in der Datei „SPT0_ISR_1836_Processing.asm“. Diese Datei ist Bestandteil des Grundgerüsts „Talkthru“ und wurde nur geringfügig angepasst. An dieser Stelle

noch ein kleiner Exkurs in die Syntax der Variablendeklaration. Mit vorangestelltem „.VAR„ werden Variablen oder Vektoren und Puffer definiert. Diese können auch in anderen Dateien des Projektes verwendet werden, wenn sie dort mit vorangestelltem „.EXTERN„ deklariert werden. Unterprogramme, die auch aus anderen Projektdateien erreichbar sein sollen, werden mir „.GLOBAL„ definiert. In den Dateien, die solche Unterprogramme aufrufen sollen, muss dieses Unterprogramm ebenfalls deklariert werden – dann mit „.EXTERN„.

Zuerst wird hier nun jedoch der Deklarationsteil für die Datei „noise_control.asm“ behandelt. Folgende Variablen wurden für den Programmteil, der für die Ein- und Ausgabe der Samples zur Aufnahme der Systemantwort zuständig ist, definiert. Sie dienen dazu, die Anzahl der aus- und einzulesenden Werte zu steuern. Die genauere Verwendung wird später im Quelltexterklärungsteil erläutert.

```
.section /pm      pm_data;              //define your user program memory variables here
.VAR    repeat_mls_samples;
.VAR    repeat_mls_samples_counter=0;
.VAR    repeat_input;
.VAR    stop_input_samples=0;
```

Natürlich müssen auch die entsprechenden Vektoren definiert werden. „mls[L]„ ist hier nur der Vollständigkeit halber aufgeführt – diese Variable wurde schon im Zusammenhang mit dem MLS-Generator deklariert und verwendet. Außerdem ist ein Vektor nötig, in dem die Samples der Systemantwort gespeichert werden können („response[L]„). Da in den mehrkanaligen Versionen ein der Kanalanzahl entsprechendes Vielfaches an Samples gespeichert werden muss, wird die Länge dieses Vektors dort als „response[LM]„ definiert.

```
.section /dm     segsdram;
.VAR    mls[L];
.VAR    response[L];
.section /pm pm_code;
.EXTERN                 Left_Channel_In0;
.EXTERN                 Right_Channel_In0;
.EXTERN                 Left_Channel_In1;
.EXTERN                 Right_Channel_In1;
.EXTERN                 Left_Channel_Out0;
.EXTERN                 Right_Channel_Out0;
.EXTERN                 Left_Channel_Out1;
.EXTERN                 Right_Channel_Out1;
.EXTERN                 Left_Channel_Out2;
.EXTERN                 Right_Channel_Out2;
.EXTERN                 Left_Channel_AD1852;
.EXTERN                 Right_Channel_AD1852;
.GLOBAL                 input_output;
```

Die mit vorangestelltem „.EXTERN„ deklarierten Variablen wurden ursprünglich in der Datei „SPT0_ISR_1836_Processing.asm“ definiert. Damit Sie in der Datei „noise_control.asm“ nutzbar sind, müssen Sie auch hier deklariert werden. Wie die Namen schon nahelegen, handelt es sich hierbei um Variablen, in denen die Samples für die Ein- und Ausgabe über die Audio-Codecs zwischengespeichert werden.

„input_output„ deklariert das Unterprogramm, damit es im Programmablauf per Sprungbefehl angesteuert werden kann. Das ist nötig, da nach Ein- und Ausgabe der Samples im Unterprogramm „Process_AD1836_Audio_Samples„ in der Datei „SPT0_ISR_1836_Processing.asm“ die Wahl getroffen werden muss, ob zum Programmteil zur Ermittlung der Systemantwort oder nach Abschluss der Systemidentifikation in das Unterprogramm mit den ANC-Algorithmen gesprungen wird.

Abschließend noch der Vollständigkeit halber die Quelltextzeilen, in denen die beiden verwendeten Vektoren initialisiert werden.

```
user_code_init:
…
b1=mls; l1=@mls;
…
b8=response; l8=@response;
```

Die Datei „SPT0_ISR_1836_Processing.asm“ mit dem Unterprogramm „Process_AD1836_Audio_Samples„ ist für die eigentliche Ein- und Ausgabe der Sound-Werte zuständig. Einige Teile des Deklarationsteils werden an dieser Stelle nicht genauer erklärt, da es sich um systemspezifische Definitionen handelt, deren genaue Funktion für das Gesamtverständnis des Projektes meines Erachtens nicht wesentlich sind. Der Deklarationsteil sieht hier folgendermaßen aus – wie in jeder zum Projekt gehörenden Programm-Datei muss auch hier zuerst die Register- und Bit-Definitionsdatei „def21161.h“ eingebunden werden.

```
/* ADSP-21161 System Register bit definitions */
#include        "def21161.h"
…
.GLOBAL         Process_AD1836_Audio_Samples;     /* Label of code listed here to get
                                                     samples into and out of*/
```

In den folgenden Zeilen werden die Puffer-Variablen für die Ein- und Ausgabe an die Audio-Codecs deklariert. Damit diese Variablen auch in anderen Programmteilen nutzbar sind, werden sie mit „.GLOBAL“ definiert. Die SPDIF-Kanäle sind für dieses Projekt eigentlich nicht nötig. Auch der AD1854-Codec wird nicht verwendet. Um das Projekt aber einfach erweiterbar zu gestalten, wurden diese Deklarationen nicht entfernt.

```
/* SPORT DMA buffers and process_audio routine*/
.GLOBAL     Left_Channel_In0;          /* These will tranfer ADC samples of
                                          interest from the*/
.GLOBAL     Right_Channel_In0;         /* Process_AD1836_Audio_Samples routine
                                        to the process_audio routine*/
.GLOBAL     Left_Channel_In1;
.GLOBAL     Right_Channel_In1;
.GLOBAL     Left_Channel_SPDIF_rx;
.GLOBAL     Right_Channel_SPDIF_rx;
.GLOBAL     Left_Channel_Out0;         /* These will bring samples for the
                                          DACs back into the*/
.GLOBAL     Right_Channel_Out0;        /* Process_AD1836_Audio_Samples
                                          routine from the process_audio routine*/
.GLOBAL     Left_Channel_Out1;
.GLOBAL     Right_Channel_Out1;
.GLOBAL     Left_Channel_Out2;
.GLOBAL     Right_Channel_Out2;
.GLOBAL     Left_Channel_AD1852;
.GLOBAL     Right_Channel_AD1852;
//.GLOBAL   Left_Channel_Out_Thru;     /* These serve the same function for a dif
                                        ferent set of audio */
//.GLOBAL   Right_Channel_Out_Thru;    /* samples destined for the DACs*/
.EXTERN     tx2a_buf;                  /* These are the DMA buffers that hold the 8
                                          channels of audio */
.EXTERN     rx0a_buf;                  /* immediately pre-transmission and post-
                                          reception */
```

Damit in die Programmteile ‚input_out„ und ‚noise_control„ gesprungen werden kann, müssen diese auch hier bekanntgegeben werden.

```
.EXTERN     input_output;              /* Label of audio algorithm code listed in an
                                          other file which executes */
.EXTERN     noise_control;             /* a processing algorithm on the audio data
                                          before it is output to the DACs */
.EXTERN     switch_noise_control;
```

Es folgt die eigentliche Deklaration der Puffer-Variablen für die Audio-Codecs:

```
.section /dm    dm_codec;
/* AD1836 stereo-channel data holders - used for DSP processing of audio data received
from codec */
.VAR        Left_Channel_In0;          /* Input values from AD1836 ADCs */
.VAR        Left_Channel_In1;
.VAR        Right_Channel_In0;
.VAR        Right_Channel_In1;
.VAR        Left_Channel_SPDIF_rx;
.VAR        Right_Channel_SPDIF_rx;
.VAR        Left_Channel_Out0;         /* Output values for AD1836 DACs */
.VAR        Left_Channel_Out1;
.VAR        Left_Channel_Out2;
.VAR        Right_Channel_Out0;
```

```
.VAR        Right_Channel_Out1;
.VAR        Right_Channel_Out2;
.VAR        Left_Channel_AD1852;
.VAR        Right_Channel_AD1852;
```

3.8.3 Vorbereitende Berechnungen für die Ausgabe der Messsignale

Der Systemidentifikationsalgorithmus ist so programmiert, dass eine große Freiheit bei der Wahl der Länge der Mess-Maximalfolge gewährleistet ist. In Kombination mit der Anzahl der Wiederholungen der MLS-Ausgabe zur Berechnung der gemittelten Systemantwort sind im Vorfeld einige Berechnungen nötig. Diese werden bereits im Unterprogramm ‚user_code_init„ vorgenommen.

Zuerst werden die Werte für die Länge der MLS (r0) und die Anzahl der Wiederholungen (r1) eingelesen. r2 wird mit 2 initialisiert – das repräsentiert die Dummy-Ausgaben vor und nach der Messung, um Effekte beim Ein- und Ausschwingen des Systems zu vermeiden.

```
r0=L;                    // calculate number of mls output samples
r1=repeat_mls;
r2=2;
```

Die folgenden Code-Zeilen berechnen die Anzahl der auszugebenden Samples inklusive der beiden Dummy-Ausgaben. Dann wird dieser Wert für die eigentliche Ausgabe in der Variable ‚repeat_mls_samples„ gespeichert. ‚dm(repeat_mls_samples)„ bedeutet, dass sich diese Variable dann im Daten-Speicher befindet.

```
r2=r1+r2;
r12=r0*r2 (SSI);
dm(repeat_mls_samples)=r12;
```

Abschließend wird noch die Anzahl der Werte berechnet, die für die Ermittlung der gemittelten Systemantwort herangezogen werden.

```
r12=r0*r1 (SSI);         // calculate number of samples for average response (after
                         first mls-sequence)
dm(repeat_input)=r12;
```

Da der Ursprungszustand der Vektoren nach dem “Bauen” des Programms nicht definiert ist, werden alle Vektoren, die an allen Stellen eine „0“ enthalten müssen, in den folgenden Programmschleifen mit „0“ initialisiert.

```
f0=0.0;
LCNTR=CWM, DO zero_loop UNTIL LCE;

zero_loop: pm(i10,m8)=f0;                          // write zeros to w_filter
LCNTR=CC, DO zero_loop_reference UNTIL LCE;
zero_loop_reference: dm(i2,m1)=f0;                 // write zeros to reference(x)
LCNTR=CWMM, DO zero_loop_filtered_reference UNTIL LCE;
zero_loop_filtered_reference: dm(i3,m1)=f0;        // write zeros to filtered_reference(r)
```

Dies wird an dieser Stelle explizit erwähnt, da für die Aufsummierung und spätere Mittelwertbildung der Input-Samples unbedingt der Null-Zustand herrschen muss.

```
LCNTR=LM, DO zero_loop_response UNTIL LCE;
zero_loop_response: pm(i8,m8)=f0;                  // write zeros to response (nesse
                                                   cary because of average calculation)
```

In den mehrkanaligen Versionen wird die MLS entsprechend der Anzahl der Kanäle verschoben ausgegeben. Das verkürzt die Messzeit, erfordert aber, dass die Länge der MLS mindestens ein Vielfaches der Kanalanzahl beträgt. In den einkanaligen Varianten kann die gesamte Länge der MLS zur Bestimmung der Impulsantwort genutzt werden.

In den zweikanaligen Versionen wird die ursprüngliche Maximalfolge und eine um etwa die halbe Länge verschobene MLS ausgegeben. Da Maximalfolgen mit unterschiedlichen Längen möglich sind und diese nicht durch 2 teilbar sind, ist etwas Rechenaufwand nötig, um den Verschiebungswert zu berechnen.

```
user_code_init:
…
r3=L;
f4=0.5;                    // calculate shift MLS by 1/2
r3=r3+1;
f3=float r3;
```

Nach dem Einlesen der Länge der Maximalfolge und des Teilers wird zu ‚L„ eine 1 addiert, um einen geraden Wert zu erhalten, der dann später durch 2 teilbar ist. Dann muss der Integer-Wert im Register r3 in einen Gleitkomma-Wert umgewandelt werden, um die Multiplikation zu ermöglichen. Division ist übrigens nicht möglich, daher wird f4 mit 0,5 initialisiert, um später die halbe Maximalfolgenlänge zu berechnen. Register mit vorangestelltem f enthalten Zahlen im Gleitkomma-Format (floating-point).

Dann folgen die eigentliche Multiplikation und anschließend die Umwandlung des Ergebnisses mit dem fix-Befehl ins Integer-Format. Damit die Verschiebung vor der Hälfte der MLS liegt, wird noch eine 1 vom Ergebnis abgezogen.

```
f3=f3*f4;
r3=fix f3;
r3=r3-1;
```

Zur späteren Verwendung werden Modifizierungsregister und die Variable ‚ir_shift„ mit dem Verschiebungswert initialisiert.

```
m0=r3;
m15=r3;
dm(ir_shift)=r3;                    // ir_shift=..., 255, 511, 1023, etc
```

Der Verschiebungswert ist also ((L+1)/2)-1 und entsprechend der Ordnung/Länge der Maximalfolge damit 63, 127, 255, 511 usw.

Nachdem r3 nun die passende Verschiebung enthält, wird in die folgende Routine ‚init_ir_shift„ gesprungen. Hier werden die beiden Modifizierungsregister m0 und m15 mit dem Verschiebungswert initialisiert.

```
init_ir_shift:
m0=r3;
m15=r3;
init_ir_shift.end:
```

Es ist nötig, hier die beiden Modifizierungsregister m0 und m15 zu verwenden, da grundsätzlich die Register m0 bis m7 nur für Vektoren benutzt werden können, die im Datenspeicher (DM) abgelegt sind (i0 bis i7). Analog dazu sind die Modifizierungsregister m8 bis m15 für die Vektoren im Programmspeicher (PM) zuständig (i8 bis i15). m0 sorgt für die Verschiebung der Maximalfolge (‚mls„ (i1)) bei der Ausgabe und m15 wird später im ANC-Algorithmus dazu verwendet, den Zeiger des Vektors mit den Impulsantworten (‚ir„ (i9)) auf das erste Sample der gewünschten Impulsantwort zu stellen.

Damit sind die Vorbereitungen für die Ein- und Ausgabe der Samples zur Ermittlung der Systemantwort abgeschlossen.

3.8.4 Programmbeschreibung Ausgabe und Einlesen der Signale

Dieser Abschnitt beschäftigt sich mit der Ausgabe der Mess-Maximalfolge und dem Einlesen der Systemantwort. Da in einem Abtastschritt jeweils ein Sample ausgegeben und eingelesen wird, wurde die Ein- und Ausgabe in einem gemeinsamen Programmteil ‚input_output„ realisiert. Grundsätzlich findet in der Datei „noise_control.asm“ nur die Verarbeitung und Speicherung der Samples statt. Zur eigentlichen Ausgabe über den AD1836 96 kHz Audio Codec wird anschließend entspre-

chend der ISR-Tabelle zurück in die Datei „SPT0_ISR_1836_Processing.asm“ – genauer gesagt in die Hauptroutine ‚Process_AD1836_Audio_Samples„ gesprungen.

Mittels der folgenden beiden kleinen Prozeduren wird entschieden, ob nur Dummy-Samples ausgegeben werden oder auch gleichzeitig die Systemantwort aufgenommen wird. In der ersten Prozedur wird geprüft, ob bereits eine volle MLS-Folge ausgegeben wurde, welche für ein Einschwingen des Systems sorgen soll. Die Variable ‚repeat_mls_samples_counter‘ wird mit jeder Ausgabe eines Samples um 1 erhöht. Solange die Differenz zwischen der MLS-Länge ‚L„ und diesem Counter größer als 0 ist, wird direkt in die Ausgaberoutine ‚samples_out‘ gesprungen.

```
input_output:
r0=L;
r12=dm(repeat_mls_samples_counter);
r11=r0-r12;
if gt jump samples_out;
r0=dm(repeat_input);
r12=dm(stop_input_samples);
r11=r0-r12;
if le jump samples_out;
```

Die zweite Prozedur ermittelt, ob bereits die nötige Anzahl an Samples für die Messung verarbeitet wurde. Die Variable ‚repeat_input‘ enthält diese für die Messung verwendete Anzahl Samples, d.h. das Vielfache der MLS-Länge entsprechend der in dem Header-File „mls.h“ definierten Anzahl der Wiederholungen zur Mittelung des Messergebnisses. ‚stop_input_samples‘ ist ein Zähler, der mit jedem Einleseschritt der Messung um 1 erhöht wird. Ab dem Moment, bei dem die Differenz zwischen diesen beiden Variablen kleiner oder gleich 0 ist, wird die Einlese-Prozedur übersprungen und nur Dummy-Samples ausgegeben. Solange also diese Differenz größer als 0 ist, wird auch die Routine zum Einlesen der Systemantwort abgearbeitet.

```
read_response_sample:
r2 = -31;
r8=dm(Left_Channel_In0);                    // read ch1 left input sample (#1)
f8 = float r8 by r2;                        // convert to floating point
```

Die vorstehenden Code-Zeilen wandeln den im Input-Puffer vorliegenden Integer-Wert in einen Gleitkomma-Wert um und skalieren ihn dabei auf einen Wertebereich zwischen -1 und +1. Dann wird dieser Wert noch mit der im Header-File definierten Amplitudenanpassung multipliziert.

```
f0=V;
f8=f8*f0;
```

Anschließend wird dieser Wert durch die Anzahl der Wiederholungen der MLS-Messung geteilt. Die Variable ‚repeat_mls" wurde in der „mls.h" festgelegt.

```
r3=repeat_mls;
f3=float r3;
f3=RECIPS f3;                    // calculate 1/repeat_mls for calculation average
f8=f8*f3;
```

Nun wird der Wert ausgelesen, der im vorhergehenden Mess-Durchlauf an dieser Stelle im Vektor ‚response[LM]" (i8), der die Systemantwort enthält, gespeichert wurde. Zu diesem wird das aktuell eingelesene Sample addiert und wieder an der gleichen Stelle des Systemantwort-Vektors gespeichert. Damit wird also erreicht, dass nach jedem Mess-Durchlauf bereits die aktuellen Durchschnittswerte im Vektor vorliegen.

```
f9=pm(i8,0);                     // read old response_1_1
f8=f8+f9;                        // add
pm(i8,m8)=f8;                    // write to response_1_1
```

In den zweikanaligen Versionen werden danach entsprechend angepasste Prozeduren auch für den anderen Kanal abgearbeitet. Diese werden an dieser Stelle nicht näher erläutert, da sie analog dazu funktionieren und nur die verwendeten Input-Puffer andere sind. In jedem Abtastschritt werden die Werte für die Kanäle hintereinander in den Response-Vektor geschrieben. Diese Speicherphilosophie zu kennen ist später für das Verständnis der Routine zur Berechnung der Impulsantwort wichtig. Zum Abschluss bleibt noch, den Zähler für die Einlese-Abtastschritte um 1 zu erhöhen.

```
r12=r12+1;
dm(stop_input_samples)=r12;
```

In der Routine ‚samples_out„ werden die Werte der Maximalfolge aus dem Vektor ‚mls[L]„ (i1) ausgelesen und in den Puffern für die Ausgabe über den Audio-Codec gespeichert. Da in den mehrkanaligen Versionen verschobene MLS ausgegeben werden, wird zuerst der Start-Pointer des MLS-Vektors in der Variable ‚pointer_save„ gespeichert, um diesen am Ende der Prozedur wieder einfach an die richtige Stelle zurücksetzen zu können.

```
samples_out:
dm(pointer_save)=i1;
```

Anschließend wird der erste Wert der MLS aus dem Vektor ausgelesen und im Register f8 abgelegt. Dann muss das Zahlenformat wieder in das für den Audio-Codec passende Integer-Zahlenformat gebracht werden. Nun kann dieses Sample an den Ausgabe-Puffer für den linken Kanal des ersten Ausgangs des Audio-Codecs übergeben werden.

```
f8=dm(i1,m1);
r2 = 31;                                    // scale the result back up to MSBs
r8 = fix f8 by r2;
dm(Left_Channel_Out0)  = r8;                // ch1 left output sample (#1)
```

Nun soll das erste Sample der verschobenen Maximalfolge auf dem rechten Kanal des Ausgangs 1 ausgegeben werden. Dazu muss der Zeiger des MLS-Vektor mittels des vorher mit dem Verschiebungswert initialisierten Modifizierungsregisters ‚m0„ verändert werden.

```
modify(i1,m0);                              // shift MLS
```

Die Anpassung des Zahlenformats und das Befüllen des Ausgabe-Puffers geschehen dann analog zum Vorgehen beim ersten Kanal. Auf die Darstellung des entsprechenden Quelltextes wurde hier verzichtet.

Nun muss der Zeiger des MLS-Vektors noch auf den nächsten Wert der MLS gestellt werden.

```
i1=dm(pointer_save);                        // shift MLS back
modify(i1,1);                               // move pointer to next MLS-sample
```

Es folgen einige abschließende Operationen, um die Zähler korrekt einzustellen. Außerdem wird mit Hilfe des ‚comp„-Befehls und anschließender ‚if„-Verzweigung festgestellt, ob die Ein- und Ausgabeprozedur verlassen und in die Routine zur Berechnung der Impulsantwort gesprungen werden kann. Dies passiert genau dann, wenn der Counter ‚repeat_mls_samples_counter„ und der negativ zählende Counter ‚repeat_mls_samples„, der ursprünglich mit der Gesamtanzahl der auszugebenden Samples initialisiert wurde, gleich groß sind. Es wurden dann also insgesamt jeweils eine Dummy-MLS vor und nach der eigentlichen Messung und dazwischen die Samples zur eigentlichen Messung ausgegeben. Hierbei sind natürlich auch die Messungswiederholungen zur Mittelwertbildung berücksichtigt. Abschließend wird der Zähler ‚repeat_mls_samples_counter„ der die Anzahl der bereits ausgegeben Werte beinhaltet, erhöht.

```
r12=dm(repeat_mls_samples_counter);
r0=dm(repeat_mls_samples);
r0=r0-1;
comp (r12,r0);
if eq jump calc_impulse_response;
r12=r12+1;
dm(repeat_mls_samples_counter)=r12;
```

Die eigentliche Ein- und Ausgabe über den Audio-Codec wurde an dieser Stelle noch ausgeklammert – der folgende Abschnitt wird sich mit diesem Programmteil beschäftigen.

3.8.5 Ein- und Ausgabe über den AD1836 96 kHz Audio Codec

In der Datei „SPT0_ISR_1836_Processing.asm“ werden die Daten von den A/D-Wandlern des AD1836 96 kHz Audio Codec eingelesen und die bearbeiteten Audio-Daten an die D/A-Wandler geliefert. Die Variablendeklarationen wurden bereits erläutert. In der Routine ‚Process_AD1836_Audio_Samples„ werden zuerst die Werte aus den Puffern der A/D-Wandler ausgelesen und in Variablen geschrieben, die dann auch in anderen Programmteilen verwendet werden können. Die SPDIF-Kanäle werden in diesem Projekt nicht bearbeitet.

```
Process_AD1836_Audio_Samples:
/* get AD1836 left channel input samples, save to data holders for processing */
r0 = dm(rx0a_buf + Internal_ADC_L0);  dm(Left_Channel_In0) = r0;
r0 = dm(rx0a_buf + Internal_ADC_L1);  dm(Left_Channel_In1) = r0;
r0 = dm(rx0a_buf + AUX_ADC_L0);       dm(Left_Channel_SPDIF_rx) = r0;
        /* get AD1836 right channel input samples, save to data holders for processing */
        r0 = dm(rx0a_buf + Internal_ADC_R0);  dm(Right_Channel_In0) = r0;
        r0 = dm(rx0a_buf + Internal_ADC_R1);  dm(Right_Channel_In1) = r0;
        r0 = dm(rx0a_buf + AUX_ADC_R0);       dm(Right_Channel_SPDIF_rx) = r0;
```

Diese eingelesenen Samples können nun in den Audio-Verarbeitungsroutinen verwendet werden. Da es in diesem Projekt zwei verschiedene Programmteile gibt die Audio-Samples verarbeiten, musste an dieser Stelle eine entsprechende Verzweigung zur richtigen Routine eingebaut werden. Zu diesem Zweck wurde die Variable ‚switsch_noise_control„ definiert. Diese Variable ist anfänglich 0 und wird auf 1 gesetzt, wenn die Systemidentifikation abgeschlossen ist. Solange r0=r0-1 ungleich 0 ist, wird nach dem Einlesen der Samples in die Routine ‚input_output„ gesprungen. Nach dem Setzen auf 1 wird das Ergebnis 0 und es wird ab dann immer in die Routine ‚noise_control„ verzweigt, die den ANC-Algorithmus enthält.

```
do_audio_processing:
        r0=dm(switch_noise_control);
        r0=r0-1;
        if ne call (pc, input_output);
        if eq call (pc, noise_control);
```

Da die Audio verarbeitenden Programmteile mit dem Befehl ‚rts„ (return from subroutine) enden, wird nach deren Abarbeiten in ‚playback_AD1836_left_DACs„ zurückgesprungen. Hier werden die Samples in die Ausgabepuffer der D/A-Wandler des Audio-Codecs geschrieben.

```
        /* ---- DSP processing is finished, now playback results to AD1836 ---- */
playback_AD1836_left_DACs:  /* output processed left ch audio samples to AD1836 */
        r0 = dm(Left_Channel_Out0);          dm(tx2a_buf + Internal_DAC_L0) = r0;
        r1 = dm(Left_Channel_Out1);          dm(tx2a_buf + Internal_DAC_L1) = r1;
        r2 = dm(Left_Channel_Out2);          dm(tx2a_buf + Internal_DAC_L2) = r2;
        r3 = dm(Left_Channel_AD1852);        dm(tx2a_buf + AUX_DAC_L0) = r3;
```

```
playback_AD1836_right_DACs: /* output processed right ch audio samples to AD1836 */
        r0 = dm(Right_Channel_Out0);        dm(tx2a_buf + Internal_DAC_R0) = r0;
        r1 = dm(Right_Channel_Out1);        dm(tx2a_buf + Internal_DAC_R1) = r1;
        r2 = dm(Right_Channel_Out2);        dm(tx2a_buf + Internal_DAC_R2) = r2;
        r3 = dm(Right_Channel_AD1852);      dm(tx2a_buf + AUX_DAC_R0) = r3;
```

In den folgenden Zeilen wird ein Counter, der die Abtastschritte zählt, nach jeweils einem Abtastschritt um 1 erhöht. Dies hat keinen Einfluss auf den Programmablauf sondern dient nur eventuellen Debug-Zwecken. An dieser Stelle sei noch einmal erwähnt, dass hier im Zuge der Abtastratenanpassung die ursprüngliche Direktive ‚rti(db)„ durch ‚rts(lr)„ ersetzt wurde.

```
tx_done:
        r0=dm(audio_frame_timer);           /* get last count */
        r0=r0+1;                            /* increment count */
        dm(audio_frame_timer)=r0;           /* save updated count */
        rts(lr);
Process_AD1836_Audio_Samples.end:
```

3.9 Berechnung der Impulsantwort

3.9.1 Einleitung

Nach der Ausgabe der (verschobenen) Mess-Maximalfolge liegt nun die Systemantwort im Vektor ‚response[LM]„ (i8) vor. Die Impulsantworten können nun, wie bereits erläutert, durch Berechnung der Kreuzkorrelation von Maximalfolge und Systemantwort ermittelt werden. Es wird auch hier die zweikanalige Variante erläutert, um bei dieser Gelegenheit gleich die Speicherphilosophie der Samples im Impulsantwort-Vektor ‚ir[LM]„ (i9) zu erläutern. Die einkanaligen Versionen sind analog programmiert. Da alle beteiligten Variablen und Vektoren bereits beschrieben wurden, wird auf einen eigenen Abschnitt für den Deklarationsteil verzichtet.

3.9.2 Programmbeschreibung Impulsantwortberechnung

Dieser Programmteil beinhaltet keine Ein- und Ausgabeoperationen. Es werden daher vor der Berechnungsroutine die Interrupts der seriellen Ports SPORT0 und SPORT2 maskiert, d.h. deaktiviert. Damit wird vermieden, dass bei auftretendem Interrupt in eine Interrupt Service Routine gesprungen wird und eventuell gerade zur Berechnung verwendete Register eine Änderung erfahren. Es werden also im Interrupt Mask Register (IMASK) die entsprechenden Bits für die SPORTS auf 0 gesetzt und damit zukünftig auftretende Interrupts ignoriert.

```
calc_impulse_response:
bit clr imask SP0I|SP2I;
```

Wie bereits im Theorieteil ausführlich erläutert wurde, kann die Impulsantwort eines Systems mittels periodischer Kreuzkorrelation von Messfolge x[n] und Systemantwort y[n] berechnet werden. Dies entspricht einer Faltung der Signale wobei noch auf die Länge der Maximalfolge L normiert wird.

$$(x * y)[n] = \frac{1}{L} \sum x[k] * y[n-k]$$

Da wie bereits erwähnt die Division nicht möglich ist, wird nach dem Einlesen der MLS-Länge und dem Umwandeln in das Gleitkommaformat der Kehrwert mittels des ‚RECIPS„-Befehls gebildet.

```
r3=L;
f3=float r3;
f3=RECIPS f3;                                   // calculate 1/L for division/multiplication
dm(pointer_save)=i9;
```

Um nach der Berechnung der Impulsantworten den Pointer des Impulsantwortvektors wieder bequem auf das erste Element zurücksetzen zu können, wurde dieser in der Variable ‚pointer_save„ gespeichert. Die Routine besteht aus zwei ineinander verschachtelten Schleifen. In der inneren Schleife ‚sum„ finden die Multiplikation der Werte und deren Aufsummierung statt. Das konnte einfach mit MAC-Operationen realisiert werden. Zunächst werden jedoch in der äußeren Schleife ‚ir_loop„ vor jedem Summationsschritt die Register f9-f10, in denen die Summen gespeichert werden, auf 0 gesetzt.

```
LCNTR=L, DO ir_loop UNTIL LCE;
f8=0.0;
f9=0.0;
f10=0.0;
```

Vor dem Ausführen der MAC-Operationen müssen zuerst die Samples der MLS (i1) und der Systemantwort (i8) aus den Vektoren ausgelesen und in Register geschrieben werden. Da die Samples der verschiedenen Systemantworten in jedem Abtastschritt hintereinander in den Vektor geschrieben wurden, können diese nun auch unkompliziert der Reihe nach ohne Pointer-Operationen wieder ausgelesen werden.

Um den Vorteil von DSPs, die MAC-Operationen in einem Taktzyklus ausführen können, auch nutzen zu können, müssen bestimmte Register gewählt werden, d.h. es sind nicht alle Registerkombinationen möglich. Die folgende Tabelle 7 ist der Hilfe der VisualDSP-Software entnommen und zeigt die möglichen Kombinationen. Besonders bei komplexen Berechnungen mit mehreren beteiligten Operanden will die Auswahl der Register also gut überlegt sein.

Tabelle 7: Registerkombinationen für MAC-Operationen

Input	Valid Sources
Multiplier X	R3-R0 (F3-F0)
Multiplier Y	R7-R4 (f7-f4)
ALU X	R11-R8 (F11-F8)
ALU Y	R15-R12 (F15-F12)

```
LCNTR=L, DO ir_loop UNTIL LCE;
f8=0.0;
f9=0.0;
LCNTR=L, DO sum UNTIL LCE;          // calculate cross-correlation
f0=dm(i1,m1),f4=pm(i8,m8);          // read mls and response
f5=pm(i8,m8);
f12=f0*f4, f8=f8+f12;
sum: f13=f0*f5, f9=f9+f13;          // multiplicate and accumulate mls and response
f8=f8+f12;
f8=f8*f3;
f9=f9+f13;
f9=f9*f3;
```

Ist die Summen-Schleife abgearbeitet, bleibt noch, die letzten Additionen auszuführen und diese Summen jeweils mit dem vorher berechneten Kehrwert der MLS-Länge zu multiplizieren. Dann werden die Ergebnisse für die verschiedenen Kanäle wieder nacheinander in den Impulsantwortvektor (i9) geschrieben. Bevor nun der nächste Schleifendurchlauf startet, muss der Pointer der Systemantwort um 2 Stellen (m10) auf den nächsten Abtastwert des ersten Kanals der Systemantwort verschoben werden, da der Pointer nach ‚L„ Schleifendurchläufen wieder beim ersten Wert des gerade abgearbeiteten Abtastwerts steht.

```
pm(i9,m8)=f8;                       // write to ir (impulse response)
pm(i9,m8)=f9;                       // write to ir (impulse response)
ir_loop: f8=pm(i8,m10);             // shift response
```

Nun wird der Zeiger des Maximalfolgenvektors wieder auf den ersten Wert eingestellt und die Maskierung der SPORT-Interrupts aufgehoben.

```
i9=dm(pointer_save);
bit set imask SP0I|SP2I;
```

Zum Schluss wird die Variable ‚switch_noise_control‘ auf 1 gesetzt. Damit wird ab jetzt nach der Ein- und Ausgabe der Samples in die Routine ‚noise_control„ mit den ANC-Algorithmen gesprungen.

```
r0=1;
pm(switch_noise_control)=r0;
call (pc,Process_AD1836_Audio_Samples);
```

3.9.3 Speicherphilosophie der Samples im Impulsantwortvektor

Die Impulsantworten liegen jetzt im Impulsantwortvektor vor. Es folgt nun ein kleiner Exkurs darüber, wie genau die Impulsantworten der einzelnen Sekundärstrecken in diesem Vektor gespeichert sind. Das ist sehr wichtig für den späteren Umgang mit den ANC-Routinen, in denen die Impulsantworten benötigt werden. Schließlich müssen exakt die richtigen zur jeweiligen Sekundärstrecke gehörenden Samples verwendet werden. In den zweikanaligen Versionen, in denen mit vier Impulsantworten acht gefilterte Referenzsignale in der Feedback-Version berechnet werden, heißt es dann, den Überblick zu behalten.

Um die Erklärungen übersichtlicher und nachvollziehbarer zu gestalten, wurden reale Messungen am Board mit kurzgeschlossenen Ein- und Ausgängen durchgeführt. Die dabei resultierenden Impulsantworten kommen der Autokorrelationsfunktion der Maximalfolge, d.h. einem Dirac-Impuls, sehr nahe und sind deshalb für eine genaue Erläuterung der Speicherphilosophie gut geeignet. Alle Messungen wurden bei einer Abtastrate von 2 kHz durchgeführt. Es wurde jeweils eine Maximalfolge der Ordnung M=11 und damit einer Länge von L=2047 Samples verwendet. Da es in diesem Fall nicht auf genaue Impulsantworten ankam und auch keine Störeinflüsse heraus gerechnet werden mussten, wurde keine Mittelung der Systemantworten vorgenommen.

Die Abbildung 26 zeigt die Impulsantwort für die beiden einkanaligen Algorithmen. Ein idealer Dirac-Impuls hätte den Peak an der ersten Stelle (n=0) des Vektors. Bedingt durch Verzerrungen des Boards vor allem beim AD/DA-Wandeln ist das Resultat der realen Messung davon natürlich etwas abweichend und der Peak der Impulsantwort liegt an der Stelle n=4. Da in den einkanaligen Versionen noch nicht mit verschobenen Maximalfolgen gearbeitet werden musste, repräsentiert der komplette Impulsantwortvektor mit seiner Länge von 2047 Samples die Impulsantwort c_{11} der Sekundärstrecke zwischen Mikrofon und Lautsprecher (hier nur zwischen Eingang CH1/L und Ausgang CH/L). Bei der späteren Verwendung in den ANC-Algorithmen wird der Impulsantwortvektor durch die Definition der Anzahl der Impulsantwortwerte mittels der Variable ‚CC„ in der Header-Datei „mls.h“ begrenzt. Die genaue

Belegung der Kanäle kann in dem späteren Abschnitt zum Versuchsaufbau (4.1) nachgelesen werden.

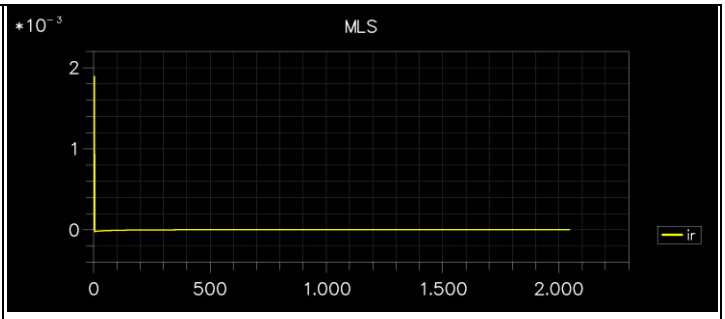
Ch1/L <- Ch1/L (ir_1_1, c_{11}) Mic 1 <- LS 1 Peak: 4 1. Wert: 0 Werte: 0, 1, 2, …	

Abbildung 26: Impulsantwort einkanalig

Die Beschreibung der einkanaligen Variante mit einer Impulsantwort wurde hier nur der Vollständigkeit halber erwähnt. Wichtig wird das Wissen um die genaue Lage der Impulsantworten im Vektor aber natürlich erst für die mehrkanaligen Algorithmen.

Da die Mess-Maximalfolge um die halbe MLS-Länge versetzt ausgegeben wird, um alle Impulsantworten in einem Messdurchgang ermitteln zu können, werden grundsätzlich auch die zu den jeweiligen Mikrofonen gehörenden Impulsantworten um diese Verschiebung versetzt im Impulsantwortvektor gespeichert. Es steht daher für die Impulsantworten im zweikanaligen Fall nur die halbe Länge der Mess-Maximalfolge zur Verfügung. Die Konvention für die Namensgebung der Impulsantworten wurde schon im Theorieteil erläutert. Hier nur noch einmal zur Wiederholung das Wichtigste. Der erste Index steht für das entsprechende Mikrofon, der zweite für den beteiligten Lautsprecher. C_{12} repräsentiert also beispielsweise die Impulsantwort der Sekundärstrecke zwischen Mikrofon 1 (Ch1/L, ‚Left_Channel_In0„) und Lautsprecher 2 (Ch1/R, ‚Right_Channel_Out0„).

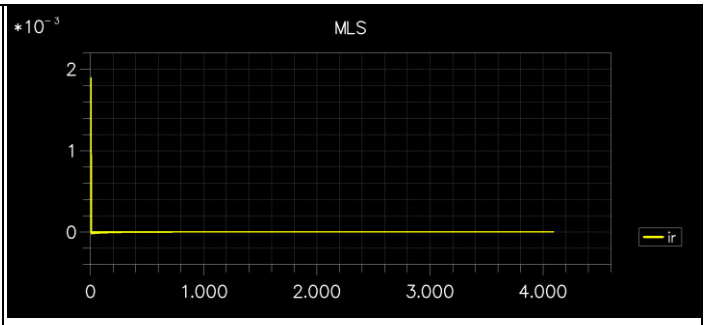
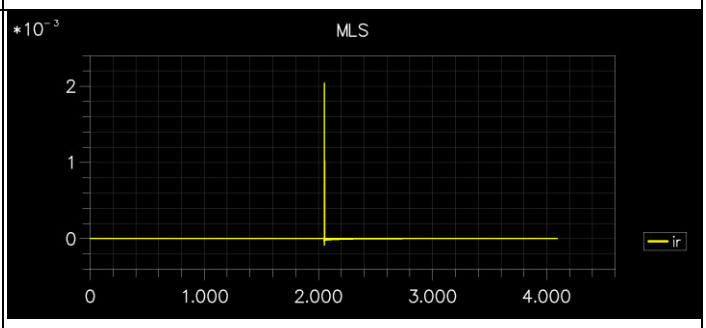
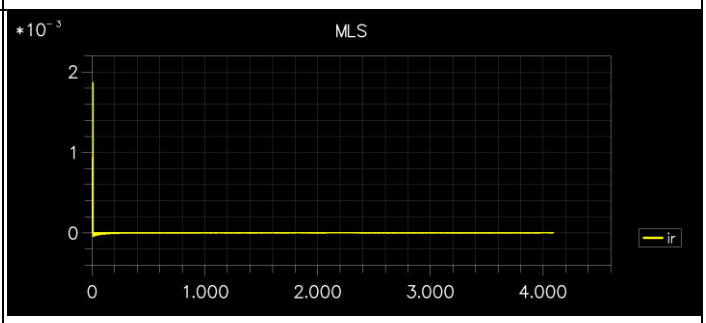
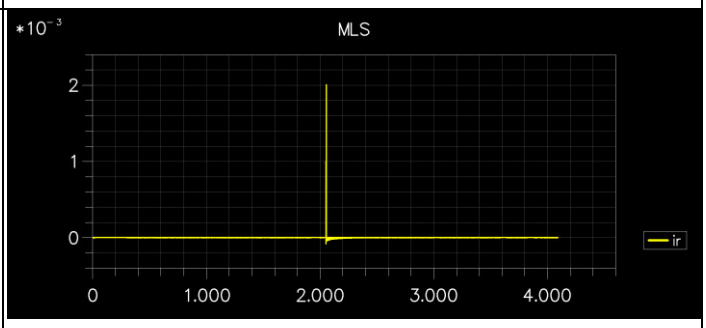
Ch1/L <- Ch1/L (ir_1_1, c_{11}) Mic 1 <- LS 1 Peak: 8 1. Wert: 0 Werte: 0, 2, 4, (Gerade)	
Ch1/L <- Ch1/R (ir_1_2, c_{12}) Mic 1 <- LS 2 Peak: 2054 1. Wert: 2* ir_shift=2046 Werte: 2046, 2048, 2050, … (Gerade)	
Ch1/R <- Ch1/L (ir_2_1, c_{21}) Mic 2 <- LS 1 Peak: 9 1. Wert: 1 Werte: 1, 3, 5, (Ungerade)	
Ch1/R <- Ch1/R (ir_2_2, c_{22}) Mic 2 <- LS 2 Peak: 2055 1. Wert: 2* ir_shift+1=2047 Werte: 2047, 2049, 2051, … (Ungerade)	

Abbildung 27: Impulsantworten zweikanalig

Die zu den jeweiligen Lautsprechern gehörenden Samples der Impulsantworten werden abwechselnd in den Vektor geschrieben. Es ist also jedes zweite Sample im Impulsantwortvektor einer bestimmten Impulsantwort zugeordnet. Im zweikanaligen Fall mit vier Impulsantworten beginnt c_{11} an der ersten Stelle des Vektors und die folgenden Samples befinden sich an den geraden Speicherstellen. C_{21} beginnt dementsprechend an der zweiten Stelle und belegt die ungeraden Speicherstellen. Damit ist auch klar, dass der Peak der Impulsantwort c_{11} nun an der Stelle 8 liegt. Analog dazu liegen die Impulsantworten c_{12} und c_{22} um die ursprüngliche MLS-Verschiebung verschoben im Impulsantwortvektor. Mit Hilfe der Abbildung 27 wird versucht, diesen etwas komplizierten Sachverhalt anschaulich darzustellen.

3.10 Programmierung der LMS-Algorithmen (filtered-x-LMS-Algorithmus)

3.10.1 Einleitung

Dieser Abschnitt beschreibt die programmierten filtered-x-LMS-Algorithmen. Die dafür benötigten Impulsantworten liegen bereits im Impulsantwortvektor vor. Es wurden zwei Varianten des filtered-x-LMS-Algorithmus, Feedforward und Feedback, realisiert. Diese werden in getrennten Abschnitten erläutert.

3.10.2 Programmbeschreibung Feedforward –Variablen- und Vektorendeklaration

In diesem Abschnitt werden die Variablen und Vektoren vorgestellt, die in den vorangehenden Programmteilen noch nicht benötigt wurden.

Es werden natürlich entsprechend den zum Feedforward-Algorithmus gehörenden Filterroutinen weitere Puffer zum Zwischenspeichern von Werten benötigt. In dem Vektor „w_filter[CWM]" werden die Adaptionsfilterkoeffizienten gespeichert.

```
.section /pm      pm_data;              //define your user program memory variables here
.VAR    w_filter[CWM];
...
```

Weiterhin sind Puffer für das Referenzsignal x („reference[CC]") und für die gefilterten Referenzsignale r („filtered_reference[CWMM]") nötig.

```
.section /dm      dm_data;              //define your user data memory variables here
.VAR    filtered_reference[CWMM];

.section /dm      seg_dmres;
.VAR    reference[CC];
...
```

In den ‚error„-Variablen werden die Fehlersignal-Samples der einzelnen Kanäle für die spätere Verwendung zwischengespeichert.

```
.section /pm        pm_data;                //define your user program memory variables here
.VAR    error_1;
.VAR    error_2;
```

Das Unterprogramm, in dem die LMS-Algorithmen berechnet werden, heißt ‚noise_control„ und muss natürlich auch im Deklarationsteil deklariert werden.

```
.section /pm pm_code;
.GLOBAL                 noise_control;
```

Die verwendeten Puffer und Vektoren sind wie folgt definiert.

```
user_code_init:
b2=reference; l2=@reference;
b3=filtered_reference; l3=@filtered_reference;
b8=response; l8=@response;
i9=ir;
```

Für den Puffer ‚w_filter„ mit den Koeffizientensätzen w wurden zwei Pointer definiert. Dies wird in der Routine zur Berechnung der Adaptionsfilterkoeffizenten zum Tragen kommen und an entsprechender Stelle genauer erläutert.

```
b10=w_filter;
b11=b10;
l10=@w_filter;
l11=l10;
```

3.10.3 Programmbeschreibung Feedforward – Einlesen der Samples

Bei diesem ANC-Projekt werden die Sound-Samples in Echtzeit verarbeitet, wobei diese Samples in jedem Abtastschritt vor der eigentlichen Berechnung mittels der ANC-Routinen vom AD1836 Audio Codec eingelesen und in Gleitkommawerte umgewandelt werden. Zuerst wird das Sample des Referenzsignals vom rechten Kanal des zweiten Eingangs in das Register f0 eingelesen. Da dieser Wert direkt im Anschluss an die Einleseprozedur verwendet wird, ist es nicht nötig, diesen in einer Variablen zwischen zu speichern. Danach werden die Fehlersignale der unterschiedlichen Kanäle eingelesen und konvertiert. Es ist hier stellvertretend für alle Kanäle nur der Quelltext für den ersten Kanal (links, erster Eingang) aufgeführt.

```
r1 = -31;                       // scale the input samples to the range of +/-1.0
r0 = DM(Right_Channel_In1);     // get right AD1836 ADC0 input sample (reference)
f0 = float r0 by r1;            // convert to floating point
r5=dm(Left_Channel_In0);        // read ch0 left input sample (error(e1))
f5 = float r5 by r1;            // convert to floating point
dm(error_1)=f5;                 // save error for later use in calc_w
```

3.10.4 Programmbeschreibung Feedforward – Berechnung des Ausgangssamples

Wie bereits im Theorieteil erläutert, setzt sich der Feedforward filtered-x-LMS-Algorithmus aus mehreren FIR-Filterroutinen zusammen. Praktisch werden diese Filter durch eine Ausführung der diskreten Faltung realisiert. Zuerst wird das Referenzsignal x mit den entsprechenden berechneten Adaptionsfilterkoeffizienten w gefiltert und so die Ausgangssignale y ermittelt. Die allgemeine Formel lautet:

$$\vec{y}_m[n] = (\vec{x}_k * \vec{w}_{mk})[n]$$

Die Anzahl der Sekundärquellen wird durch den Index m gekennzeichnet und k entspricht der Anzahl der Referenzsignale (Feedforward arbeitet nur mit einem Referenzsignal)

Der besseren Übersichtlichkeit halber ist hier die Faltungsfunktion zur Berechnung des ersten Ausgangssignales y_1 angegeben:

$$\vec{y}_1[n] = (\vec{x}_1 * \vec{w}_{11})[n] = \sum_{k=1}^{K} \vec{x}_1[k]\vec{w}_{11}[n-k]$$

Um den Zeiger auf das erste Element des Referenzsignalvektors (i2) nach der Filterung wieder zurücksetzen zu können, wird dieser zunächst in der Variable ‚pointer_save„ zwischengespeichert. Das ist nötig, da diese Vektoren zwar Ringpuffer sind, diese aber nicht zwangsläufig die gleiche Länge haben. Die Länge der Puffer wird in der Datei „mls.h“ mittels der Variablen ‚CC„ (Länge des Referenzsignal-Puffers) und ‚CW„ (Länge des Koeffizienten-Puffers) definiert.

```
calc_output_y_sample_1:
dm(pointer_save)=i2;
```

Im Register f0 befindet sich noch das Referenzsignal-Sample – dieses wird nun im Referenzsignal-Puffer (i2) gespeichert. Außerdem wird im selben Zyklus der erste Wert aus dem Anpassungsfilterkoeffizientenvektor w_{11} (i10) in das Register f4 geschrieben. Das kann hier im gleichen Zyklus geschehen, da Speicherzugriffe auf Datenspeicher und Programmspeicher gleichzeitig möglich sind. Dann werden noch vor der eigentlichen FIR-Filterschleife die ersten Multiplikationen berechnet und bereits neue Werte aus den Puffern ausgelesen. Damit ist es möglich, in der Filterschleife mit einem Berechnungsschritt für die MAC-Operation auszukommen. Nach Abarbeiten der Schleife muss darum auch noch die letzte MAC-Operation und die letzte Summation zur Berechnung des Ausgangssamples y abgearbeitet werden.

```
calc_output_y_sample_1:
dm(pointer_save)=i2;
dm(i2,m2)=f0, f4=pm(i10,m10);          // store x(n) in delay line (reference(x)), f4=w0(n)
f8=f0*f4, f0=dm(i2,m2), f4=pm(i10,m10);        // f8= x(n)*w0(n), f0= x(n-1), f4= w1(n)
f12=f0*f4, f0=dm(i2,m2), f4= pm(i10,m10);      // f12= x(n-1)*w1(n), f0= x(n-2), f4= w2(n)}
lcntr=CW-3, do calc_output_y_loop_1 until lce;
calc_output_y_loop_1: f12=f0*f4, f8=f8+f12, f0=dm(i2,m2), f4= pm(i10,m10);
                        // f12= x(n-i)*wi(n), f8= sum of prod, f0= x(n-i-1), f4= wi+1(n)}
f12=f0*f4, f8=f8+f12;                          // f12= x(n-N+1)*wN-1(n)
f12=f8+f12;
```

Die Schrittweite ‚m2„ beim Auslesen der Referenz-Samples beträgt -1, da es unabhängig von der Anzahl der Kanäle des Feedback-Algorithmus nur ein Referenzsignal gibt. Anders verhält es sich bei dem Koeffizienten-Vektor. In diesem Vektor werden in der zweikanaligen Version auch zwei Koeffizientensätze (w_{11}, w_{21}) gespeichert. Da die entsprechenden Samples im Wechsel im Vektor gespeichert sind, muss in der Filterroutine nur jedes zweite Sample aus dem Vektor ausgelesen werden (analog zur bereits erläuterten Speichertaktik der Impulsantworten). Für die Schrittweite m10 muss hier also 2 verwendet werden – in der einkanaligen Version beträgt die Schrittweite m10=1.

Nun wird noch der Zeiger des Referenzsignal-Vektors wieder auf den ersten Wert gesetzt.

```
i2=dm(pointer_save);                   // move pointer of reference(x) back to base-address
```

Mit der Variable VO, die in der Headerdatei „mls.h“ definiert wurde, wird nun der Ausgangswert y multipliziert, um eventuell eine Pegelanpassung zu ermöglichen. Anschließend muss dieser Wert wieder in das für den DA-Wandler benötigte Festkomma-Format umgewandelt werden um dann in den Ausgabe-Puffer des linken Kanals des ersten Ausgangs geschrieben zu werden.

```
output_sample_1:
f0=VO;
f12=f12*f0;
        r1 = 31;                       /* scale the result back up to MSBs */
        r8 = fix f12 by r1;            /* convert back to fixed point */
        DM(Left_Channel_Out0) = r8;    /* send left channel result to AD1836 Right DAC0 */
```

Abschließend wird der Zeiger des Koeffizienten-Vektors eine Stelle weiter gesetzt, damit in der folgenden Berechnung des zweiten Ausgangssamples (rechter Kanal des ersten Ausgangs) mit dem entsprechenden zweiten Koeffizientensatz w_{21} gefiltert wird.

```
f0=pm(i10,1);
```

Diese kleine Filterroutine wird nun entsprechend der Anzahl der Kanäle des Algorithmus für bis zu zwei Ausgangssamples ausgeführt. Da das analog zur gerade erläuterten Routine geschieht, wird darauf an dieser Stelle nicht weiter eingegangen.

Der Zeiger des Koeffizientenvektors muss zum Abschluss wieder auf die erste Stelle gesetzt werden. Im zweikanaligen Fall muss dieser Zeiger also um eine Stelle zurück gesetzt werden.

```
f0=pm(i10,-1);
```

3.10.5 Programmbeschreibung Feedforward – Berechnung der gefilterten Impulsantworten

Zur Berechnung der mit den entsprechenden Impulsantworten gefilterten Referenzsignale wurden nun ebenfalls FIR-Filterroutinen programmiert. Diesmal wird das Referenzsignal x (i2) mit einer Impulsantwort gefiltert. Die allgemeine Formel lautet:

$$\vec{r}_{lmk}[n] = (\vec{x}_k * \vec{c}_{lm})[n]$$

(l – Fehlersensoren; m – Sekundärquellen; k – Referenzsignale)

Die Faltungsformel zur Berechnung des ersten gefilterten Referenzsignales r_{111} lautet also:

$$\vec{r}_{111}[n] = (\vec{x}_1 * \vec{c}_{11})[n] = \sum_{k=1}^{K} \vec{x}_1[k]\vec{c}_{11}[n-k]$$

Es wird nun wieder die bekannte Filterroutine abgearbeitet. Die Schrittweite beim Auslesen des Impulsantwortvektors (i9) ist wieder 2, da wie bereits erläutert im zweikanaligen Fall die Samples von zwei Impulsantworten (c_{11}, c_{21}, und c_{12}, c_{22}) abwechselnd im Vektor gespeichert sind. Beispielhaft hier der Quellcode zur Berechnung des gefilterten Referenzsignals r_{111} durch Filterung des Referenzsignals x mit der Impulsantwort c_{11}.

```
calc_filtered_reference_1_1_1:          // r(n)=sum(c(n)*x(n)), (c(n)=ir_1_1(n))
dm(pointer_ir)=i9;                      // save base adress of ir (first sample of ir_1_1)
f0=dm(i2,m2), f4=pm(i9,m10);                    // store x(n) in delay line, f4=ir0(n)
f8=f0*f4, f0=dm(i2,m2), f4=pm(i9,m10);          // f8= x(n)*ir0(n), f0= x(n-1), f4= ir1(n)
f12=f0*f4, f0=dm(i2,m2), f4= pm(i9,m10);        // f12= x(n-1)*ir1(n), f0= x(n-2), f4= ir2(n)}
lcntr=CC-3, do c_filter_loop_1_1_1 until lce;
c_filter_loop_1_1_1: f12=f0*f4, f8=f8+f12, f0=dm(i2,m2), f4= pm(i9,m10);
                                // f12= x(n-i)*iri(n), f8= sum of prod, f0= x(n-i-1), f4= iri+1(n)}
f12=f0*f4, f8=f8+f12;                           // f12= x(n-N+1)*irN-1(n)
f12=f8+f12;
dm(i3,1)=f12;                                   // write to filtered_reference(r)
```

Nach Speicherung des berechneten Samples im Vektor mit den gefilterten Referenzsignalen (i3) muss nun der Zeiger des Impulsantwortvektors an die erste Stelle zurückgesetzt werden. Dazu wurde diese Speicherstelle bereits im Vorfeld in der Variable ‚pointer_ir„ gespeichert. Dann wird der Zeiger auf das erste Sample der nächsten zu verwendenden Impulsantwort gesetzt. In diesem Fall ist das c_{21} und der Zähler muss nur um eine Stelle auf den zweiten Wert im Vektor verschoben werden.

```
i9=dm(pointer_ir);          // move pointer of ir back to base-address
f0=pm(i9,1);                // i9 -> ir move pointer to next sample (first sample of ir_2_1)
```

Nun kann in der folgenden Routine mit c_{21} ein Sample des gefilterten Referenzsignals r_{211} berechnet werden. Nachdem dieser Wert ermittelt wurde, muss zum ersten Sample der Impulsantwort c_{12} gesprungen werden, um ein Sample von r_{121} zu berechnen. Dieser Wert ist an der Stelle 2*m15 gespeichert. In dem Modifizierungsregister m15 wurde bereits der Wert der MLS-Verschiebung, die bei der Ermittlung der Impulsantworten benutzt wurde, gespeichert. Da im zweikanaligen Fall die Samples von zwei Impulsantworten hintereinander im Vektor vorliegen, muss diese Verschiebung natürlich verdoppelt werden.

```
i9=dm(pointer_ir);      // move pointer of ir back to base-address
modify(i9,m15);
modify(i9,m15);         // i9 -> ir move pointer to 2*ir_shift sample (first sample of ir_1_2)
```

Insgesamt wurden nun in der zweikanaligen Version vier gefilterte Referenzsignalsamples berechnet. Bevor im letzen Schritt die Adaptionsfilterkoeffizienten berechnet werden können, müssen die Pointer der Vektoren noch an die richtigen Stellen gesetzt werden. Da die folgende Routine zuerst mit dem ersten Sample des gefilterten Referenzsignals r_{211} rechnet, wird der Zeiger gleich beim Speichern des letzten berechneten Samples von r_{221} an diese Stelle, also um zwei Stellen zurück, gesetzt. Anschließend wird noch der Zeiger des Impulsantwortvektors für die Berechnungen im nächsten Abtastschritt wieder an die erste Stelle gesetzt.

```
dm(i3,m4)=f12;          // write to filtered_reference(r), shift pointer to first sample of r_2_1_1
                        (m4=-2)
i9=dm(pointer_ir);      // move pointer of ir back to base-address
```

3.10.6 Programmbeschreibung Feedforward – Adaptionsfilterkoeffizienten

Bei dem Algorithmus zur Berechnung der Adaptionsfilterkoeffizienten handelt es sich nicht um eine einfache FIR-Filterroutine. In den mehrkanaligen Versionen ist dieser Algorithmus etwas unübersichtlich. Deshalb wird an dieser Stelle zum einfacheren Verständnis zuerst die einkanalige Variante erläutert. Es sollen aus den aktuellen Adaptionskoeffizienten w[n] die Koeffizienten w[n+1] für die Filterung im

nächsten Abtastschritt berechnet werden. Die zugrunde liegende allgemeine Formel lautet:

$$\vec{w}_{mk}[n+1] = \vec{w}_{mk}[n] - \alpha \sum_{l=1}^{L} \vec{r}_{lmk}[n] e_l[n]$$

(l – Fehlersensoren; m – Sekundärquellen; k – Referenzsignale)

Aus programmiertechnischen Gründen wird nun zuerst der Subtrahend berechnet. Dazu werden die Werte von α, e[n] und r[n] in Register eingelesen und diese anschließend multipliziert.

```
calc_w:                     // update weights (w): w(n+1)=w(n)-alpha*r(n)*e(n)
f2=A;                       // read alpha (STEPSIZE)
f5=dm(error);               // read error(e)
f1=f2*f5, f4=dm(i3,m2);     //f1= STEPSIZE*e(n), f4= r(n)
f0=f1*f4, f12=pm(i10,m8);   //f0= STEPSIZE*e(n)*r(n), f12= w0(n)
```

Im Register f1 ist das für diesen Abtastschritt konstante Produkt aus α und e[n] gespeichert. In der Schleife ‚update_weights„ muss darum nur der entsprechende Wert des gefilterten Referenzsignales r[n] in das Register f4 eingelesen, mit f1multipliziert und vom entsprechenden Wert des aktuellen Koeffizientenvektors w[n] subtrahiert werden, der ebenfalls in jedem Schleifendurchlauf in das Register f12 eingelesen wird.

```
lcntr=CW-1, do update_weights until lce;
f8=f12-f0, f4=dm(i3,m2), f12=pm(i10,m8);      //f8= wi(n+1), f4= r(n-i-1), f12= wi+1(n)
update_weights:  f0=f1*f4, pm(i11,m8)=f8;     //f0= STEPSIZE*e(n)*r(n-i-1), store wi(n+1)
```

Die Schleife wurde nur (CW-1)-mal durchlaufen, da ein Koeffizient außerhalb der Schleife berechnet wurde. Es bleibt nun noch, die letzte Subtraktion zu vollziehen, um diesen letzten Koeffizienten zu berechnen und im Koeffizientenvektor (i10, i11) zu speichern. Gleichzeitig wurde der Zeiger auf den Puffer mit den gefilterten Referenzsignalsamples (i3) für die Berechnungen im nächsten Abtastschritt eine Stelle weiter gesetzt.

```
f8=f12-f0, f0=dm(i3,1);        //f8= wN-1(n+1), i3 -> r(n+1) location in delay line
pm(i11,m8)=f8;                 //store wN-1(n+1)
```

Wie bereits im Abschnitt mit den Variablendeklarationen erwähnt, wurden zwei Pointer auf den Koeffizientenvektor definiert. Das ist nötig, da in der Schleife ein „alter“ Koeffizient ausgelesen wird und an dieser Stelle im nächsten Programmzyklus gleich wieder der neu berechnete Wert gespeichert wird.

In der zweikanaligen Variante wird die Routine zur Berechnung der Adaptionskoeffizienten komplexer, da mit zwei Fehlersignalsamples und zwei gefilterten Referenz-

signalen gerechnet werden muss. Außerdem muss diese Routine zweimal zur Berechnung der zwei Koeffizientenvektoren w_{11} und w_{21} ausgeführt werden. Die ausführliche Formel für die Berechnung des ersten Koeffizientensatzes w_{11} lautet:

$$\vec{w}_{11}[n+1] = \vec{w}_{11}[n] - \alpha(\vec{r}_{311}[n]e_1[n] + \vec{r}_{211}[n]e_2[n] + \vec{r}_{111}[n]e_3[n])$$

Zuerst wird wieder die Sprungweite α gespeichert, diesmal im Register f1. Dann werden die Fehlersamples eingelesen und mit α multipliziert. Im gleichen Zyklus wird jeweils auch ein Wert von r[n] eingelesen. Konkret sind das für die Berechnung von w_{11} die Werte von r_{211} und r_{111}. Diese werden in umgekehrter Reihenfolge ausgelesen, da ansonsten ein zusätzlicher Zyklus für das Umstellen des Zeigers des r-Puffers nötig wäre. Die Einzelheiten dieser Routine sollten sich mit Hilfe der Kommentare im Quelltext erschließen lassen.

```
calc_w_1_1:                 // update weights(w): w(n+1)=w(n)-alpha*r(n)*e(n)
f1=A;                       // read alpha (STEPSIZE)
f2=dm(error_1);             // read error_1(e1)
f2=f1*f2, f5=dm(i3,m2);     // f2= STEPSIZE*e1(n), f5= r_2_1_1(n)
f3=dm(error_2);             // read error_2(e2)
f3=f1*f3, f4=dm(i3,m6);
// f3= STEPSIZE*e2(n), f4= r_1_1_1(n), m6=-3 (move pointer to r_2_1_1(n-1))
f0=f2*f4, f11=pm(i10,m10);  // f0= STEPSIZE*e1(n)*r_1_1_1(n), f12= w0(n)
f1=f3*f5;                   // f1= STEPSIZE*e2(n)*r_2_1_1(n)
lcntr=CW-1, do update_weights_1_1 until lce;
f12=f0+f1, f5=dm(i3,m2);
// f0= (STEPSIZE*e1(n)*r_1_1_1(n))+(STEPSIZE*e2(n)*r2_1_1(n))
f1=f3*f5, f8=f11-f12, f4=dm(i3,m6), f11=pm(i10,m10);
update_weights_1_1:  f0=f2*f4, pm(i11,m10)=f8; // store wi(n+1)
f12=f0+f1;
f8=f11-f12, f0=dm(i3,m3);   // f8= wN-1(n+1), i3 -> r_2_2_1(n) location in delay line
```

Nachdem die Programmschleife abgearbeitet wurde, bleibt nun noch, die Pointer auf den Koffizientenvektor (i10, i11) auf das erste Sample des Koeffizientensatzes w_{21} zu bringen.

```
pm(i11,m8)=f8;              // store wN-1(n+1)
modify(i10,1);              // move pointer to first sample of w_2_1
```

Nachdem auch der zweite Koeffizientensatz w_{21} berechnet wurde, müssen die Zeiger wieder auf den ersten Koeffizienten von w_{11} gesetzt werden.

```
pm(i11,m8)=f8;              // store wN-1(n+1)
modify(i10,-1);             // move pointer back to first sample of w_1_1
```

Nun wird mit dem Befehl ‚rts„ wieder in die Routine ‚Process_AD1836_Audio_Samples„ in der Datei „SPT0_ISR_1836_Processing.asm“

gesprungen und im nächsten Abtastschritt neue Samples vom AD1836 Audio Codec eingelesen.

```
rts;
noise_control.end:
```

Danach beginnt die Berechnung des Feedforward-Algorithmus„ in ‚noise_control„ für die aktuellen Abtastwerte erneut.

3.10.7 Programmbeschreibung Feedback – Variablen- und Vektorendeklaration

Der Feedback-Algorithmus arbeitet nicht mit einem einzulesenden Referenzsignal, sondern es werden für jeden Übertragungsweg Referenzsignale geschätzt. Darum werden im Vergleich zu Feedforward einige weitere Variablen und Vektoren benötigt, die an dieser Stelle kurz vorgestellt werden. Für die zweikanalige Version werden jeweils zwei Variablen benötigt, in denen die Samples für die geschätzten Referenzsignale und die berechneten Ausgangssamples zwischengespeichert werden.

```
.section /pm      pm_data;      //define your user program memory variables here
.VAR    reference_sample_1;
.VAR    reference_sample_2;
.VAR    y_sample_1;
.VAR    y_sample_2;
```

Für die zusätzliche Filterroutine zur Schätzung der Referenzsignale wird ein weiterer Puffer für die Ausgangssamples y benötigt.

```
.section /dm      dm_data;      //define your user data memory variables here
.VAR    y_buffer[CCM];
```

Außerdem ist zu beachten, dass nun bis zu zwei Referenzsignale und bis zu 8 gefilterte Referenzsignale gespeichert werden müssen. Dazu wurden in der Header-Datei „mls.h“ zusätzlich die Variablen ‚CCM„ und ‚CWMMM„ definiert, die die Länge der Puffer bestimmen.

```
.section /dm      seg_dmres;
.VAR    reference[CCM];
.VAR    filtered_reference[CWMMM];
```

3.10.8 Programmbeschreibung Feedback – Berechnung der Ausgangssamples

Nachdem die Samples der Fehlersignale eingelesen wurden, werden nun entsprechend dem Feedback-Algorithmus die Ausgangssamples y berechnet. Einlesen wurde nicht näher erläutert, da dies nahezu analog zum Feedforward-Algorithmus funktioniert. Der einzige Unterschied besteht darin, dass kein Referenzsignal eingelesen werden muss – diese werden später in einer Routine berechnet. Die allgemeine Formel für die Berechnung der Ausgangssignale y lautet:

$$\vec{y}_m[n] = \sum_{k=1}^{K} (\vec{x}_k * \vec{w}_{mk})[n]$$

(m – Sekundärquellen; k=l – Referenzsignale)

Das Sample des ersten Ausgangssignals y_1 in der zweikanaligen Variante berechnet sich demnach wie folgt:

$$\vec{y}_1[n] = \sum_{k=1}^{K} \vec{x}_1[k]\vec{w}_{11}[n-k] + \sum_{k=1}^{K} \vec{x}_2[k]\vec{w}_{12}[n-k] + \sum_{k=1}^{K} \vec{x}_3[k]\vec{w}_{13}[n-k]$$

Die einzelnen Terme werden nun nacheinander mit einfachen FIR-Filterroutinen berechnet. Nachfolgend die Berechnung des ersten Terms mittels Filterung des geschätzten Referenzsignales x_1 mit den Adaptionsfilterkoeffizienten w_{11}. Das im letzten Abtastschritt berechnete und in der Variable ‚reference_sample_1„ vorliegende Referenzsample x_1 wird vorher in das Register f0 eingelesen und im Referenzsignalvektor gespeichert.

```
calc_output_y_sample_1:
dm(pointer_save)=i2;
f0=dm(reference_sample_1);
dm(i2,m4)=f0, f4=pm(i10,m11);          // store x(n) in delay line (reference(x)), f4=w0(n)
f8=f0*f4, f0=dm(i2,m4), f4=pm(i10,m11);        // f8= x(n)*w0(n), f0= x(n-1), f4= w1(n)
f12=f0*f4, f0=dm(i2,m4), f4= pm(i10,m11);      // f12= x(n-1)*w1(n), f0= x(n-2), f4= w2(n)}
lcntr=CW-3, do calc_output_y_loop_1_1 until lce;
calc_output_y_loop_1_1: f12=f0*f4, f8=f8+f12, f0=dm(i2,m4), f4= pm(i10,m11);
                                  // f12= x(n-i)*wi(n), f8= sum of prod, f0= x(n-i-1), f4= wi+1(n)}
f12=f0*f4, f8=f8+f12;                          // f12= x(n-N+1)*wN-1(n)
f13=f8+f12;                                    // f12= y(n)
```

Im Register f13 ist jetzt das Ergebnis für die spätere Addition mit den anderen Termen gespeichert. Bevor nun der nächste Term berechnet werden kann, muss noch der Zeiger des Referenzsignalvektors auf die Speicherstelle des ersten Samples des Referenzsignals x_2 und der des Koeffizientenvektors auf den ersten Wert von w_{12} gesetzt

werden. Dies wird natürlich analog dazu auch nach den anderen Berechnungsroutinen getan.

```
i2=dm(pointer_save);        // move pointer of reference(x) back to base-address
f0=dm(i2,1);                // move pointer of reference(x) to first sample of x2
f0=pm(i10,1);               // move pointer to first sample of w_1_2
```

Nachdem dann beide Terme berechnet und in den Registern f13 und f12 vorliegen, werden diese addiert. Zur späteren Verwendung bei der Schätzung der Referenzsignale werden diese Ausgangssamples y außerdem in der Variable ‚y_sample_1„ usw. zwischengespeichert.

```
f12=f12+f13;
dm(y_sample_1)=f12;
```

Es bleibt nun noch, das berechnete Ausgangssample y in dem Ausgabepuffer des Audio-Codecs zu speichern. Vorher kann dieser Wert noch mittels der in der Header-Datei „mls.h“ definierten Variable ‚VO„ skaliert werden. Außerdem wird dieser Wert in das für den D/A-Wandler nötige Festkommaformat konvertiert.

```
output_sample_1:
f0=VO;
f12=f12*f0;
        r1 = 31;                    /* scale the result back up to MSBs */
        r8 = fix f12 by r1;         /* convert back to fixed point */
        DM(Left_Channel_Out0) = r8; /* send left channel result to AD1836 Right DAC0 */
```

Nach der Berechnung des zweiten Ausgangssamples y_2 müssen abschließend die Pointer wieder auf die ersten Werte der Vektoren für die nächsten Berechnungsschritte gesetzt werden.

```
i2=dm(pointer_save);        // move pointer of reference(x) back to base-address
f0=dm(i2,-1);               // move pointer of reference(x) to first sample of x1
f0=pm(i10,-3);              // move pointer to first sample of w_1_1
```

3.10.9 Programmbeschreibung Feedback – Schätzung der Referenzsignale

Die Ausgangswerte y_1, y_2 und y_3 liegen nun vor und werden in diesem Programmteil zur Schätzung der Referenzsignale benötigt. Die zugrunde liegenden Formeln lauten:

$$\vec{x}_k[n] = e_l + \sum_{m=1}^{M} (\vec{y}_m * \vec{c}_{lm})[n]$$

$$\vec{x}_1[n] = e_1 + (\vec{y}_1 * \vec{c}_{11})[n] + (\vec{y}_2 * \vec{c}_{12})[n] + (\vec{y}_3 * \vec{c}_{13})[n]$$

$$\vec{x}_1[n] = e_1 + \sum_{k=1}^{K} \vec{y}_1[k]\vec{c}_{11}[n-k] + \sum_{k=1}^{K} \vec{y}_2[k]\vec{c}_{12}[n-k] + \sum_{k=1}^{K} \vec{y}_3[k]\vec{c}_{13}[n-k]$$

Wie schon in den Routinen zur Berechnung der Ausgangswerte werden zuerst die einzelnen Terme berechnet, diese addiert und anschließend noch der entsprechende Fehlersignalwert hinzuaddiert.

Nachdem das zuvor berechnete und in der Variable ‚y_sample_1" gespeicherte Ausgangssample in das Register f0 und danach in den Ausgangssignalpuffer eingelesen wurde, läuft wieder die bekannte FIR-Filterroutine ab. Entsprechend der Variablen ‚CC„ aus „mls.h" wird nun mit der entsprechenden Anzahl von Werten der passenden Impulsantwort gefiltert.

```
calc_reference_1:
f0=dm(y_sample_1);
dm(i4,m4)=f0, f4=pm(i9,m10);                    // store y1(n) in delay line, f4=ir0(n)
f8=f0*f4, f0=dm(i4,m4), f4=pm(i9,m10);          // f8= y(n)*ir0(n), f0= y(n-1), f4= ir1(n)
f12=f0*f4, f0=dm(i4,m4), f4= pm(i9,m10);        // f12= y(n-1)*ir1(n), f0= y(n-2), f4= ir2(n)}
lcntr=CC-3, do x_filter_loop_1_1 until lce;
x_filter_loop_1_1: f12=f0*f4, f8=f8+f12, f0=dm(i4,m4), f4= pm(i9,m10);
                   // f12= y(n-i)*iri(n), f8= sum of prod, f0= y(n-i-1), f4= iri+1(n)}
f12=f0*f4, f8=f8+f12;                           // f12= y(n-N+1)*irN-1(n)
f13=f8+f12;                                     // f13=sum(c(n)*y(n))
```

Wie immer in diesen Routinen ist es wichtig, die Pointer für die nächsten Berechnungen an die richtige Position zu setzen. In diesem Fall wird der Zeiger des Inpulsantwortvektors auf die Speicherstelle des ersten Samples von c_{12} gestellt. Der Zeiger des Ausgangspuffers muss eine Stelle weiter gestellt werden, um dort später den berechneten Wert y_2 ablegen zu können.

```
i9=dm(pointer_ir);      // move pointer of ir back to base-address
modify(i9,m15);
modify(i9,m15);         // i9 -> ir move pointer to 2*ir_shift sample (first sample of ir_1_2)
f0=dm(i4,1);            // i4 -> y move pointer to first sample of y2
```

Die zwei Terme werden nun addiert. Anschließend wird das passende Fehlersignal e_1 in das Register f5 eingelesen und addiert.

```
f12=f12+f13;                        // add results of filters
f5=dm(error_1);                     // read error(e)
f12=f12+f5;                         // add error(e)
dm(reference_sample_1)=f12;         // write to reference_1(x)
```

Das Referenzsignalsample wird dann in der Variable ‚reference_sample_1„ gespeichert und die Pointer auf die richtigen Stellen für die Routine zur Berechnung des nächsten (hier zweiten) Referenzsignalsamples gesetzt.

```
i9=dm(pointer_ir);          // move pointer of ir back to base-address
f0=pm(i9,1);                // i9 -> ir move pointer to next sample (first sample of ir_2_1)
f0=dm(i4,-1);               // i4 -> y move pointer to first sample of y1
```

Analoge Routinen werden nun für jedes Referenzsignalsample durchlaufen und es können die gefilterten Referenzsignale berechnet werden.

1.1.1 Programmbeschreibung Feedback – Berechnung der gefilterten Referenzsignale und Adaptionsfilterkoeffizienten

Die gefilterten Referenzsignale r werden prinzipiell wie auch im Feedforward-Algorithmus berechnet. Allerdings sind hier bis zu acht Referenzsignale zu filtern. Auf die Darstellung der Quellcodes und der Formeln kann an dieser Stelle verzichtet werden, da sich die Filterroutinen nur durch die Modifizierungsregister und damit durch die Sprungweiten beim Auslesen der Samples von denen des Feedforward-Algorithmus,, unterscheiden. Wichtig ist hier wieder, dass die Zeiger auf die Vektoren nach jeder Filterung auf die richtigen Speicherstellen für die folgende Filterung gesetzt werden. Anhand der Kommentierung sollte das gut nachvollziehbar sein.

```
calc_filtered_reference_1_1_1:                  // r(n)=sum(c(n)*x(n))
f0=dm(i2,m4), f4=pm(i9,m10);                    // store x(n) in delay line, f4=ir0(n)
f8=f0*f4, f0=dm(i2,m4), f4=pm(i9,m10);          // f8= x(n)*ir0(n), f0= x(n-1), f4= ir1(n)
f12=f0*f4, f0=dm(i2,m4), f4= pm(i9,m10);        // f12= x(n-1)*ir1(n), f0= x(n-2), f4= ir2(n)}
lcntr=CC-3, do c_filter_loop_1_1_1 until lce;
c_filter_loop_1_1_1: f12=f0*f4, f8=f8+f12, f0=dm(i2,m4), f4= pm(i9,m10);
                        // f12= x(n-i)*iri(n), f8= sum of prod, f0= x(n-i-1), f4= iri+1(n)}
f12=f0*f4, f8=f8+f12;   // f12= x(n-N+1)*irN-1(n)
f12=f8+f12;
dm(i3,1)=f12;           // write to filtered_reference(r)
i9=dm(pointer_ir);      // move pointer of ir back to base-address (first sample of ir_1_1)
modify(i9,1);           // i9 -> ir move pointer to first sample of ir_2_1
```

Die Berechnung der Adaptionsfilterkoeffizienten w erfolgt nun ebenfalls analog zur Berechnung im Feedforward-Algorithmus. Auch hier müssen jedoch mehr Koeffizientensätze berechnet werden.

```
calc_w_1_1:                     // update weights(w): w(n+1)=w(n)-alpha*r(n)*e(n)
f1=A;                           // read alpha (STEPSIZE)
f2=dm(error_1);                 // read error_1(e1)
f2=f1*f2, f5=dm(i3,m2);         // f2= STEPSIZE*e1(n), f5= r_2_1_1(n)
f3=dm(error_2);                 // read error_2(e2)
f3=f1*f3, f4=dm(i3,m6);         // f3= STEPSIZE*e2(n), f4= r_1_1_1(n), m6=-7 (move
                                pointer to r_2_1_1(n-1))
f0=f2*f4, f11=pm(i10,m11);      // f0= STEPSIZE*e1(n)*r1_1_1(n), f12= w0(n)
f1=f3*f5;                       // f1= STEPSIZE*e2(n)*r2_1_1(n)
lcntr=CW-1, do update_weights_1_1 until lce;
f12=f0-f1, f5=dm(i3,m2);
// f0= (STEPSIZE*e1(n)*r1_1_1(n))+(STEPSIZE*e2(n)*r2_1_1(n))
f1=f3*f5, f8=f11+f12, f4=dm(i3,m6), f11=pm(i10,m11);
update_weights_1_1:  f0=f2*f4, pm(i11,m11)=f8;          // store wi(n+1)
f12=f0-f1;
f8=f11+f12, f0=dm(i3,2);        // f8= wN-1(n+1), i3 -> r_2_1_2(n) location in delay line
pm(i11,5)=f8;                   // store wN-1(n+1)
modify(i10,1);
```

3.11 Bedienung des ANC-Systems

3.11.1 Ausführliche Schritt-für-Schritt-Anleitung

Die Installation und Einrichtung der Visual-DSP-Oberfläche wurde bereits in den Abschnitten 2.6.4 und 2.6.5 beschrieben und kann dort nachgelesen werden. Hier soll nun eine Schritt-für-Schritt-Anleitung mit Erläuterungen gegeben werden, wie das ANC-System bedient wird. Abbildung 28 zeigt einen Screenshot, um die wichtigsten Buttons und Menüs zu lokalisieren.

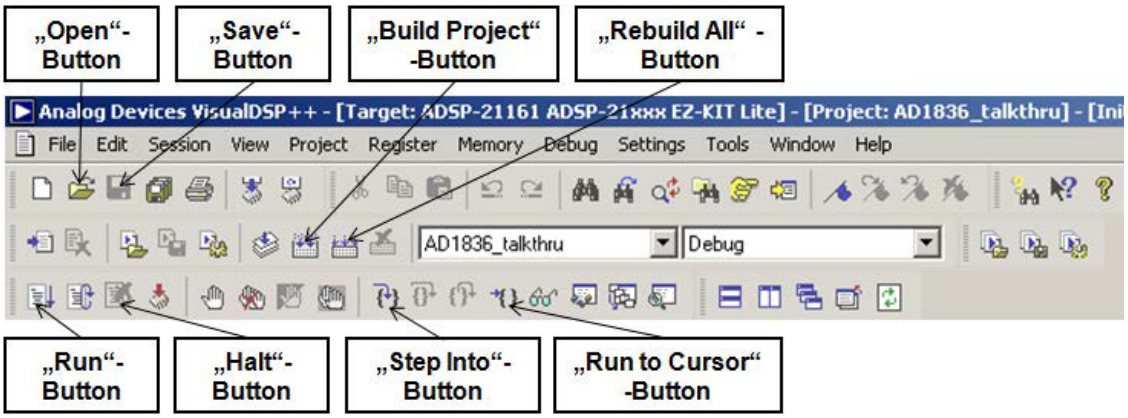


Abbildung 28: Positionen der Menüs und Buttons

- Visual-DSP-Umgebung starten
- Korrekte Session für ADSP-21161 auswählen oder neu kreieren (Menü „Session“), siehe Abbildung 16
  - Simulator (ohne Board)
    - Debug target: ADSP-2116x Family Simulator
    - Platform: ändert sich entsprechend Debug target
    - Processor: ADSP-21161
    - Session name: frei wählbar, kann so belassen werden (ADSP-21161 ADSP-2116x Simulator)
  - EZ-KIT Lite (mit Board)
    - Debug Target: EZ-KIT Lite(ADSP-21xxx)
    - Platform: ändert sich entsprechend Debug target
    - Processor: ADSP-21161
    - Session name: frei wählbar, kann so belassen werden (ADSP-21161 ADSP-21xxx EZ-KIT Lite)

- Projekt öffnen
  - Menü „File-Open-Project“ oder „Open“-Button
    - Die Projekte heißen für alle Varianten „noise_control.dpj“, um trotzdem erkennen zu können, welcher Algorithmus gerade geöffnet ist, findet sich im linken Projektfenster eine Textdatei mit entsprechendem Namen (z.B. „ff_1_ch.txt“ für die einkanalige Feedforward-Variante)
    - Die verschiedenen Varianten sind in den entsprechenden Unterverzeichnissen von „..\lms_final\“ gespeichert und haben selbsterklärende Namen (z.B. „..\lms_final\feed_forward_1_ch\ noise_control.dpj“ für die einkanalige Feedforward-Variante)
- Parameter wie Abtastrate, Koeffizientenanzahl oder α konfigurieren
  - Doppelklick auf die Header-Datei „mls.h“ im linken Projekt-Fenster
    - Die Parameter in dieser Header-Datei wurden bereits erläutert (Abschnitt 3.5, Abbildung 23 und Abbildung 24: Header-File „mls.h“ (Feedback, zweikanalig))
    - Nachdem Änderungen vorgenommen wurden, sollte die Datei gespeichert werden (Menü „File-Save“ oder „Save“-Button). Allerdings wird im nächsten Schritt auch noch eine Abfrage gemacht, sollte eine Datei des Projekts noch nicht gespeichert worden sein
- Projekt für den Start des Algorithmus„ vorbereiten
  - Projekt „bauen“
    - Nach dem Öffnen des Projekts (oder wenn Änderungen am Programmcode gemacht wurden) sollte dazu „Rebuild All“ angeklickt werden (entweder entsprechenden Button oder im Menü „Project“).
    - Auch nach Ausführen des ANC-Programms muss vor einem erneuten Start immer neu „gebaut“ werden. Hierzu genügt dann auch „Build Project“ (F7). Allerdings dauert die Ausführung genauso lange wie bei „Rebuild All“.
    - Wurde der Befehl ordnungsgemäß ausgeführt, erscheinen keine Fehlermeldungen Im Output-Fenster (unten) und der Cursor steht im Quellcode-Fenster beim ersten Befehl des Projekts (Datei „Init_21161_EZKit.asm“).

- ANC-Programm verwenden
  - Ausführen des ersten Teils Impulsantwortermittlung
    - Vor dem eigentlichen ANC-Algorithmus müssen die Impulsantworten ermittelt werden. Dabei sollte das Störsignal noch nicht eingeschaltet sein. Durch viele Mittelungsdurchläufe lassen sich aber auch mit Störsignal brauchbare Impulsantworten bestimmen.
    - Damit nur dieser erste Teil abgearbeitet wird, sollte das Programm nach der Impulsantwortermittlung gestoppt werden. Dazu ist der Cursor an einer geeigneten Stelle im Code (z.B. bei ‚r=0;„ am Ende der Routine ‚calc_impulse_response„) in der Datei „noise_control“ zu platzieren. Dazu muss zuvor diese Datei durch Doppelklick im linken Projekt-Fenster geöffnet werden und an die richtige Stelle gescrollt werden. Dann kann durch den Befehl „Run to Cursor“ (Button oder im Menü „Debug“) das Progamm gestartet werden – die Ausführung wird nach der Impulsantwortberechnung gestoppt. Leider habe ich keine einfachere Vorgehensweise gefunden – Breakpoints funktionieren nicht wie gewünscht.
  - Ausführen des zweiten Teils ANC-Algorithmus
    - Störquelle Einschalten
    - Das Programm kann nun durch den „Run“-Button (F5) oder im Menü „Debug“ fortgefahren werden.
    - Die Ausführung kann jederzeit durch den „Halt“-Button (Shift F5) oder im Menü „Debug“ gestoppt werden. Das Fortsetzen des Programms an der gleichen Stelle ist wieder mit „Run“ möglich.
  - Erneutes Ausführen des Programms
    - Hierzu ist es nötig, das Projekt neu zu „Bauen“. Entweder durch „Build Project“ (F7) oder „Rebuild All“.
    - Danach muss wieder die Datei „noise_control“ durch Doppelklick links (oder im Menü „Window“) in den Vordergrund geholt werden, um „Run to Cursor“ auszuführen. Wurde der Cursor inzwischen nicht bewegt, steht er noch an der richtigen Stelle im Quellcode nach der Impulsantwortberechnung.

- Anderes Projekt öffnen
  - Vor dem Öffnen eines anderen Projekts muss unbedingt das aktuelle Projekt geschlossen werden (Menü „File-Close“)
  - Die folgende Frage, ob alle Quell-Fenster geschlossen werden sollen, sollte mit „ja“ beantwortet werden, um Durcheinander zu vermeiden.
  - Nun kann ein anderes Projekt nach bekanntem Muster geöffnet, bearbeitet und/oder gestartet werden.

4 Test des Verfahrens am Antischallfenster

4.1 Versuchsaufbau

Da der Aufbau gerade für die mehrkanaligen Systeme etwas komplexer ist, wird im Folgenden detailliert auf die verwendeten Geräte, deren Einstellungen und die Anschlusskonfiguration eingegangen. Neben dem DSP-Board werden für die zweikanaligen Varianten vier Tiefpass-Filter zum Anti-Aliasing, zwei Verstärker und mindestens 2 Mikrofone und Lautsprecher, die in diesem Fall in ein Antischall-Fenster eingebaut sind, benötigt. Weiterhin sind ein Generator zur Erzeugung des Störsignals und ein Lautsprecher für dessen Abstrahlung nötig. Diese Konfiguration wurde insbesondere für die Tests während der Programmierphase verwendet. Der veränderte Aufbau für die ANC-Versuche wird im nächsten Abschnitt beschrieben. Die folgende Aufstellung führt die verwendeten Geräte im Einzelnen auf:

- EZ-KIT Lite Board mit dem ADSP-21161N SHARC® Processor
- PC/Laptop mit USB und installierter VisualDSP++® Entwicklungsumgebung
- 2x2 TP-Filter
- 2 Stereo-Verstärker
- Antischall-Fenster mit Mikrofonen und Lautsprechern
- Signalgenerator
- Lautsprecherbox mit eingebautem Verstärker zum Abstrahlen des Störsignals
- Diverse Kabel

Um einen Überblick zu gewinnen, zeigt Abbildung 29 den prinzipiellen Aufbau des Active Noise Control Systems (einkanalig). Die gestrichelte Linie deutet an, dass diese Verbindung in der Feedback-Variante nicht hergestellt werden muss.

Die wichtigste Information, um mit dem ANC-System arbeiten zu können, ist wohl die Anschlusskonfiguration am Board. Die Abbildung 30 zeigt, an welche Anschlüsse die externen Geräte anzuschließen sind. Um die Cinch-Buchsen für Kanal 1 (rechts/links) benutzen zu können, ist zuerst die Einstellung für Jumper JP11 zu ändern, da diese auf den Klinkenanschluss voreingestellt ist. Etwas verwirrend ist die Tatsache, dass die Bezeichnung auf dem Board (ch1 und ch2) nicht mit den im Programm vordefinierten Variablen für die Kanäle (Left_Channel_In0, Right_Channel_In0 und Left_Channel_In1, Right_Channel_In1) korrespondiert. Zudem wurden in Bezug auf das ANC-System die Kanäle 1-3 definiert, da die Stereo-Links/Rechts-Notation hier keinen Sinn macht.

An den Ein- und Ausgängen des Boards sind Tiefpassfilter zur Vermeidung von Aliasing zwischengeschaltet. Für die zweikanaligen Versionen sind also vier Tiefpassfilter nötig.

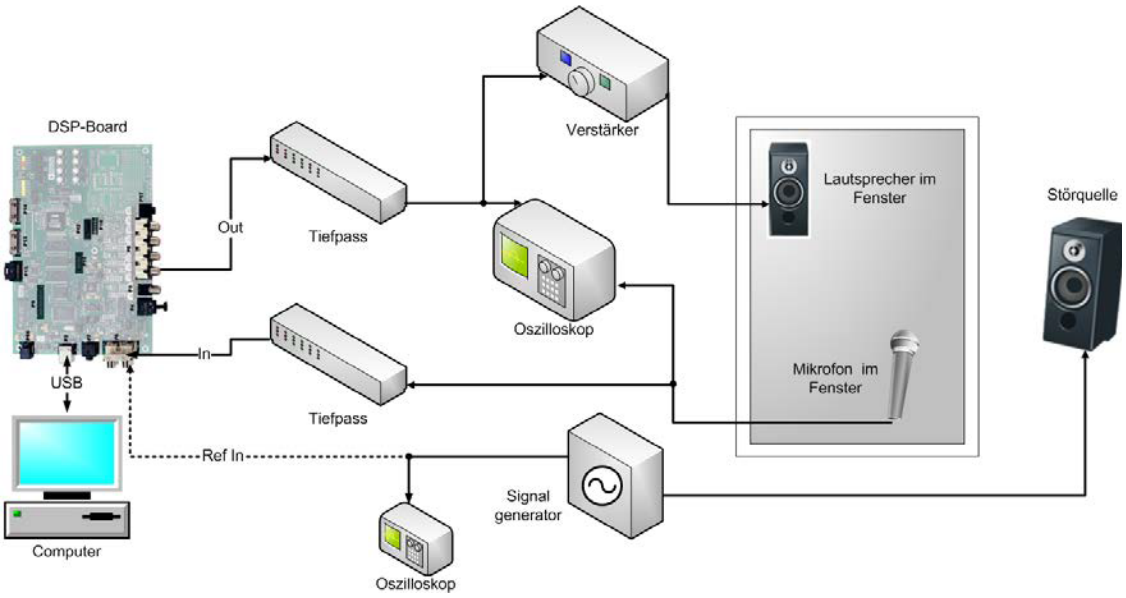


Abbildung 29: Versuchsaufbau des ANCs (einkanalig)

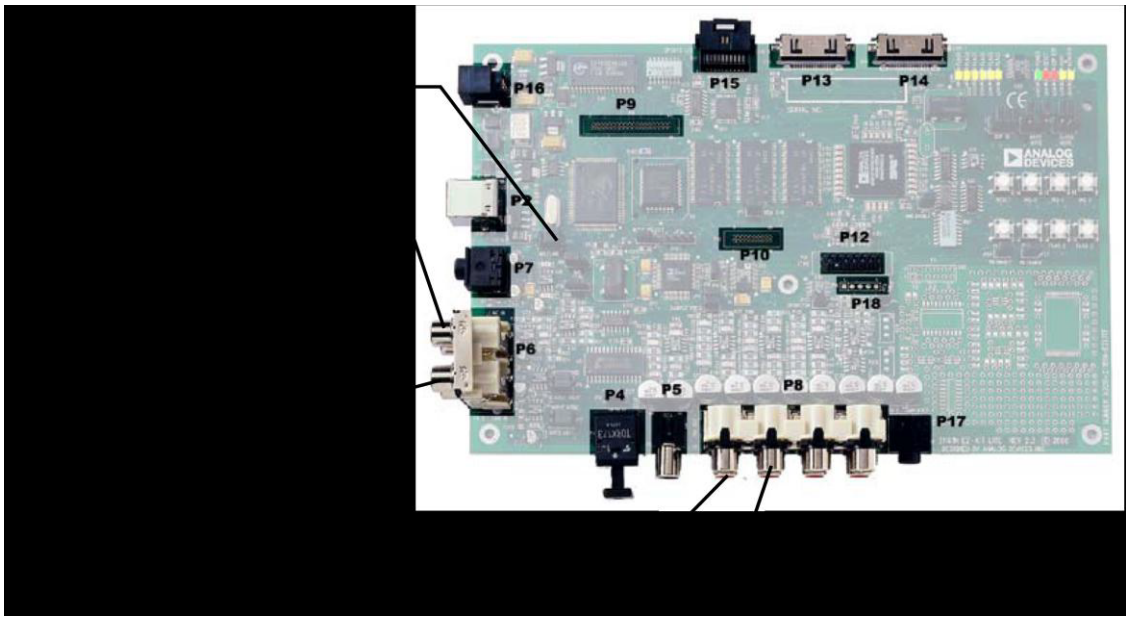


Abbildung 30: Anschlusskonfiguration

4.2 ANC-Versuche

4.2.1 Vorbereitende Messungen und Einstellungen

Um die Parameter bei unbekannten Aufbauten oder Umgebungen sinnvoll einstellen zu können, sollten vorab mit einer einkanaligen Programmversion einige Testläufe gefahren werden. Insbesondere die ungefähre Länge der Impulsantworten muss bestimmt werden, um die Länge des Impulsantwortvektors definieren und außerdem abschätzen zu können, mit welcher Abtastrate gearbeitet werden kann. Das Programm sollte also einige Male bis einschließlich Impulsantwortberechnung ausgeführt und die Länge der Impulsantworten im Debug-Plot ermittelt werden (siehe Abschnitt 2.6.5). Dann sollten einige Durchläufe mit ANC durchgeführt werden, um die Pegel der Verstärker und eventuell des Störsignals auf einen vernünftigen Bereich einzustellen. Dazu bietet es sich an, mit einem sinusförmigen Störsignal zu arbeiten und die Signale an einem Oszilloskop zu verfolgen (siehe nächsten Abschnitt 4.2.2). Ist keine korrekte Funktionsweise zu erkennen, hilft auch hier oft ein Blick in einen Debug-Plot, um die Amplituden der Vektoren zu kontrollieren. Diese sollten naturgemäß unter „1“ bleiben, damit es nicht zu Verzerrungen kommt. Wird das Ausgangssignal y (z.B. im Vektor „Left_Channel_Out0“) größer als „1“, muss der Ausgangspegel der Verstärker erhöht werden.

4.2.2 Test der ANC-Algorithmen

Um die programmierten Algorithmen grundsätzlich auf korrektes Arbeiten zu überprüfen, wurden diese im Vorfeld (und beim Programmieren) mit einem 100 Hz Sinus als Störquelle getestet. Die Abbildung 31 und Abbildung 32 zeigen Fotos der Anzeige des Oszilloskops mit und ohne eingeschalteten ANC-Algorithmus (zweikanalig). Der erste (oben) und dritte Kanal des Oszilloskops stellen das Störsignal, die Kanäle zwei und vier (unten) die vom ANC generierten Gegensignale dar.

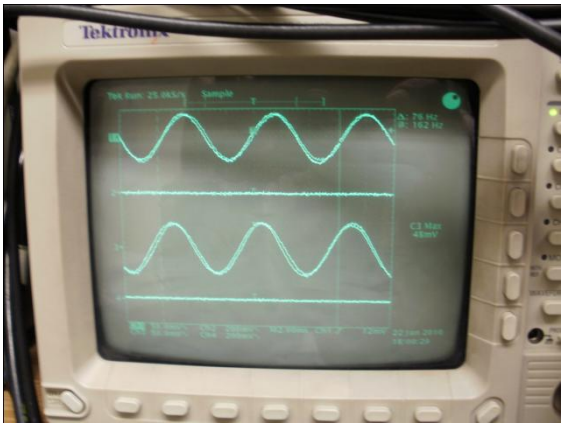


Abbildung 31: ohne ANC

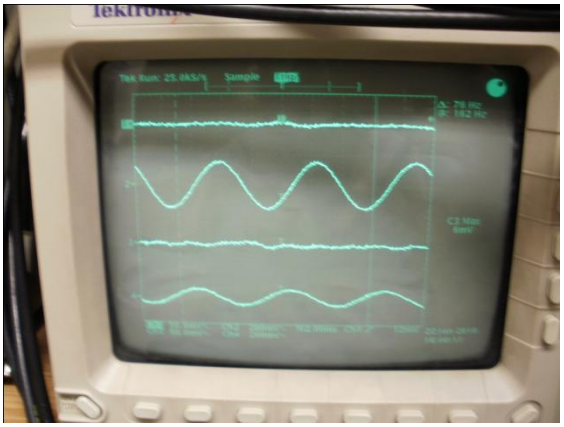


Abbildung 32: mit ANC

4.2.3 Test unter realen Bedingungen

Getestet wurde das beschriebene ANC-System abschließend am Anti-Schall-Fenster. Als Störsignal diente hierbei ein praxisorientiertes weißes Rauschen bis 300 Hz. Mit einem Sound Analyzer (Typ 110, Tippkemper) wurden die Pegel der Terzen 16 Hz bis 10 kHz bei aus- und eingeschaltetem ANC-System gemessen, um die Verbesserung der Schalldämmung zu bestimmen. Die Tiefpassfilter wurden auf 800 Hz eingestellt. Die Abbildung 33 zeigt den Versuchsaufbau am Fensterprüfstand.

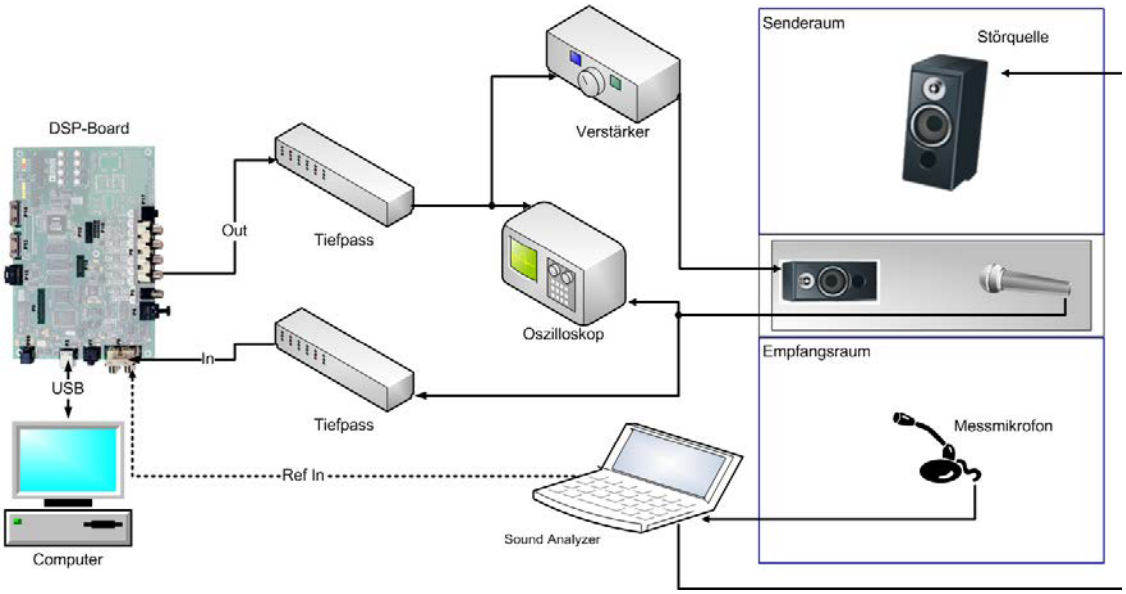


Abbildung 33: Versuchsaufbau Fensterprüfstand

Mit dem vom Sound Analyzer bereitgestellten Messsignal wurde der im Senderaum aufgestellte Lautsprecher angesteuert. Im Empfangsraum wurde ein Messmikrofon auf einem rotierenden Ständer aufgestellt, um den mittleren Schalldruckpegel zu messen. Es wurden nun Messungen unter Verwendung der verschiedenen Algorithmen (einkanalig/zweikanalig, Feedback/Feedforward) und bei abgeschaltetem ANC-System vorgenommen. Die Messdauer betrug dabei jeweils eine Minute. Außerdem wurden die Anregungspegel variiert (hoch/niedrig). Im Folgenden werden einige Ergebnisse dargestellt.

Abbildung 34 zeigt die Messergebnisse für die ein- und zweikanaligen Feedforward-Systeme. Es wurde mit 1000 Adaptionsfilterkoeffizienten und einem α=0,4 (einkanalig) und α=0,08 (zweikanalig) gemessen. Es wurden bei eingeschaltetem ANC Verbesserungen um etwa 4 dB (einkanalig) und 5 dB (zweikanalig) erreicht. Wie auch erwartet zeigt sich, dass die zweikanalige Variante bessere Ergebnisse erzielt.

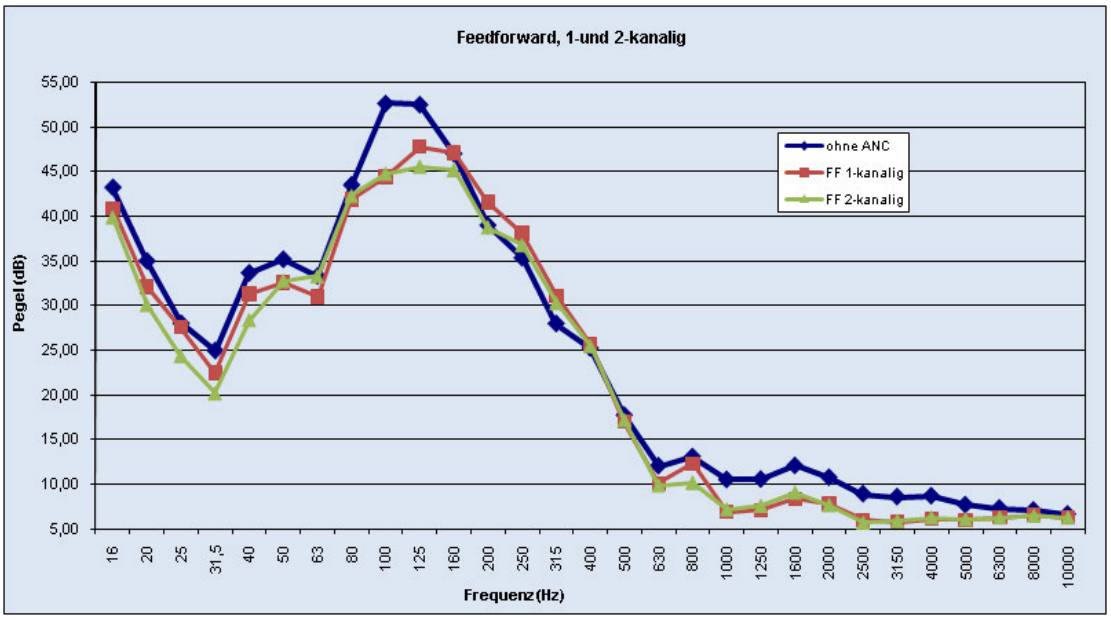


Abbildung 34: Vgl. Feedforward ein- und zweikanalig

Die Messungen mit den Feedback-Algorithmen zeigen geringere Verbesserungen gegenüber dem passiven Fenster (Abbildung 35). Diese Messung wurde hier nur mit 500 Adaptionsfilterkoeffizienten vorgenommen, was eine Verringerung der Anpassungsgenauigkeit zur Folge hat. Eine weitere Erklärung für die schlechteren Ergebnisse dürfte auch die generell langsamere Anpassungsgeschwindigkeit der Feedback-Algorithmen sein.

Es wurde nur mit einem festen α gemessen (α=10 einkanalig, α=60 zweikanalig). Hier ließen sich mit Finetuning dieser Schrittweite α des Adaptionsfilters noch Verbesserungen erreichen.

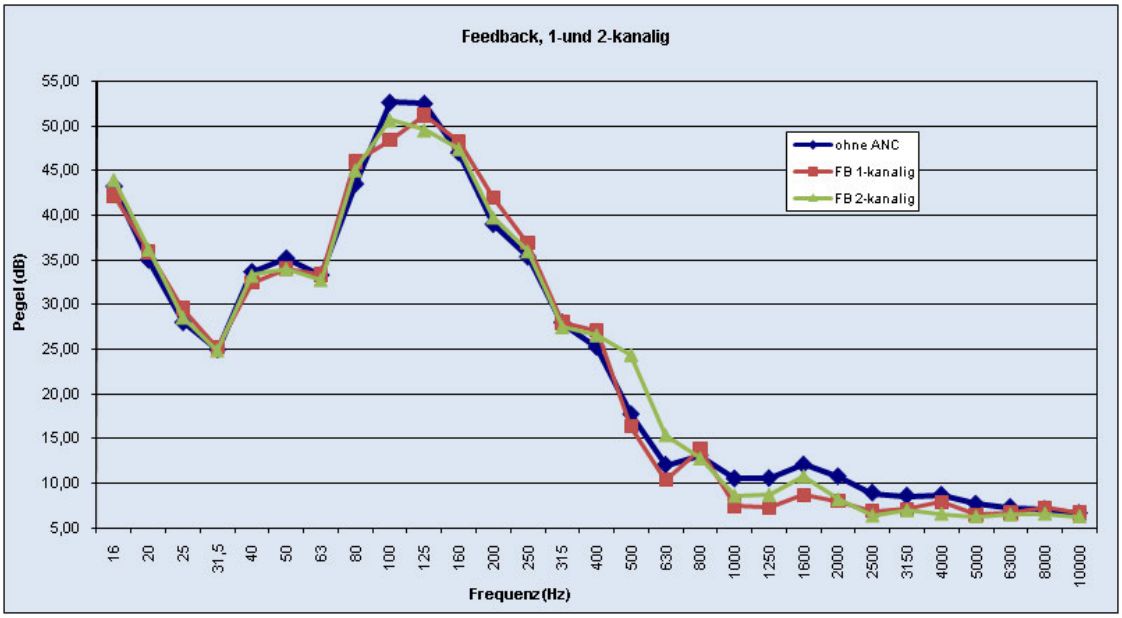


Abbildung 35: Vgl. Feedback ein- und zweikanalig

Die kompletten Messergebnisse sind auf der beigefügten CD-ROM als Excel-Datei abgelegt.

Insgesamt erscheint die durch das Active Noise Control System erreichte Lärmminderung recht gering. Allerdings sollten diese Versuche nur die grundsätzliche Funktionstüchtigkeit der programmierten Algorithmen zeigen. Ein Grund dafür ist wohl eine ungenaue Systemidentifikation, da nur eine Mess-Maximalfolgeausgesendet und auf Mittelung mehrerer verzichtet wurde. Außerdem eignen sich ANC-Systeme besser für stationäre, tonale Störgeräusche und weniger für (Pseudo-)Rauschen. Durch weitere Versuche mit dem System können durch Variieren verschiedener Parameter wie Koeffizientenanzahl, α oder auch Pegeleistellungen noch Verbesserungen erzielt werden.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Es wurde ein funktionierendes Active Noise Control System auf dem vorhandenen DSP-Entwicklungssystem (ADSP-21161N EZ-KIT Lite® Evaluation Kit) in Assembler implementiert. Auf der beiliegenden CD-ROM finden sich alle erstellten Programme und die Messdaten der Versuchsmessung am Schallschutzfenster.

Die Sekundärstreckenmessung des Versuchsaufbaus (Schallschutzfenster im Fensterprüfstand) wurde unter Anwendung der Maximalfolgentechnik realisiert. Um die für die Messung benötigten Maximalfolgen (MLS) bereitstellen zu können, wurde außerdem ein Maximalfolgen-Generator programmiert.

Als eigentliche ANC-Algorithmen kamen x-filtered-LMS-Algorithmen in den Varianten Feedforward und Feedback zum Einsatz.

Dieses Anti-Schall-System ermöglicht Untersuchungen und Versuche im Bereich der aktiven Lärmbekämpfung und lässt sich prinzipiell auch in der Praxis einsetzen. Das System kann einkanalig und zweikanalig betrieben werden. Nach erfolgreichem Test dieser ein- und zweikanaligen Programme wurde zusätzlich versucht, das System auch dreikanalig verwenden zu können, da das DSP-Board dies mit der Anzahl seiner Ein- und Ausgänge noch zugelassen hätte. Die erforderlichen Programme wurden auch geschrieben, konnten aber letztendlich nicht erfolgreich getestet werden. Wahrscheinlich verhindert ein kleiner, aber im Rahmen dieser Arbeit nicht auszumerzender Fehler bei der Ermittlung der Impulsantworten das ordnungsgemäße Arbeiten des Systems mit drei Kanälen. Die dreikanaligen Programmversionen liegen ebenfalls der beigefügten CD-ROM bei.

In dieser Magisterarbeit wurden die theoretischen Grundlagen der Identifikation akustischer Systeme, der Maximalfolgentechnik und der aktiven Lärmbekämpfung, insbesondere mit x-filtered-LMS-Algorithmen, dargestellt. Die verwendete Hardware und die Versuchsanordnungen wurden umfassend dokumentiert und erläutert. Da ein besonderes Augenmerk auf Nachvollziehbarkeit und Erweiterbarkeit des Systems gelegt werden sollte, wurden die programmierten Routinen und Algorithmen ausführlich anhand des Quellcodes beschrieben. Damit mit diesem System auch ohne genaue Kenntnis der technischen Einzelheiten gearbeitet werden kann, wurden außerdem eine kurze Beschreibung der Entwicklungsumgebung und eine Schritt-für-Schritt-Anleitung zur Bedienung des ANC-Systems erarbeitet.

Abschließend wurde das Anti-Schall-System in einem in Fensterprüfstand mit eingebautem Schallschutzfenster unter realen Bedingungen getestet und die Ergebnisse dargestellt und diskutiert.

5.2 Schlussbemerkungen

Das hier vorgestellte ANC-System ist wie auch angestrebt natürlich noch ausbaufähig. So ist es wie bereits erwähnt möglich, das System dreikanal-tauglich zu machen. Ein Weg, die bereits dreikanalig programmierten LMS-Routinen auf Funktion zu überprüfen, wäre, die Ermittlung der Impulsantworten extern oder auch mit diesem System einzeln zu ermitteln diese in Form eines gespeicherten Vektors einzubinden. Dazu müssen die Werte natürlich entsprechend der erläuterten Reihenfolge im Vektor gespeichert werden. Vielleicht wird dabei auch klar, warum die Systemidentifikation nicht richtig funktioniert. Mit den vier Ein- und Ausgängen des Boards würde sich der Feedback-Algorithmus sogar 4-kanalig realisieren lassen, da kein Eingang für das Referenzsignal benötigt wird.

Die Impulsantwortberechnung und die LMS-Routinen wurden im SISD-Mode (Single Instruction-Single Data) programmiert. Um die Leistungsfähigkeit des Systems zu erhöhen und zum Beispiel mehr Filterkoeffizienten zu ermöglichen, könnten die Echtzeit-Routinen auch im SIMD-Mode (Single Instruction-Multiple Data) programmiert werden. Das EZ-KIT Lite Board beherrscht auch diesen Berechnungsmodus, bei dem in einem Befehlszyklus zwei Berechnungen ausgeführt werden können.

Um das System bedienungsfreundlicher zu gestalten wäre auch eine Lösung zu wünschen, die das Editieren der Header-Datei vereinfacht, indem z.B. die Länge der MLS aus der vorgegebenen Ordnung automatisch berechnet würde. Da sich das m.E. in der Header-Datei selbst nicht bewerkstelligen lässt, könnte das auch über ein externes z.B. Visual Basic-Programm mit Eingabemaske, das die Werte dann in die Header-Datei schreibt, geschehen.

Das Forschungsgebiet der aktiven Lärmbekämpfung (und auch der aktiven Schwingungsminderung) erfährt in den letzten Jahren immer mehr Interesse. So verdoppelt sich die Anzahl wissenschaftlicher Publikationen zu diesem Thema in den letzten Jahrzehnten im Fünf-Jahres-Intervall [1]. Auch das Interesse an kommerziellen Anwendungen vergrößert sich stetig. Einige Entwicklungen haben es inzwischen bereits zu Marktreife und Serienproduktion geschafft.

ANC-Systeme kommen inzwischen in Fahrzeugen zur Unterdrückung der Motor- und Fahrgeräusche, in Flugzeugen, zur Minderung der Schallabstrahlung verschiedenster Maschinen und Geräte oder auch in Abluftkanälen zum Einsatz. Eine weitere Anwendung sind aktive Kopfhörer, und inzwischen gibt es auch Versuche, aktive Lärmschutzfenster ähnlich dem hier vorgestellten zur Marktreife zu bringen.

Abschließend möchte ich mich bei Dr. André Jakob für die Unterstützung und seine schier unendliche Geduld bedanken.

6 Literaturverzeichnis

[1] D. Guicking, *Aktive Lärm- und Schwingungsminderung - Von einer Laborkuriosität zum technischen Produkt*., 2007, http://www.physik3.gwdg.de/~guicking/ANVC-Ueberblick.pdf.

[2] S. J. Elliot, P. A. Nelson, "Active Noise Control," *Noise/News International*, pp. 75-98, June 1994.

[3] D. D. Rife, J. Vanderkooy, "Transfer-function measurements with maximum-length sequences," *Journal of the Audio Engineering Society*, vol. 37, no. 6, pp. 419-444, 1989.

[4] S. K. Olesen, et al., "An Improved MLS Measurement System For Acquiring Room Impulse Responses," in *NORSIG2000 Proceedings*, Kolmården, Sweden, 2000, http://www.es.isy.liu.se/norsig2000/publ/page117_id096.pdf.

[5] Jens Hee. Jens Hee Signal processing site. [Online]. http://jenshee.dk/signalprocessing/mls.pdf

[6] Otto-von-Guericke-Universität Magdeburg, Fakultät Elektrotechnik und Informationstechnik, Lehrstuhl für Kognitive Systeme. (2004) Laborpraktikum Digitale Signalverarbeitung. [Online]. http://iesk.et.uni-magdeburg.de/ko/lehre/disi-praktikum/2004/versuch4.pdf

[7] J. Fasbender, D. Günzel, "Ein Meßsystem für rechnergestützte Impulsmessungen in der Akustik," *ACUSTICA*, vol. 45, p. 151 ff., 1980.

[8] A. Jakob, M. Möser, C. Ohly, "Ein aktives Doppelglas-Fenster mit geringem Scheibenabstand," in *Fortschritte der Akustik - DAGA 2002*, Bochum, Deutschland, 2002, http://www.advacoustic.de/jakob_daga2002_1.pdf.

[9] A. Jakob, M. Möser, "Enhancement of the transmission loss of double panels by means of actively controlling the cavity sound field," in *Proceedings of ACTIVE 99*, Fort Lauderdale, Florida, USA, 1999, http://www.advacoustics.de/jakob_active99.pdf.

[10] A. Jakob, M. Möser, "Aktive Verbesserung der Schalldämmung eines Doppelfensters: Vergleich zwischen Feedback- und Feedforward-Anordnungen," in *Fortschritte der Akustik - DAGA 2001*, Hamburg, Deutschland, 2001, http://www.advacoustics.de/jakob_daga2001.pdf.

[11] A. Jakob, "Antischall-Fenster in Theorie und (Labor)-Praxis," in *Fortschritte der Akustik - DAGA 2005*, München, Deutschland, 2005, http://www.advacoustic.de/jakob_daga2005_1.pdf.

[12] A. Jakob, "Schallschutzfenster mit active noise control - Motivation, Lösungen und Visionen," in *ift Schallschutzfachtagung 2006, Institut für Fenstertechnik - ift*, Rosenheim, Deutschland, 2006.

[13] Martin Meyer, *Signalverarbeitung: Analoge und digitale Signale, Systeme und Filter*. Wiesbaden: Vieweg & Teubner, GWV Fachverlage GmbH, 2009.

[14] A. Domann, *Modellierung des Übertragungsverhaltens einer schwingenden Platte, Diplomarbeit*. Berlin, 1997.

[15] S. Müller, "Messtechnik," in *Handbuch der Audiotechnik*, Stefan Weinzierl, Ed. Berlin, Heidelberg: Springer Verlag, 2008.

[16] M. Möser, Ed., *Messtechnik der Akustik*. Berlin, Heidelberg: Springer-Verlag, 2010.

[17] Michael Wolff, André Jakob, Michael Möser, "Systemidentifikation von nicht-linearen Sekundärstrecken in active noise control Systemen mit Hilfe Neuronaler Netzwerke," , München, 2005.

[18] M. Vorländer, "Anwendungen der Maximalfolgentechnik in der Akustik," in *Fortschritte der Akustik – DAGA ´94*, Frankfurt (M.), 1994.

[19] W. D. T. Davies, *System Identification for self-adaptive control*.: Wiley-Interscience, 1970.

[20] F. Dachselt, "Zufall und C(h)aos," *KC-News*, Mar. 1996, http://www.iee.et.tu-dresden.de/~kc-club/02/KCN96-03/KCN96-03-08.HTML.

[21] R. L. Peterson, R. L. Ziemer, D. E. Borth, *Introduction to Spread Spectrum Communications*. Upper Saddle River, NJ, USA: Prentice Hall, 1995.

[22] A. Schmitz, M. Vorländer, "Messung von Außenstoßantworten mit Maximalfolgen-Hadamard-Transformation und deren Anwendung bei Inversionsversuchen," *ACUSTICA*, no. Vol. 71, pp. 257-268, 1990.

[23] Bernard Widrow, Samuel Stearns, *Adaptive Signal Processing*.: Prentice-Hall, 1985.

[24] S. J. Elliot, P. A. Nelson, "Algorithm for multichannel LMS adaptive filtering," *Electronics Letters*, no. 21, pp. 979-981, 1985.

[25] A. Jakob, M. Möser, "A modal model for actively controlled double-glazed windows," *ACUSTICA - acta acustica*, vol. 89, no. 3, pp. 479-493, 2003.

[26] A. Jakob, M. Möser, "Parameter study with a modal model for actively controlled double-glazed windows," *ACUSTICA - acta acustica*, vol. 90, no. 3, pp. 467-480, 2004.

[27] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, Second Edition ed. San Diego, CA: California Technical Publishing, 1999, http://www.dspguide.com/.

[28] W. Heinrich, *Signalprozessor Praktikum zum Starterkit EZ-Kit-Lite von Analog Devices*. Poing: Francis" Verlag GmbH, 1999.

[29] G. Doblinger, *Signalprozessoren – Architekturen, Algorithmen, Anwendungen*, 2nd ed. Wilburgstetten: J. Schlembach Fachverlag, 2004.

[30] Analog Devices Inc., *ADSP-21161 SHARC® DSP Hardware Reference*, Third Edition ed. Norwood, MA, USA, 2002.

[31] Analog Devices, Inc. (2011) AD1836A: Multichannel 96 kHz Codec. [Online]. http://www.analog.com/en/audiovideo-products/audio-codecs/ad1836a/products/product.html

[32] Analog Devices, Inc. (2011) AD1852: Stereo, 24-Bit, 192KHz, Multibit Sigma Delta DAC. [Online]. http://www.analog.com/en/digital-to-analog-converters/audio-da-converters/ad1852/products/product.html

[33] Analog Devices Inc., *ADSP-21161N EZ-KIT LITE Evaluation System Manual*, 20th ed. Norwood, MA, USA, 2003.

[34] Analog Devices Inc., *Visual DSP++® 3.5 User's Guide for 32-Bit Processors*, 10th ed. Norwood, MA, USA, 2004.

[35] Dr-Jordan-Design. (2011) Signal Generator. [Online]. http://www.dr-jordan-design.de/signalgen.htm

7 Abbildungsverzeichnis

Abbildung 1: MLS-Folge der Ordnung 7 8

Abbildung 2: Schieberegisterkette der Ordnung m=7 10

Abbildung 3: Kreuzkorrelation zwischen MLS und Systemantwort bei kurzgeschlossenen Anschlüssen 10

Abbildung 4: Prinzip der Ermittlung der Impulsantwort 12

Abbildung 5: Blockdiagramm FB 14

Abbildung 6: Blockdiagramm FF 14

Abbildung 7: Prinzipschaltbild des filtered-x-LMS-Algorithmus„ (Feedforward) 15

Abbildung 8: Prinzipschaltbild des filtered-x-LMS-Algorithmus„ (Feedback) 16

Abbildung 9: Antischallfenster 18

Abbildung 10: Lautsprecher vor dem Einbau 19

Abbildung 11: Schalldämmmaßkurve des Fensters 19

Abbildung 12: ADSP-21161N EZ-KIT Lite® Evaluation Kit Board 24

Abbildung 13: ADSP-21161 SHARC Blockschaltbild 25

Abbildung 14: AD1836 Blockschaltbild 26

Abbildung 15: AD1852 Blockschaltbild 27

Abbildung 16: Screenshots „New Session“ 29

Abbildung 17: Screenshot der Visual-DSP-Oberfläche 31

Abbildung 18: Top-Down Design von DSP-Anwendungen 34

Abbildung 19: Dateien des Projekts 36

Abbildung 20: Memory-Mapping - Speichersektionen 40

Abbildung 21: Memory- Mapping – speicherbare Datenworte 40

Abbildung 22: Programmablauf der Abtastratenanpassung 43

Abbildung 23: Header-File „mls.h“ (Feedforward, zweikanalig) 44

Abbildung 24: Header-File „mls.h“ (Feedback, zweikanalig) 46

Abbildung 26: Speicherbelegung DSP-Board 47

Abbildung 27: Impulsantwort einkanalig 68

Abbildung 28: Impulsantworten zweikanalig 69

Abbildung 28: Positionen der Menüs und Buttons 83

Abbildung 30: Versuchsaufbau des ANCs (einkanalig) 88

Abbildung 31: Anschlusskonfiguration 88

Abbildung 32: ohne ANC 90

Abbildung 33: mit ANC 90

Abbildung 34: Versuchsaufbau Fensterprüfstand 90

Abbildung 35: Vgl. Feedforward ein- und zweikanalig 91

Abbildung 36: Vgl. Feedback ein- und zweikanalig 92

8 Tabellenverzeichnis

Tabelle 1: Testsignale 7

Tabelle 2: Anwendungen von DSPs 23

Tabelle 3: Operationen pro Abtastzyklus 39

Tabelle 4: Speicherbelegung Feedforward 48

Tabelle 5: Speicherbelegung Feedback 49

Tabelle 6: Echtzeit-Berechnungsgrenzen Feedforward 50

Tabelle 7: Registerkombinationen für MAC-Operationen 66

9 Anhang

9.1 Dateitypen im ADSP-21161N EZ-KIT Lite® Evaluation Kit for SHARC® processors

Extension	Name	Purpose
.ASM	Assembly source file	Source file comprising assembly language instructions
.C	C source file	Source file comprising ANSI standard C code and Analog Devices extensions
.CPP .CXX .HPP .HXX	C++ source file	Preprocessed compiler files that are inputs to the C/C++ compiler. These files comprise ANSI standard C++ code.
.DPJ	Project file	Contains a description of how your source files combine to build an execut-able program
.LDF	Linker Description File	Linker command source file is a text file that contains commands for the linker in the linker"s scripting language
.IS .PP .S	Intermediate files	Preprocessed assembly files generated by the preprocessor
.DOJ	Assembler Object file	Binary output of the assembler
.DLB	Archiver file	Archiver"s binary output in ELF format
.H	Header file	Dependency file used by the pre-proces-sor, and a source file for the assembler and compiler
.H	Loader output	For ADSP-2192-12 processors, the C-language header file output of the boot loader utility (BLU)

.DAT	Data file	Dependency file used by the assembler for data initialization
.DSP	Assembly source file	Source file comprising assembly language instructions
.MAK .MK	Makefiles	The output make rule file is used for project builds
.DPG	Project group	An .XML file containing information about projects
.IDL	VCSE input	VCSE Interface Definition Language (VIDL) specification
.XML	Manifest	Used by New Component Package Wizard when packaging a component for distribu-tion

9.2 Kurzanleitung

- Visual-DSP-Umgebung starten
  - Korrekte Session für ADSP-21161 auswählen oder neu kreieren (Menü „Session“)
  - Simulator (ohne Board)
  - EZ-KIT Lite (mit Board)
- Projekt öffnen
  - Menü „File-Open-Project“
  - „noise_control.dpj“ in z. B „..\lms_final\feed_forward_1_ch\“ öffnen
- Parameter wie Abtastrate, Koeffizientenanzahl oder α konfigurieren
  - Doppelklick auf die Header-Datei „mls.h“ im linken Projekt-Fenster
  - Änderungen vornehmen
- Projekt für den Start des Algorithmus„ vorbereiten
  - Projekt „bauen“ („Rebuild All“/„Build Project“ (F7))
- ANC-Programm verwenden
  - Ausführen des ersten Teils Impulsantwortermittlung
    - Störquelle aus
    - Cursor zum Ende der Routine „calc_impulse_response“ („r=0;“)
    - Start des Programms mit „Run to cursor“-Button
  - Ausführen des zweiten Teils ANC-Algorithmus
    - Störquelle einschalten
    - Fortsetzen mit „Run“-Button (F5)
    - Unterbrechen mit „Halt“-Button (Shift F5)/ Fortsetzen mit „Run“-Button (F5)
  - Erneutes Ausführen des Programms
    - „Build Project“ (F7) oder „Rebuild All“.
    - Datei „noise_control“ in den Vordergrund
    - Run to Cursor“ auszuführen (Cursor vorher wieder an die „calc_impulse_response“ („r=0;“)
- Anderes Projekt öffnen
  - Geöffnetes Projekt schließen (Menü „File-Close“)
  - Die folgende Frage, ob alle Quell-Fenster geschlossen werden sollen, mit „ja“ beantworten
  - anderes Projekt öffnen