

# **Flexibles Patientenalarmierungssystem**

Semesterarbeit  
Wintersemester 06/07

Autoren:  
Simon Landolt, Ronald Reichmuth

Betreuer:  
Prof. Dr. Guido M. Schuster

Rapperswil, 9. Februar 2007

HSR Hochschule für Technik Rapperswil  
Oberseestrasse 10  
Postfach 1475  
CH - 8640 Rapperswil

**Angaben zur Semesterarbeit:**

Abteilung:	Elektrotechnik	
Vertiefungsmodul:	Digitale Signalverarbeitung	
Autoren:	Ronald Reichmuth	rreichmu@hsr.ch
	Simon Landolt	slandolt@hsr.ch
Betreuer:	Prof. Dr. Guido M. Schuster	guido.schuster@hsr.ch
	Dipl. El. Ing. FH Reto Ansorge	ransorge@hsr.ch

# Inhaltsverzeichnis

<b>1</b>	<b>Kurzfassung</b>	<b>5</b>
<b>2</b>	<b>Aufgabenstellung</b>	<b>7</b>
2.1	Ausgangslage . . . . .	7
2.2	Aufgabenstellung . . . . .	7
2.3	Erwartete Ergebnisse . . . . .	8
2.4	Arbeitsweise . . . . .	8
<b>3</b>	<b>Konzept und Projektplanung</b>	<b>9</b>
3.1	Funktionsweise . . . . .	9
3.2	Zeitplan . . . . .	10
<b>4</b>	<b>Verwendete Hardware und Software</b>	<b>11</b>
4.1	Hardware . . . . .	11
4.1.1	SmartRF04EB . . . . .	11
4.1.2	CC2430EM . . . . .	12
4.1.3	SoC CC2431 . . . . .	12
4.2	Software . . . . .	13
4.2.1	IAR Embedded Workbench . . . . .	13
4.3	Oszillator . . . . .	14
<b>5</b>	<b>Abtastung und Quantisierung</b>	<b>15</b>
5.1	Abtastung . . . . .	15
5.2	Quantisierung . . . . .	16
5.2.1	DMA (Direct Memory Access) . . . . .	17
5.2.2	Auswertung der ADC-Daten . . . . .	18
<b>6</b>	<b>Voice Activity Detection</b>	<b>21</b>
6.1	Zero Crossing Rate (ZCR) . . . . .	21
6.2	Root-Mean-Square (RMS) . . . . .	22
6.3	Simulation und Implementation auf dem SoC . . . . .	22
6.3.1	Simulation mit Matlab . . . . .	22
6.3.2	Alarmauslösung . . . . .	23
<b>7</b>	<b>Audio Signalerzeugung</b>	<b>25</b>
7.1	PWM als Digital Analog Konverter . . . . .	25
7.2	Analoges Rekonstruktionsfilter . . . . .	28
<b>8</b>	<b>C-Programm</b>	<b>31</b>
8.1	Jitter Buffer . . . . .	31

8.2	Patient (Master) . . . . .	32
8.2.1	Kommunikation mit der Zentrale . . . . .	33
8.3	Zentrale (Slave) . . . . .	34
8.3.1	Sende-Ringbuffer . . . . .	34
8.3.2	Kommunikation mit dem Patienten (Master) . . . . .	35
8.4	Daten Senden und Empfangen . . . . .	36
8.4.1	ZigBee . . . . .	37
8.4.2	Uebertragung . . . . .	37
8.5	Verbindungskontrolle . . . . .	38
8.6	RS232-Schnittstelle . . . . .	39
<b>9</b>	<b>Messungen</b>	<b>41</b>
9.1	Messungen mit Simulink . . . . .	41
9.2	Quantisierungsrauschen . . . . .	42
9.3	Messresultate der SNR . . . . .	42
9.4	Versuch mit 16kHz mittels Up/Downsampling . . . . .	44
9.4.1	Messresultate . . . . .	44
<b>10</b>	<b>Ausblick</b>	<b>47</b>
<b>11</b>	<b>Rückblick</b>	<b>49</b>
<b>A</b>	<b>Anhang</b>	<b>51</b>
A.1	Datenblätter CC2430DK Development Kit . . . . .	52
A.1.1	Schema Audio Interface . . . . .	52
A.1.2	TPA4411 Stereo Headphone Driver . . . . .	53
A.1.3	LTC1799 Oscillator . . . . .	54
A.1.4	Errata-File Ausschnitt . . . . .	55

# 1 Kurzfassung

Ziel dieser Arbeit ist es, eine bidirektionale Sprachkommunikation zwischen zwei Teilnehmern aufzubauen. Die eine Seite ist die Zentrale und die andere Seite die Patienten. Beim Patienten wird mit Spracherkennung ermittelt, ob die Person ein Problem hat. Wenn dieser Fall eintritt, wird die Kommunikation zur Zentrale aufgebaut und der Patient kann sich mit dem Pfleger an der Zentrale unterhalten. Wenn das Problem gelöst ist, wird die Verbindung wieder freigegeben und ist wieder offen für andere Patienten. Wenn ein Patient eine Verbindung zur Zentrale aufbauen will, aber diese ist schon besetzt, wird der zusätzliche Alarm bei der Zentrale angezeigt. Sobald die Verbindung wieder frei ist, wird der Kontakt mit dem anderen Patienten hergestellt.

Um zu erkennen ob der Patient ein Problem hat, wird eine Voice Activity Detection (VAD) angewendet. Es werden der quadratische Mittelwert (RMS) und die Anzahl Nulldurchgänge (Zero Crossing) berechnet. Bei genügend VAD wird die Verbindung zur Zentrale hergestellt. Das Sprachsignal wird dann mit 8kHz abgetastet und mit einem ADC eingelesen. Sobald ein Paket von 120 Samples gefüllt ist, werden die Sprachdaten dem Gegenüber gesendet. Dieser gibt die Daten mittels PWM (Pulsweitenmodulation) an den Ausgang, wo sie mit einem Rekonstruktionsfilter wiederhergestellt werden und an einen Lautsprecher ausgegeben werden.

Das Ganze wird mit dem SmartRF04EB realisiert, welches ein Audio-Interface enthält. Auf dieses Evaluations Board (EB) kann der SOC (System On Chip) CC2431 aufgesteckt werden. Der CC2431 beinhaltet einen 8051 Intel Prozessor und ist ZigBee fähig. ZigBee ist eine Variante der Funkübertragung, welche wir für unsere Übertragung der Sprachdaten gebrauchen.

Um die Kommunikation zu testen, hat sich die Übertragung eines Sinus als sehr hilfreich erwiesen. Das Spektrum des beim Empfänger ausgegebenen Sinus konnte mit Matlab gut analysiert werden. Damit konnten die kleinsten Fehler sowie die groben, welche sich durch Phasenfehler ausdrücken, festgestellt und nach Möglichkeit behoben werden.



## 2 Aufgabenstellung

### 2.1 Ausgangslage

Die Schweizerische Gesellschaft für Muskelkranke SGMK führt jährlich ein Lager für muskelkranke Menschen durch. Die Lagerteilnehmer haben einen sehr hohen Betreuungsbedarf, da sie auf Grund ihrer Krankheit alltägliche Handlungen nur äusserst beschränkt oder gar nicht vornehmen können. Um während der Nacht die Betreuung sowie das rasche Eingreifen bei Notfällen zu ermöglichen wird ein Alarmsystem eingesetzt.

Die Lager der SGMK werden in normalen Ferienhäusern durchgeführt, die nicht eine permanente Alarmierungsanlage haben, wie sie etwa in Spitälern oder Krankenheimen vorhanden sind. Zurzeit erfolgt die Alarmierung behelfsmässig mittels Babyphone sowie Funk-Türklingeln. Dabei benutzen mehrere Sender jeweils den gleichen Funkkanal.

Diese Konstellation hat folgende Nachteile:

- Nur Einwegkommunikation Patient-Betreuer
- Keine Kontrolle über die Funktionsfähigkeit
- Keine eindeutige Zuordnung zwischen Endgerät und Benutzer
- Geringe Reichweite
- Hohe RF Störempfindlichkeit
- Geringe Toleranz auf Umgebungsgeräusche (Beatmungsgerät)
- Keine Privatsphäre bei Telefongesprächen des Patienten

### 2.2 Aufgabenstellung

Entwicklung eines Zweiweg Patientenalarmierungssystems auf Chipcon Basis. Alarmauslösung über Stimme und Knopfdruck, wobei Störgeräusche keinen Fehlalarm auslösen sollen. Die Zentrale sollte die rasche Identifikation des Rufenden ermöglichen und einen Stausüberblick über die Sender ermöglichen.

Das System soll die Nachteile der jetzigen Lösungen vermeiden und zusätzlich (in zweiter Priorität) folgende Eigenschaften haben:

- Unterstützung von bis zu 25 Sendern
- Anschluss eines externen Alarmschalters am Sender

- Getrennter Daten-/Signalisierungs- und Sprachkanal
- 2-3 Sprachkanäle (mittels mehrerer Frequenzen oder Sprachkompression)
- Akkubetrieb der Sender für min. 16 h / aufladen in weniger als 8 Stunden
- Reichweitenvergrößerung mittels automatischer Relaisfunktion
- Evtl. Computeranschluss der Zentrale

## **2.3 Erwartete Ergebnisse**

- Dokumentation der Theorie und der Lösung
- Ein funktionsfähiges System (Hard und Software) welches diese bidirektionale Sprachkommunikation ausführt.
- Je ein Laborbuch

## **2.4 Arbeitsweise**

- Sie führen ein persönliches Laborbuch, wo Sie aufschreiben wann Sie was für wie lange machen und was die Ergebnisse sind
- Sie schicken mir vor jeder Sitzung eine Zusammenfassung welche dokumentiert, was Sie in der letzten Woche gemacht haben.

# 3 Konzept und Projektplanung

## 3.1 Funktionsweise

Mittels eines Mikrophones und geeigneter Analyse können Geräusche welche beim Patienten im Zimmer vorkommen analysiert werden. Bei kritischen Situationen (lautes Krachen oder Hilferufe) wird automatisch Alarm ausgelöst und der Zentrale gemeldet. Ein Betreuer kann dann direkt mit dem Patienten kommunizieren. Bei der Zentrale kann der Betreuer auf einem Display oder einem Computer erkennen, welcher Patient gerade ein Alarm ausgelöst hat. Dem Betreuer wird angezeigt von welchem der Patienten die Zentrale eine Antwort erhält. Dies ermöglicht einen Überblick über die Sender und man bemerkt, wenn ein Sender nicht mehr erreichbar ist.

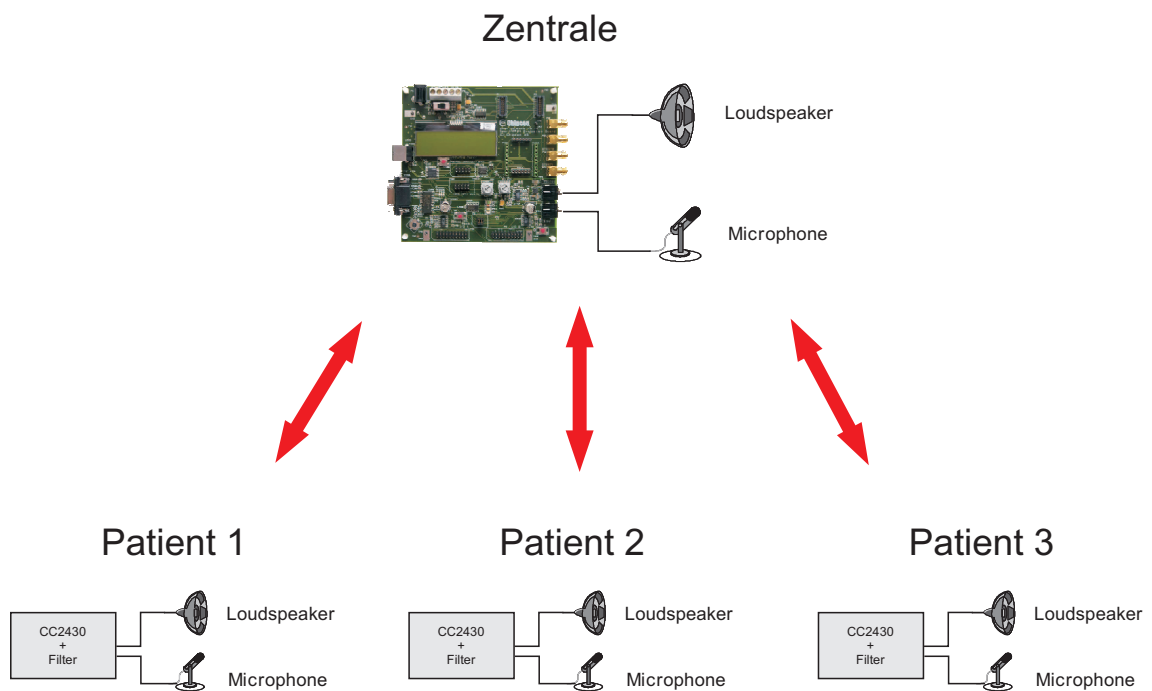


Abbildung 3.1: Projektkonzept

Das Einlesen der Sprache, das Versenden und die Ausgabe des Sprachsignals wurden mit dem Chip CC2431 realisiert. Dieser wird mit dem dazugehörigen Evaluationsboard im Kapitel 4 (Verwendete Hardware und Software) näher beschrieben.

Der grobe Ablauf der Kommunikation ist in Abbildung 3.2 dargestellt. Der ADC (Analog Digital Converter) tastet das Sprachsignal ab und sendet die Daten an den jeweiligen

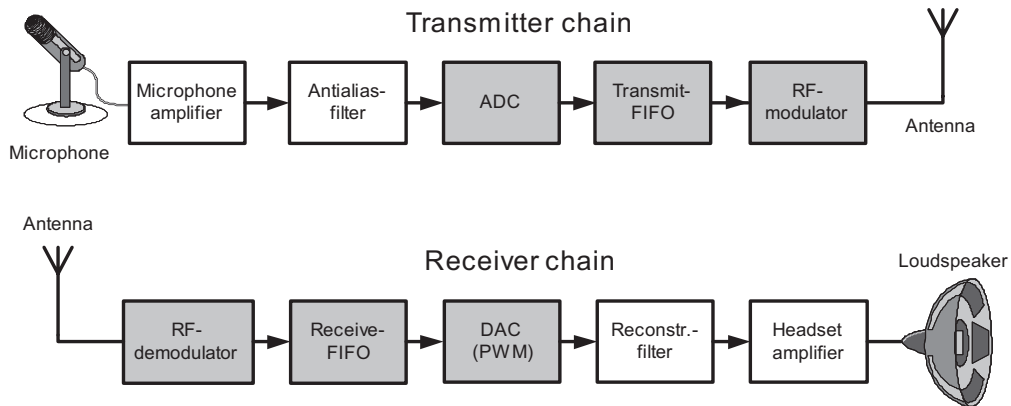


Abbildung 3.2: System Überblick und Signalverlauf

Empfänger. Dieser empfängt die Daten und erzeugt das Audiosignal mittels PWM (Pulsweitenmodulation). Umgekehrt sendet auch dieser seine eingelesenen Sprachdaten und der Andere erzeugt das Sprachsignal. Somit entsteht eine bidirektionale Sprachkommunikation.

### 3.2 Zeitplan

Arbeit	KW	43	44	45	46	47	48	49	50	51	52	1	2	3	4	5	6
Einarbeiten ins Thema	Soll																
	Ist																
Inbetriebnahme der vorgegebenen HW	Soll																
	Ist																
Bidirektionale Sprachkommunikation aufbauen	Soll																
	Ist																
Spracherkennung/Filter	Soll																
	Ist																
Signalisierung	Soll																
	Ist																
Dokumentation schreiben	Soll																
	Ist																

Im Wesentlichen konnten wir unseren Zeitplan gut einhalten. Wir machten uns mit dem Thema und der Hardware vertraut und machten uns dann an die Programmierung des CC2431. Jedoch benötigten wir für den Aufbau der bidirektionalen Sprachkommunikation deutlich länger als erwartet. Wir hatten relativ schnell eine funktionierende Verbindung, doch diese war mit Fehlern behaftet welche wir nur allmählich beheben konnten. Diese Fehler haben uns sehr viel Zeit gekostet und sind mehrheitlich auf unvorhergesehene Probleme mit dem Chip CC2431 zurückzuführen.

# 4 Verwendete Hardware und Software

## 4.1 Hardware

### 4.1.1 SmartRF04EB

Das Main Evaluation Board (EB) SmartRF04EB besitzt ein Audio Interface, USB Interface, RS232 Interface, LCD, Buttons, Potentiometer, LEDs und mehr. Da man das CC2430EM (4.1.2) auf den SmartRF04EB aufstecken kann, ist es möglich einfach mit dem CC2431 (4.1.3) zu arbeiten. Die verschiedenen Komponenten können nach Bedarf benutzt werden. Wir benutzten das USB Interface um den CC2431 programmieren zu können und das Audio Interface um die Sprachdaten einzulesen und auszugeben. Bei der Programmierung des CC2431 steht die IAR Embedded Workbench (4.2.1) zur Verfügung, sie wird als Compiler und als Debugger benutzt.

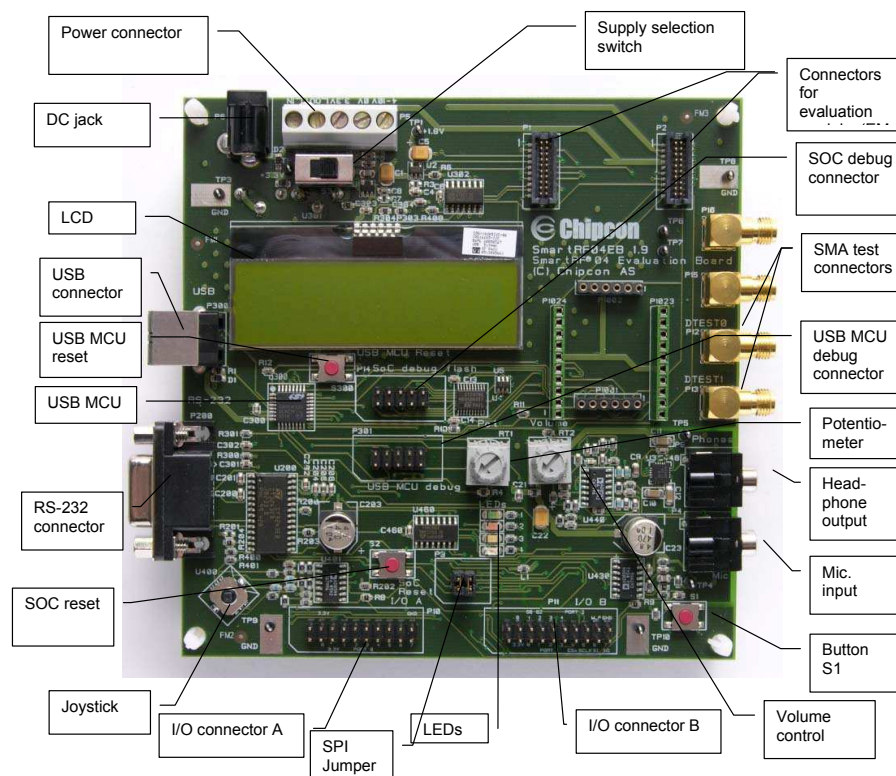


Abbildung 4.1: SmartRF04EB Main Evaluation Board [8]

### 4.1.2 CC2430EM

Das Evaluation Module CC2430EM wird über den Stecker P1 und P2 (siehe in Abbildung 4.1, *connectors for evaluation*) auf das SmartRF04EB gesteckt. Das Module beinhaltet den CC2431 mit externer Beschaltung für den Clock und den RF Teil mit der 2.4 GHz Antenne für ZigBee.



Abbildung 4.2: CC2430EM

### 4.1.3 SoC CC2431

Der CC2431 ist ein SOC (System-On-Chip) für kabellose Netzwerk-Lösungen mittels ZigBee. Durch die vielen Komponenten welche der CC2431 bereits enthält, ist es möglich ZigBee schnell und billig zu realisieren. Dadurch ist die ZigBee Lösung auf diesem SOC die konkurrenzfähigste auf dem Markt. Der CC2431 beinhaltet einen 8051 Mikroprozessor, 128kB Flash Speicher, 8kB RAM und viele weitere leistungsfähige Komponenten wie ein DMA, 14-Bit ADC (Analog-Digital-Wandler), Timer und mehr. Ein Überblick über die Komponenten des CC2431 ist in Abbildung 4.3 gegeben. Einige der Eigenschaften des CC2431 sind unten aufgelistet, jedoch sind das längst nicht alle. Um weitere Informationen zu erhalten sollte das Datenblatt [2] konsultiert werden.

Der CC2431 ist sehr gut für ultra low power Anwendungen geeignet. Dies wird durch verschiedene Funktionsmodi erreicht.

#### Eigenschaften:

- 32MHz 8-Bit 8051 Mikroprozessor
- 2.4 GHz IEEE 802.15.4 konformer RF Transceiver
- 128 KB programmierbarer Flash-Speicher
- 4 flexible Powermodi für geringeren Stromverbrauch
- Ultra low power Betrieb möglich
- ZigBee Protokoll Stack(Z-Stack)
- 2 USARTs welche serielle Protokollen unterstützen

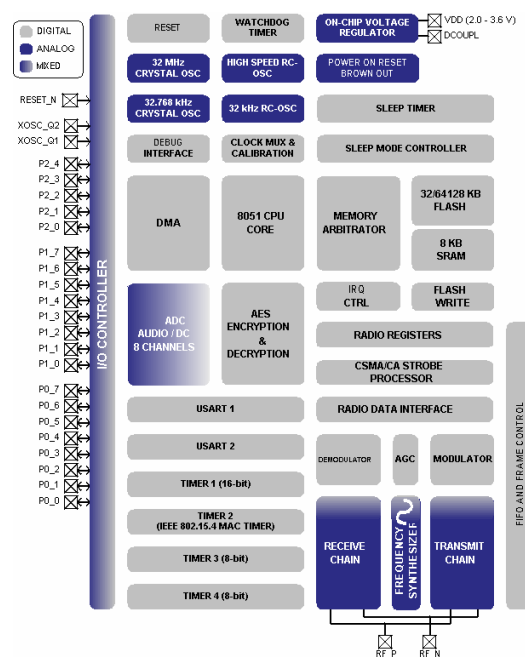


Abbildung 4.3: Aufbau des CC2431 [2]

## Verwendete Hardware und Software

---

- Programmierbarer Watchdog-Timer
- Ein IEEE 802.15.4 MAC Timer, ein 16-Bit Timer und zwei 8-Bit Timer
- On-chip Temperatur Sensor
- 21 I/O Pins
- Hardware unterstütztes Debugging

## 4.2 Software

### 4.2.1 IAR Embedded Workbench

Die IAR Embedded Workbench ist eine voll integrierte Entwicklungsumgebung für die Entwicklung von eingebetteten Systemen. Die Software enthält notwendige Funktionen wie einen Projektmanager, einen Texteditor, C/C++ Compiler, Assembler, Bibliotheken und verschiedene Debugger. Um die Entwicklungsumgebung in Betrieb zu nehmen kann nach dem Chipcon IAR Usermanual 1.2 [3] vorgegangen werden. Um das EB möglichst schnell Anzusteuern, kann auf der Chipcon Homepage bestehendes Projekt [9] für den CC2431 heruntergeladen werden. Darin ist folgendes enthalten:

- Hardware Abstraction Layer (HAL)
- Chipcon Utility Library (CUL)

Damit müssen bei den Projektoptionen noch zusätzlich folgende Einstellungen vorgenommen werden, die unter: Projekt → Option → C/C++ Compiler → Preprocessor, zu finden sind:

```
$TOOLKIT_DIR$\INC\  
$TOOLKIT_DIR$\INC\CLIB  
$PROJ_DIR$\..\..\..\Library\cc2430\CUL\include  
$PROJ_DIR$\..\..\..\Library\cc2430\CUL\source  
$PROJ_DIR$\..\..\..\Library\cc2430\HAL\include  
$PROJ_DIR$\..\..\..\Library\cc2430\HAL\source  
$PROJ_DIR$\..\..\..\Library\cc2430\EB\include  
$PROJ_DIR$\..\..\..\Library\cc2430\EB\source  
$PROJ_DIR$\..\include
```

Damit findet der Compiler die nötigen h- und c-Files. Um den C-Spy Debugger von IAR nutzen zu können, muss in den Projektoptionen unter Linker → Output → Format der Radio Button (Debug information for C-SPY) aktiviert werden.

### 4.3 Oszillator

Der DAC und die PWM Ausgabe wird mit einem externen Oszillator von Linear Express (siehe Datenblatt im Anhang A.1.3) gesteuert. Für die Abtastung benötigen wir ein Frequenz von 8kHz. Mit Hilfe des Widerstandes R kann die Frequenz eingestellt werden. Der Wert berechnet sich wie folgt:

$$R = \frac{10^{11}}{f_{OSC} \cdot N} = \frac{10^{11}}{8kHz \cdot 100} = 125k\Omega \rightarrow 124,7k\Omega \quad (4.1)$$

Wir wählten aufgrund der Normreihen einen Widerstand von 124,7kΩ, daraus resultiert eine Oszillatorfrequenz von 8.02kHz. Da beide EB mit der selben Frequenz arbeiten, stört das nicht weiter. Besonders ist darauf zu achten dass der P2.0 vom SoC nicht mit dem P2.0 von vom EB verbunden ist, sondern mit P2.4 beschriftet ist.

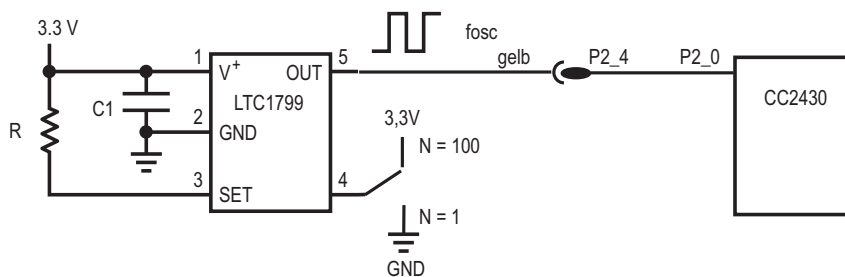


Abbildung 4.4: Beschaltung Oszillator

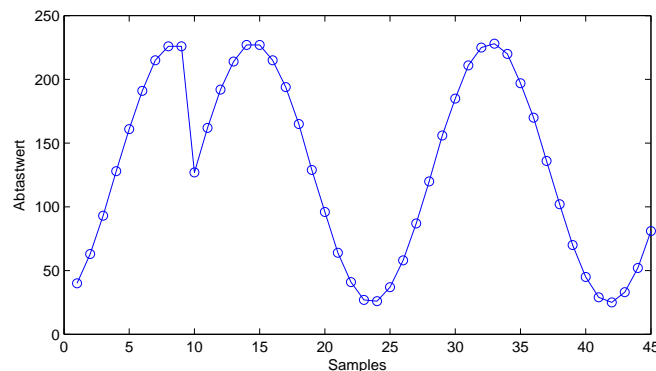
# 5 Abtastung und Quantisierung

## 5.1 Abtastung

Das Sprachsignal wird bereits vom SmartRF04EB von der Audio-Buchse an den Pin P0.0 vom Chip CC2431 gelegt. Somit kann das Signal direkt eingelesen werden. Bei der Sprachübertragung ist es üblich, Frequenzen bis 4 kHz zu verwenden. Die erforderliche Abtastrate beträgt somit 8 kHz, damit das Abtasttheorem eingehalten wird.

$$f_s \geq 2 * f_{MAX}$$

Für die Abtastung verwenden wir den ADC (Analog Digital Converter), welchen wir zuerst mit dem Timer1 triggern wollten. Der Timer hatte eine Timerperiode von 125us, was der Abtastrate von 8 kHz entsprach. Jedoch wurde der Timer1 Interrupt nicht regelmässig aufgerufen. Beim Einlesen eines Sinus kann man die Fehler, welche sich durch Phasensprünge ausdrücken, gut erkennen (siehe Abbildung 5.1).



**Abbildung 5.1:** Fehler bei der Abtastung

Das Problem liegt beim Timer 1. Wenn der Timer 1 läuft und währenddessen mit dem SFR (Special Function Register) gearbeitet wird, setzt der Timer teilweise einfach aus. Diesen Fehler haben wir lange nicht bemerkt und erst als wir das Error File des CC2430 hatten, kamen wir dem Problem näher. Ein Auszug dieses Error Files ist im Anhang zu finden (A.1.4).

Um diese Fehler zu verhindern haben wir den ADC extern über den Pin P2.0 getriggert. Um diesen Clock zu erzeugen, benutzen wir den Oscillator LTC1799 (siehe Kapitel 4.3), welchen wir mit einem Widerstand auf unsere Frequenz von 8kHz einstellen. Der externe Interrupt wird regelmässig ausgelöst und somit entstehen auch keine Fehler mehr bei der Abtastung.

```

//-----
// ADC Einstellungen
//-----

ADC_ENABLE_CHANNEL(0);
ADC_SEQUENCE_SETUP(ADC_REF_P0_7 | ADC_10_BIT | ADC_AIN0); // 10 Bit Auflösung

ADCCON1 &= ~0x10; // Trigger from P2_0
ADCCON1 &= ~0x20; //
ADCCON1 |= 0x40; // ADC starten
  
```

Bit	Name	Reset	R/W	Description
7	EOC	0	R H0	End of conversion Cleared when both ADCH and ADCL has been read. If a new conversion is completed before the previous data has been read, the EOC bit will remain high.  0 conversion not complete 1 conversion complete
6	ST	0	R/W1	Start conversion. Read as 1 until conversion has completed  0 no conversion in progress 1 start a conversion sequence if ADCCON1.STSEL = "11" and no sequence is running.
5:4	STSEL[1:0]	11	R/W	Start select. Selects which event that will start a new conversion sequence.  00 External trigger on P2_0 pin. 01 Full speed. Do not wait for triggers. 10 Timer 1 channel 0 compare event 11 ADCCON1.ST = 1
3:2	RCTRL[1:0]	00	R/W	Controls the 16 bit random generator. When written "01" or "10", the setting will automatically return to "00" when operation has completed.  00 Normal operation. (13x unrolling) 01 Clock the LFSR once (no unrolling). 10 Seeding from modulator. NOTE: The ADC must be running in order for the seeding to start. 11 Stopped. Random generator is turned off.
1:0	-	11	R/W	Reserved. Always set to 11.

Abbildung 5.2: ADCCON1 (0xB4) - ADC Control 1 [2]

## 5.2 Quantisierung

Alle 125us wird ein neuer Wert des Sprachsignals vom ADC eingelesen. Dabei werden 10 Bit eingesetzt, jedoch werden nur die höchsten 8 Bit verwendet (mehr ergibt keinen Sinn, weil die PWM-Ausgabe eine maximale Auflösung von 8 Bit hat), die niedrigsten 2 Bit werden weggelassen um Störsignale zu verringern. Das Sprachsignal kann Werte von 0V bis 3.3V annehmen, was den ADC-Werten 0 bis 255 entspricht (siehe auch Abbildung 5.1). Da nicht unnötige Rechenzeit des Chips verloren werden soll, wird für das Einlesen der Daten ein DMA verwendet.

### 5.2.1 DMA (Direct Memory Access)

Mittels einem DMA (übersetzt: direkter Speicherzugriff) können Daten von verschiedenen Speicherorten innerhalb des Chip transferiert werden. Da dies parallel zum CPU geschieht, geht dafür keine Rechenzeit verloren. Hier wird er benutzt um die Daten, welche vom ADC eingelesen werden, direkt in den Speicher zu schreiben. So lässt sich die Ausführungsgeschwindigkeit erhöhen.

```
//-----
// DMA Einstellungen
//-----

// Setzt die Source Adresse des DMA, wo die Daten geholt werden sollen
// Wird auf des High Register des ADC gesetzt
SET_WORD(dmaAdc.SRCADDRH, dmaAdc.SRCADDRL, &X_ADCL);

// Setzt die Destination Adresse des DMA, wo die Daten hingespeicher werden
// Wird zuerst auf den Buffer 1 (adcValues_1) gesetzt
SET_WORD(dmaAdc.DESTADDRH, dmaAdc.DESTADDRL, &adcValues_1);

// Legt die Transferlänge fest, entspricht gerade den Anzahl Samples
SET_WORD(dmaAdc.LENH, dmaAdc.LENL, BUFF);

// Der DMA soll als Transferlänge gerade die Anzahl Counts verwenden
dmaAdc.VLEN          = VLEN_USE_LEN;

// Dem DMA wird die höchste Priorität zugeordnet, damit er nicht unterbrochen wird
dmaAdc.PRIORITY      = PRI_HIGH;

// Die Transferlänge soll 8 Bit betragen
dmaAdc.M8            = M8_USE_8_BITS;
// Interrupt Maske ist wird ausgeschaltet
dmaAdc.IRQMASK       = FALSE;
// Die Zieladresse soll immer um eins erhöht werden,
//damit der Buffer fortlaufend gefüllt wird
dmaAdc.DESTINC       = DESTINC_1;
// Die Sourceadresse soll immer gleich bleiben
dmaAdc.SRCINC        = SRCINC_0;
// DMA ist auf Kanal 0 des ADC getriggert
dmaAdc.TRIG          = 21;
// DMA transferiert getriggert vom ADC die Anzahl Samples
dmaAdc.TMODE         = TMODE_SINGLE_REPEATED;
// Transfergrösse
dmaAdc.WORDSIZE      = WORDSIZE_WORD;
// Dem DMA wird die Adresse seiner Konfigurationsdaten angegeben
DMA_SET_ADDR_DESCO(&dmaAdc);

// DMA wird aktiviert
DMA_ARM_CHANNEL(0);

//-----
```

Die Transferlänge ist auf 120 gesetzt, somit werden jeweils 120 Werte eingelesen. Sobald der DMA den Transfer abgeschlossen hat, wird das entsprechende Bit im `DMAIRQ` Register auf 1 gesetzt. Hier verwenden wir den Channel 0, somit muss das nullte Bit (`DMA_CHANNEL_0`) abgefragt werden (siehe Abbildung 5.3).

Wenn der Transfer abgeschlossen ist, muss die DMA Zieladresse neu gesetzt werden. Die Zieladresse ist der Buffer, der momentan nicht für die Auswertung der ADC Daten genutzt wird.

Bit	Name	Reset	R/W	Description
7:5	-	000	R/W0	Not used
4	DMAIF4	0	R/W0	DMA channel 4 interrupt flag. 0 : DMA channel transfer not complete 1 : DMA channel transfer complete/interrupt pending
3	DMAIF3	0	R/W0	DMA channel 3 interrupt flag. 0 : DMA channel transfer not complete 1 : DMA channel transfer complete/interrupt pending
2	DMAIF2	0	R/W0	DMA channel 2 interrupt flag. 0 : DMA channel transfer not complete 1 : DMA channel transfer complete/interrupt pending
1	DMAIF1	0	R/W0	DMA channel 1 interrupt flag. 0 : DMA channel transfer not complete 1 : DMA channel transfer complete/interrupt pending
0	DMAIF0	0	R/W0	DMA channel 0 interrupt flag. 0 : DMA channel transfer not complete 1 : DMA channel transfer complete/interrupt pending

Abbildung 5.3: DMAIRQ (0xD1) - DMA Interrupt Flag [2]

```
//-----
// Wenn Datentransfer beendet -> Zieladresse ändern
//-----

if((DMAIRQ & DMA_CHANNEL_0))
{
    // Bit in DMAIRQ Register muss zurückgesetzt werden
    DMAIRQ &= ~(DMA_CHANNEL_0);

    // Abfrage, in welchen Buffer gerade Daten eingelesen wurden
    if(dmaDest == adcValues_1)
    {
        // adcValues_1 (Buffer 1) kann für die Auswertung genutzt werden
        samples = adcValues_1;
        // adcValues_2 (Buffer 2) kann für den nächsten Transfer genutzt werden
        dmaDest = adcValues_2;
    }
    else // Daten wurden gerade in den Buffer 2 transferiert
    {
        // adcValues_2 (Buffer 2) kann für die Auswertung genutzt werden
        samples = adcValues_2;
        // adcValues_1 (Buffer 1) kann für den nächsten Transfer genutzt werden
        dmaDest = adcValues_1;
    }

    // DMA Destinationadresse muss neu gesetzt werden
    // Destinationadresse ist der Buffer, der nicht für die Auswertung genutzt wird
    SET_WORD(dmaAdc.DESTADDRH, dmaAdc.DESTADDRL, dmaDest);
    ...
    ...
}
}
```

## 5.2.2 Auswertung der ADC-Daten

Die beiden Register `ADCL` und `ADCH` des ADC werden vom DMA an die eingestellte Zieladresse hingespeichert. Die tieferwertigen Bits werden in `ADCL` und die höherwertigen in `ADCH` abgespeichert (siehe Abbildung 5.4 und 5.5). Der Zeiger `samples` verweist auf den Speicherort der am nächsten zu verarbeitenden Daten, also dorthin wo sich 120 eingelesene

ADC-Daten befinden. Das ADCL Register liegt vor dem ADCH Register, da es sich um einen Intel Prozessor handelt, der mit Little Endian arbeitet.

Bit	Name	Reset	R/W	Description
7:2	ADC[5:0]	0x00	R	Least significant part of ADC conversion result.
1:0	-	00	R0	Not used. Always read as 0

Abbildung 5.4: ADCL (0xBA) - ADC Data Low [2]

Bit	Name	Reset	R/W	Description
7:0	ADC[13:6]	0x00	R	Most significant part of ADC conversion result.

Abbildung 5.5: ADCH (0xBB) - ADC Data High [2]

```
//-----
// Maskierung und Verschiebung der Samples in den Sendebuffer
//-----

sampI8=(INT8*)samples;
for(i=0;i<BUFF;i++)
{
    u=(((*sampI8<<1)&(INT8)0xFE)); sampI8++;
    sendBuffer[sendBuffNuIn][i]=u|((*sampI8>>7)&(INT8)0x01); sampI8++;
}

```

Bei der Maskierung muss darauf geachtet werden, dass der Prozessor mit Little Endian arbeitet. Die Samples werden Byteweise manipuliert und in den Sendebuffer kopiert. Das Bit 15 ist ein Vorzeichenbit und wird nicht verwendet. Das höchstwertige Bit (MSB) ist somit Bit 14 und für unsere Anwendung das tiefstwertige (LSB) Bit 7. Die Daten werden wie in Abbildung 5.6 aufgezeigt in den Sendebuffer geschrieben. Das Umschreiben in den Sendebuffer sieht auf den ersten Blick etwas ungewohnt aus. Es wurde darum so gemacht weil das Rechnen mit 8Bit-Operationen wesentlich schneller abläuft als mit 16Bit. Dadurch haben wir für das Umrechnen mehr als die Hälfte der Zeit eingespart.

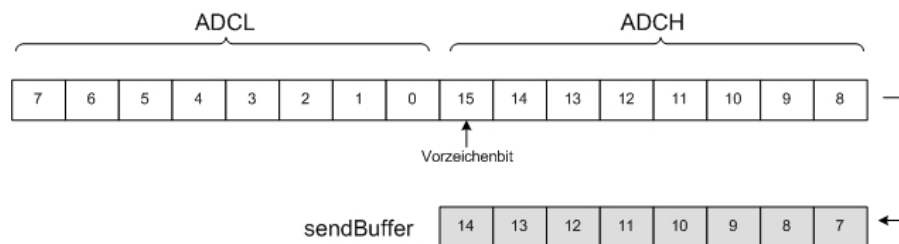


Abbildung 5.6: Maskierung der ADC Daten



## 6 Voice Activity Detection

Ein Nachteil des bisher verwendeten System ist die Störanfälligkeit gegenüber Umgebungsgeräuschen wie Beatmungsgerät und andere medizinische Überwachungsapparaturen. Um solche Geräusche von einer menschlichen Stimme unterscheiden zu können, müssen verschiedene Merkmale der Audiodaten untersucht werden. Typische Merkmale sind beispielsweise das Spektrum, gewonnen durch die Fouriertransformation, der Amplitudenverlauf und die Verteilung der Frequenzen in einem diskreten Zeitraum. Mit diesen Informationen lassen sich bereits einige typische Merkmale von Audiodaten beschreiben. Einfachstes Merkmal ist hier wahrscheinlich die Anzahl Nulldurchgänge (Zero Crossing Rate).

### 6.1 Zero Crossing Rate (ZCR)

Der ZCR Algorithmus zählt die Anzahl Nulldurchgänge der Audiodaten in einem Zeitfenster. Da bei unserem System ein Paket gerade 15ms (120 samples) Audiodaten enthält, werden die Nulldurchgänge in diesem Paket gezählt. Bei reinen Sinusschwingungen sind die Anzahl Nulldurchgänge gerade proportional zur Frequenz. In der Praxis kommen jedoch reine Sinusschwingungen nur selten vor. Jedoch lässt sich aus der ZCR schliessen ob die Audiodaten aus vorwiegend hohen oder tiefen Frequenzanteilen besteht. So lassen sich Sprachsignale die Zischlaute wie „s“ „z“ oder „sch“ (Geräusche wie sie von einem Beatmungsgerät entstehen könnten) deutlich von anderen Lauten wie Vokalen (typisch in Sprachdaten) unterscheiden.[1]

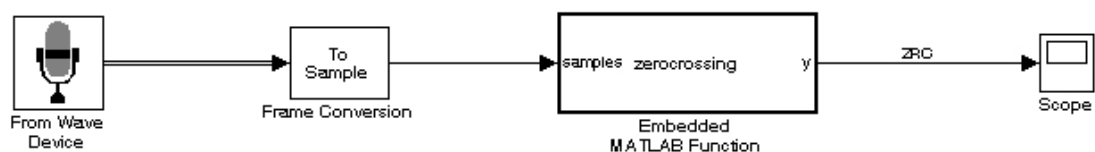


Abbildung 6.1: Simulink Model zur ZCR Messung

Der C-Code zur Embedded Matlab Function:

```
function y = zerocrossing(samples)
i=0;
for m = 1:120-1
    if sign(samples(m))==sign(samples(m+1))
    else
        i=i+1;
    end
end
y=i;
```

## 6.2 Root-Mean-Square (RMS)

Die RMS Kurve kann als die vom Menschen wahrgenommene Lautstärke dienen und ist einfach zu berechnen. Die allgemeine Formel zur Berechnung lautet wie folgt:

$$RMS = \sqrt{\frac{1}{N} \sum_{k=0}^{N-1} s^2(k)} \quad (6.1)$$

$s(k)$ : Audio Signal

$N$ : Anzahl der Samplewerte im Zeitfenster

Es werden also in einem Zeitfenster der Länge  $N$  alle Werte quadriert, aufsummiert und gemittelt.

Da der Rechenaufwand für das Dividieren und Wurzelziehen sehr rechenintensiv ist, wurde bewusst darauf verzichtet. Dafür die Schwellwerte für den RMS und die ZCR angepasst. Die Sample Werte die wir vom ADC erhalten haben einen Wertebereich von 0 bis 255. Durch die Vorspannung des Mikrophones erhalten wir in der Ruhestellung den Wert 128 (0x80) den wir vor dem Quadrieren zuerst abziehen müssen.

C-Code für die Berechnung der ZCR und des RMS:

```

double rms;
double zcr;

for(i=0;i<BUFF;i++)
{
  rms += ((byteSamplesArr[i]-(0x80)) * (byteSamplesArr[i]-(0x80)));
  if(i<BUFF-1)
  {
    if((byteSamplesArr[i]>(0x80))!=(byteSamplesArr[i+1]>(0x80)))zcr++;
  }
}

//rms = sqrt(rms/BUFF); //dauert zu lange! ca. 5ms
  
```

## 6.3 Simulation und Implementation auf dem SoC

### 6.3.1 Simulation mit Matlab

Um die ZCR und den RMS der Umgebungsgeräusche und der Stimme zu messen, erstellten wir mit Hilfe von Simulink ein Modell. Dabei verwendeten wir den Audioeingang des PC an dem ein handelsübliches Mikrophon angeschlossen wurde. Das Modell in Simulink ist in Abbildung 6.2 dargestellt.

Als Schwellwertschalter wurden zwei Relays verwendet. Bei Relay1 (ZCR) wurde die Schwelle auf 40 (ZCR) gesetzt und Relay2 (RMS) auf 0.003, was bei RMS nur ein Erfahrungswert ist und je nach Audioeingang des PCs variiert. Auf der Abbildung 6.3 ist ein 20 Sekunden langer Ausschnitt einer Aufzeichnung dargestellt.

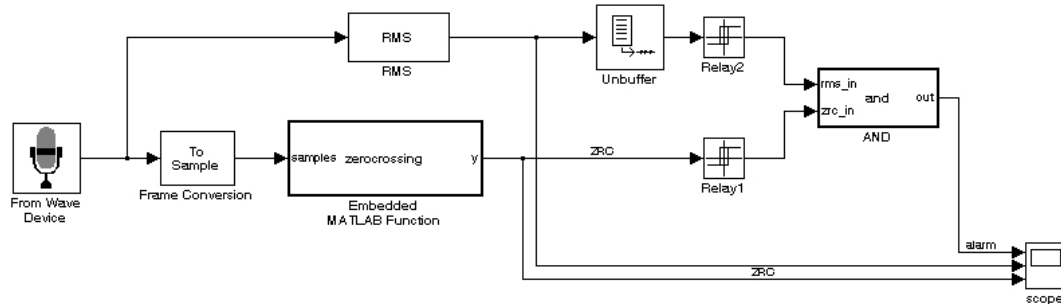


Abbildung 6.2: Simulation ZCR und RMS mittels Simulink

### 6.3.2 Alarmauslösung

Für jedes Paket, das 15ms Audiodaten enthält, wird der RMS und die ZCR berechnet und mit einem Schwellwert verglichen. Damit nicht bei jeder Spitze Alarm ausgelöst wird, werden die ausgelösten Alarme in der Variable `alarm_value` aufsummiert und beim Unterschreiten der Schwelle wieder abgebaut. Damit das System trotzdem schnell auf ausgelöste Alarme reagiert, wird der Alarm doppelt so schnell aufgebaut als abgebaut. Sobald `alarm_value` die Schwelle welche in `ALARM_SCHWELLE` festgelegt wurde erreicht, wird das Bit `alarm` gesetzt und an der Zentrale angemeldet. Auf dem SoC haben wir das wie folgt realisiert:

```

//-----
// Schwellwerteschalter für die Voice Activity Detection
//-----
..
BOOL alarm=0;           //Akkustisch Alarm ausgelöst?
..
if(!alarm)
{
    //Schwellwerte für VAD
    if((rms > 15000) && (zcr < 20))
    {
        //inkrement
        if(alarm_value < ALARM_SCHWELLE) alarm_value = alarm_value + 10;
    }
    else
    {
        //dekrement
        if(alarm_value > 5) alarm_value = alarm_value - 5;
        else alarm_value = 0;
    }

    //Falls Alarmschwelle erreicht
    if(alarm_value >= SCHWELLE)
    {
        alarm = TRUE;
    }
}
}

```

Als Vergleichswerte für den RMS und die ZCR die wir empirisch ermittelten, wählten wir für den RMS 15'000 und für die ZCR 20. Diese Werte müssten dann in der endgültigen akustischen Umgebung angepasst werden, oder mittels Potentiometer Eingestellt werden können. Eine weitere Möglichkeit wäre, die Schwellen adaptiv anzupassen.

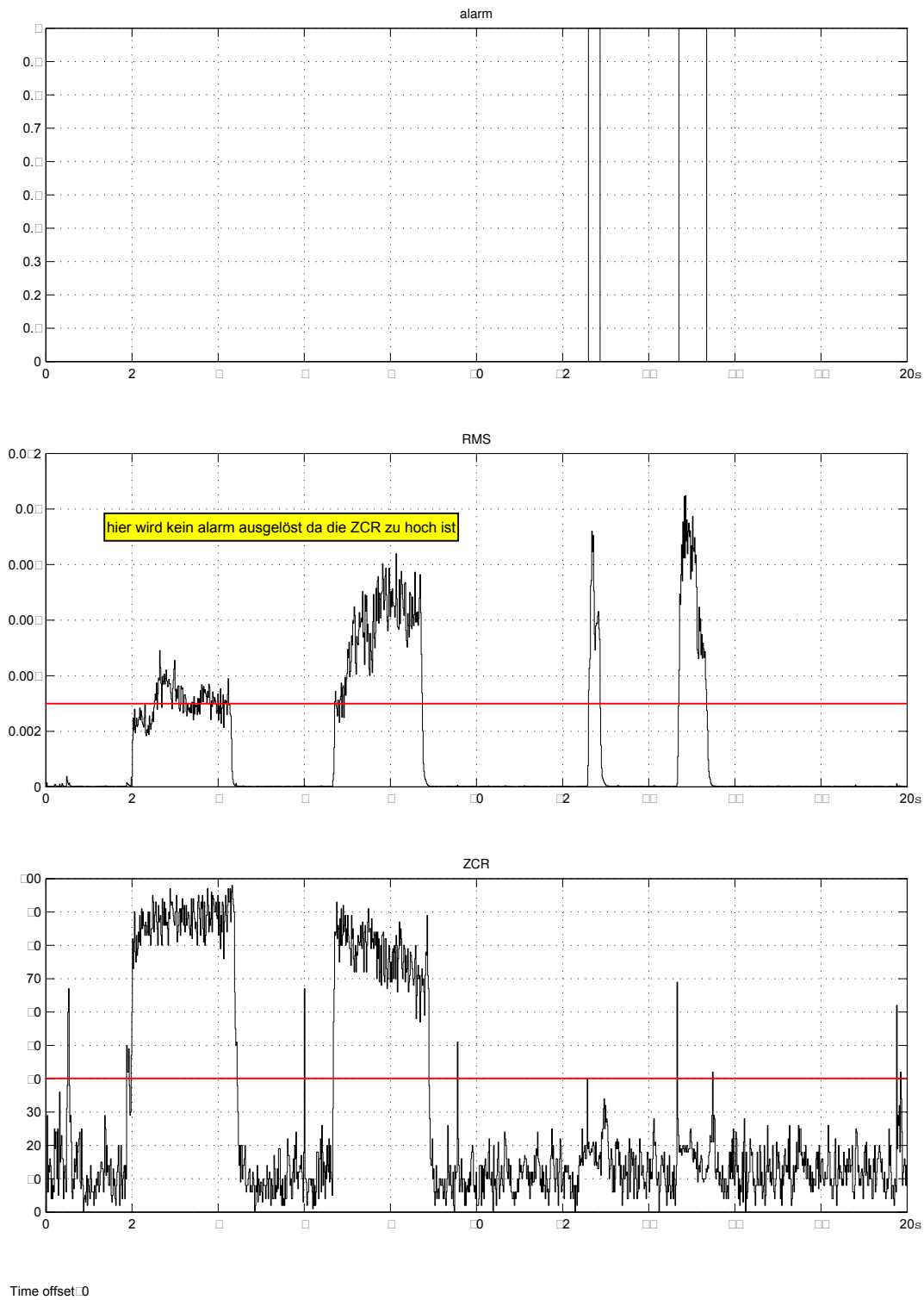


Abbildung 6.3: 20 Sekunden Aufzeichnung des Scopes

## 7 Audio Signalerzeugung

Zur Erzeugung des Audio Signals verwenden wir die vorgegebene Hardware auf dem EB. Das EB ist bereits mit einem Tiefpassfilter und einem Verstärker ausgestattet. An der 3,5mm Cinch Buchse kann direkt ein Headset eingesteckt werden und mittels Potentiometer (RT2) kann die Lautstärke verstellt werden. Für die Messungen verwendeten wir den Testpunkt (TP5).

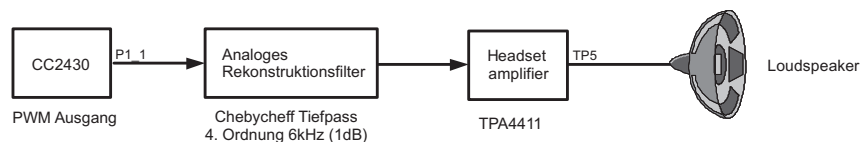


Abbildung 7.1: Blockdiagramm des Audioausganges auf dem EB

### 7.1 PWM als Digital Analog Konverter

Da auf dem Chip kein integrierter DAC (Digital Analog Converter) eingebaut ist, verwenden wir PWM (Pulsweitenmodulation) um unser digitales Signal zu konvertieren und auszugeben. Der Vorteil gegenüber eines herkömmlichen DAC ist dass für die Ausgabe lediglich einen Pin benötigt wird.

Die Trägerfrequenz ist bei der Pulsweitenmodulation konstant, mit der Pulsbreite variiert man dann die Leistung des Ausgangssignals, welche sich proportional zur Amplitude des Ausgangssignals verhält (siehe Abbildung 7.2). Filtert man dieses Signal mit einem analogen Tiefpassfilter, erhält man das rekonstruierte analoge Signal.

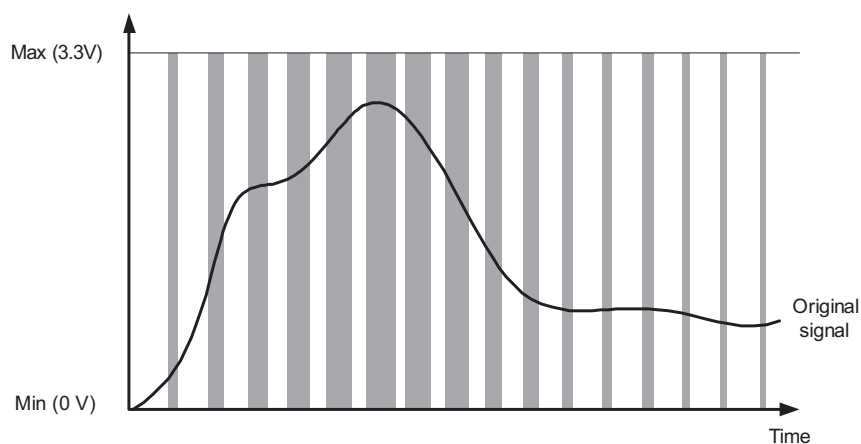


Abbildung 7.2: Pulsweitenmodulation [4]

Eine Analyse des Frequenzspektrums des PWM Signals ergibt, dass Peaks bei vielfachen der Trägerfrequenz des PWM Signals auftreten. Da sich auf dem EB bereits ein Ausgangsfilter mit der Grenzfrequenz von 6kHz befindet, müssen wir eine genügend hohe Frequenz für den PWM Ausgang wählen (Nach dem Nyquist Theorem mindestens doppelt so hoch wie die maximale Signalfrequenz). Da wir beim Einlesen mit einer Auflösung von 8bit arbeiten, wollen wir diese Auflösung auch bei der Ausgabe umsetzen. Dazu betreiben wir den Timer 4 im Freerunning Mode. Wie in Abbildung 7.3 ersichtlich zählt der Counter von 0 bis 255 und startet danach wieder bei 0.

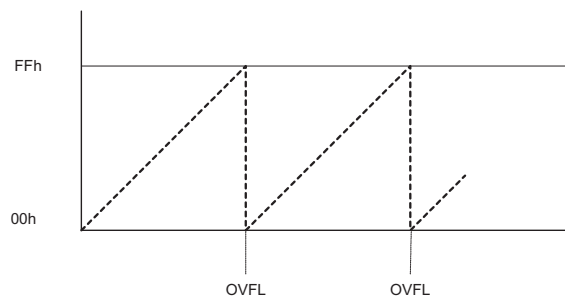


Abbildung 7.3: Timer 4 Free-running mode

Unser Chip wird mit einem 32Mhz Oszillator getaktet. Dies entspricht somit folgender PWM Periodendauer:

$$T_{PWM} = \frac{1}{32 \cdot 10^6 \text{ Hz}} \cdot 255 = 7.97 \mu\text{s} \quad (7.1)$$

Dies entspricht einer PWM Frequenz von  $f_{PWM} = 125,5 \text{ kHz}$  und lässt sich mit dem bereits vorhandenen analogen Filter problemlos filtern.

```

//-----
// Initialisierung des Timer 4 für den Freerunning Mode
//-----

void init_pwm (void)
{
  IO_FUNC_PORT_PIN(1,1,IO_FUNC_PERIPH); //Ausgabe an Pin P1_1
  T4CTL = 0x04; //free running
  T4CCTL0 = 0x00;
  T4CC0 = 0x01;
  T4CCTL1 = 0x24; //Clear output on compare-up
  T4CC1 = 0x01; //Value (Dies entspricht der Pulsbreite)
  T4IE = TRUE; //Interrupt für Timer 4 einschalten
  T4CC0 = 0xff; //Setting timer value.
  T4CTL = (T4CTL | 0x10); //Starten
}
  
```

Um nun die Samples im Abstand von  $125 \mu\text{s}$  ausgeben zu können, benötigen wir einen zweiten Timer. Dafür verwendeten wir zuerst den Timer 3 der alle  $125 \mu\text{s}$  eine Interrupt Subroutine aufruft und das Register `t4cc1` mit dem Wert des Samples setzt. Aus den selben Gründen wie bereits bei Timer 1 im Kapitel 5.1 erwähnt, funktionierte auch der Timer 3 nicht zuverlässig. Die einzige Möglichkeit ist, die Sample Ausgabe auch extern zu Takten. Dafür verwenden wir den selben Pin (P2.0), mit dem wir auch den ADC takten. Dazu benutzen wir den externen Interrupt an Port 2.

## Audio Signalerzeugung

```
//-----
// Initialisierung des externen Interrupts an Port 2 für die Ausgabe
//-----

void init_audio_out (void)
{
    P2IFG = 0x00;
    PICTL |= 0x04;           // Enable interrupt from P0_1 low
    PICTL |= 0x20;           // Falling edge gives interrupt
    INT_ENABLE(INUM_P2INT, INT_ON); // Enabling interrupt from P0
    INT_GLOBAL_ENABLE(TRUE);

}

```

Nun wird immer bei fallender Flanke an Port 2 Pin 0 die Interrupt Subroutine `P2INT_VECTOR` aufgerufen. Mit dem Status Flag `P2IFG = 0x01` überprüfen wir nun, ob es sich um eine fallende Flanke an Pin 0 handelt. Danach können die Samples, die sich im Ringbuffer `buffer[outBuff][n]` befinden, gesetzt werden. Dabei verweist `outBuff` auf das aktuelle Segment im Ringbuffer und `n` auf das aktuelle Sample.

```
//-----
// ISR für die Audio Ausgabe
//-----

#pragma vector=P2INT_VECTOR
__interrupt void P2_IRQ(void){
    if(P2IFG & 0x01){

        EA = FALSE;
        INT_SETFLAG(INUM_P2INT, FALSE);

        BYTE pwm = buffer[outBuff][n]; // Zuweisung des jeweiligen PWM-Wertes

        if(pwm <=0)  pwm=1;
        if(pwm >=255)  pwm=254;

        while(T4CNT < 240); // Warten bis der Timer-Counter 240 erreicht hat

        T4CC1  = pwm;      // Setzen des PWM-Werts

        n++;
        if(n==BUFF)// Wenn am Ende eines Paketes angelangt
        {
            n=0;
            outBuff++;
            if(outBuff == inBuff) // Gleiche Ausgabe wie Eingabe --> Fehler
            {
                outBuff--;      // Vorheriges Paket wird ausgegeben
            }
            if(outBuff==NOBUFF){ // Steuerung des Ringbuffers für die Empfangenen Daten
                if (inBuff!=0)
                    outBuff=0;
                else
                    outBuff=NOBUFF-1;
            }
        }
        P2IFG &= ~0x01;
    }
    P2IF = FALSE;
    EA = TRUE;
}

```

## Probleme mit der PWM Ausgabe

Bei der Ausgabe des Signales entstanden immer wieder Fehler bei der PWM, welche als Knistern wahrzunehmen waren. Die Fehler traten nur sporadisch auf und waren schwer mit dem KO zu erfassen. Nach langem Debuggen fanden wir heraus, dass der Timer 4 im freerunning Mode die Werte im Compare Register  $\tau_{4CC1}$  sofort übernimmt und nicht erst am Ende der PWM-Periode. Ist nun der Counter ( $\tau_{4CNT}$ ) noch nicht an der Stelle vorbei gekommen, an dem er den Ausgang umschalten soll ( $\tau_{4CC1}$ ), und wird der Wert hinter den  $\tau_{4CNT}$  gesetzt, schaltet der Ausgang erst bei der nächsten Periode um.

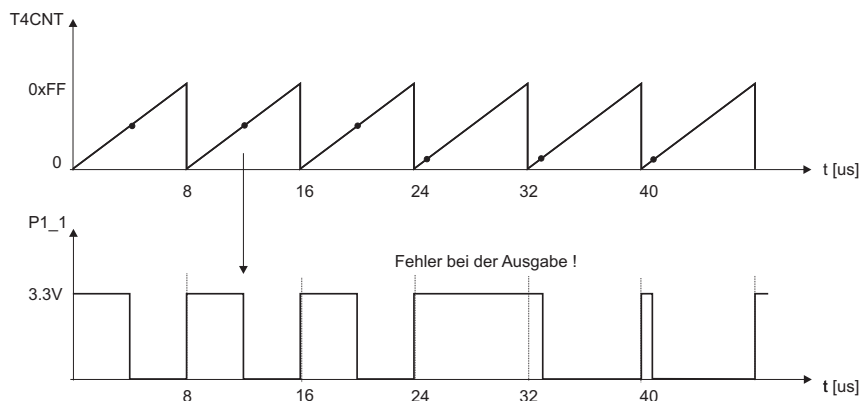


Abbildung 7.4: PWM Fehler bei der Ausgabe

Diese Fehler sind als leichtes Knistern wahrzunehmen. Als Lösung dieses Problemes müssten wir den Timer 4 in einem anderen Modus betreiben oder um die PWM Frequenz beizubehalten eine Warteschleife in der ISR einbauen, bei dem das neue Sample erst nach Ablauf der aktuellen Periode setzt:

```

..
while(T4CNT < 240); // Warten bis der Timer-Counter 240 erreicht hat
T4CC1 = pwm;      // Setzen des PWM-Werts
..

```

Jetzt wird der neue Wert erst gesetzt, wenn der Counter von Timer 4 den Wert 240 erreicht hat. Somit ist sichergestellt, dass der Counter bereits an der Stelle vorbei gekommen ist, an dem er den Ausgang schalten musste. Den Wert 240 haben wir deshalb gewählt, da der Mikrokontroller noch einige Instruktionen für die Abfrage und das setzen des Registers benötigt.

## 7.2 Analoges Rekonstruktionsfilter

Zur Filterung des PWM Signals verwenden wir das Tiefpassfilter, welches sich bereits auf dem EB befindet. Leider handelt es dabei um ein 6kHz Tiefpassfilter. Für unser Ausgangssignal müssten wir eigentlich ein Tiefpassfilter mit einer Grenzfrequenz von 4kHz oder tiefer wählen. Bei einem Redesign der Schaltung müsste dies beachtet werden. Bei dem Filter auf dem EB handelt es sich um ein Chebycheff Tiefpass mit folgenden Spezifikationen:

## Audio Signalerzeugung

---

- Chebycheff Tiefpass
- 4. Ordnung
- 1 dB Ripple im Durchlassbereich
- 6 kHz cutt-off Frequenz

Das Schema des Audio Interface ist im Anhang A.1.1 zu finden. Nach dem Chebycheff Tiefpassfilter wird das Signal auf einen Stereo Headphone Treiber A.1.2 geführt. Der Treiber entfernt den vorhandenen DC Anteil des Signals, um es dann auf einem Headphone ausgeben zu können. Der Treiber hat eine maximale Leistung von 80mW pro Kanal. Die SNR ist bei 40mW Ausgangsleistung mit 98dB angegeben.



## 8 C-Programm

Wir beschreiben hier nicht das gesamte C-Programm im Detail, sondern nehmen Bezug auf die wichtigsten Programmkomponenten und deren Ablauf. Die Listings der Programme für die Zentrale (Slave) und den Patienten (Master) sind auf der CD zu finden.

### 8.1 Jitter Buffer

Um Schwankungen bei der Funkübertragung zu kompensieren, implementierten wir einen Jitterbuffer (siehe Abbildung 8.1), den wir als Ringbuffer betreiben, um so die Verzögerungen auszugleichen. Jedes Segment enthält 15ms Sprachdaten. Dabei entsteht allerdings eine kleine End-zu-End Verzögerung von mehr als  $4 \cdot 15\text{ms}$ , die jedoch kaum bemerkbar ist.

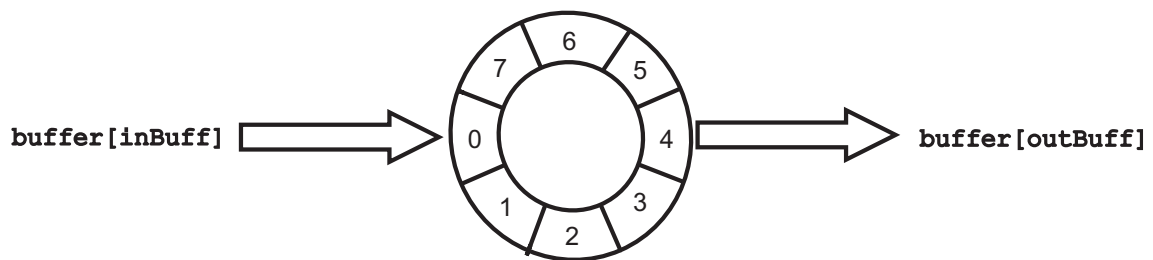


Abbildung 8.1: Jitter Ringbuffer

```
//-----  
// Deklarationen für den Ringbuffer  
//-----  
  
#define BUFF 120  
#define NOBUFF 8  
BYTE buffer[NOBUFF][BUFF]; //Jitter Buffer  
BYTE outBuff; //Zeiger auf Abspielbuffer  
BYTE inBuff; //Zeiger auf Empfangsbuffer
```

Der Jitterbuffer besteht aus einem zweidimensionalen Array, die Anzahl Reihen entspricht der Anzahl Segmente (`NOBUFF`) und die Länge aus der Anzahl Samples (`BUFF`) in einem Paket. Sobald Daten empfangen worden sind, werden diese mittels `memcpy` in den Empfangsbuffer kopiert und `inBuff` um eins inkrementiert. Die Empfangs- und Sendefunktionen werden im Kapitel 8.4 genauer erläutert.

```

//-----
// Empfangene Daten in den Ringbuffer schreiben
//-----

if((sppRxStatus == RX_COMPLETE))
{
    pBuffer = rxDaten.payload;
    length = rxDaten.payloadLength;
    sender = rxDaten.srcAddress;

    if((length == BUFF) && (sender == RECEIVER_ADDR))
    {
        memcpy(inbuffer,pBuffer,BUFF); //Daten in Eingangsbuffer kopieren

        inBuff++;                       //Ringbuffer inkrementieren
        if(inBuff==NOBUFF)
        {inBuff=0;}
    }

    sppReceive(&rxDaten);              //Empfang wieder aktivieren
}
  
```

## 8.2 Patient (Master)

Erste Versuche bei der Kommunikation zwischen der Zentrale und dem Patienten haben nicht zuverlässig funktioniert, da die beiden Sendefunktionen ständig kollidierten. Wir entschieden uns schliesslich, eine Master/Slave Beziehung zwischen den beiden Teilnehmern zu implementieren.

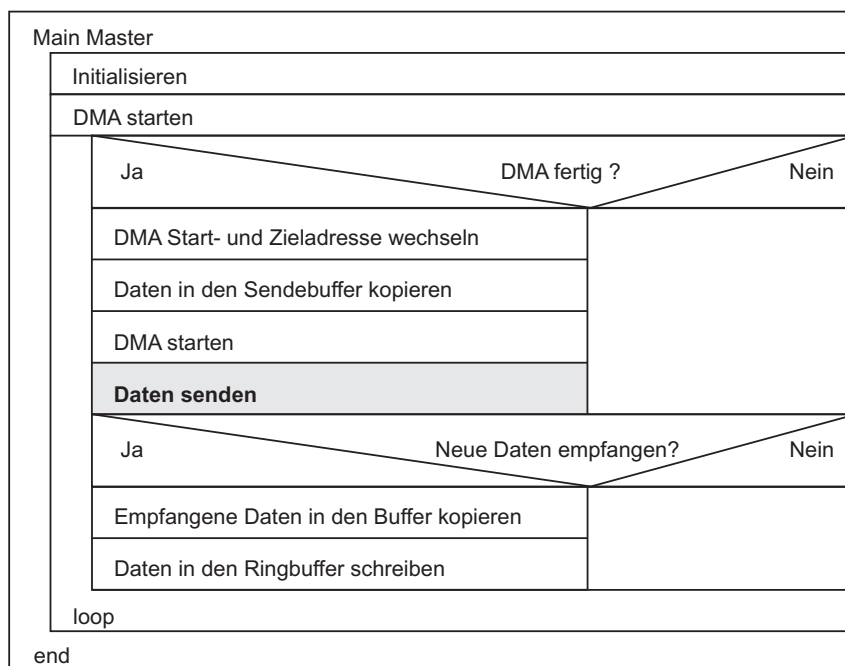


Abbildung 8.2: Struktur des C-Programmes beim Patienten (Master)

Da beim Patienten ständig das Signal analysiert werden muss, läuft der DMA bereits beim Einschalten des EBs. Somit ist der Takt, mit dem die Daten im Falle eines Alarmes verschickt werden müssen, schon vorgegeben. Der Patient fungiert somit als Master, das heisst die Sprachdaten werden zuerst versendet, und erhält erst dann von der Zentrale die Sprachdaten. Werden keine Daten vom Patienten versendet, kommen auch keine Daten von der Zentrale zurück. Damit stellen wir sicher, dass die beiden Sendefunktionen nicht kollidieren. Der Ablauf des C-Programmes ist in Abbildung 8.2 dargestellt.

### 8.2.1 Kommunikation mit der Zentrale

Da die Kapazität der Zentrale und des Luftraumes begrenzt ist, und um sicherzustellen dass keine Kollisionen entstehen, müssen die Gespräche zwischen den verschiedenen Patienten koordiniert werden. Dafür definierten wir verschiedene Bytes, welche der Zentrale signalisieren, welcher Patient gerade einen Alarm pendent hat und mit der Zentrale kommunizieren möchte. Der Verbindungsaufbau ist in Abbildung 8.3 dargestellt und ist eine Anlehnung an den SIP Standard.

```
//-----
// Deklarationen für die Kommunikation
//-----

#define READY          0xCC //Daten können gesendet werden
#define BYE            0xBB //Verbindungsabbau
#define INVITE        0xAA //Verbindungsaufbau
#define BUSY          0xDD //Besetzt
#define HELLO         0xEE //Verbindungskontrolle
```

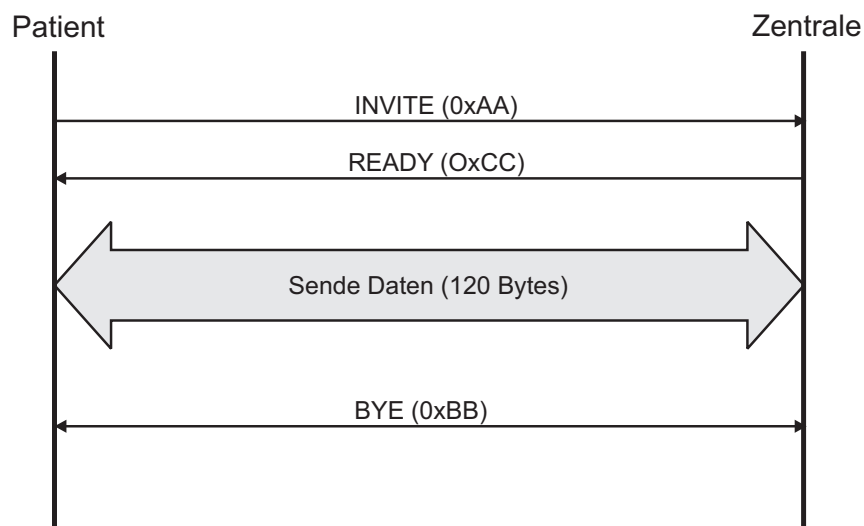


Abbildung 8.3: Kommunikationsprotokoll

Der Ablauf kann in der Abbildung 8.4 verfolgt werden. Sobald das EB eingeschaltet wird (Punkt RESET im Diagramm), werden DMA und Sendeteil initialisiert. Wird nun beim Patienten durch die Voice Activity Detection ein Alarm ausgelöst, wird das Bit `alarm` auf `TRUE` gesetzt und der Zentrale mittels `INVITE` gemeldet. Danach wird gewartet bis neue

Daten empfangen werden. Falls innerhalb einer Zeitspanne kein Signalisationsbyte von der Zentrale empfangen wurde, wird davon ausgegangen dass die Zentrale das `INVITE` nicht erhalten hat, danach wird das `INVITE` erneut an die Zentrale gesendet bis die Anzahl Versuche, die in `TRIES` festgelegt wurde, erreicht ist. Danach wird wieder in den Ruhezustand geschaltet und auf den nächsten Alarm gewartet.

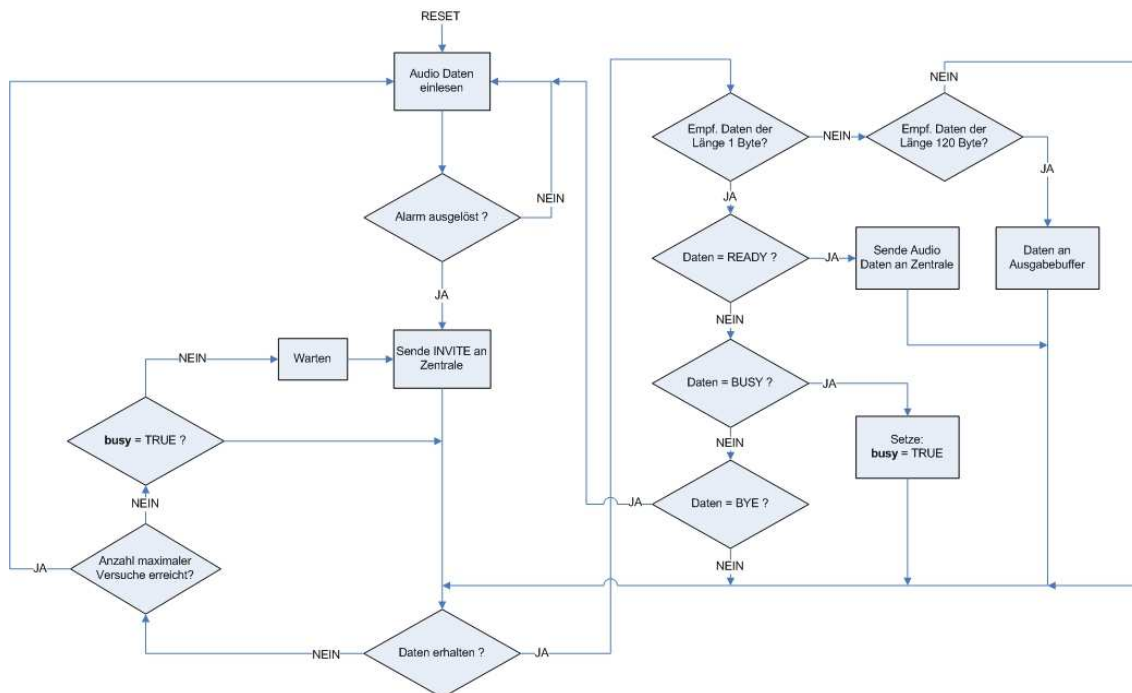


Abbildung 8.4: Flowchart C-Programm Patient

## 8.3 Zentrale (Slave)

Im Gegensatz zum Patienten schaltet die Zentrale den DMA erst dann ein, wenn sie vom Patienten einen Alarm erhält. Die Daten werden, sobald ein Paket fertig eingelesen ist, in einen Sende-Ringbuffer (siehe Kapitel 8.3.1) geschrieben. Wenn die Zentrale ein Datenpaket mit Sprachdaten vom Patienten erhalten hat, werden diese in einen Ausgabe-Ringbuffer abgelegt und die Daten des Sende-Ringbuffers werden an den Patienten gesendet (siehe Abbildung 8.5).

Dies wird darum so gehandhabt, um einen Konflikt bei der Übertragung zu vermeiden. Es wird sichergestellt dass der Patient gerade erst gesendet hat und noch genug Zeit ist, die Daten der Zentrale zu senden, solange der Kanal noch frei ist.

### 8.3.1 Sende-Ringbuffer

Der Sende-Ringbuffer funktioniert vom Prinzip her gleich wie der Ringbuffer für die Ausgabe (siehe Kapitel 8.1). Jedoch ist er nur halb so gross, um nicht unnötig Speicherplatz zu verschwenden.

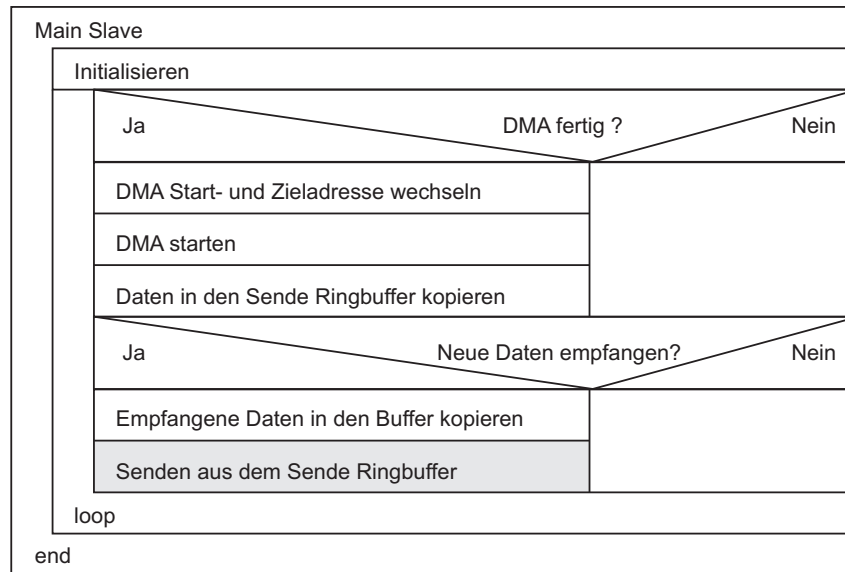


Abbildung 8.5: Programmablauf Zentrale (Slave)

```

//-----
// Deklarationen für den Sende-Ringbuffer
//-----

#define BUFF 120
BYTE sendBuffer[4][BUFF]; // Send Buffer
BYTE sendBuffNuIn; // Nummerierung für Sende-Buffer Eingabe
BYTE sendBuffNuOut; // Nummerierung für Sende-Buffer Ausgabe
  
```

### 8.3.2 Kommunikation mit dem Patienten (Master)

Aus den selben Gründen wie der Patient (siehe Kapitel 8.2.1) muss der Datenaustausch mit dem jeweiligen Patienten zum richtigen Zeitpunkt erfolgen, um den Austausch der Daten zu ermöglichen. Der Ablauf der Kommunikation mit dem Patienten ist in Abbildung 8.6 illustriert. Sobald Daten empfangen wurden, wird überprüft ob es sich um ein Sprachpaket (Länge 120 Byte) oder um ein Signalisationsbyte (Länge 1 Byte) handelt.

Wurde ein Signalisationsbyte empfangen, wird überprüft ob es sich um ein `bye` handelt. Wenn dies der Fall ist, wird der DMA für das Abtasten der Sprachdaten und die PWM-Ausgabe gestoppt. Wenn `ALARM` gesendet wurde, wird mit `onwork` überprüft, ob die Zentrale schon mit einem anderen Patienten verbunden ist. Wenn dies der Fall ist wird das Bit `busyBit` gesetzt. Andernfalls wird dem Patienten `ok` gesendet und der DMA für das Abtasten der Sprachdaten sowie die PWM-Ausgabe werden gestartet.

Wurden Sprachdaten empfangen, wird noch überprüft ob das Bit `busyBit` gesetzt ist. Falls dies der Fall ist, wird dem entsprechenden Patienten signalisiert, dass die Zentrale besetzt ist (`busy`). Zusätzlich wird noch überprüft, ob die Zentrale die Verbindung getrennt hat (`cancel`, wird gesetzt wenn Button gedrückt wird), was die Zentrale dazu veranlassen würde dem Patienten ein `bye` zu senden. Anschliessend werden die empfangenen Sprachdaten in den Ausgabe-Ringbuffer abgelegt und die eigenen Sprachdaten aus dem Sende-Ringbuffer dem Patienten, mit welchem man verbunden ist, gesendet.

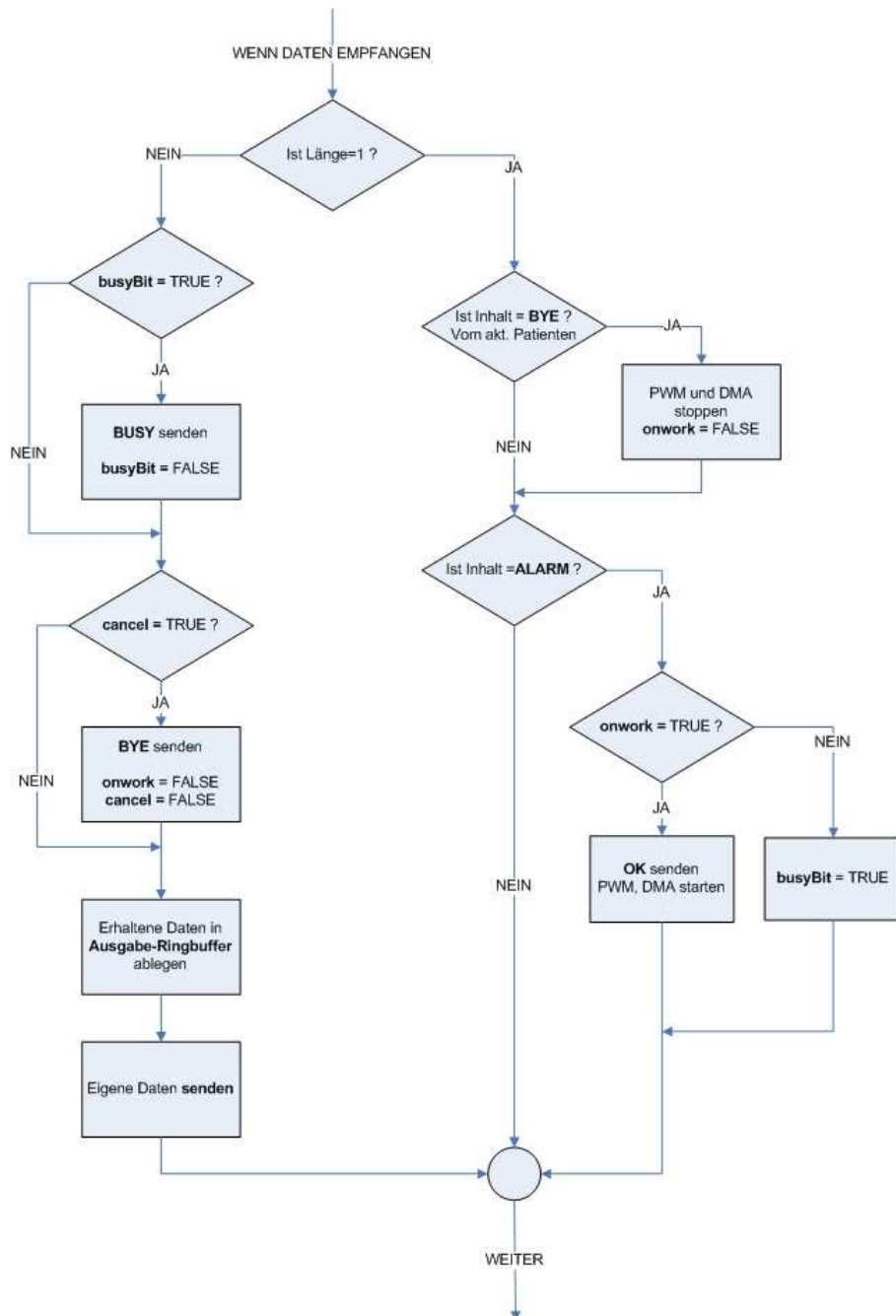


Abbildung 8.6: Ablauf wenn Daten erhalten

## 8.4 Daten Senden und Empfangen

Für den Datenaustausch über eine Funkübertragung zwischen der Zentrale und den Patienten benutzen wir Zigbee. Dies ermöglicht uns relativ einfach eine Kommunikation aufzubauen.

### 8.4.1 ZigBee

ZigBee IEEE 802.15.4 ist ein Standard zur RF Funkübertragung mit einer Reichweite von 64m x 64m. Die Übertragungsfrequenz beträgt 2.4 GHz und die Datenübertragungsrate 250kbps.[6] Neben einem niedrigen Energieverbrauch der Endgeräte zielt der Standard vor allem auf die Sicherheit und Stabilität der Verbindung ab. Weil er eine geringe Reaktionszeit vorweist ist er gut für Echtzeitanwendungen verwendbar, wie wir es benötigen.[5]

### 8.4.2 Uebertragung

Für das Senden und Empfangen benutzen wir das vorgegebene Protokoll `simple packet protocol` (SPP) mit dessen Funktionen aus `cul.h`. SPP ist ein einfaches und funktionelles Übertragungsprotokoll, welches von Interrupts gesteuert wird. Dank dem und der Tatsache dass für die Datentransfers der DMA benutzt wird, kann während dem Senden und Empfangen von Daten die Rechenzeit für andere Operationen benutzt werden.[7]

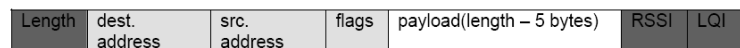


Abbildung 8.7: Simple Packet Protocol (SSP)[7]

In Abbildung 8.7 ist der Aufbau eines Pakets von SPP aufgezeigt. Die dunkelgrauen Felder werden von der Chip Hardware gesetzt, die hellgrauen sind vom SPP Protokoll und die weissen Felder stehen dem Benutzer zur Verfügung (125 Byte). `Length` ist die Anzahl Byte im Paket (ohne `Length` selber). Die Ziel- und Quelladressen sind je 1 Byte gross. Werden Pakete an die Adresse `0x00` gesendet, werden sie von allen empfangen. Das Byte `flags` beinhaltet ein ACK Bit, ein ACK Antwort (`DO_ACK`), einen Indikator zur erneuten Übertragung (`retransmission indicator`) und ein Sequenzbit. Das Sequenzbit und der `retransmission indicator` machen es für den Empfänger möglich, zwischen neuen und zuvor empfangenen Daten zu unterscheiden. RSSI bezeichnet die empfangene Signalstärke und LQI die Qualität der Verbindung. [7]

```
//-----
//Aufbau der Struct für senden und empfangen aus cul.h
//-----
typedef struct{
    BYTE payloadLength;
    BYTE destAddress;
    BYTE flags;
    BYTE *payload;
}SPP_TX_STRUCT;

typedef struct{
    BYTE payloadLength;
    BYTE destAddress;
    BYTE srcAddress;
    BYTE flags;
    BYTE payload[SPP_MAX_PAYLOAD_LENGTH + SPP_FOOTER_LENGTH];
}SPP_RX_STRUCT;
```

Zusammen mit den Strukturen (`SPP_RX_STRUCT`, `SPP_TX_STRUCT`), welche für den SPP-Standard angelegt werden, kann die Grösse der Daten bestimmt werden, welche pro Paket übertragen werden. Wenn der ganze Header zusammengerechnet wird, kommen wir mit einer Payload

von 120 Byte auf eine gesamte Datenmenge pro Paket von:

$$d = (1 + 1 + 1 + 1 + 120 + 1 + 1) * 8Bit = 1008Bit$$

Dies entspricht bei der angegebenen Übertragungsrate einer Übertragungszeit von:

$$t = \frac{1008Bit}{250kbps} \approx 4.1ms$$

Um die Kommunikation zu überprüfen, messen wir die Signalenergie (RSSI), welche auf unserer benutzten Frequenz vorhanden ist. Mit dem folgenden Befehl wird die Signalenergie auf dem Pin `PI_7` ausgegeben. `LOW` bedeutet dass Energie vorhanden ist, `HIGH` zeigt an dass gerade nichts gesendet wird.

```
IOCFG1 = 0x40;
```

Bei der Messung (Abbildung 8.8) der Übertragungszeit über die Messung der Signalenergie kommen wir auf einen Sendedauer von

$$t \approx 4.6ms$$

Dies stimmt ungefähr mit der Berechnung überein. Denn dort wurde die Zeit, welche für die Initialisierung gebraucht wird (ca.  $450\mu s$ ), nicht beachtet.

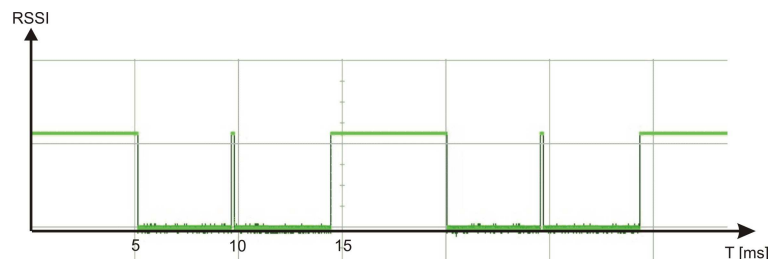


Abbildung 8.8: Signalstärke beim Senden

Ein grober Überblick über den Ablauf der Sende- und Empfangsfunktion wird in Abbildung 8.9 gegeben. Das Diagramm zeigt den unterschiedlichen Ablauf der Funktion, abhängig von den Einstellungen des Parameters `ACK`. Falls eine Bestätigung (`ACK`) verlangt wird und keine gesendet wird, werden nach einer bestimmten Zeit die Daten erneut gesendet.

## 8.5 Verbindungskontrolle

Die Verbindungskontrolle wird von der Zentrale ausgeführt, und wird nur in der Zeit durchgeführt, wenn keine Sprach-Verbindung zu einem Patienten besteht. Sie schickt allen Patienten ein `HELLO`, dies wird in einem gewissen Zeitabstand (5s) wiederholt. Wenn ein Patient das Signal empfangen hat, sendet er ein `HELLO` an die Zentrale zurück. Somit weiss die Zentrale, dass dieser Patient zu erreichen ist und setzt ihn in die Senderliste.

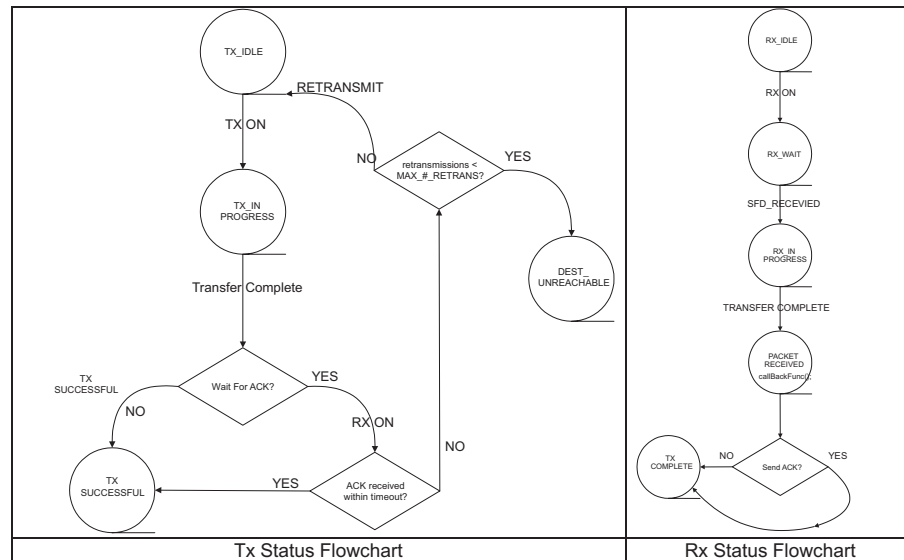


Abbildung 8.9: Flowchart Sende- und Empfangsfunktion [7]

```

//-----
// Verbindungskontrolle
//-----
if(*pBuffer==HELLO && !onwork)           // Wenn ein HELLO von einem Patienten erhalten
{
  BOOL newSender=TRUE;

  for(int r=0; r<anzSender; r++)         // Durchsuchen ob Patient schon eingetragen ist
  {
    if(senderListe[r]==sender)           // schon eingetragen
    {
      r=anzSender;                       // fertig
      newSender=FALSE;
    }
    senderTestListe[r] = TRUE;          // Eintrag dass Sender noch immer aktiv ist
  }

  if(newSender)                         // Wenn der Patient noch nicht eingetragen
  {
    senderListe[anzSender]=sender;        // Patient -> eintragen
    senderTestListe[anzSender] = TRUE;    // Sender ist aktiv
    anzSender++;
  }
}

```

Bei der Ausgabe (Abbildung 8.10) der aktiven Sender (Patienten) auf den Bildschirm (via RS232), werden die Sender welche nicht mehr aktiv sind aus der Liste gestrichen. So kann erkannt werden, wenn zu wenig Sender zu erreichen sind.

## 8.6 RS232-Schnittstelle

Weil man das LCD nach dem Gebrauch des externen Timers (siehe Kapitel 5.1) mit der vorhandenen Hardware nicht mehr gebrauchen kann, senden wir nun die Statusangaben

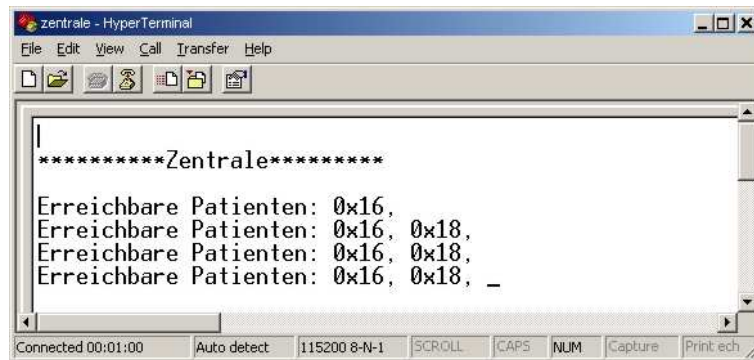


Abbildung 8.10: Verbindungskontrolle im Hyper Terminal

des Gerätes über die RS232-Schnittstelle des Evaluationsboards an den Computer. Der folgende C-Code zeigt die Initialisierung der RS232-Schnittstelle. Mit dem Befehl `printf` können dann einfach die gewünschten Daten gesendet werden.

```

//-----
//RS232 Initialisierung
//-----

IO_PER_LOC_UART0_AT_PORT0_PIN2345 (); // Pin Konfiguration einstellen
UART_SETUP(0, 115200, HIGH_STOP); // Setzen der Parameter
UTXOIF = 1; // Interrupt Flag setzen

printf((char*)"Zentrale\n"); // "Zentrale" an RS232 senden
  
```

Am Computer können dann die empfangenen Daten mit einem entsprechenden Programm angezeigt werden. Wir verwendeten dafür Hyper Terminal. Die Ausgabe sieht dann zum Beispiel so aus.

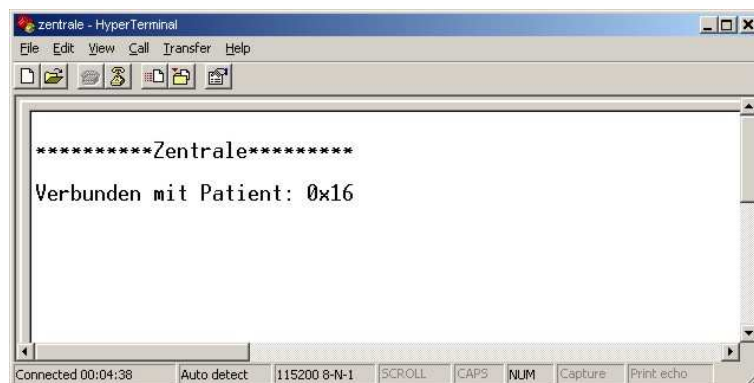


Abbildung 8.11: Hyper Terminal Anzeige

## 9 Messungen

Um den Sprachkanal testen zu können, und für die Fehlersuche suchten wir nach einem einfachen Verfahren um die Kommunikation zu testen. Dazu benützten wir Matlab und Simulink und die Soundkarte des PCs, welche für Audiosignale geeignet sind.

### 9.1 Messungen mit Simulink

Um einfach einen Sinus generieren zu können, verwendeten wir Simulink. Der Headphone Ausgang des PCs ist für Frequenzen im Audiobereich geeignet. Mit dem Modul „Sine Wave“ wird ein reiner Sinus mit der vorgegebenen Frequenz ausgegeben. „Chirp“ ist ein Generator mit Variabler Frequenz der in einem bestimmten Bereich abfährt. Die detaillierten Einstellungen sind in Abbildung 9.2 zu entnehmen.

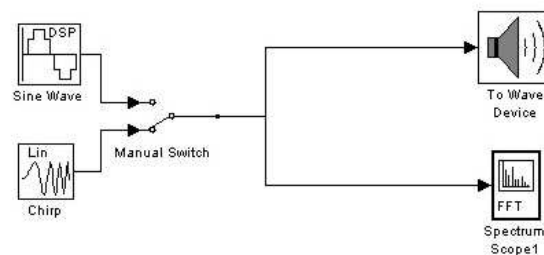
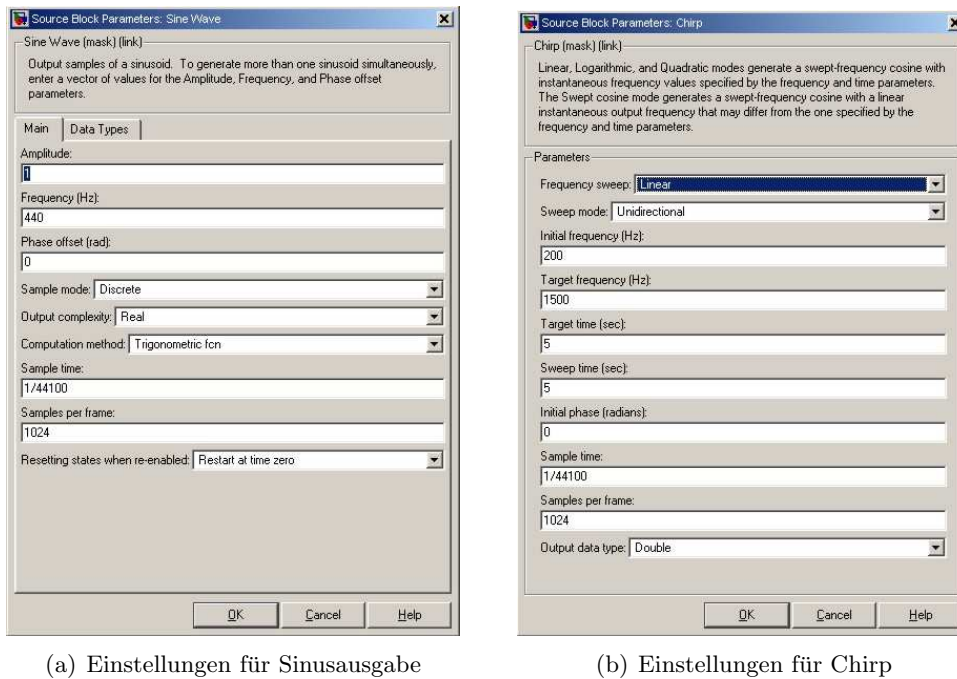


Abbildung 9.1: Sinusausgabe mittels Simulink

Den Sinus, den wir so mit Simulink erzeugen, können wir von der Headphone Buchse des PCs entnehmen. Mit einem 3,5mm Cinchkabel wird das Signal auf die „Mic“ Buchse des Patienten oder der Zentrale gespiesen. Hier ist wichtig, dass man darauf achtet, dass das Signal nicht übersteuert ist. Eventuell muss die Lautstärke am PC etwas zurück gestellt werden. Nach der Übertragung mittels ZigBee kann das Signal vom Gesprächspartner wieder von den „Phone“ Buchsen entnommen werden. Um das Signal analysieren zu können, benützen wir auch Simulink. Dazu verwenden wir den Mikrophone Eingang des PC. Auch hier ist es wichtig, dass das Signal nicht übersteuert ist. Mit Simulink und dem „Spectrum Scope“ ist es nun möglich die FFT zu betrachten.

Beim Spectrum Scope wählten wir folgende Einstellungen:

- Window Type: Hanning
- Window Sampling: periodic (1024 Samples per Frame)
- Number of spectral averages 200



(a) Einstellungen für Sinusausgabe

(b) Einstellungen für Chirp

Abbildung 9.2: Einstellungen für Sine Wave und Chirp

## 9.2 Quantisierungsrauschen

Bei der linearen PCM<sup>1</sup> bei einer sinusförmigen Spannung und Vollpegelsteuerung lässt sich das Quantisierungsrauschen wie folgt berechnen:

$$SNR = 10 \cdot \log\left(\frac{3 \cdot 2^{2 \cdot N}}{2}\right) = 10 \cdot \log\left(\frac{3 \cdot 2^{2 \cdot 8}}{2}\right) = 49.9 \text{ dB} \quad (9.1)$$

N entspricht der Anzahl Bits. Somit ist bei optimalen Bedingungen eine SNR von 49.9 dB zu erwarten.

## 9.3 Messresultate der SNR

Bei unseren Messungen kamen wir auf eine **SNR von etwa 45 dB** (siehe Abbildungen 9.3 und 9.4). Besonders auffällig sind die Aliasfrequenzen um 8 kHz, die durch die Abtastung entstehen und durch unser Filter zu wenig gedämpft werden (Messungen mit 16 kHz siehe Kapitel 9.4). Weitere Gründe für die eher etwas tiefe SNR sind sicherlich auch dass der ADC nicht voll angesteuert ist, und das nicht optimale Schaltungsdesign des EB, welche zusätzliches Rauschen erzeugen.

<sup>1</sup>Puls-Code-Modulation

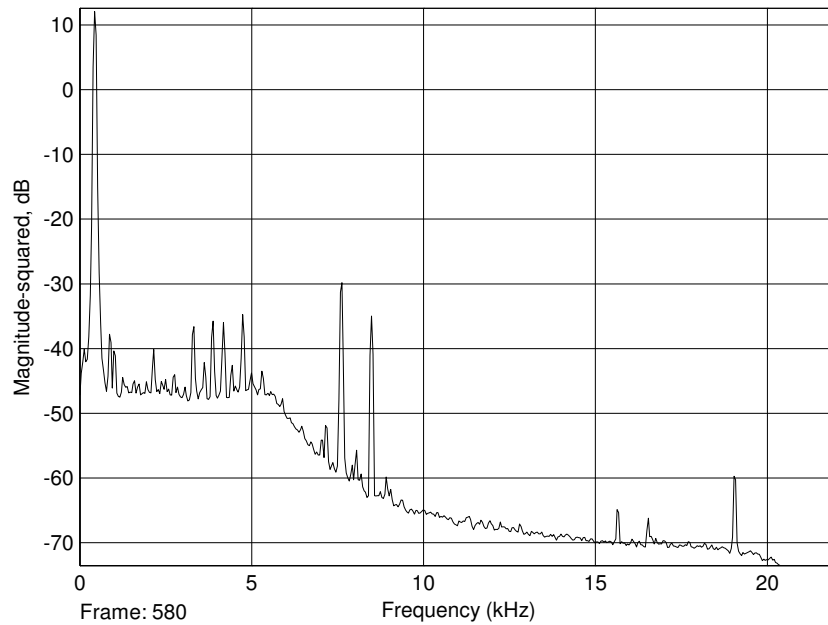


Abbildung 9.3: FFT des Kanals vom Patienten zur Zentrale

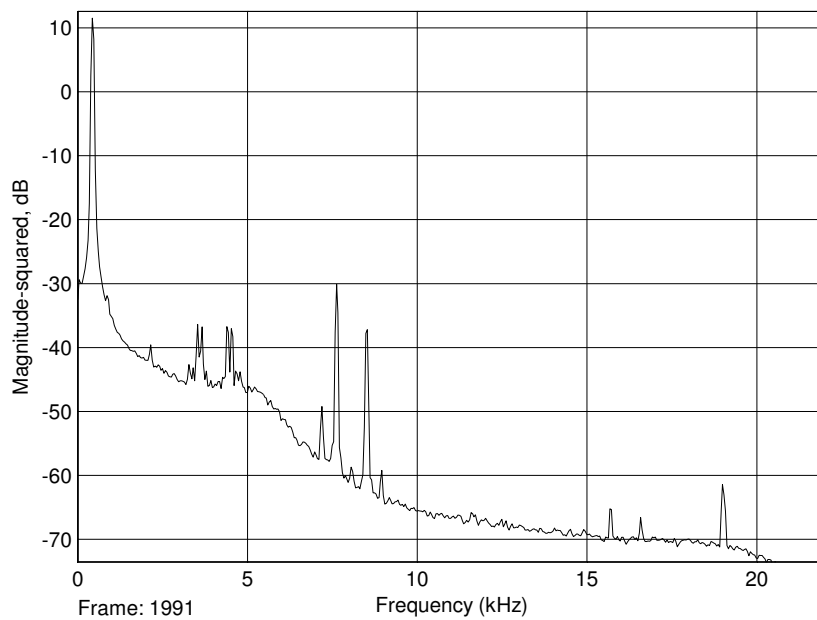


Abbildung 9.4: FFT des Kanals von der Zentrale zum Patienten

## 9.4 Versuch mit 16kHz mittels Up/Downsampling

Um trotzdem die vorhandene Hardware mit dem 6kHz Rekonstruktionsfilter benutzen zu können, haben wir zu Testzwecken den ADC und die Ausgabe mit 16kHz getaktet. Dabei erhalten wir vom ADC doppelt so viele Samples (240) und müssen diese aufgrund der begrenzten Übertragungsrate zuerst mittels Downsampling auf 120 Bytes herunter rechnen. Nach der Übertragung werden die Werte mit Upsampling wieder auf 240 Werte hoch gerechnet. Dafür verwenden wir eine lineare Interpolation.

```

//-----
// Berechnungen für das Up/Down Sampling
//-----

#define BUFF          120      //Anz. Bytes beim versenden
UINT16 temp[BUFF];
BYTE   temp2[BUFF];
BYTE   inbuffer[BUFF];

BYTE   byteSamplesArr[240];    //Samples vom ADC
BYTE   buffer[NOBUFF][240];   //Jitter Buffer
BYTE   inbuffer[BUFF];        //Empfangsbuffer
..

//Downsampling
for(i=0;i<BUFF;i++){
  temp[i] = byteSamplesArr[2*i]+byteSamplesArr[2*i+1];
  temp2[i] = temp[i]/2;
}
..

//Upsampling:
for(i=0;i<BUFF;i++){
  buffer[inBuff][2*i] = inbuffer[i];
  if(i<BUFF-1) buffer[inBuff][2*i+1] = (inbuffer[i]+inbuffer[i+1])/2;
}
buffer[inBuff][240-1] = inbuffer[120-1]; //letzten Wert wiederholen
..
  
```

### 9.4.1 Messresultate

Bei der Messung mit Simulink und dem „Spectrum Scope“ sind deutlich die zusätzlich entstandenen Frequenzen zu erkennen. Es hat sich herausgestellt, dass die Frequenzen durch die lineare Interpolation entstanden sind. Da die Interpolation alles lineare Operationen sind, dürften theoretisch keine zusätzlichen Frequenzen entstehen. Die FFT von einer 3,2 kHz Sinusschwingung mittels linearer Interpolation ist in Abbildung 9.5 dargestellt.

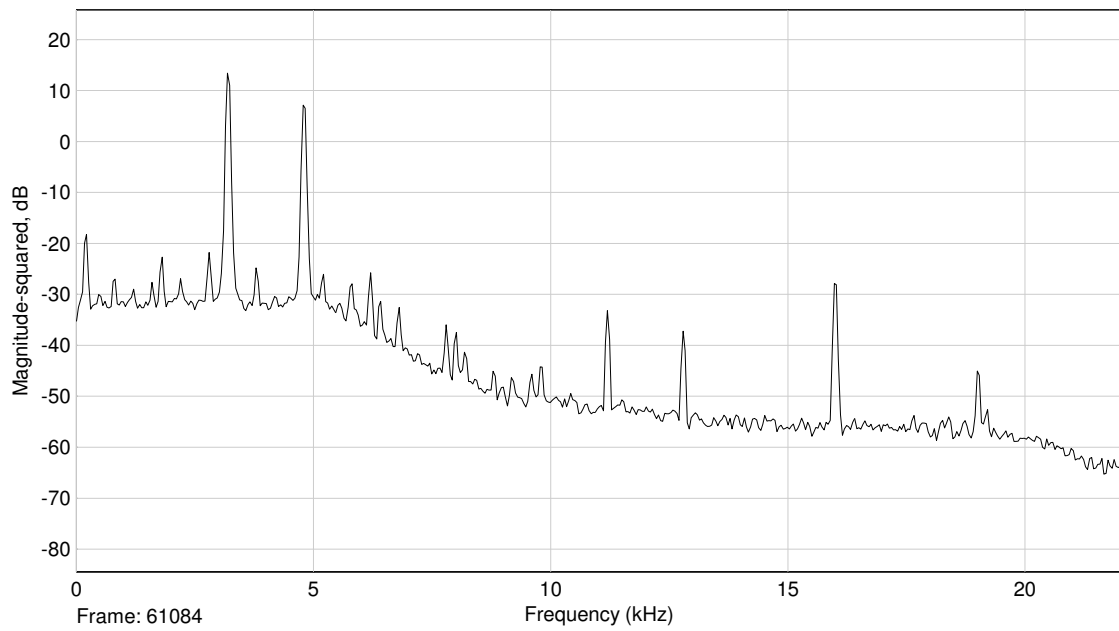


Abbildung 9.5: Sinus 3,2kHz mittels linearer Interpolation



# 10 Ausblick

Die Bidirektionale Verbindung zwischen den beiden Teilnehmer wurde soweit realisiert. Die Nachteile der existierenden Lösung von der Schweizerischen Gesellschaft für Muskelkranke konnten grösstenteils behoben werden. Zusätzlich müssten noch folgende Features implementiert werden:

- Getrennter Daten-/Signalisierungs- und Sprachkanal
- 2-3 Sprachkanäle (mittels mehrerer Frequenzen oder Sprachkompression)
- Akkubetrieb der Sender
- Reichweitenvergrößerung mittels automatischer Relaisfunktion
- Computeranschluss der Zentrale
- Redesign der Schaltung mit getrenntem Audioteil und den erforderlichen Filter und den korrekten Grenzfrequenzen. Sowie zusätzliche Taster und einem LCD Display bei der Zentrale. Eventuell Einbau in eine Telefon-Tischstation.



# 11 Rückblick

Während unserer Semesterarbeit lernten wir viel im Bereich der digitalen Signalverarbeitung. Themen, welche wir gerade in den Vorlesung behandelten, konnten gleich praktisch angewendet werden. Mit wenig Erfahrungen im Bereich der hardwarenahen Programmierung stürzten wir uns über die Handbücher und konnten bereits erste Erfolge feiern. Bald mussten wir aber feststellen, dass nicht immer alles so funktioniert wie es sollte. Manchmal beschäftigen wir uns bis spät Abends mit der Fehlersuche. Immer wieder konnten wir Probleme beheben und schätzen unser Laborjournal, in welchem wir unsere Erkenntnisse und Fortschritte dokumentierten. Wir möchten uns bei Prof. Dr. Guido Schuster und bei Reto Ansorge für Ihre Betreuung während der Arbeit bedanken.

Rapperswil, 7. Juli 2006

Landolt Simon

Reichmuth Ronald







## A.1.2 TPA4411 Stereo Headphone Driver



TPA4411

www.ti.com

SLOS430–AUGUST 2004

### 80-mW CAPLESS STEREO HEADPHONE DRIVER

#### FEATURES

- Ground-Referenced Outputs Eliminate DC-Bias Voltages on Headphone Ground Pin
  - No Output DC-Blocking Capacitors
    - Reduced Board Area
    - Reduced Component Cost
    - Improved THD+N Performance
    - No Degradation of Low-Frequency Response Due to Output Capacitors
- Wide Power Supply Range: 1.8 V to 4.5 V
- 80-mW/Ch Output Power into 16-Ω at 4.5 V
- Independent Right and Left Channel Shutdown Control
- Short-Circuit and Thermal Protection
- Pop Reduction Circuitry
- Space Saving Pb-Free Packages
  - 20-pin, 4 mm × 4 mm ThinQFN
  - 16-ball, 2 mm × 2 mm WCSP (Product Preview)

#### APPLICATIONS

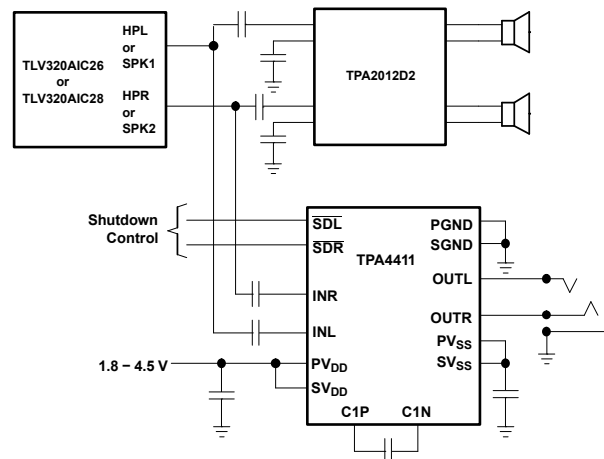
- Notebook Computers
- CD / MP3 Players
- Smart Phones
- Cellular Phones
- PDAs

#### DESCRIPTION

The TPA4411 is a stereo headphone driver designed to allow the removal of the output dc-blocking capacitors for reduced component count and cost. The TPA4411 is ideal for small portable electronics where size and cost are critical design parameters.

The TPA4411 is capable of driving 80 mW into a 16-Ω load at 4.5 V. The TPA4411 has a fixed gain of -1.5 V/V and headphone outputs have ±8-kV IEC ESD protection. The TPA4411 has independent shutdown control for the right and left audio channels.

The TPA4411 is available in a 20-pin, 4 mm × 4 mm ThinQFN package.



Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this data sheet.

UNLESS OTHERWISE NOTED this document contains PRODUCTION DATA information current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.

Copyright © 2004, Texas Instruments Incorporated

### A.1.3 LTC1799 Oscillator



LTC1799

1kHz to 33MHz  
Resistor Set SOT-23 Oscillator

#### FEATURES

- One External Resistor Sets the Frequency
- Fast Start-Up Time: < 1ms
- 1kHz to 33MHz Frequency Range
- Frequency Error  $\leq 1.5\%$  5kHz to 20MHz ( $T_A = 25^\circ\text{C}$ )
- Frequency Error  $\leq 2\%$  5kHz to 20MHz ( $T_A = 0^\circ\text{C}$  to  $70^\circ\text{C}$ )
- $\pm 40\text{ppm}/^\circ\text{C}$  Temperature Stability
- 0.05%/V Supply Stability
- 50%  $\pm 1\%$  Duty Cycle 1kHz to 2MHz
- 50%  $\pm 5\%$  Duty Cycle 2MHz to 20MHz
- 1mA Typical Supply Current
- 100 $\Omega$  CMOS Output Driver
- Operates from a Single 2.7V to 5.5V Supply
- Low Profile (1mm) SOT-23 (ThinSOT™ Package)

#### APPLICATIONS

- Low Cost Precision Oscillator
- Charge Pump Driver
- Switching Power Supply Clock Reference
- Clocking Switched Capacitor Filters
- Fixed Crystal Oscillator Replacement
- Ceramic Oscillator Replacement
- Small Footprint Replacement for Econ Oscillators

#### DESCRIPTION

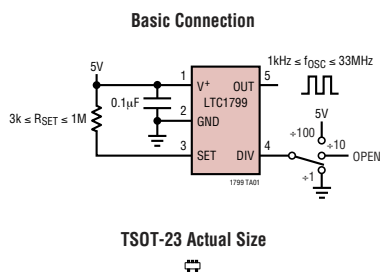
The LTC<sup>®</sup>1799 is a precision oscillator that is easy to use and occupies very little PC board space. The oscillator frequency is programmed by a single external resistor ( $R_{SET}$ ). The LTC1799 has been designed for high accuracy operation ( $\leq 1.5\%$  frequency error) without the need for external trim components.

The LTC1799 operates with a single 2.7V to 5.5V power supply and provides a rail-to-rail, 50% duty cycle square wave output. The CMOS output driver ensures fast rise/fall times and rail-to-rail switching. The frequency-setting resistor can vary from 3k to 1M to select a master oscillator frequency between 100kHz and 33MHz (5V supply). The three-state DIV input determines whether the master clock is divided by 1, 10 or 100 before driving the output, providing three frequency ranges spanning 1kHz to 33MHz (5V supply). The LTC1799 features a proprietary feedback loop that linearizes the relationship between  $R_{SET}$  and frequency, eliminating the need for tables to calculate frequency. The oscillator can be easily programmed using the simple formula outlined below:

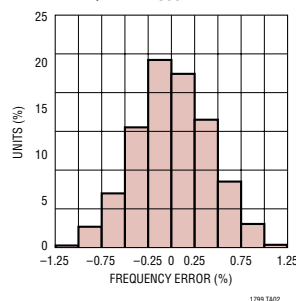
$$f_{osc} = 10\text{MHz} \cdot \left( \frac{10\text{k}}{N \cdot R_{SET}} \right), N = \begin{cases} 100, & \text{DIV Pin} = V^+ \\ 10, & \text{DIV Pin} = \text{Open} \\ 1, & \text{DIV Pin} = \text{GND} \end{cases}$$

LTC and LT are registered trademarks of Linear Technology Corporation. All other trademarks are the property of their respective owners. ThinSOT is a trademark of Linear Technology Corporation. Protected by U.S. Patents including 6342817 and 6614313.

#### TYPICAL APPLICATION



Typical Distribution of Frequency Error,  
 $T_A = 25^\circ\text{C}$  ( $5\text{kHz} \leq f_{osc} \leq 20\text{MHz}$ ,  $V^+ = 5\text{V}$ )



1

## A.1.4 Errata-File Ausschnitt

### 12 Timer 1, 3, and 4 interrupts are ignored when writing SFR registers

#### 12.1 Description

If interrupts from timers 1, 3, and 4 are triggered in the same clock cycle as any SFR register is written, the interrupt flag may not be set and the interrupt will be ignored. If A triggers produced by the same events as these interrupts will also be ignored in these situations.

#### 12.2 Suggested workaround

There is no workaround for this problem.



# Abbildungsverzeichnis

3.1	Projektkonzept . . . . .	9
3.2	System Überblick und Signalverlauf . . . . .	10
4.1	SmartRF04EB Main Evaluation Board [8] . . . . .	11
4.2	CC2430EM . . . . .	12
4.3	Aufbau des CC2431 [2] . . . . .	12
4.4	Beschaltung Oszillator . . . . .	14
5.1	Fehler bei der Abtastung . . . . .	15
5.2	ADCCON1 (0xB4) - ADC Control 1 [2] . . . . .	16
5.3	DMAIRQ (0xD1) - DMA Interrupt Flag [2] . . . . .	18
5.4	ADCL (0xBA) - ADC Data Low [2] . . . . .	19
5.5	ADCH (0xBB) - ADC Data High [2] . . . . .	19
5.6	Maskierung der ADC Daten . . . . .	19
6.1	Simulink Model zur ZRC Messung . . . . .	21
6.2	Simulation ZCR und RMS mittels Simulink . . . . .	23
6.3	20 Sekunden Aufzeichnung des Scopes . . . . .	24
7.1	Blockdiagramm des Audioausganges auf dem EB . . . . .	25
7.2	Pulsweitenmodulation [4] . . . . .	25
7.3	Timer 4 Free-running mode . . . . .	26
7.4	PWM Fehler bei der Ausgabe . . . . .	28
8.1	Jitter Ringbuffer . . . . .	31
8.2	Struktur des C-Programmes beim Patienten (Master) . . . . .	32
8.3	Kommunikationsprotokoll . . . . .	33
8.4	Flowchart C-Programm Patient . . . . .	34
8.5	Programmablauf Zentrale (Slave) . . . . .	35
8.6	Ablauf wenn Daten erhalten . . . . .	36
8.7	Simple Packet Protocol (SSP)[7] . . . . .	37
8.8	Signalstärke beim Senden . . . . .	38
8.9	Flowchart Sende- und Empfangesfunktion [7] . . . . .	39
8.10	Verbindungskontrolle im Hyper Terminal . . . . .	40
8.11	Hyper Terminal Anzeige . . . . .	40
9.1	Sinusaussage mittels Simulink . . . . .	41
9.2	Einstellungen für Sine Wave und Chirp . . . . .	42
9.3	FFT des Kanals vom Patienten zur Zentrale . . . . .	43
9.4	FFT des Kanals von der Zentrale zum Patienten . . . . .	43

9.5 Sinus 3,2kHz mittels linearer Interpolation . . . . . 45

# Literaturverzeichnis

- [1] Homepage Fraunhofer-Institut  
Internet:  
[http://www.ipsi.fraunhofer.de/merit/forschung/papers/Merkmale\\_digitaler\\_Audiodaten.pdf](http://www.ipsi.fraunhofer.de/merit/forschung/papers/Merkmale_digitaler_Audiodaten.pdf)  
21.01.07
  
- [2] Datenblatt CC2430,  
Internet: [www.chipcon.com/files/CC2430\\_Data\\_Sheet\\_rev1p03.pdf](http://www.chipcon.com/files/CC2430_Data_Sheet_rev1p03.pdf)  
13.01.07
  
- [3] Chipcon IAR usermanual 1.2  
Internet: [www.chipcon.com/files/Chipcon%20IAR%20IDE%20usermanual\\_1\\_2.pdf](http://www.chipcon.com/files/Chipcon%20IAR%20IDE%20usermanual_1_2.pdf)  
08.02.07
  
- [4] Chipcon Homepage Application Notes C1010  
[http://www.chipcon.com/files/AN\\_026\\_Wireless\\_Audio\\_1\\_0.pdf](http://www.chipcon.com/files/AN_026_Wireless_Audio_1_0.pdf)  
05.02.07
  
- [5] Zitat über ZigBee von Wikipedia  
<http://de.wikipedia.org/wiki/Zigbee>  
05.02.07
  
- [6] Daten des CC2431  
[http://www.chipcon.com/files/CC2431\\_Brochure.pdf](http://www.chipcon.com/files/CC2431_Brochure.pdf)  
05.02.07
  
- [7] Usermanual zu HAL/CUL/EB  
[http://www.chipcon.com/files/HAL\\_CUL\\_EB\\_Software\\_%20User\\_Manual\\_1\\_1.pdf](http://www.chipcon.com/files/HAL_CUL_EB_Software_%20User_Manual_1_1.pdf)  
05.02.07
  
- [8] CC2430ZDK User Manual  
Internet: [www.chipcon.com/files/2430ZDK\\_User\\_Manual\\_rev\\_1.2.1.pdf](http://www.chipcon.com/files/2430ZDK_User_Manual_rev_1.2.1.pdf)  
07.02.07
  
- [9] CC2430 Examples and user library files  
Internet: [http://www.chipcon.com/files/cc2430\\_lib\\_and\\_app\\_1.0.zip](http://www.chipcon.com/files/cc2430_lib_and_app_1.0.zip)  
08.02.07