Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Roman Plessl, Stéphane Racine

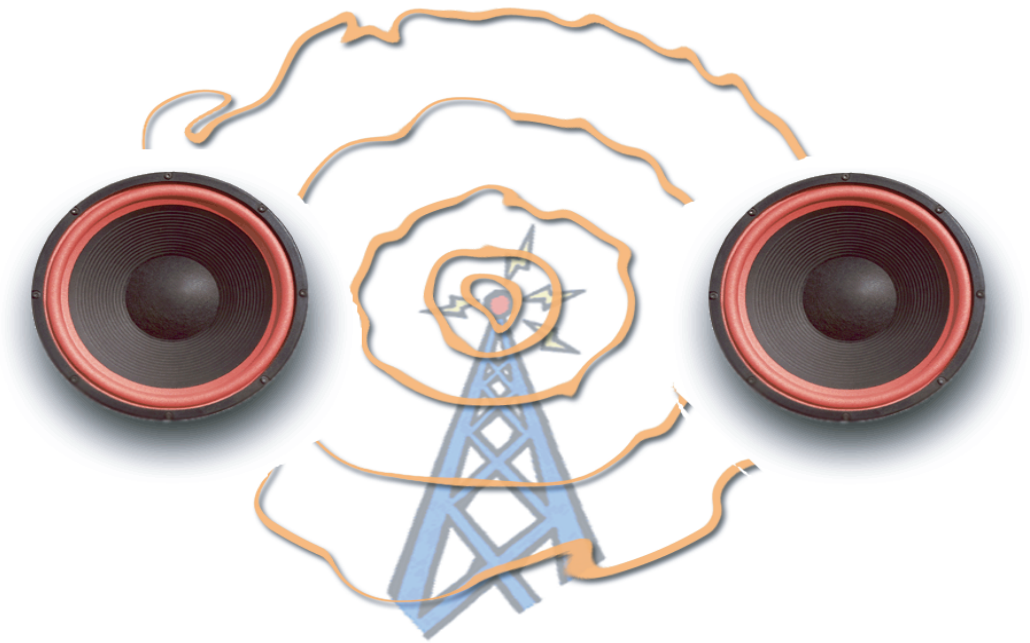
Wireless Stereo Speakers

Student Thesis SA-2003.20
Summer Term 2003

Tutor: Philipp Blum

Supervisor:
Prof. Dr. Lothar Thiele

31.7.2003



WIRELESS STEREO SPEAKERS

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Vorwort

Der Grundgedanke dieser Semesterarbeit basiert auf einer Idee von Philipp Blum, welcher sich seit längerem mit dem Thema der Uhrensynchronisation auf Standard Computern beschäftigt. Diverse Semester- und Diplomarbeiten wurden in diesem Feld beim Institut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich schon ausgeführt, jedoch fehlte bis anhin ein System, welches die unterschiedlich Methoden zur Synchronisation braucht und in einer gewissen Weise darstellen kann.

Das grosse Interesse unsererseits für dieses Thema und schlussendlich der kompetente Eindruck den unser Betreuer bei der ersten Besprechung hinterliess, haben uns dazu bewogen diese Semesterarbeit auszuwählen. Die präsentierte Aufgabenstellung schien in unseren Augen realistisch, um in der zur Verfügung stehenden Zeit umgesetzt werden zu können. Ebenfalls ist es uns beiden wichtig gewesen, dass die hier vorliegende Arbeit zu einem abgeschlossenen und fertigen Produkt führte.

Abschliessend möchten wir an dieser Stelle ganz herzlich unserem Betreuer Philipp Blum danken, für seine Geduld und seine allzeit vorhandene tatkräftige Unterstützung. Zürich, den 13. Juli 2003

Stéphane Racine

Roman Plessl

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Einleitung

Das menschliche Ohr kann zwar nicht die gleiche Menge an Information aufnehmen wie das Auge, dafür erbringt es ganz erstaunliche Leistungen in Bezug auf die Präzision mit der gewisse Phänomene wahrgenommen werden. Beispielsweise werden zufällige Verzögerungen von wenigen Mikrosekunden auf einem Kanal eines Stereosignals (man spricht von Jitter) bereits als Rauschen wahrgenommen.

Diese hohen Ansprüche an die Verbindungsqualität haben es bisher verhindert, dass Mehrkanal-Soundsysteme auf der Basis von billigen Standard Datennetzen gebaut werden konnten. In der Regel werden die verschiedenen Kanäle zentral (und damit synchron) von digital nach analog gewandelt und dann auf einem dedizierten Kabel mit höchst konstanter Laufzeit verteilt.

Alle momentan bestehenden Lösungen verfolgen im Prinzip den gleichen Ansatz: das Datennetzwerk soll irgendwie dazu gebracht werden, Audiodaten hoch synchron übertragen zu können. Mit dem Ziel bestehende Ethernet Infrastruktur mitzubenutzen scheint dies kaum vereinbar zu sein, da insbesondere wechselnde Lastverhältnisse im Netzwerk die Verbindungscharakteristika massiv verändern.

Deshalb verfolgt die Startup Firma BridgeCo zusammen mit dem TIK einen alternativen Ansatz: Bei der Verteilung der Daten wird bewusst eine schlechte Qualität in Kauf genommen. Synchron laufende Uhren in den Endknoten erlauben die Audiodaten direkt beim Ausspielen wieder zeitlich zu ordnen und somit die ursprüngliche Qualität des Klangbildes wieder herzustellen. Bei diesem alternativen Ansatz liegt nun das Problem bei den zu synchronisierenden Knoten. Am TIK wurden Algorithmen entwickelt, welche Standard PC Uhren über 802.11b Wireless LAN in der Grössenordnung von 10 Mikrosekunden synchronisieren.

1.2 Aufgabenstellung

Im Rahmen der nun hier vorliegenden Semesterarbeit sollte ein System gebaut werden, mit dem die Qualität einer Uhrensynchronisation hörbar gemacht werden kann.

Jedoch soll das ganze System nicht nur für einen ausgezeichneten Algorithmus funktionieren, sondern für beliebig andere. Die erstellte Lösung realisiert diese Trennung durch das Ausführen von vier parallel laufenden Prozessen, wie in der Abbildung 1.1 dargestellt ist.
In dem nun hier vorliegenden Demonstrator mussten verschiedene Teilaufgaben gelöst werden, auf welche hier überblickend eingegangen werden soll. Grundlegend für das System sind verteilte Uhrenobjekte, wie dies in Abbildung 1.1 schematisch dargestellt ist. Auf diesem Uhrenobjekt aufbauend ist ein Synchronisationsalgorithmus implementiert worden, welcher mehrere Uhren aufeinander abgleicht. Diese gestellten Uhren werden von einem Audio-Player für ein zeitsynchrones Ausspielen eines Audio-Files verwendet. Zur einfacheren Bedienung wurde ein graphisches Frontend als Aufsatz entwickelt.

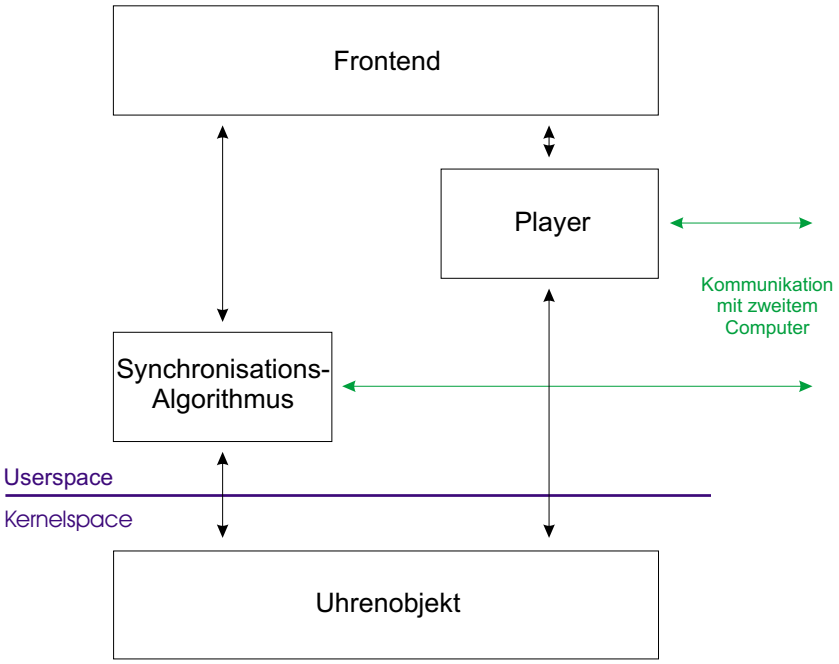


Abbildung 1.1: Die vier Teilprobleme (Uhrenobjekt, Uhrensynchronisation, Player und Frontend) wurden in nebenläufigen Prozessen realisiert. Mit den vertikalen Pfeilen ist die Kommunikation zwischen den einzelnen Prozessen symbolisiert, die horizontalen Pfeile entsprechen der Kommunikation zwischen den beteiligten Computern.

1.3 Aufbau des Berichtes

Im *ersten Kapitel* werden wir eine kurze Einführung in das Projekt geben und schon grob das System umschreiben.

Das *zweite Kapitel* befasst sich mit der Uhrensynchronisation.

Kapitel drei beschreibt den Audioplayer, welcher die synchronisierten Uhren anhand von Audio-Daten hörbar macht.

Der im *vierten Kapitel* beschriebene Kanalmischer ist vor allem für ein Ausprobieren und die Vortrags-Demonstration wichtig.

Im *Anhang* sind diverse Messresultate, Codebeschreibungen und Checklisten untergebracht.

Kapitel 2

Uhrensynchronisation

Die Uhrensynchronisation, wie wir sie in unserer Arbeit benutzt haben, hat zum Ziel, eine Uhr auf einem Client (oder mehreren Clients) mit einer Uhr auf einem Server abzugleichen. Im Idealfall laufen danach beide Uhren gleich schnell und zeigen immer die gleiche Zeit an. Der in Kapitel 3 beschriebene Player kann über eine definierte *API*[1] auf die Uhren zugreifen und damit den Zeitpunkt des Abspielens bestimmen.

2.1 Allgemeines zur Uhrensynchronisation

Im Folgenden wird oft die Rede von *Uhrenobjekten* (*Clockobjects*) sein. Darunter soll man sich in Gewissem Sinne eine Art *Uhr* vorstellen, von dem jeder Prozess eines Rechners die Zeit ablesen kann. Die Einheit der ausgelesenen Zeit ist sekundär. Ebenfalls ist es möglich dieses Uhrenobjekt zu stellen und somit dessen Wert und dessen Geschwindigkeit zu verändern.

Die Rechner mit ihren Uhrenobjekten sind über ein Computernetzwerk miteinander verbunden. Ein ausgezeichneter Referenz-Server soll nun die zu synchronisierende Zeit vorgeben. Durch Nachrichtenaustausch über besagtes Netzwerk werden die Uhren der Clients auf die Zeit und Laufgeschwindigkeit des Servers gestellt. Einen Überblick über das Gesamtsystem gibt Abbildung 2.1.

Für unsere Arbeit haben wir ein *ad-hoc Wireless LAN* mit zwei Rechnern (Client und Server) aufgebaut.

2.2 Synchronisationsalgorithmus

Der Schwerpunkt unserer Arbeit bestand im Wesentlichen aus der Entwicklung eines Demonstrators für präzise Uhrensynchronisation, der die Qualität des benutzten Algorithmus hörbar macht, und nicht in der Ausarbeitung eines guten Synchronisationsalgorithmus.

[1] Application Programming Interface

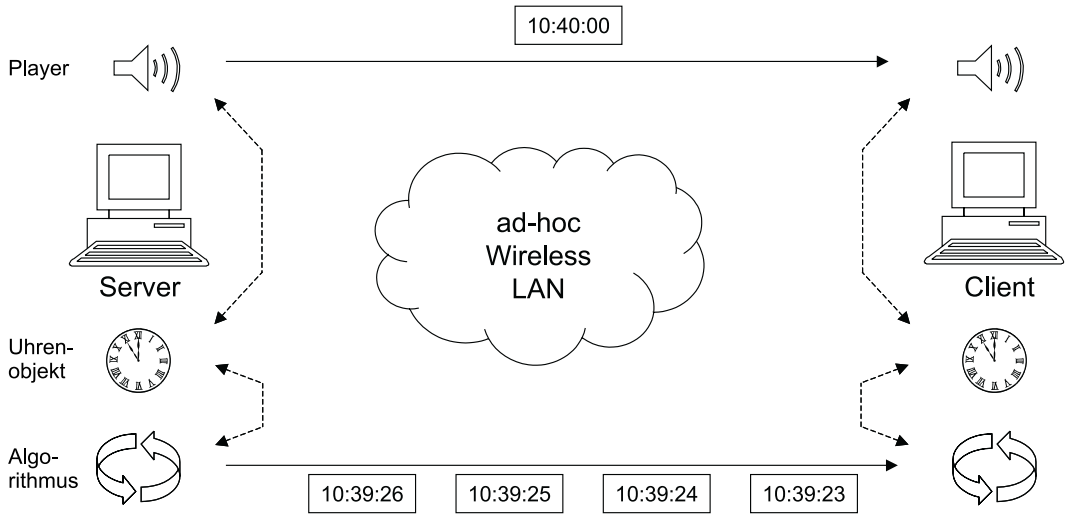


Abbildung 2.1: Systemüberblick: Server (links) und Client (rechts) mit jeweils dem Synchronisationsalgorithmus, der Uhr und dem Audioplayer. Gestrichelt dargestellt sind die Schnittstellen zum Uhrenobjekt.

Nichstdestotrotz musste ein Algorithmus benutzt werden, um den Demonstrator testen zu können. Die Wahl des Algorithmus fiel uns ziemlich leicht. Wir haben den von unserem Betreuer Philipp Blum und Men Muheim in [1] vorgestellen *Local Selection (LS) Algorithmus* implementiert.

2.2.1 Beschreibung des Algorithmus

Der Server, der als Zeitreferenz dient, liest über eine Schnittstelle den Wert der eigenen Uhr und versendet ihn über das Wireless LAN (als UDP-Datengramm). Der Client empfängt dieses Datenpaket und liest daraus den Zeitstempel. Dem ersten Paket folgt eine unbestimmte Anzahl an Datengrammen.

Folgende Aufzählung beschreibt den Algorithmus in groben Zügen:

- Zeitstempel und Uhr lesen liefert: $Zeit_{Server}(i)$, $Zeit_{Client}(i)$, $\beta_{Client}(i)$
- $i = 0$: Initialisierung der Clientuhr beim ersten Durchlauf
- $i > 0$: eigentlicher Algorithmus
  - Fall 1: $Zeit_{Server}(i) > Zeit_{Client}(i)$
    - setze Uhr des Client auf $Zeit_{Server}(i)$
    - $\Delta T = Zeit_{Client}(i) - Zeit_{Client}(i-1)$
    - $\phi = Zeit_{Server}(i) - Zeit_{Client}(i)$
    - setze β_{Client} auf $\beta_{Client}(i) + \kappa_1 \cdot \frac{\phi}{\Delta T}$

- Fall 2: $Zeit_{Server}(i) \leq Zeit_{Client}(i)$
  - setze β_{Client} auf $\beta_{Client}(i) - \kappa_2 \cdot \Delta T$

Der Drift der Clientuhr wird mit β_{Client} bezeichnet. Dieser Wert wird vom Algorithmus laufend angepasst und neu berechnet. Zu einem späteren Zeitpunkt werden wir ebenfalls der Variable α begegnen. Diese beschreibt den Offset der Clientuhr. Mit den Konstanten κ_1 und κ_2 kann man eine gewisse Feineinstellung des Algorithmus erreichen. Die von uns gewählten Werte wurden anhand von Messungen empirisch erhoben. ΔT ist die Differenz von zwei aufeinanderfolgenden Zeitstempeln. Die Differenz zwischen der Client- und der Serverzeit wird in ϕ berechnet.

2.3 Implementation

Um das Problem der Uhrensynchronisation besser bearbeiten zu können, haben wir die Aufgabe in folgende Teilprobleme zerlegt und in den aufgeführten Code-Dateien implementiert:

- Kreieren von Uhrenobjekten
  (implementiert in *clkobj.c*, *clkobj.h*, *wrapperfunctions.s*)
- Ausprogrammieren des Synchronisationsalgorithmus
  (implementiert in *sync_client.c*, *sync_server.c*)
- Nachrichtenaustausch über ein Comuternetzwerk ermöglichen
  (implementiert in *sync_client.c*, *sync_server.c*)
- Schnittstellen zwischen Uhr und Player definieren
  (implementiert in *clkobj.c*, *clkobj.h*, *wrapperfunctions.s*)
- objektives Messen (bzw. Aufzeichnen) der Genauigkeit der synchronisierten Uhren
  (implementiert in *parallelport_client.c*, *parallelport_server.c*, *tsc_log.c*, *tsc_log.h*)

Die bis anhin noch nicht erwähnten Dateien haben folgende Funktionen:

- Netzwerkfunktionalität (vergleiche [2] und Abschnitt 2.3.3):
  unpconf.h, *unp.h*, *libunp.a*
- Paketformate:
  tsc_types.h
- Parallelport-Modul:
  tscpp.c, *tscpp.h*
- Makefile:
  Makefile

- Script zum Laden des Parallelport-Moduls:
  load_tscpp
- Matlab-Dateien zur Auswertung:
  parallelport_analyse.m, *sync_analyse.m*

2.3.1 Uhrenobjekt

Die Implemenation der Uhrenobjekte basiert weitgehend auf einer Vorgängerarbeit [3], wobei jedoch einige Anpassungen, auf die später eingegangen wird, gemacht wurden. Es handelt sich um ein Kernelmodul, das geladen werden muss, um ein Uhrenobjekt zu instanzieren und darauf zugreifen zu können. Der Quellcode des Uhrenobjektes (*clkobj.c*, *clkobj.h*) befindet sich auf der CD und ist im Anhang nachzulesen.

Ein zentrales Element dieser Uhrenobjekte ist der *TSC* (*Time Stamp Counter*), der auf jeder *x86 Prozessor-Architektur* vorhanden ist. Es handelt sich um ein 64-Bit breites Register, das mit jedem Taktzyklus um eins inkrementiert wird. Der TSC kann somit bereits als eine Art Uhr betrachtet werden. Der TSC gibt die Anzahl seit dem Starten des Rechners abgelaufener Taktzyklen wieder, und trägt deshalb die Einheit *Ticks*. Eine Umrechnung in die für den Menschen verständlichere Einheit *Sekunden* kann nach Gleichung 2.1 erfolgen.

$$Zeit_{Sek}\ [\text{Sekunden}] = \frac{Zeit_{Ticks}\ [\text{Ticks}]}{CPU\,Taktfrequenz\ [\text{Hz}]} \tag{2.1}$$

Wird das Uhrenmodul (*clkobj.o*) wie im Anhang D.1.2 beschrieben geladen, so wird im Kernel ein einzelnes Uhrenobjekt instanziert und seine Zeit beginnt zu laufen.

Wie bereits in der Vorgängerarbeit [3] und auch kurz in [4] beschrieben, ist Gleichung 2.2 die zentrale Formel zur Berechnung der Zeit in Ticks. Die Laufgeschwindigkeit wird durch β, beziehungsweise $1 + \beta$, bestimmt. Der Offset ist durch α gegeben.

$$Zeit_{Ticks}\ [\text{Ticks}] = \alpha + (1 + \beta) \cdot TSC \tag{2.2}$$

Vorgenommene Anpassungen

Um die Schnittstellen-Problematik vereinfachen zu können, haben wir uns entschlossen, im Gegensatz zur Vorgängerarbeit, uns auf nur eine Uhr auf jedem Rechner zu beschränken. Dadurch brauchen wir keine Kommunikation zwischen den einzelnen Prozessen (Synchronisationsalgorithmus, Audioplayer) aufzubauen, ermöglichen aber den Zugriff von allen Programmen auf dasselbe Uhrenobjekt. Das Uhrenobjekt wird beim Laden des Kernelmoduls automatisch kreiert und initialisiert. Aus diesem Grund muss man beim Verwenden der Funktionen, welche die Uhr einstellen, nicht mehr explizit angeben auf welches Uhrenobjekt man zugreifen möchte. An den Stellen des Vorgängercodes, wo ein Clockdeskriptor (`int`

`clk_desc`) als Parameter übergeben werden musste, reicht es jetzt den Dummy-Wert 0 hinzuschreiben. Die Funktion weiss selber, welches Uhrenobjekt zu verwenden ist. Aus diesem Grund sollten die Anwendungsprogramme von der zur Verfügung stehenden Funktion `initclk()` keinen Gebrauch machen. Diese kommt nur bei der Initialisierung des Kernelmoduls zum Einsatz.

Auf Serverseite wäre eigentlich gar kein Uhrenobjekt, das vom Modul erzeugt wird, notwendig, da wir als Referenzzeit den TSC-Wert – in Nanosekunden umgerechnet – benutzen. Da man aber über das Uhrenobjekt und der Funktion `getrawtime()` äusserst bequem den TSC und die CPU-Taktfrequenz lesen kann, wurde auch auf Serverseite dieses Uhrenobjekt benutzt.

2.3.2 Algorithmus

Mit Hilfe der TSC-Werte `packet.tsc_tx` (TSC-Wert des Servers kurz vor Versand des Pakets) und `packet.tsc_rx` (TSC-Wert des Clients kurz nach Empfang des Pakets) aus dem UDP-Datengramm, den Werten der Clientuhr (`clock_values.alpha`, `clock_values.beta`) und der minimalen *Round Trip Time* (*RTT*) (`minRttTICKS`) aus den Datenpaketen, werden die beiden Werte `timeServerNSEC` ($Zeit_{Server}$) und `timeClientNSEC` ($Zeit_{Client}$) berechnet. Diese beiden Zeiten werden dem Algorithmus übergeben und von ihm zur Berechnung der neuen Werte der Clientuhr benutzt.

Die RTT wird vom Server folgendermassen berechnet: Kurz vor dem Versenden des Datenpaketes wird ein Zeitstempel ins Paket geschrieben (`tscServerSend`). Der Client empfängt das Datenpaket und schickt es sogleich ohne Änderung zurück. Wieder beim Server angelangt wird ein weiterer Zeitstempel (`tscServerRecv`) ins Paket eingefügt. Somit kann der Server die Differenz dieser beiden Zeitstempel (dies entspricht der RTT) berechnen, sich den kleinsten aller RTT-Werte merken und ihn ins nächste Synchronisationspaket einfügen.

Der gesamte Algorithmus ist in den Dateien *sync_server.c* und *sync_client.c* implementiert.

Parameter

Eine Art Feineinstellung des Algorithmus kann durch Verändern der folgenden Parameter erreicht werden. Welche Auswirkung ihr Verändern hat, wird im Kapitel über die Messresultate im Anhang B.1 gezeigt.

- Nachrichtenintervall
- ΔT_{maxSum} (`deltaTMaxSum`)
- κ_1 (`kappa1`)
- κ_2 (`kappa2`)
- `MAXDRIFT`

Parameter	ermittelter Wert
Nachrichtenintervall [ms]	50
`deltaTMaxSum` (ΔT_{maxSum}) [ns]	10^8
`kappa1` (κ_1) [1]	10^{-2}
`kappa2` (κ_2) [$\frac{1}{\text{ns}}$]	10^{-11}
`MAXDRIFT` [ppm]	50
Last	nein
Netzwerktyp	WLAN
Messintervall [ms]	1000

Tabelle 2.1: Parameter mit ihren ermittelten Werten

Die von uns zum Schluss verwendeten Werte kann man der Tabelle 2.1 entnehmen.

Implementation des Algorithmus

In den folgenden Zeilen soll etwas genauer auf die Implementation des Synchronisationsalgorihtmus eingegangen werden. Die Codeteile stammen aus der Datei *sync_client.c* und werden von oben nach unten betrachtet. Zum besseren Verständnis ist es sicherlich sinnvoll, die Kommentare im Code ebenfalls zu lesen. Es wird davon ausgegangen, dass der erste Punkt aus der Beschreibung des Algorithmus in Abschnitt 2.2.1 bereits ausgeführt wurde und die entsprechenden Zeiten in den Variablen `timeServerNSEC` und `timeClientNSEC` gespeichert sind.

Beim ersten Durchlauf erfolgt die Initialisierung der Clientuhr auf plausible Werte α und β. Der Wert α wird mit der Funktion `settime()` gesetzt, β mit der Funktion `setrate()`. Man beachte, dass der Wert β vor Aufruf der Funktion `setrate()` auf einen ganzzahligen Integer umgerechnet werden muss (siehe dazu [3]).

```
...

/* Initialisierung der Uhr auf maximal moegliches Beta
   und ersten Zeitwert vom Server */
if(init == 0){
    /* 2^24 = 16777216 */
    betaInitFLOAT = -MAXDRIFT * (16777216.0 / 1e6);
    betaInitINT   = (int)betaInitFLOAT;
    setrate(0, &betaInitINT);

    nsec2ticks(&timeServerNSEC, &timeInitTICKS,
               clock_values.cpu_khz);
    settime(0, &timeInitTICKS, &packet.tsc_rx);

    timeClientOldNSEC  = timeServerNSEC;
    timeClientPrevNSEC = timeServerNSEC;
```

```
    init = 1;
}
```

Bei allen anderen Durchläufen, also der eigentlichen Synchronisation, werden zuerst die von den beiden Fällen unabhängigen Variablen berechnet. An dieser Stelle sei noch erwähnt, dass die vorliegende Implementation nicht ganz mit der Beschreibung aus Abschnitt 2.2.1 übereinstimmt. Der Grund ist, dass wir eine Art *Buffer* implementiert haben, damit der Algorithmus auch unter starker Netzwerklast befriedigende Resultate liefert. Dies ist uns aber nur unbefriedigend gelungen, so dass wir nicht allzustark darauf eingehen werden. Eine mögliche Lösung, wäre es weiter mit `deltaTMaxSum` oder anderen Parametern zu experimentieren, oder die Bufferfunktionalität anzupassen. Gegen Ende unserer Arbeit hat sich unser Assistent Philipp Blum noch mit dieser Problematik beschäftigt. Die von ihm vorgenommenen Änderungen an der Datei *sync_client.c* sind auf der CD im Ordner *code_philipp/* zu finden.

```
/* falls Initialisierung bereits geschehen
   --> eigentlicher Algorithmus */
else{

    /* berechne deltaTClient */
    if(timeClientNSEC >= timeClientPrevNSEC){
      deltaTClient =
           (signed long long)(timeClientNSEC - timeClientPrevNSEC);
    }
    else{
      deltaTClient =
           (signed long long)(timeClientPrevNSEC - timeClientNSEC);
      deltaTClient = -1 * deltaTClient;
    }
    timeClientPrevNSEC = timeClientNSEC;

    /* berechne deltaTServer (braucht es nicht wirklich) */
    if(timeServerNSEC >= timeServerPrevNSEC){
      deltaTServer =
           (signed long long)(timeServerNSEC - timeServerPrevNSEC);
    }
    else{
      deltaTServer =
           (signed long long)(timeServerPrevNSEC - timeServerNSEC);
      deltaTServer = -1 * deltaTServer;
    }
    timeServerPrevNSEC = timeServerNSEC;
```

Anschliessend werden die zwei Fälle unterschieden. Beim ersten Fall ist die Zeit des Servers (welche aus dem Datenpaket stammt) grösser als die Zeit des Clients. Beim zweiten Fall ist es genau umgekehrt.

Tritt Fall 1 ein, so wird als erstes mit der Funktion `settime()` der Wert α angepasst und danach die Werte von ΔT und ϕ berechnet. Die Anschliessende

Aufsummierung der ΔT's und ϕ's hängt mit dem schon erwähnten Buffer zusammen. Hat die Summe einen bestimmten Wert erreicht, so wird β neu berechnet und angepasst.

```
/* 1. Fall: timeServerNSEC > timeClientNSEC */
if(timeServerNSEC > timeClientNSEC){

  /* setze die Uhr auf die vom Server empfangene Zeit */
  nsec2ticks(&timeServerNSEC, &timeServerTICKS,
             clock_values.cpu_khz);
  /* hier macht die Uhr einen Sprung!? */
  settime(0, &timeServerTICKS, &packet.tsc_rx);

  /* berechne deltaT */
  if(timeClientNSEC >= timeClientOldNSEC){
      deltaT =
        (signed long long)(timeClientNSEC - timeClientOldNSEC);
  }
  else{
      deltaT =
        (signed long long)(timeClientOldNSEC - timeClientNSEC);
      deltaT = -1 * deltaT;
  }
  timeClientOldNSEC = timeClientNSEC;

  /* berechne phi */
  phi = (signed long long)(timeServerNSEC - timeClientNSEC);

  /* counterTillValidClientTime muss groesser sein
     als counterTillValidRtt!!! */
  if(counterTillValidClientTime < 6){
      counterTillValidClientTime =
             counterTillValidClientTime + 1;
  }
  else{
      phiSum    = phiSum + phi;
      deltaTSum = deltaTSum + deltaT;
  }

  /* setze neues Beta, falls eine genuegend grosse Zeitspanne
     verstrichen ist */
  if(deltaTSum >= deltaTMaxSum){
      /* berechne aktuelles Beta */
      getrawtime(0, &clock_values);
      /* 2^24 = 16777216 */
      betaFLOAT = ((float)(clock_values.beta))/(16777216.0);

      /* berechne neues Beta */
      betaNewFLOAT = betaFLOAT +
             (kappa1 * (((float)phiSum)/((float)deltaTSum)));
```

```
            if(betaNewFLOAT > MAXDRIFT*1e-6){
              betaNewFLOAT = MAXDRIFT*1e-6;
            }
            /* 2^24 = 16777216 */
            betaNewINT = (int)(betaNewFLOAT * 16777216);

            /* setze neues Beta */
            setrate(0, &betaNewINT);

            phiSum    = 0;
            deltaTSum = 0;
          }

        }
```

Tritt der zweite Fall ein, so wird nur β neu berechnet und angepasst.

```
        /* 2. Fall: timeServerNSEC <= timeClientNSEC */
        else{

          /* berechne aktuelles Beta */
          getrawtime(0, &clock_values);
          /* 2^24 = 16777216 */
          betaFLOAT = ((float)(clock_values.beta))/(16777216.0);

          /* berechne neues Beta */
          betaNewFLOAT = betaFLOAT - kappa2 * deltaTClient;

          if(betaNewFLOAT < -MAXDRIFT*1e-6){
              betaNewFLOAT = -MAXDRIFT*1e-6;
          }

          /* 2^24 = 16777216 */
          betaNewINT = (int)(betaNewFLOAT * 16777216);

          /* setze neues Beta */
          setrate(0, &betaNewINT);
        }

    }

    ...
```

2.3.3 Nachrichtenaustausch über das Netzwerk

Die implementierte Netzwerkfunktionalität ist stark an die Beschreibungen in [2] angelehnt und teilweise übernommen worden. Ziel war es nicht, das Funktionieren der Netzwerkschnittstelle unter Linux genau zu verstehen, sondern mit einem

geringen Aufwand einen funktionierenden Nachrichtenaustausch zu implementieren. Ausgewählt wurde das Versenden der Daten per UDP. Die Implementierung des Datenaustausches ist ebenfalls in den Dateien *sync_server.c* und *sync_client.c* vorgenommen worden.

Als kurzer Hinweis sei noch angefügt, dass man zu Testzwecken ohne vorhandene Netzverbindung auch die Loopback-Adresse 127.0.0.1 verwenden kann.

filter()-Funktion

Damit gewährleistet ist, dass die Round Trip Time (RTT) möglichst genau berechnet werden kann, und keine (bzw. eine möglichst geringe) Zeitverzögerung zwischen dem Schreiben des TSC und dem Versenden des Pakets entsteht, wurde eine Art Paketfilter implementiert. Allen Netzwerkpaketen, die für den Synchronisationsalgorithmus bestimmt sind, sollen deshalb die TSC-Werte erst kurz vor Verlassen, beziehungsweise kurz nach Eintritt in die Rechner eingefügt werden. Um dies mit einem vernünftigen Aufwand bewerkstelligen zu können, haben wir beschlossen alle Synchronisationspakete über den zufällig gewählten UDP-Port 9879 zu Versenden und zu Empfangen. Ist das Uhrenobjekt-Modul geladen, so fängt die implementierte `filter()`-Funktion solche Pakete ab und fügt die TSC-Werte (`tsc_tx`, `tsc_rx`, `tscServerSend`, `tscServerRecv`) in die Pakete ein.

Werden Datenpakete für die Synchronisation nicht an den Port 9879 geschickt, so wird der Algorihtmus nicht funktionieren, da in solche Pakete keine Zeitstempel eingefügt werden.

Paketformat

Das von uns verwendete Paketformat zum Nachrichtenaustausch ist im Headerfile *tsc_types.h* folgendermassen definiert:

```
/* Paketformat, 'unsigned long long's sind nicht aufgeteilt */
typedef struct {
    unsigned long long alpha;
    int                beta;
    unsigned long long tsc_tx;
    unsigned int       cpu_khz;
    int                betashift;
    unsigned long long tsc_rx;

    unsigned long long tscServerSend;
    unsigned long long tscServerRecv;

    unsigned long long minRttTICKS;
} packet_clk;
```

Die Serverwerte `alpha` und `beta` sind bereits aus vorherigen Erklärungen bekannt, werden aber, wie bereits erklärt wurde, vom Client nicht benötigt (Referenzzeit basiert nur auf dem TSC-Wert des Servers). `tsc_tx` ist der TSC-Wert des

Servers kurz bevor das Paket verschickt wird. `tsc_rx` ist der TSC-Wert des Clients kurz nach Empfangen des Pakets. `betashift` wird zur Umrechnung von β in einen Integer benutzt. Die letzten drei Werte (`tscServerSend`, `tscServerRecv`, `minRttTICKS`) werden zur Berechnung der minimalen Round Trip Time benötigt.

Das hier im Anschluss aufgeführte Paket ist eine umgewandelte Form vom `packet_clk`-Paket. Es dient nur dazu, die Daten korrekt über das Netzwerk versenden zu können.

```
/* Paketformat, 'unsigned long long's sind aufgeteilt */
typedef struct {
    unsigned long alpha_high;
    unsigned long alpha_low;
    int           beta;
    unsigned long tsc_tx_high;
    unsigned long tsc_tx_low;
    unsigned int  cpu_khz;
    int           betashift;
    unsigned long tsc_rx_high;
    unsigned long tsc_rx_low;

    unsigned long tscServerSend_high;
    unsigned long tscServerSend_low;
    unsigned long tscServerRecv_high;
    unsigned long tscServerRecv_low;

    unsigned long minRttTICKS_high;
    unsigned long minRttTICKS_low;
} packet_tx_rx;
```

Da das Versenden von Variablen mit einer Auflösung von 64-Bit (`unsigned long long`) nicht ganz unproblematisch ist, wird das Paket vom typ `packet_clk` vor dem Versenden mit der Funktion

```
void fill_packet_clk(packet_clk *packet,
                     packet_tx_rx *packet_tx_udp)
```

(siehe *tsc_common.c*) in ein versandbereites Paket vom Typ `packet_tx_rx` umgewandelt. `unsigned long long` Variablen (1 mal 64 Bit) werden dort in zwei `unsigned long` Variablen (2 mal 32 Bit) aufgeteilt. Beim Empfangen von Paketen geschieht der umgekehrte Vorgang mit der Funktion

```
void read_packet_clk(packet_clk *packet,
                     packet_tx_rx *packet_rx_udp).
```

Man beachte, dass zur Herstellung der korrekten Byte-Order für die Übertragung ebenfalls die Funktionen `htonl()` bzw. `ntohl()` benutzt werden.

2.3.4 Schnittstellen

Um ein Programm zu schreiben, das auf das Uhrenobjekt zugreifen soll, muss man die Datei *clkobj.h* im Include-Bereich des Codes aufführen und das Uhrenobjekt im

Kernel laden. Danach stehen die Funktionen, welche das Uhrenobjekt schreiben und lesen zur Verfügung. Ebenfalls zugänglich sind dann die Umrechnungsfunktionen von *Ticks* in *Nanosekunden* und umgekehrt.

Will man aus einem Anwendungsprogramm die Uhren lesen, so ist wie folgt beschrieben vorzugehen.

- um $Zeit_{Server}$ zu lesen, der Reihe nach:
  - `int getrawtime(0, struct clkval *cv);`
  - ```
    int ticks2nsec(unsigned long long *ticks,
                   unsigned long long *nsec,
                   unsigned int cpu_khz);
    ```

  wobei die Werte `ticks` und `cpu_khz` aus der Struktur `cv`, die mit `getrawtime()` beschrieben wurde, entnommen werden müssen.

- um $Zeit_{Client}$ zu lesen, der Reihe nach:
  - `int getrawtime(0, struct clkval *cv);`
  - ```
    int readtime(int clk_desc, unsigned long long *now);
    ```
    bzw.
    ```
    int readtime(0, unsigned long long *now);
    ```
  - ```
    int ticks2nsec(unsigned long long *ticks,
                   unsigned long long *nsec,
                   unsigned int cpu_khz);
    ```

Systemcall-Funktionen

Die in der folgenden, nicht vollständigen, Aufzählung aufgeführten Funktionen stehen dem User dank dem Uhren-Kernel-Modul zur Verfügung. Voraussetzung dafür ist, dass das Kernelmodul geladen und im Code die Anweisung

```
#include "clkobj.h"
```

nicht vergessen wurde.

- `int readtime(int clk_desc, unsigned long long *now);`
- ```
  int setclk(int clk_desc, unsigned long long *new_time,
             int *new_beta);
  ```
- ```
  int settime(int clk_desc, unsigned long long *new_time,
              unsigned long long *tsc_rx);
  ```
- `int setrate(int clk_desc, int *new_beta);`
- `int getrawtime(int clk_desc, struct clkval *cv);`
- ```
  int calctime(unsigned long long *alpha,
               unsigned long long *tsc, int *beta,
               unsigned long long *now);
  ```

- `int nsec2ticks(unsigned long long *nsec, unsigned long long *ticks, unsigned int cpu_khz);`
- `int ticks2nsec(unsigned long long *ticks, unsigned long long *nsec, unsigned int cpu_khz);`

Wir werden nicht weiter auf die Eigenschaften dieser Funktionen eingehen, sie sind genügend gut im Code (*clkobj.c*) oder in [3] erklärt. Es soll aber nochmals erwähnt werden, dass wir die aus [3] übernommenen Funktionen teilweise abgeändert haben. Jede der Funktionen mit einem Übergabeparameter `int clk_desc` wurde so angepasst, dass dieses Argument nicht mehr ausgewertet wird, sondern immer auf das gleiche Uhrenobjekt zugegriffen wird. Ein solcher Funktionsaufruf geschieht demzufolge beispielsweise in dem man an dieser Stelle eine Null übergibt: z.B. `readtime(0, now);`

2.3.5 Messung der Qualität des Algorithmus

Die Schwierigkeit der Messung liegt darin, beide Uhren auch wirklich gleichzeitig zu lesen, um danach deren Abweichung berechnen zu können. Zu diesem Zweck haben haben wir die Parallelport-Programme (*parallelport_server.c*, *parallelport_client.c*), basierend auf der Arbeit von Philipp Blum, geschrieben. Voraussetzung für erfolgreiches Messen ist, dass die beiden Rechner (Server und Client) miteinander über das spezielle Parallelport-Kabel verbunden sind.

Wird die Messung gestartet, so löst der Server einen Interrupt am Parallelport aus. Dieses Signal geht sowohl an den Client, als auch direkt wieder an den Server zurück. Sobald beide Rechner das Interrupt-Signal erhalten werden die Uhren gelesen (praktisch gleichzeitig) und die Zeiten in einer Variable (beim Client: `/dev/tsc_latch`) gespeichert. Der Server sendet seine gemessene Zeit über das Netzwerk an den Client. Dieser empfängt das UDP-Paket, liest das Device `/dev/tsc_latch` und protokolliert beide Zeiten in einem Logfile.

Messvorgehen

Das Vorgehen, um die Genauigkeit der Uhrensynchronisation zu messen ist das folgende: Wie im Anhang D.1.2 beschrieben, muss das *tscpp*-Modul geladen werden. Mit diesem Modul ist es möglich, die Uhr beim Server und beim Client gleichzeitig zu lesen und die gemessenen Zeitpunkte in einem Logfile zu speichern, um diese zu einem späteren Zeitpunkt mit *Matlab* auswerten zu können.

Die in Matlab erfolgte Auswertung ist in der Datei *parallelport_analyse.m* implementiert. Wird in Matlab `pallelport_analyse('logfile.log')` aufgerufen, wobei `logfile.log` der Name des Auszuwertenden Logfiles ist, so wird die die Differenz zwischen der Zeit des Clients und des Servers berechnet und in einer Grafik, wie sie im Anhang B.1 dargestellt ist, ausgegeben.

Kapitel 3

Alsa Player

Die Idee hinter der Verwirklichung des vorliegenden Audio-Abspielprogramms ist es einen Hardware nahen Audio-Player zu schreiben, welcher den Gewinn von synchronisierten Uhren möglichst gut zur Audiowiedergabe nützt. Gesucht war somit ein API zur Soundkarte, welche aus dem Userspace "direkt" auf die Hardware zugreifen konnte.

3.1 Konzept und Teilprobleme

Die Erstellung dieses Audioplayers war ein fortlaufender Vorgang – Stück für Stück wurde gesondert implementiert, getestet und anschliessend zum fertigen Player zusammengesetzt werden. Beim hier vorliegenden Wireless Audio Speaker Player sind dabei die folgenden Teilprobleme aufgetreten:

- **Audioausgabe:**
  Die meisten für Linux vorhandene Audioplayer sind in einem sehr fortgeschrittenem Stadium. Dadurch ist es trotz ausgedehntem Studium des Quelltexts nur schwer zu verstehen, wie die Soundkarte korrekt initialisiert, konfiguriert und an welcher Code-Stelle dann anschliessend die Audioausgabe gestartet wird. Eine somit nahe liegende Idee war einen Audioplayer von Grund auf neu zu schreiben.

  Eine Möglichkeit, welche auch in der Semesterarbeitsausschreibung vorgeschlagen wurde, war vorhandenes Wissen und Codestücke im Departement zu den ALSA Libraries[1] zu nützen und mit diesen ein Ausspielprogramm zusammenzusetzen.

  Aufgrund der verständlichen API und einiger guter Vorlagebeispiele fiel der Entscheid zugunsten von ALSA (Advanced Linux Sound Architecture) und auf dieser Basis wurde der Wireless Audio Speaker Player - im Folgenden kurz mit WAS abgekürzt - implementiert.

[1] Weitere Möglichkeiten und Informationen zu ALSA werden im Unterkapitel 3.2.1 kurz vorgestellt.

- **Aushandeln eines gemeinsamen Abspielzeitpunkt:**
  Eine zentrale Fragestellung der synchronisierten Audioausgabe ist das Problem des *synchronen Startens*. Grob gesagt muss, unter Verwendung der synchronisierten Uhren, ein gemeinsamer Startzeitpunkt definiert werden, auf welchen dann die Audioplayer mit der Wiedergabe beginnen.

  Es stellte sich heraus, dass für dieses Teilproblem der gewählte Weg - einen Player von Grunde auf neu zu schreiben - sehr günstig war, denn so konnte ein sehr einfaches Kommunikationsprotokoll integriert werden: Wie in Abbildung 2.1 angedeutet, gibt es zwei Rollen, welche der Player einnehmen kann: Entweder er agiert als *Server* oder er nimmt die Rolle eines *Clients* an.

  Um möglichst einfach einen gemeinsamen Ausspielzeitpunkt zu bestimmen, reicht es prinzipiell, wenn der Server dem Client (bzw. den Clients) eine Zeit "vorschreibt". Wird bei der Wahl des Zeitpunktes eine allfällige Verzögerungen der Übertragung berücksichtigt, sollten alle Computer in der Lage sein zum gewählten Zeitpunkt mit der Audioausgabe starten zu können.

  Genau dieses Konzept wurde in den WAS Player eingebaut: Nach dem Starten der Uhrensynchronisation wird als Erstes ein Initalisierungsvorgang durchgeführt und anschliessend wartet der Client in blockierender Weise auf ein UDP-Paket des Servers, welches den Abspielzeitpunkt in der Einheit des Uhrenobjektes enthält. Wie im Kapitel 2 beschrieben, kann mittels der Uhrenobjekte eine *gemeinsame Zeit* erzeugt werden und mittels des obgenannten Konzeptes ein gemeinsamer Abspielzeitpunkt.

  In der gewählten Implementierung wird nach dem Nachrichtenaustausch solange die "Uhrzeit" des Uhrenobjektes *gepollt*, bis der Abspielzeitpunkt erreicht oder überschritten wurde, danach beginnen die Audioausgaberoutinen.

- **Vergleich einer Userspace gegenüber einer Kernelspace Implementation:**
  Für die Implementation des WAS Player haben sich zwei Möglichkeiten angeboten: Entweder den Player mit den vorhandenen ALSA API komplett im Userspace zu implementieren oder gewisse ALSA Funktionen anzupassen, sowohl im User- wie auch im Kernelspace. Beide Varianten sind in Abbildung 3.1 kurz skizziert.

  Aus zeitlichen Gründen wurde eine reine Userspace Implementation vorgezogen (die linke Variante in der Abbildung 3.1), unter anderem da es für diese Lösung einen grösseren Grundstock an Materialien gibt und nicht in die ALSA Treiber eingegriffen werden muss.

  Für die Mischform spricht hingegen die Tatsache, dass in dieser Variante eine Verzögerung nach dem Pollen auf den Startzeitpunkt vermieden werden und somit eine zeitlich genauere Audioausgabe erreicht werden kann.

- **Kompensation von Host-spezifischen Verzögerungen:**
  Der Audio-Player wird im allgemeinen Fall auf unterschiedlich schnellen Stan-

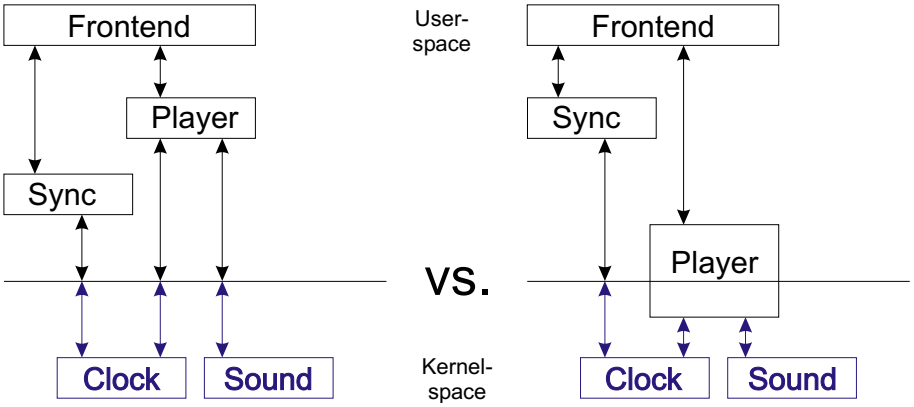


Abbildung 3.1: Vergleich Implementation des Players im Userspace gegenüber einer gemischten Implementation User- und Kernelspace

dard Computer eingesetzt. Da in diesem Fall die Software-Routinen des Programms unterschiedlich schnell abgearbeitet werden, muss dieser Punkt mittels einer Korrektur der Host-spezifischen Verzögerungen berücksichtig werden.

Insbesondere ist diese Korrektur für die Synchronisation des Abspielzeitpunktes entscheidend: Wie beschrieben, wird der Startzeitpunkt durch den *Server* gewählt und anschliessend *pollen* sowohl der *Server*, als auch der *Client*, die Zeit des Uhrenobjekts bis der Startzeitpunkt erreicht wurde. Anschliessend werden einige Funktionsaufrufe zur Audioausgabe aufgerufen, welche folglich unterschiedlich schnell abgearbeitet werden.

Überblickend eine Auflistung der zu kompensierenden Verzögerungen:

- unterschiedliche Geschwindigkeiten der Computer, welche Einfluss auf die Abarbeitung der Betriebssystem- und die implementierten Software-Routinen haben
- Host-abhängige Zeit zwischen dem Ausspielbefehl in Software, und dem Zeitpunkt bei welchem dann ein Audiosignal am Soundkartenausgang erscheint
- Die Geschwindigkeit des Ausspielens wird von der Soundkarte vorgegeben; unterschiedliche Karten[2] und Konfigurationen haben somit unterschiedliche Verzögerungen

- **Driftkorrektur:**
  In dem vorliegenden Konzept für den WAS Player wird der Startzeitpunkt der Audio-Wiedergabe synchronisiert. Der Drift der beiden Soundkarten, das heisst die Geschwindigkeitsdifferenz der Oszillatoren der Karten, aber nicht

[2] auch desselben Typs: Oszillatoren laufen in der Regel nicht exakt gleichschnell

berücksichtigt[3]. Als Vorgriff auf die später folgenden Messung sei hier erwähnt, dass der auftretende Drift grösser ist, als lange Zeit angenommen wurde. Aus diesem Grunde fehlt beim WAS Player eine entsprechende Implementierung.

Um diesen Drift zu kompensieren sind zwei unterschiedliche Ansätze denkbar:

1. Periodisches Weglassen oder Verdoppeln eines Samples:
   Anstatt nur einer Synchronisation des Abspielpunktes, wird eine regelmäessige Resynchronisation der Abspielzeitpunkte der einzelnen Frames durchgeführt. Da der Drift erfahrungsgemäss nicht allzugross ist, sollte eine solche Synchronisation machbar und eventuell für den Höhrvorgang nicht bemerkbar sein.
2. Kontinuierliche Kompensation:
   Die zweite Idee entspricht eher dem Ansatz einer "dynamischen Driftkomensation". Dabei wird das digitale Audio-Signal eines Computers so gestreckt oder gestaucht (Stichworte: *Dezimation* und *Interpolation*), so dass resultierend kein Drift mehr messbar ist.

Ausführlichere Informationen zu diesen Vorschlägen zur Driftkompensation befinden sich im Abschnitt C.18 (S. 82).

3.2 Implementation des Wireless Audio Speaker

3.2.1 Audioausgabe unter Linux

Einführung

Linux ist ursprünglich nicht dafür entwickelt worden Multimedia wiederzugeben und somit wurden unter anderem Sound-Teiber erst in den letzten Jahren geschaffen, verbessert oder von anderen Treibern abgelöst. Dadurch existieren oft für ein und dieselbe Soundkarte mehrere Treiber und APIs, welche sich durch ihre Funktionalität und ihren Aufbau unterscheiden.

Es koexistieren momentan folgende Ansätze:

- ALSA [5]
- OSS Commercial [6]
- OSS Free [7]
- jack [8]
- Linux Treiber vom Hersteller der Soundkarte oder von einem Ausrüster.

[3] Messungen dazu sind im Abschnitt B.2.4 auf Seite 61 zu finden.

Eine gute Übersicht der aktuellen APIs und Treiber, sowie Audio Programme für Linux bietet [9].

ALSA im Überblick

Bei der Realisierung des Wireless Audio Speaker Player ist der Entscheid auf die ALSA[4] API gefallen. Das ALSA Projekt entstand aus der sehr unbefriedigenden Situation der Audio-Unterstützung im Kernel 2.4, welcher im Wesentlichen noch auf der OSS[5] API aufbaut.

ALSA erweitert das Linux OS um Audio und Midi Funktionalität mit den folgenden wichtigen Eigenschaften:

- Effiziente Unterstüzung für alle möglichen Arten von Audio Schnittstellen, von der einfachen Soundkarte bis hin zu professionellen Schnittstellen mit Mehrkanal Audio.
- Total modularisierte Sound Treiber.
- SMP und thread safe design.
- Userspace Bibliothek (ALSA-lib), welche das Applikationsprogrammieren vereinfacht und eine Funktionalität auf höherer Ebene bietet.
- Unterstützung für die alte OSS API und somit Binärkompatibilität zu den meisten OSS Programmen.

ALSA wurde designiert das OSS in den Folgeversionen des Kernels abzulösen - im Entwicklerast ist dieser Schritt bereits schon vollzogen worden. Für den Standard-Kernel ist dieser Vorgang für ein spätere Version als der aktuellen Version 2.4.21 vorgesehen; mit grösster Wahrscheinlichkeit geschieht ein solche Änderung erst in der Version 2.6 oder 3.0.

Für die Applikations- und Treiberentwicklung wurde ein *API Freeze* bei ALSA vollzogen, so dass bei einem späteren Versionswechsel von ALSA, der WAS-Player ohne Änderungen als Programm gestartet oder neukompiliert werden sollte.

API von ALSA

Für das Ansprechen der ALSA-Kerneltreiber gibt es für den *Userspace* ein gut dokumentiertes Interface [10], u.a. für C, so dass der Punkt der hardware-nahen

[4] Advanced Linux Sound Architecture [5]

[5] Open Sound System [7] [6]: erhältlich in einer aktuellen kommerziellen Version und in einer "älteren" freien Version, welche jeweils unter die GPL gestellt wird. Unter anderem aus diesen Lizenz Gründen ist und war OSS der Linux-Community schon lange ein Dorn im Auge.

Programmierung erfüllt werden kann. Die API besitzt eine reiche Schar an gut beschriebenen Funktionen.

3.2.2 Steuerung / Optionen von WAS

WAS wird mittels eines *Commandline Interfaces* aus einer *Shell* angesprochen und besitzt verschiedene Variationsparameter, mit welchen dem Player des gewünschte Verhalten diktiert werden kann. Die Software kann als *Server*, als *Client* oder für Testzwecke auch als ein *eigenständiger Player* agieren, bestimmt durch die gewählten Parameter.

Folgende Einstellungen können getroffen werden:

- Netzverhalten:
  - Player und zusätzlich Server in einem Netzwerk
  - Player und zusätzlich Client in einem Netzwerk
  - Eigenständiger Betrieb für Testzwecke (Standalone)
- Abspielmethoden:
  - Abspielen der digitalen Audiodaten mit Funktionen auf *IO Calls* aufbauend
  - Wiedergabe der Audiodaten mit *Memory Mapped* Methoden
- Einstellung von Formaten:
  - Einstellen des PCM Formates (Codierung)
  - Variation der Samplerate der generierten oder vorliegenden Audiodaten
  - Wahl der Anzahl Kanäle (Mono, Stereo, Mehrkanal)
- Audiodatenwahl:
  - Erzeugen von künstlich generierten Audiodaten (Sinusschwingung, einseitige Sinusschwingung, Rechtecks- und Schrittfunktionen)
  - Auslesen und Wiedergeben von WAVE - Dateien

3.2.2.1 Commandline Argumente von WAS:

Die Commandline des WAS Player lässt sich mit kurzen, sowie auch einprägsamen langen Kommandozeilen Parametern starten:

```
Usage: was [-h] [[-t|-l] [-i=IP] [-p=Port]]
           [-m|-w] [-F=Format] [-R=Samplerate] [-C=Channels]
           [-s|-a|-r|-c|-k|-S|-f [FILE]]

-h,--help       help
```

```
-t,--talker     acts as an server
-l,--listener   acts as a client

-i,--ip         use the following IP
-p,--port       use the following Port

-m,--mmap       use mmaped audio functions
-w,--writei     use ioctl audio functions

-F,--format     use the following ALSA PCM format
-R,--samplerate use the following samplerate
-C,--channels   use the following channels (1 = mono, 2 = stereo)

-s,--sinus      generates a sine audio signal
-a,--abssinus   generates an abs sine audio signal
-r,--rect       generates a rectangular audio signal [from sine]
-S,--step       generates a step audio signal
-c,--const      generates a constant high audio signal
-k,--konst      generates a constant low audio signal

-f,--file       play the following .WAV file

-d,--delta      start the audio playout at the client nSecs later
```

3.2.3 Implementation des WAS in C

Der C-Code wurde in fünf Teile gegliedert: Funktionen und Variablen werden nach Bedarf für den Aufruf aus anderen Strukturen als `extern` definiert, so dass in diesem Sinne ein "Klassen-Konzept" - ähnlich des von C++ - in C aufgebaut werden kann:

- **AlsaSound:**
  Hier befinden sich die Funktionen mit welchen man eine Soundkarte initialisieren, einen *Stream* zu dieser Karte schicken oder dann diesen Stream ausspielen kann. Dabei wurden sowohl Funktionen welche *Memory Mapped* agieren, als auch solche welche mit *IO-Calls* arbeiten, implementiert.

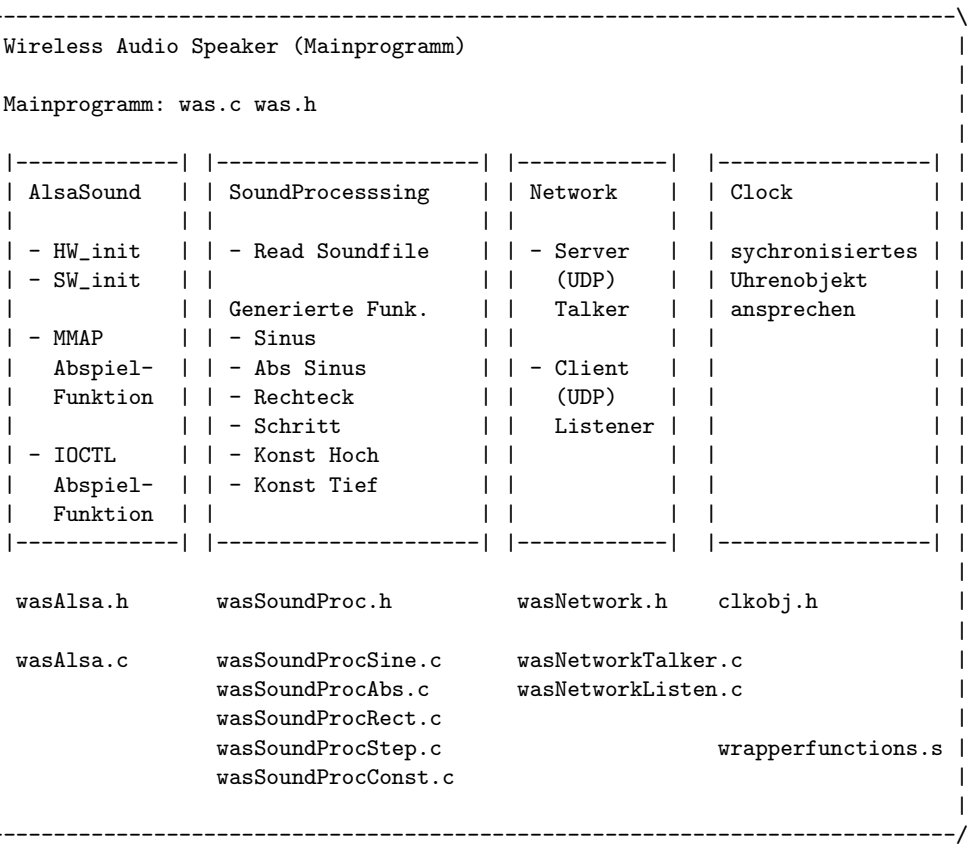
- **SoundProcessing:**
  Erzeugt einen Stream von konkreten Wellenformen (Sinus, Absolutwert des Sinus, Rechtecksfunktion, Schrittfunktion, Konstanter Wert hoch und tief) oder wandelt ein .WAV-File in einen Stream um.

- **Network:**
  Funktionalität für einen UDP-Datengramme sendenden Netzwerk Server, bzw. für einen auf diese UDP-Datengramme wartenden Netzwerk Client. Der Client ist in der gewählten Implementierung blockierend.

- **Clock:**

Einbindung der Funktionen zum Ansprechen des lokalen Kernel-Uhrenobjektes aus dem Userspace.

- **Main:**
  Vereint die obigen vier Bestandteile und enthält im Wesentlichen noch die Auswertung der Kommandozeilenparameter, das Aushandeln eines gemeinsamen Abspielzeitpunktes und das Aufsetzen und Starten der gewünschten Ausspielart.



Eine ausführliche Beschreibung der einzelnen Dateien und Funktionen befindet sich im Anhang C (S. 62). Der Sourcecode selbst befindet sich am Schluss auf Seite 90ff.

3.2.4 Die 6 Phasen im Player Programm

1. **Init:** Initialisierung der Variationsparameter und weiterer Hilfsvariablen
2. **Parameter:** Auswertung der *Commandline* und Setzen von Variablen
3. **ALSA:** Initialisierung der ALSA Umgebung
4. **Clock:** Auslesen des Clock-Objektes

5. **Netzwerk:** Aushandeln des Abspielpunktes

   In der getroffen Implementierung schickt der Server dem Client ein simples UDP-Paket, welches drei Timestamps enthält. Genauer bestehet dieses Paket aus dem letzten gelesenen Timestamp des Servers, dem aktuell gelesenen Timestamp des Clients (bei der Erstellung auf dem Server noch leer) und dem gemeinsamen Abspielzeitpunkt.

   Der gemeinsame Abspielzeitpunkt wird vom Server[6] nach folgenden Formeln berechnet:

```
AbspielzeitpunktServer = aktTimestampServer +
                         TRANSFER_TIME;

AbspielzeitpunktClient = aktTimestampServer +
                         TRANSFER_TIME +
                         DELTA_TIMEOK_PLAYOUT +
                         DELTA_PLAYOUT_STATIONS;
```

   Dabei enthalten die Variablen und Konstanten folgende Werte:

   - `aktTimestampServer`: Variable, welche den Wert des letzten ausgelesen Timestamps des Server-Uhrenobjektes enthält.
   - `TRANSFER_TIME`: Konstante, welche auf einen Wert von 10ms gesetzt ist. Dadurch sollte erreicht werden, dass auch im *worst-case* der Client das Paket erhalten und verarbeiten kann, bevor der Server schon mit seiner Ausgabe angefängt.
   - `DELTA_TIMEOK_PLAYOUT`: Maschinenabhängige Konstante: Mit diesem Wert wird die Differenz der Zeiten angegeben (Server vs. Client), welche gebraucht wird zwischen dem "Ausspielbefehl" in Software und dem tatsächlichen Auftreten des Audiosignals an der Soundkarte.
     Der Wert `DELTA_TIMEOK_PLAYOUT` berechnet sich nach der Formel:
     `DELTA_TIMEOK_PLAYOUT = TIMEOK_PLAYOUT_Client - TIMEOK_PLAYOUT_Server`
   - `DELTA_PLAYOUT_STATIONS`: Maschinenabängige Konstante: Mit dieser Konstante wird die unterschiedliche Geschwindigkeit der Computer berücksichtigt.
     Der Wert `DELTA_PLAYOUT_STATIONS` berechnet sich nach der Formel:
     `DELTA_PLAYOUT_STATIONS = PLAYOUT_Client - PLAYOUT_Server`

   Der Client ersetzt bei Eingang des Paketes den noch leeren "Client" Timestamp durch seine aktuelle Zeit. Diese Information ist nicht unbedingt notwendig, kann aber nützlich für ein Debugging sein.

[6] Würde man WAS mit mehr als einem Client betreiben, so müssten die Addition von `DELTA_TIMEOK_PLAYOUT` und `DELTA_PLAYOUT_STATIONS` zur Clientseite verlagert werden, da diese Werte für jedem Client spezifisch sind.

6. **Player:** Warten auf Triggerzeit und synchrones Starten des Players

   In den beiden Instanzen von WAS, das heisst im Server Player und im Client Player, wird mittels eines *Polling* die Zeit des lokalen Uhrenobjektes ermittelt und sobald der ausgehandelte Abspielzeitpunkt erreicht ist, wird mit der Audio Ausgabe gestartet.

3.2.5 Hörprobe

Nach der Erstellung des Players ist ein weiteres Teilproblem aufgetaucht: Die Frage, wie man jetzt die synchronisierte Ausgabe für eine Hörprobe nutzen kann.

Es ist nur mit einer gewissen Anstrengung und einem geschulten Gehör möglich einen allfälligen Unterschied beim Auspielen einer WAVE-Datei zu hören. Ein besserer Weg ist es speziell generierte Daten zu erzeugen, welche eine Unterscheidung ermöglichen:

Damit bei einer Hörprobe des WAS Players ein Unterschied in der Synchronisation feststellbar ist, wurde das Programm so modifiziert, dass pro Iterationsschritt immer nur das Erste von zehn Frames, das heisst eine $\frac{1}{10}$ Sekunde, mit digitalen Samples gefüllt wird. Die restlichen Frames bleiben auf Null. Dadurch erscheint ein "piepsendes" Sinussignal. Wenn ein Kanal zwischen dem Server und dem Client umgeschaltet wird, so ist nach einer kleinen Angewöhnungszeit das Delta gut hörbar. Die Graphik 3.2 gibt einen Überblick einer so erzeugten Sequenz. Die ersten 100ms sind mit einem Sinus ausgefüllt, die restlichen 900ms der Sekunde verbleiben auf dem Wert Null.

3.2.6 Messungen

Zur Feineinstellung und Verifikation der synchronen Audioausgabe wurden unterschiedliche Messungen getätigt. Einerseits wurde mittels eines Oszilloskopes die Wellenformen und Sound-Frames untersucht, anderseits wurden verschiedene Computer- und Implementationsspezifische Verzögerungen mit einem Logic Analyzer ausgewertet:

- Wie sehen die digitalen Samples nach der Audiokarte aus. Zum Beispiel ist eine generierter Sinusstream auch nach der D/A Wandlung immer noch ein exakter Sinus?
- Füllen die implementierten Funktionen jeweils das ganze Frame mit Werten oder beginnt, beziehungsweise endet, der Datenstrom dazwischen?
- Wie gross sind die Host-spezifischen Delays? Die Verzögerungen zwischen den Hosts?
- Wie gross ist der Drift?

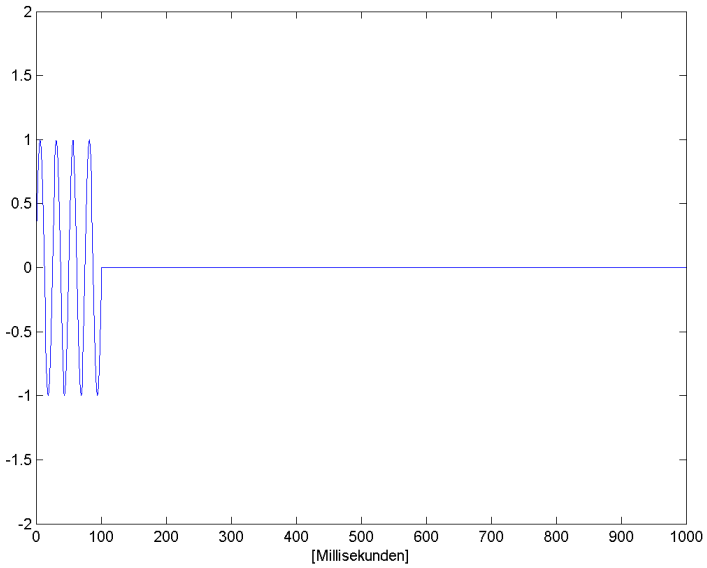


Abbildung 3.2: Ausspielweise des WAS Players für die Hörprobe. Die ersten 100ms werden mit dem gewünschten Signal gefüllt - im Beispiel mit einer Sinusfunktion - die restlichen 900ms verbleiben auf dem Nullwert.

- Wie gut synchronisiert ist denn nun der Ausspielzeitpunkt?

Das genaue Messvorgehen und die Messresultate sind im Anhang im Kapitel B.2 zu finden. An dieser Stelle möchten wir aber schon kurz die Resultate der Messungen angeben:

- **Delay** zwischen Softwarebefehl und effektivem Ausspielen des 800MHz Rechner: 496.8 μs
- **Delay** zwischen Softwarebefehl und effektivem Ausspielen des 1000MHz Rechner: 396.0 μs
- **Drift**: 5 Millisekunden innerhalb einer Laufzeit von 43 Sekunden
- **synchrones Ausspielen**: Innerhalb eines Unsicherheitsfensters von 200 μs spielen beide Computer synchron ihre Daten aus. Es treten durch die Userspace Implementation jedoch Ausreiser auf, welche sich im Millisekundenbereich bewegen.

3.2.7 Subjektiver Höreindruck

Wird bei WAS Player ein grosses Delay (10 ms bis 2 ms) zwischen dem Server und dem Client gewählt, so ist die Verschiebung deutlich hörbar; die Ausgaben der

beiden Computer scheinen einander zu folgen, die Schwingungen sind gesondert festzustellen.

Wird hingegen nur ein kleines Delay im Bereich von 2 ms bis 100 μs eingestellt, so erscheint der Server als ein bisschen lauter und als Ursprungsort des Tones (Links- oder Rechtsverschiebung des Klangbildes).

Wird keine Verzögerung gewählt so erscheint ein synchronisiertes Abspielen. Es macht sich jedoch schon bald der beschriebene Drift bemerkbar und somit verschiebt sich wiederrum das Klangbild, bis hin zu dem Punkt, bei dem die Audioausgaben gesondert wahrnehmbar sind.

3.3 Wireless Stereo Speaker Frontend

Beim Wireless Stereo Speaker Frontend - im folgenden auch mit WSS abgekürzt - handelt es sich um ein QT-GUI, welches die Funktionalität der schon beschriebenen kommandozeilen-orientierten Programme Uhrensynchronisation (Kap.2), Wireless Audio Speaker (Kap.3.2), sowie einiger Zusatztools vereint und graphisch dem User darreicht.

3.3.1 Idee zum Graphischen Benutzer Interface (GUI)

Die initiale Idee hinter der Realisierung eines graphischen Frontends war ein einfaches und wenig fehleranfälliges Eingabe-Tool für den WAS Player. Die unterschiedlichen Optionen des Kommandozeilen Programms kĄönnen so sicherlich bequemer, weniger fehleranfällig und fĄür einen ungeschulten Benutzer auch wesentlich verständlicher ausprobiert werden.

Das *QT-Toolkit*[7] bietet ein gut durchdachtes Programmier - Konzept und leistungsstarke Funktionen zur schnellen Erzeugung eines graphischen Interfaces. UnterstĄützend dabei ist das Vorhandensein eines *Rapid Prototyping Tools*[8] zum Zusammenklicken eines GUIs. Hilfreich bei der Entwicklung ist die MĄöglichkeit zur Verwendung des C++ Klassenkonzeptes und der Vererbung, mit welchen die (leeren) Eventhandlers des GUIs mit den ausprogrammierten Funktionen überschrieben werden können.

3.3.2 Benutzung von WSS

Das Wireless Stereo Speaker Programm und der vom Windowmanager zu verwendende *Style* werden folgendermassen gestartet:

```
Usage: ./WirelessStereoSpeaker [-style STYLE]

-style          QT feature, styles included in QT-3.0.5:
                windows, motif, motifplus, platinum, sgi, cde
```

Die in der Übersicht Abb.3.3 bis Abb.3.8 dargestellten Fenster kurz erklärt:

- Wireless Stereo Speaker Hauptfenster:
  Oben am Fensterrand befindet sich eine Menüleiste. Die Menupunkte unter File und Help sind selbsterklärend; die Einträge unter dem Eintrag Tools werden hier kurz vorgestellt:

  ⋆ Der Eintrag AlsaMixer ruft ein X-Terminal auf, in welchem das Programm `alsamixer` – ein zeichenbasierter Software - Audiomischer – ausgeführt wird.

[7] Unter Unix wird das QT-Toolkit immer häufiger zur Realisierung von GUIs verwendet, u.a. als AblĄösung des eingestellten Motif-Toolkits. Am berĄühmtesten ist die damit erstellte KDE-Benutzeroberfläche und deren Programme. Mehr Infos zum Toolkit unter http://www.trolltech.com.

[8] qt-designer, oft unter dem Namen designer installiert

Abbildung 3.3: Das Menu Tools des Wireless Stereo Spaker Programms

- ⋆ Der Eintrag XMMS ruft den Audioplayer `XMMS` auf.
- ⋆ Nach Betätigung des Eintrags WaveChecker wird ein Perl Programm und das Unix Tool `file` gestartet, welche eine .WAV Datei analysieren und die Einstellungsdaten dieser Wave-Datei wiedergeben. Die Datei kann unter Sound Processing -> Load File -> Filename eingegeben werden, mit dem rechts vom Eingabefenster befindlichen Button oder falls noch keine Datei ausgewählt wurde mittels einer OpenFile Dialogbox.
- ⋆ WaveConvert ruft das Tool `mpg123` auf, welches ein .MP3 File in ein .WAV File umwandelt. Dafür wird nach dem Anklicken des Menüeintrages ein OpenFile (.MP3) und anschliessend ein SaveFile (.WAV) Dialogfenster dargestellt; der Umwandlungsvorgang erscheint in einer separaten Shell.

Unter der Menüleiste befindet sich eine Auswahlbox mit drei RadioButtons:

- ⋆ Network Server: Der WAS Player wir als Server, d.h. als Talker konfiguriert.
- ⋆ Network Client: Der WAS Player wir als Client, d.h. als Listener konfiguriert.
- ⋆ Standalone: Startet den WAS Player ohne Netzwerkfunktionalität und eignet sich somit für ein unsynchronisiertes Ausprobieren.

In der Zeile zuunterst befinden sich vier Buttons:

- ⋆ Exit: Beendet das Wireless Stereo Speaker Frontend;
- ⋆ Clear: Lädt die Default-Einstellungen.
- ⋆ Stop Play: Bricht die Audio Ausgabe des Players WAS ab.
- ⋆ Start Play: Startet eine Audio Ausgabe mit dem *Commandline Tool* WAS.

- Tab Sound Processing:

Abbildung 3.4: Tab: Sound Processing

- ⋆ Manner of Playing: WAS wurde dafür ausgelegt, eine Welle zu generieren oder Samples aus einem File auszulesen und diese Daten in Frames zu verpacken. Diese kann anschliessend entweder mittels einer Memory Mapped agierenden Funktionen oder mittels IO Calls ausgespielt werden.
- ⋆ Sound Source: Auswahl, ob eine Sample Frame generiert oder aus einer Datei ausgelesen werden soll.
- ⋆ Generation of Waves[9]:
  - ◇ Sine: Sinus mit der Frequenz von 440 Hz (a)
  - ◇ Abs: Absolutwert eines Sinus mit der Frequenz von 440 Hz; erscheint dem Gehör als ein Sinus mit 880 Hz
  - ◇ Rectangular: Gestreckte Vorzeichenfunktion eines Sinus mit der Frequenz von 440 Hz
  - ◇ Step: Schrittfunktion bei welchem man den Ort des Schritts einstellen kann.
  - ◇ Constant high: Konstant auf Maximalwert
  - ◇ Constant low: Konstant auf Null
- ⋆ Load File: Angabe des Dateinamen und Pfads einer .WAV-Datei. Alternativ kann durch ein Anklicken des Buttons eine .WAV-Datei in einem

[9]Dabei ist zu beachten, dass momentan nur jedes zehnte Frame mit Werten ungleich Null gefüllt wird.

OpenFile Dialog auswählt werden.

- Tab Network Player:

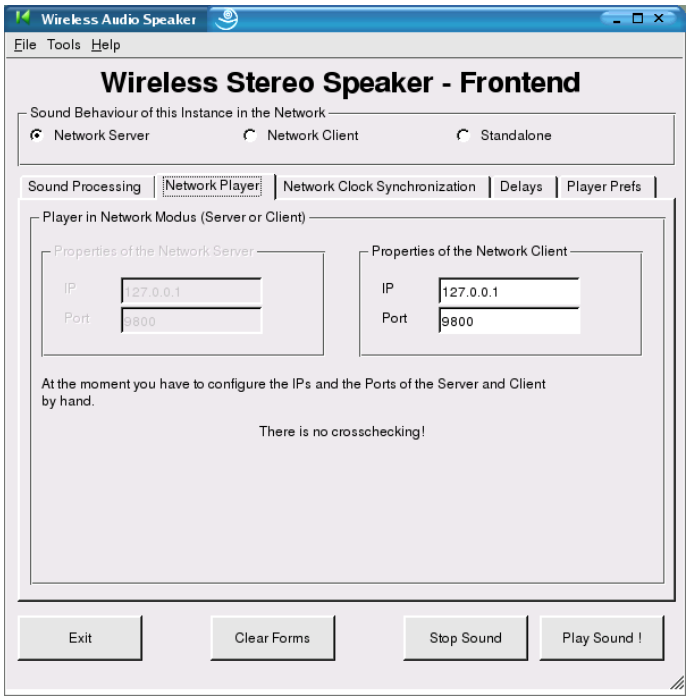


Abbildung 3.5: Tab: Network Player

  - ⋆ Properties of the Network Server:
    IP und Portnummer des *Talkers*[10]
  - ⋆ Properties of the Network Client:
    IP und Portnummer des *Listeners*[11]

- Tab Network Clock Synchronization:

  Wie im Kapitel 2 über Uhrensynchronisation beschrieben wurde, können hier die Parameter des Synchronisationsprogramms eingestellt werden. Mit Start Server, bzw. Client wird der Synchronisationsalgorithmus im Hintergrund gestartet; mit Stop Server, bzw. Client dementsprechend gestoppt. Die Status-Icons zeigen den aktuellen Zustand der Synchronisation an.

[10] Diese Einträge sind aktiviert, wenn Network Client ausgewählt wurde.
[11] Diese Einträge sind aktiviert, wenn Network Server ausgewählt wurde.

Abbildung 3.6: Tab: Network Clock Synchronization

- Tab Delays:
  Wenn eine absichtlich gewünschte Verzögerung der Ausgabe beim Client erwünscht ist, kann diese mittels der folgenden Felder eingegeben werden
  - ⋆ slider: Schiebebalken, auf welchem man die Verzögerung in Millisekunden einstellen kann. Die LCD-Box rechts vom Balken zeigt den gewählten Wert an.
  - ⋆ nummeric: nummerische Eingabe der Verzögerung in Mikrosekunden.
- Tab Player Prefs:
  Der implementierte WAS Player besitzt für .WAV - Dateien keine automatische Formaterkennung der Codierung. Als Hilfsmittel kann eine solche Formaterkennung mit dem Wavechecker Tool[12] durchgeführt und anschliessend im aktuellen Tab manuell eingestellt werden.
  - ⋆ Sound Format: Wahl der Codierung und der *Byte Order*.
  - ⋆ Channels: Wahl der Anzahl Kanäle (1 ≡ mono, 2 ≡ stereo).
  - ⋆ Sample Rate: Wahl der Rate mit welcher die Samples ausgespielt werden sollen.

[12]Menu: Tools -> WaveChecker

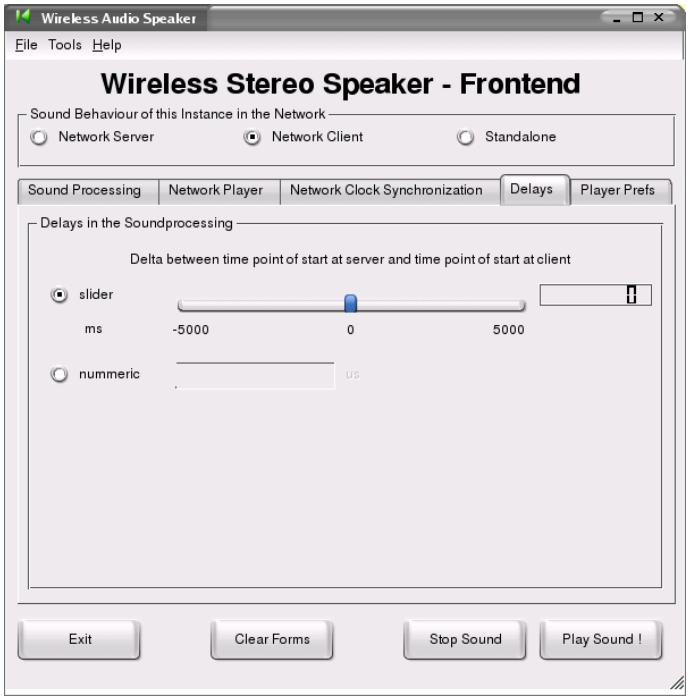


Abbildung 3.7: Tab: Delays

Abbildung 3.8: Tab: Player Prefs

3.3.3 Implementation des Wireless Stereo Speakers

Das erstellte QT GUI wurde mittels C++ programmiert und dabei das Klassenkonzept verwendet. Dadurch gab es die Möglichkeit zur Funktionsüberdeckung, welches bei der Implementation verwendet wurde.

Klassenhierarchie und Kompilierfluss

Durch *drag-and-drop* entstand in einem Entwurfeditor das vorher gezeigte Frontend, welches in einem Zwischenformat (`Frontend.ui`) zwischengespeichert wird. Das QT-Toolkit umfasst diverse Werkzeuge, welche dieses User-Interface Format in einen C++-Code umwandelt. Wie in der Abbildung 3.9 angedeutet ist, werden die Funktionsrümpfe mit der gewünschten Funktionalität überdeckt.

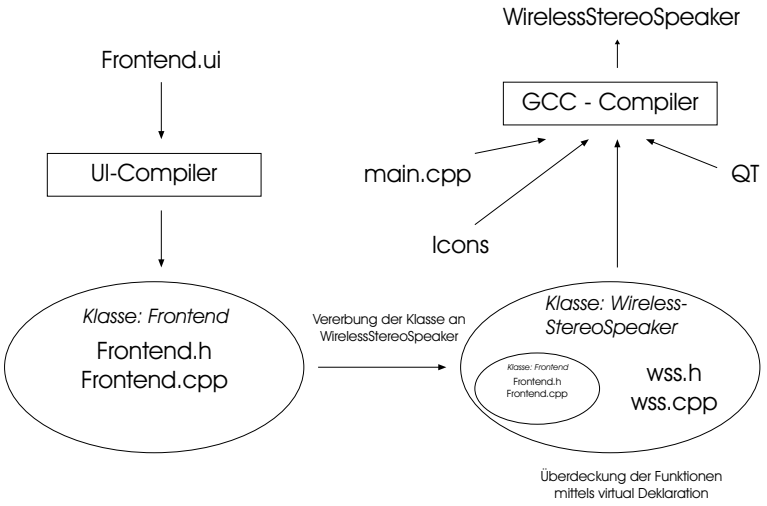


Abbildung 3.9: Zusammenhänge der Klassen und Libraries im WSS Frontend

Das WirelessStereoSpeaker Frontend ist als separater Prozess implementiert; auf den Synchronisationsalgorithmus, den Wireless Audio Speaker Player und die integrierten Tools wird mit den C-Funktionen[13] `system` und `exec` zugegriffen.

[13] Dabei wurde auf die Funktionen `fork`, `exec` und `kill` zurückgegriffen, für welche [11] verständliche Beispiele besitzt. Die Idee, den C++-Code des Frontends zu überladen entstammt der QT-Dokumentation [12].

Kapitel 4

Hardware: Audiokanalmischer

4.1 Von der Idee zum Mischer

Um die synchronisierte Audioausgabe testen zu können, müssen die Audiokanäle beider Computer dem Benutzer zur Verfügung gestellt werden: In einem ersten Test wurde dies mittels eines Knopf- und eines normalen Kopfhörers bewerkstelligt, jedoch zeigte sich bei diesem Testverfahren, dass die eingesetzten Kopfhörerlautsprecher eine sehr unterschiedliche Dynamik aufweisen und somit ein Hörvergleich schwieriger wird.

Gesucht war somit ein "Mischpult", mit welchem man die Audioausgaben der beiden Computer mischen und auf einen einzelnen Kopfhörer ausgeben kann. Die Idee war es, einen Mischer zu erstellen, welcher als Eingang die beiden Stereo-Signale entgegen nimmt, und bei welchem man mit Schalter oder Taster das Ausgangssignal zusammenstellen kann.

Für eine bessere Testbarkeit[1] wurden als Eingangsstecker Cinch-Stecker verwendet. Das "selektierte" Ausgangssignal wird auf einen 2,5 mm - Klinkenstecker geliefert, an welchem dann ein qualitativ guter Kopfhörer angeschlossen werden kann.

4.2 Mischungsmöglichkeiten

Mit der entworfenen und in Bild 4.1(a) aufgezeigten Schaltung können die in Tabelle 4.1 dargestellten Audio-Mischungen vorgenommen werden:

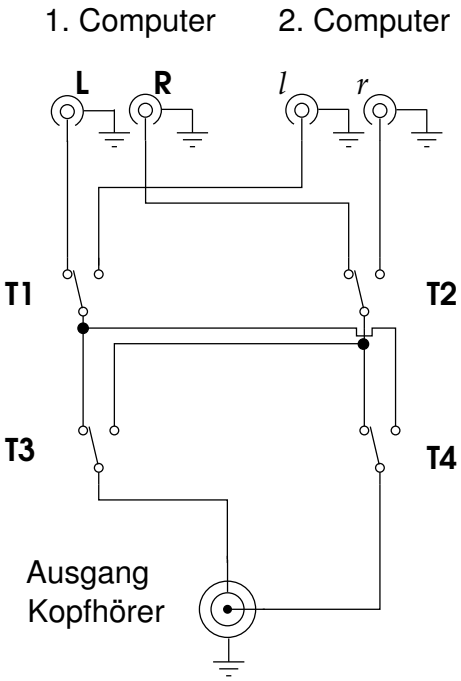
[1] Es gibt kleine Steckadapter von Cinch auf BNC, welche das Messen mit einem Oszilloskop erheblich vereinfachen und mit welchen der Messvorgang deutlich weniger Fehlerstellen aufweist.

Taster 4	Taster 3	Taster 2	Taster 1	Ausgangssignal
				$L\ R$
			√	$\mathsf{l}\ R$
		√		$L\ \mathsf{r}$
		√	√	$\mathsf{l}\ \mathsf{r}$
	√			$R\ R$
	√		√	$R\ R$
	√	√		$\mathsf{r}\ \mathsf{r}$
	√	√	√	$\mathsf{r}\ \mathsf{r}$
√				$L\ L$
√			√	$\mathsf{l}\ \mathsf{l}$
√		√		$L\ L$
√		√	√	$\mathsf{l}\ \mathsf{l}$
√	√			$R\ L$
√	√		√	$R\ \mathsf{l}$
√	√	√		$\mathsf{r}\ L$
√	√	√	√	$\mathsf{r}\ \mathsf{l}$

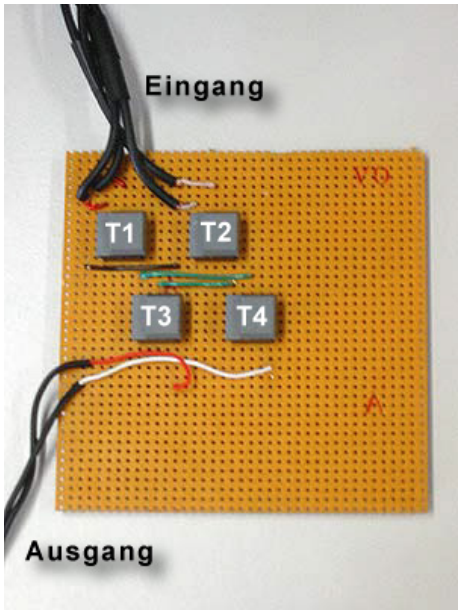
Tabelle 4.1: Ausgangsalphabet des gebauten Audiomischers. Dabei wurden die Stereo-Kanäle des ersten Computer mit L und R abgekürzt, die Kanäle des zweiten mit l und r.

Aus der Tabelle ist zu erkennen, dass alle Kombinationen der beiden Computer so gemischt werden können, dass ein direkter Vergleich linker versus rechter Kanal, beziehungsweise vorher gegen nachher gezogen werden kann.

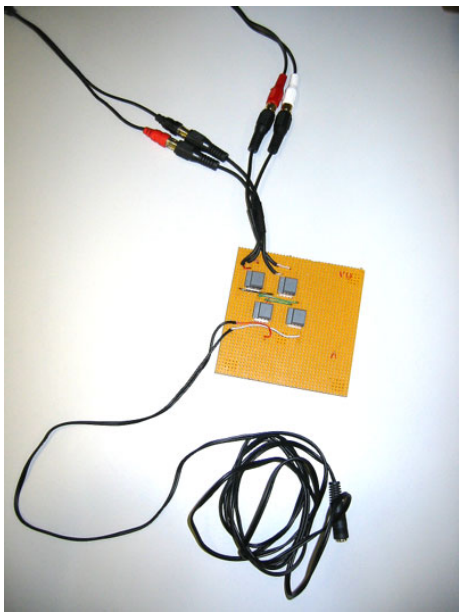
4.3 Blockschema und Prototypenboard



(a) Blockschema



(c) Vergrösserte Ansicht des Prototypenboards



(d) Prototypenboard

Abbildung 4.1: Prototypenboard für den Audio-Mischer: Durch die vier Knöpfe des Prototypenboards (mit T1 bis T4 bezeichnet), kann das linke und rechte Audiosignal des Ausganges beliebig aus den Eingangs-Stereosignalen gemischt werden. Genaueres in der Tabelle 4.1.

Kapitel 5

Ausblick und Fazit

Rückblickend betrachtet sind wir mit den von uns erreichten Resultaten zufrieden und sind der Meinung, dass wir die gesteckten Ziele einen Demonstrator zu erstellen erreicht haben.

Nichtsdestotrotz ist das System ausbaufähig. Folgende, nicht abschliessende, Aufzählung soll für weitere Gedankengänge anregen:

- Feintuning der Algorithmenparameter:
  Die momentan verwendeten Parameterwerte wurden aufgrund mehrerer Messungen empirisch bestimmt und müssen je nach Systemumgebung angepasst werden. Eine grössere Messserie würde wahrscheinlich bessere Resultate bei der Synchronisation liefern.

- Synchronisationsalgorithmus unter erhöhter Last:
  Unsere Messungen zeigten, dass der Algorithmus unter Last unbefriedigende Resultate lieferte. Die von uns getätigte Implementation scheint im Prinzip (ohne Last) zu funktionieren, jedoch haben wir, in Zusammenarbeit mit Philipp Blum, beim vertieften Testen festgestellt, dass gewisse Werte – für uns nicht nachvollziehbar – gerundet werden. Wir gehen davon aus, dass dieses Problem mit einem zusätzlichen Zeitaufwand zu lösen ist.

- Andere Algorithmen als den LS implementieren, beziehungsweise hinzufügen

- In der Implementation die Synchronisations-Pakete bevorzugen:
  Die Einführung einer Paket-Priorisierung (Stichwort QoS) würden bei Last dem Synchronisationsalgorithmus ermöglichen seine Pakete zuverlässig und ohne grössere Delays zu empfangen.

- Frontend für Server und Client:
  Momentan muss sowohl auf Server-, wie auch auf der Client-Seite das Frontend für die Demonstration gestartet und entsprechend konfiguriert werden. Mit etwas Aufwand könnte man das Frontend so gestalten, dass die Einstellungen in geeigneter Form vom Server an die Clients verteilt werden.

- Resampling und Resynchronisation:
  Wie schon beim Player erwähnt wurde (Abschnitt 2), ist in der gegenwärtigen Implementation weder *Resynchronisation* noch *Resampling* vorgesehen.

Anhang A

Setup

A.1 Rechner

- PC: ein Pentium III 1 GHz sowie ein Pentium III 800 MHz

A.2 Soundkarten

- Soundblaster PCI 64 von Creative Labs
- Chipset: ES1371

A.3 Software

- Linux OS: Suse 8.1
- C: GCC 3.2
- C++: GCC 3.2
- ASM: Nasm
- ALSA: Version 0.9.2

A.4 Wireless LAN

- Wirelesskarten: Orinoco Gold 11 Mbit/s
- statische IP-Adressen
  - Server: `192.168.1.5`
  - Client: `192.168.1.4`
- Subnetmask: `255.255.255.0`

- Wireless Settings
  - Operating mode: `ad-hoc`
  - Network name (ESSID): `SYNC_TIK`
  - Encryption key: `s:1asdas`

Anhang B

Messresultate

B.1 Uhrensynchronisation

Messung 1: mit Abbruch des Algorithmus

Die Synchronisation wurde vor der Parallelport-Messung gestartet, für 9 min 30 s (570 s) laufengelassen, danach abgebrochen und die Messung für weitere 30 s laufengelassen. Aus Abbildung B.1 (unten) ist gut ersichtlich, dass sobald der Algorithmus nicht mehr aktiv ist, die beiden Uhren auseinanderdriften. Abbildung B.1 (oben) zeigt die gewählte Konfiguration.

Messung 2: ohne Algorithmus

Es wurde für eine Dauer von 2 min eine Parallelport-Messung durchgeführt, ohne die Synchronisation laufen zu lassen. In Abbildung B.2 (unten) ist das Resultat graphisch dargestellt. Wie zu erwarten war, driften beide Uhren mit zunehmender Zeit immer mehr auseinander. Abbildung B.2 (oben) zeigt die gewählte Konfiguration.

Messung 3: ohne Abbruch des Algorithmus

Die in Abbildung B.3 (unten) gezeigte, 10 min dauernde, Messung wurde unter normaler Kunfiguration (siehe Abbildung B.3 (oben)) und bei laufendem Synchronisationsalgorithmus durchgeführt. Es ist zu erkennen, dass sich die Kurve mit der Zeit ein wenig der x-Achse nähert. Dies hat damit zu tun, dass ein immer besserer Wert für die minimale RTT gefunden wurde. Die mittlere Abweichung liegt bei zirka $+15\mu s$.

Messung 4: mit Last

Hier wird gezeigt wie sich die von uns implementierte Version des Algorithmus unter Last verhält. Als erstes wurde der Algorithmus und dann die der Messvorgang gestatet. Nach zirka einer Minute haben wurde eine 50 MB grosse Datei über das

Parameter	**gewählter Wert**
Nachrichtenintervall [ms]	50
`deltaTMaxSum` (ΔT_{maxSum}) [ns]	10^8
`kappa1` (κ_1) [1]	10^{-2}
`kappa2` (κ_2) [$\frac{1}{\text{ns}}$]	10^{-11}
`MAXDRIFT` [ppm]	50
Last	nein
Netzwerktyp	WLAN
Messintervall [ms]	1000

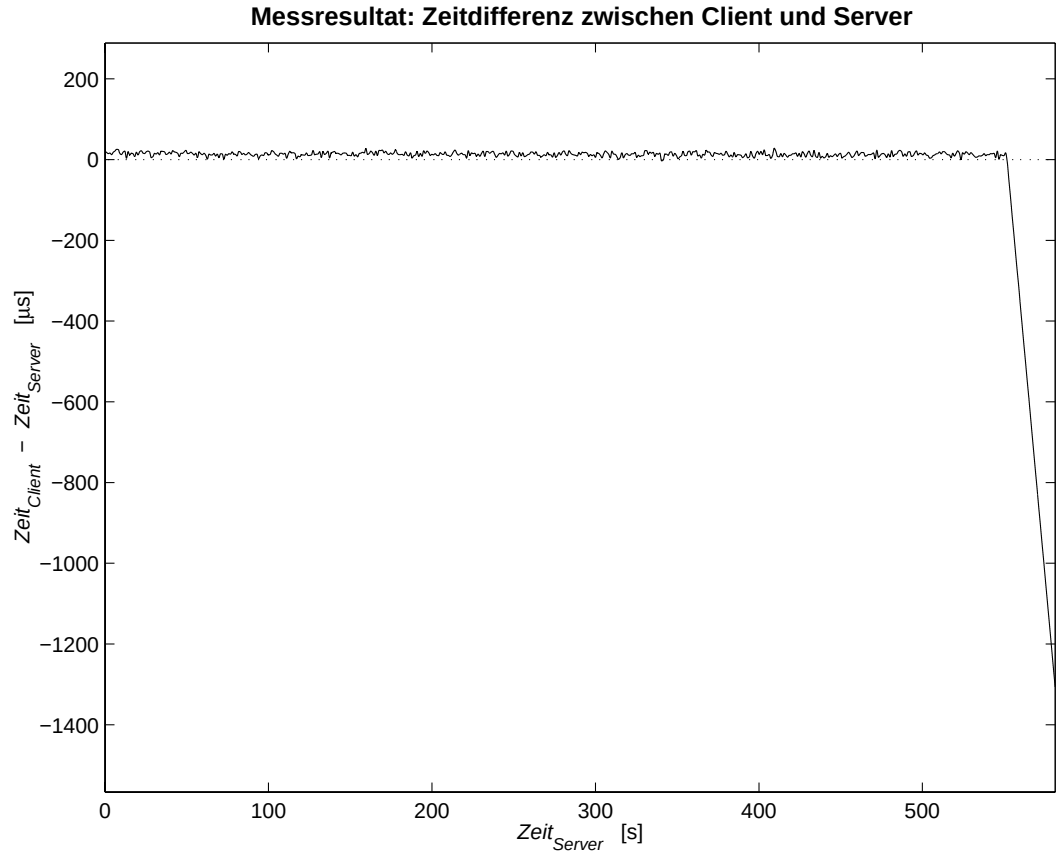


Abbildung B.1: Oben: Konfiguration Messung 1 (mit Abbruch des Algorithmus); Unten: Resultat Messung 1 (mit Abbruch des Algorithmus)

Parameter	gewählter Wert
Nachrichtenintervall [ms]	50
`deltaTMaxSum` (ΔT_{maxSum}) [ns]	10^8
`kappa1` (κ_1) [1]	10^{-2}
`kappa2` (κ_2) [$\frac{1}{\text{ns}}$]	10^{-11}
`MAXDRIFT` [ppm]	50
Last	nein
Netzwerktyp	WLAN
Messintervall [ms]	1000

Abbildung B.2: Oben: Konfiguration Messung 2 (ohne Algorithmus); Unten: Resultat Messung 2 (ohne Algorithmus)

Parameter	gewählter Wert
Nachrichtenintervall [ms]	50
`deltaTMaxSum` (ΔT_{maxSum}) [ns]	10^8
`kappa1` (κ_1) [1]	10^{-2}
`kappa2` (κ_2) [$\frac{1}{\text{ns}}$]	10^{-11}
`MAXDRIFT` [ppm]	50
Last	nein
Netzwerktyp	WLAN
Messintervall [ms]	1000

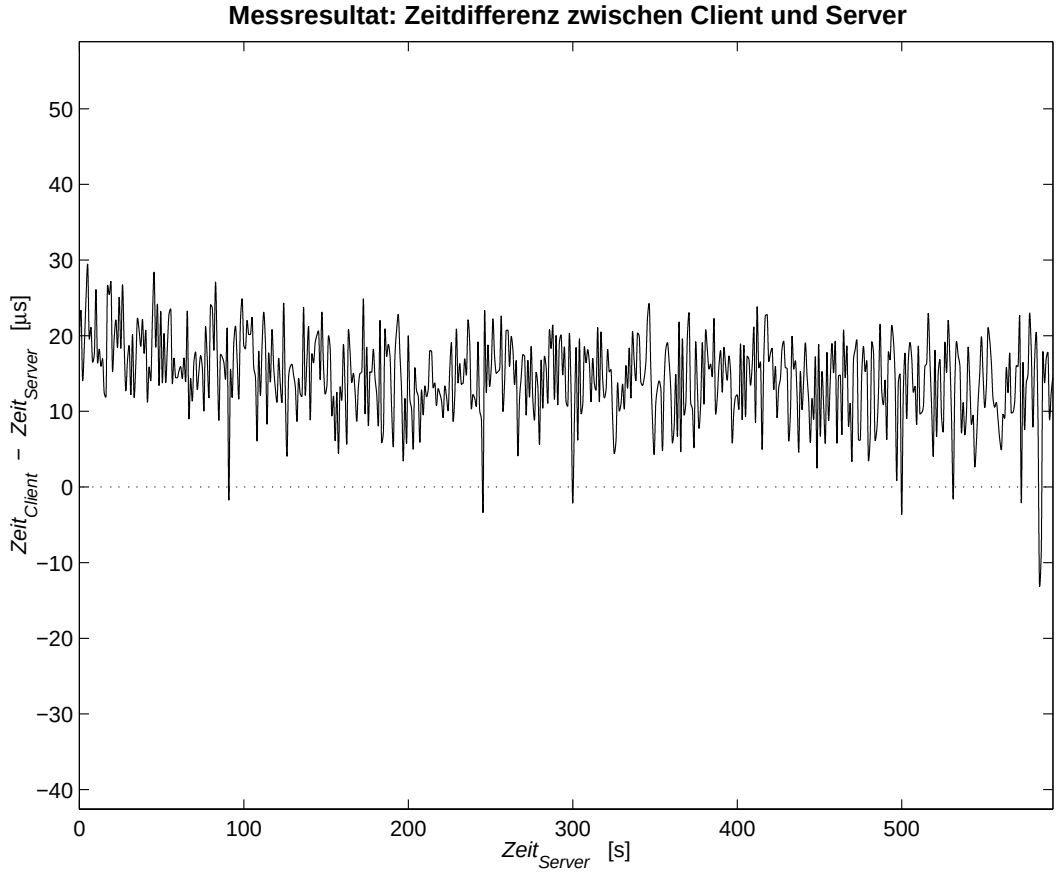


Abbildung B.3: Oben: Konfiguration Messung 3 (ohne Abbruch des Algorithmus); Unten: Resultat Messung 3 (ohne Abbruch des Algorithmus)

Wireless LAN gesendet. In Abbildung B.4 (unten) wird deutlich, dass die Qualität der Synchronisation während einer grossen Netzbelastung unbefriedigend ist. Abbildung B.4 (oben) zeigt die gewählte Konfiguration.

Messung 5: grosses Nachrichtenintervall

Wird das Intervall mit dem der Server seine Zeitstempel veschickt zu gross, so wird das Resultat unbefriedigend. Abbildung B.5 (unten) zeigt wie die Uhren zueinander stehen, wenn nur alle Sekunden eine Nachricht versendet wird. Die verwendete Konfiguration ist aus Abbildung B.5 (oben) ersichtlich.

Messung 6: Änderung von κ_1

In dieser Messung haben wir den Parameter κ_1 verändert. Verglichen mit der Messung in Abschnitt B.1 ist nur eine geringfügige Verschlechterung der Synchronisationsqualität festzustellen. Abbildung B.6 (unten) zeigt die graphische Darstellung dieser Messung, Tabelle B.6 (oben) die gewählten Parameterwerte.

Messung 7: Änderung von κ_2

Für die in dieser Messung verwendete Konfiguration (vergleiche Abbildung B.7 (oben)) haben wir den Wert von κ_2 viel zu klein gewählt, so dass mit fortschreitender Zeit nur noch der zweite Fall im Algorithmus eintrat, und somit die Uhr des Clients wegdriftete. Dies ist eindrücklich in Abbildung B.7 (unten) zu sehen.

Parameter	gewählter Wert
Nachrichtenintervall [ms]	50
`deltaTMaxSum` (ΔT_{maxSum}) [ns]	10^8
`kappa1` (κ_1) [1]	10^{-2}
`kappa2` (κ_2) [$\frac{1}{\text{ns}}$]	10^{-11}
`MAXDRIFT` [ppm]	50
Last	ja
Netzwerktyp	WLAN
Messintervall [ms]	1000

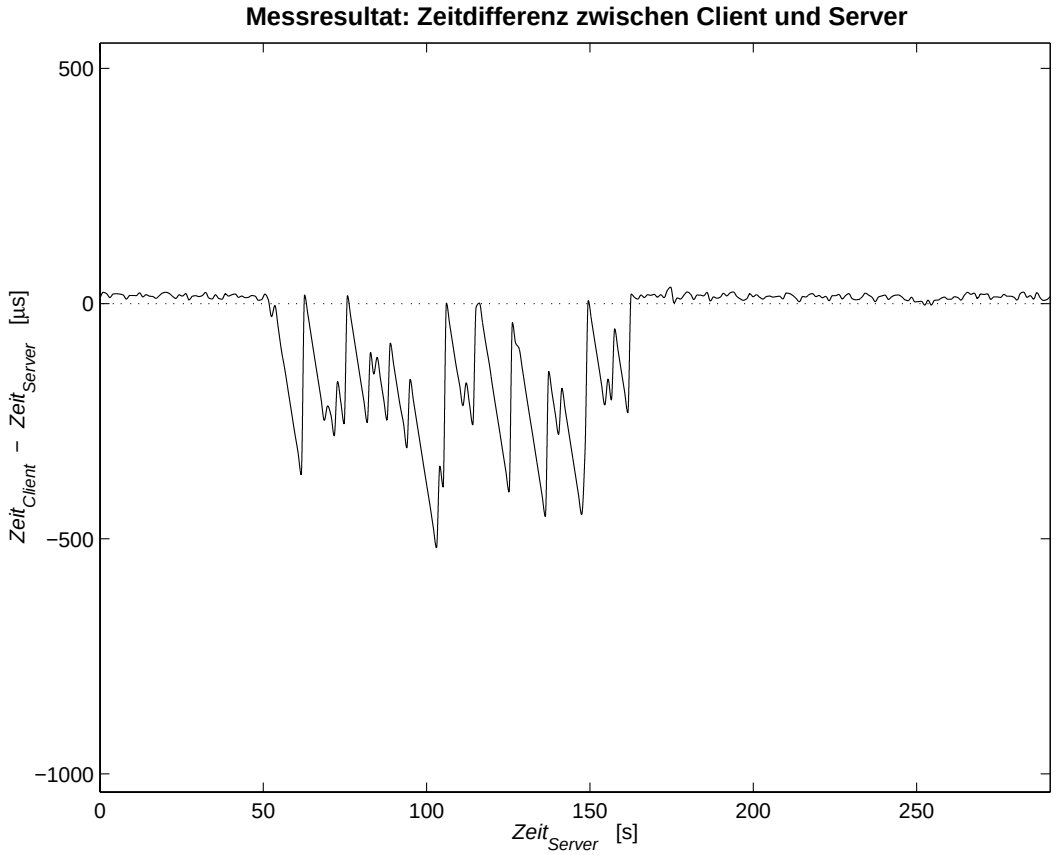


Abbildung B.4: Oben: Konfiguration Messung 4 (mit Last); Unten: Resultat Messung 4 (mit Last)

Parameter	**gewählter Wert**
Nachrichtenintervall [ms]	1000
`deltaTMaxSum` (ΔT_{maxSum}) [ns]	10^8
`kappa1` (κ_1) [1]	10^{-2}
`kappa2` (κ_2) [$\frac{1}{\text{ns}}$]	10^{-11}
`MAXDRIFT` [ppm]	50
Last	nein
Netzwerktyp	WLAN
Messintervall [ms]	1000

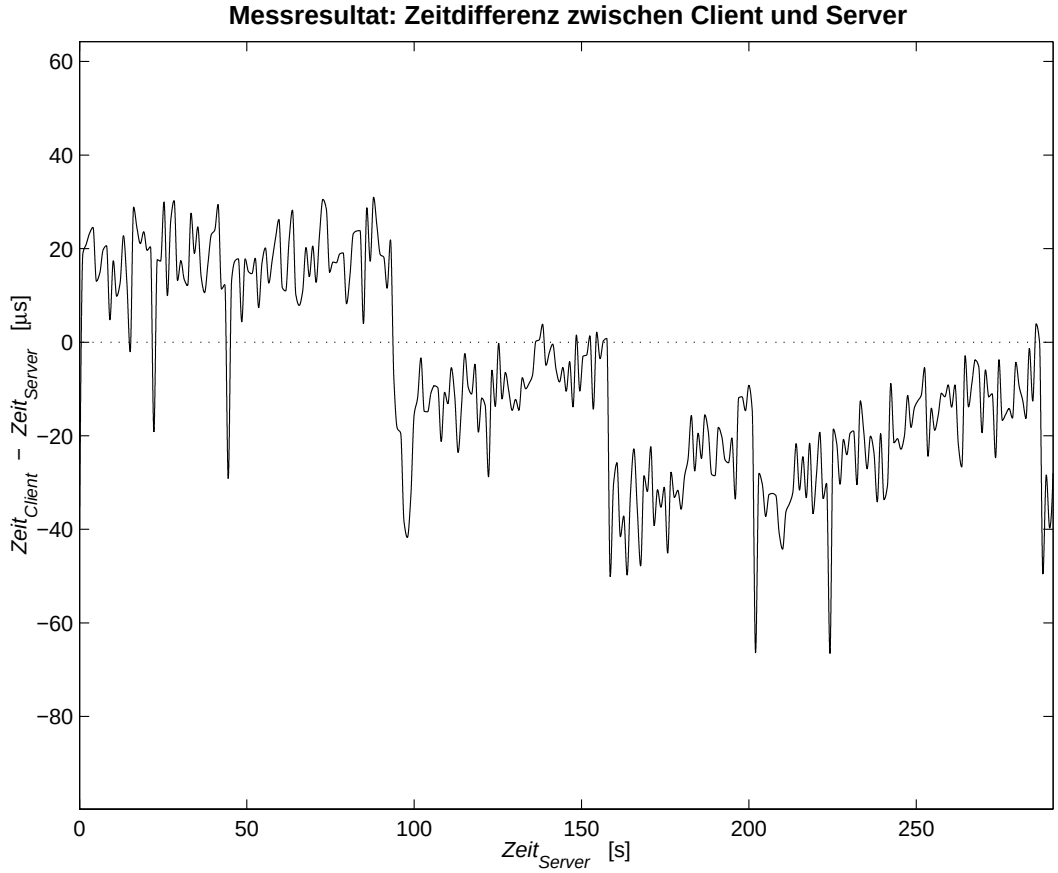


Abbildung B.5: Oben: Konfiguration Messung 5 (grosses Nachrichtenintervall); Unten: Resultat Messung 5 (grosses Nachrichtenintervall)

Parameter	gewählter Wert
Nachrichtenintervall [ms]	50
`deltaTMaxSum` (ΔT_{maxSum}) [ns]	10^8
`kappa1` (κ_1) [1]	10^{-4}
`kappa2` (κ_2) [$\frac{1}{\text{ns}}$]	10^{-11}
`MAXDRIFT` [ppm]	50
Last	nein
Netzwerktyp	WLAN
Messintervall [ms]	1000

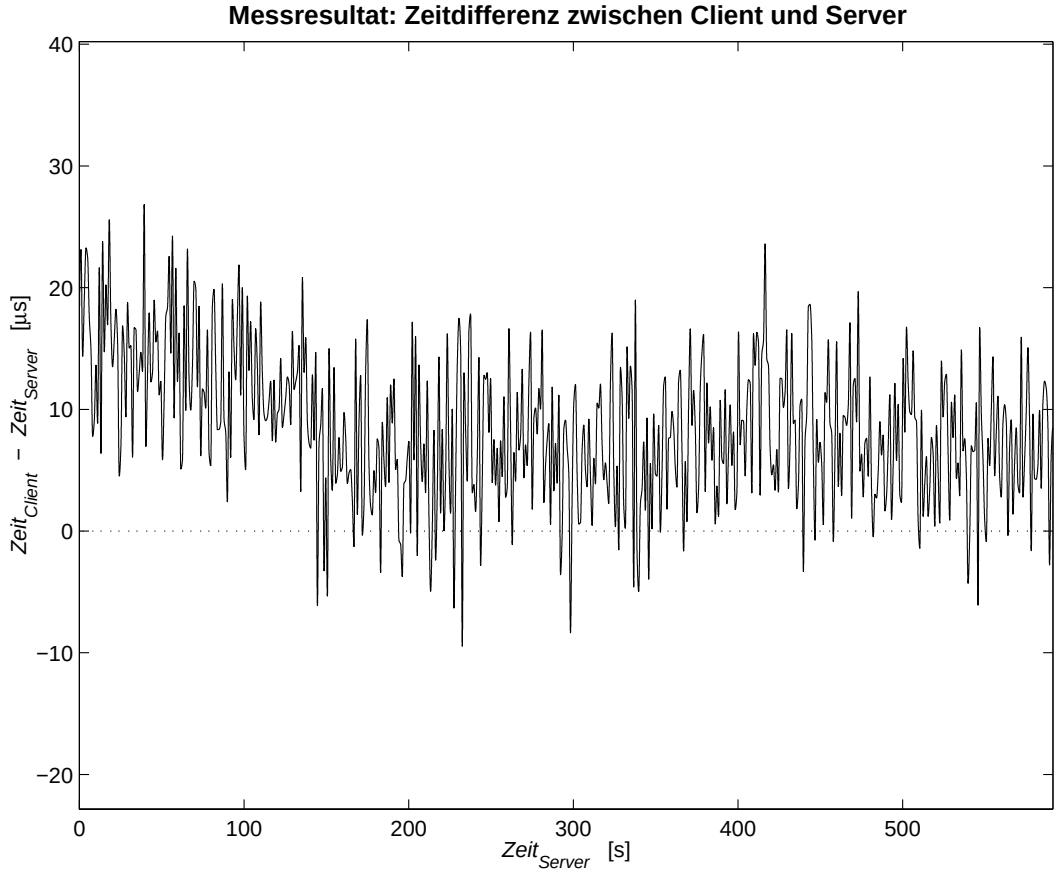


Abbildung B.6: Oben: Konfiguration Messung 6 (Änderung von κ_1); Unten: Resultat Messung 6 (Änderung von κ_1)

Parameter	**gewählter Wert**
Nachrichtenintervall [ms]	50
`deltaTMaxSum` (ΔT_{maxSum}) [ns]	10^8
`kappa1` (κ_1) [1]	10^{-2}
`kappa2` (κ_2) [$\frac{1}{\text{ns}}$]	10^{-20}
`MAXDRIFT` [ppm]	50
Last	nein
Netzwerktyp	WLAN
Messintervall [ms]	1000

Abbildung B.7: Oben: Konfiguration Messung 7 (Änderung von κ_2); Unten: Resultat Messung 7 (Änderung von κ_2)

B.2 Audioplayer

Die Fragen die bei der Implementation des Audioplayers auftauchen waren unter anderem:

- Wie sehen die digitalen Samples nach der Audiokarte aus. Zum Beispiel ist eine generierter Sinusstream auch nach der D/A Wandlung immer noch ein exakter Sinus?
- Füllen die implementierten Funktionen jeweils das ganze Frame mit Werten oder beginnt, beziehungsweise endet, der Datenstrom dazwischen?
- Wie gross sind die Host-spezifischen Delays? Die Verzögerungen zwischen den Hosts?
- Wie gross ist der Drift?
- Wie gut synchronisiert ist denn nun der Ausspielzeitpunkt?

Um diese Fragen zu beantworten wurde die Audioausgabe zuerst auf dem Oszilloskop betrachtet und anschliessen für diverse Zeitmessend ein Logic Analyzer benützt. Im Folgenden möchten wir exemplarisch ein paar Messungen anhand des Sinussignales zeigen.

B.2.1 Abspielen von generierten Wellenformen

Wie in den Abschnitten 3.2.2 und C erwähnt, kann der WAS Player sowohl mit IO Calls als auch mit MMAP ALSA-Funktionen die Sounddaten ausspielen. Im Folgenden möchten wir diese beiden Varianten vorstellen.

1. Abspielen eines Frames mit den IO Call Funktionen, wird in der Abbildung B.8 gezeigt.
2. Analog zum vorherigen Abspielen mit der IO Call Funktion (Abb. B.8) hier ein Beispiel, bei welchem mit den MMAP Funktionen die Audiowiedergabe bewerkstelligt wird (Abb. B.9).

   Die gewählten 500'000ns Buffergrösse erwiess sich als ein ungünstige gewählter Wert, welcher phänomenologisch betrachtet eine Art Bufferüberlauf oder Aliasing hervorrief. Frames wurden in regelmässigen Abständen nicht ganz gefüllt[1]. Durch eine Testserie stellte sich heraus, dass ein Buffer von 300'000ns dieselben Ausspielcharakteristik hat (Ausspielzeitpunkt der Samples nach Generierung), bei dieser Grösse aber keine der beobachteten und unerwünschten Phänome auftreten.

[1] Weitere Untersuchungen dieses Verhaltens wurden nicht bewerkstelligt. Eventuell handelt es sich bei diesem Verhalten um einen Fehler in ALSA. Prinzipiell basiert die MMAP Funktionen auf einem Ringbuffer, dessen Wert nach unserer Ansicht innerhalb einer Schranke "beliebig" gewählt werden kann.

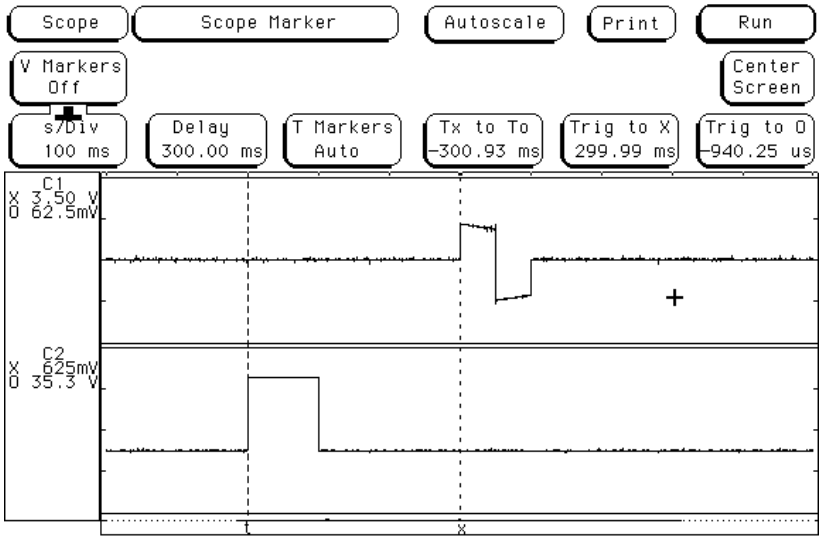


Abbildung B.8: Rechtecksfunktion mit `writei` ausgespielt. Die Buffergröesse wurde auf eine Grösse von 500'000 ns gesetzt. Das Audiosignal - eine Schrittfunktion - ist unter `C1` dargestellt. Das Signal `C2` entspricht einem Flag, welches bei gerade vor der softwareseitigen Audioausgabe gesetzt wurde. Die Verzögerung zwischen dem Softwarebefehl und dem entgültigem Ausspielen beträgt somit rund 300ms.

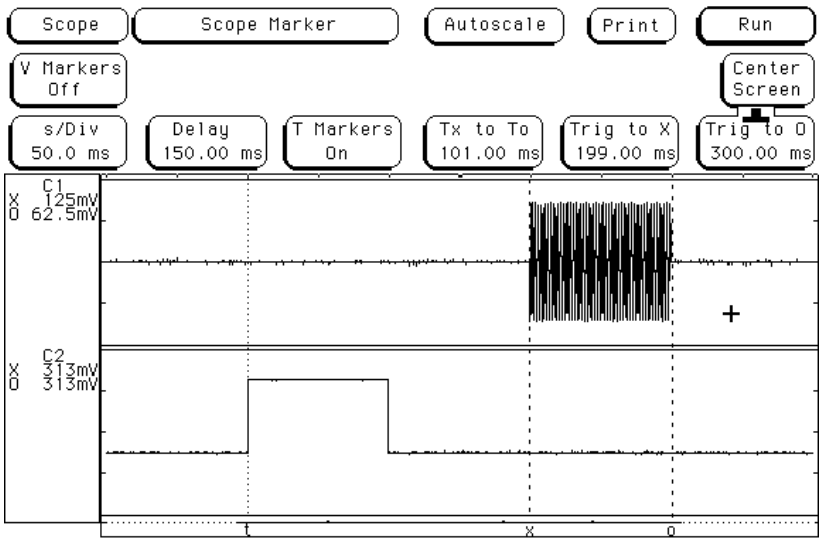


Abbildung B.9: Sinusfunktion mit `mmap_commit` ausgespielt mit einer gewählten Buffergröesse von 500'000ns. Das Audiosignal - eine Sinusfunktionen - ist unter `C1` dargestellt. Das Signal `C2` entspricht einem Flag, welches bei gerade vor der softwareseitigen Audioausgabe gesetzt wurde. Die Verzögerung zwischen dem Softwarebefehl und dem entgültigem Ausspielen beträgt somit rund 300ms.

Als nächstes folgt die exemplarische Sinusfunktion, welche genauer auf dem Oszilloskop untersucht wurde. Die Sinusfunktion wurde im Test mit den MMAP Funktionen der ALSA Library ausgespielt und dabei die Puffergrösse auf 300'000 ns festgesetzt.

Die Abbildungen B.10 bis B.13 zeigen jeweils die gleiche Sinusfunktion in unterschiedlichen Messungen. In Messungen des Oszilloskop entspricht jeweils die erste horizontale Welle, welche mit `C1` markiert ist, dem erwünschten Audiosignal; mit `C2` wird der Wechsel eines internen Software-Flags (gesetzt gerade bevor die Ausspielfunkion aufgerufen wird) dargestellt.

In der ersten Messung (Abb. B.10) wird ein Frame dargestellt, welches nach dem Konzept der Hörprobe (Abschnitt 3.2.5) erstellt wurde: 100ms wurden mit einer Sinuswelle gefüllt, die Werte vor und nach diesem speziellen Frame sind Null. Abbildung B.11 entspricht einem Strom von mehreren Sinuszügen: Jedes zehnte Frame entspricht dem unter Abbildung B.10 genauer gezeigten Frame. Die beiden Abbildungen B.12 und B.13 zeigen den Anfang und das Ende der Sinusschwingung. Es ist zu erkennen, dass der Phasengang sowohl bei Null beginnt, wie auch endet.

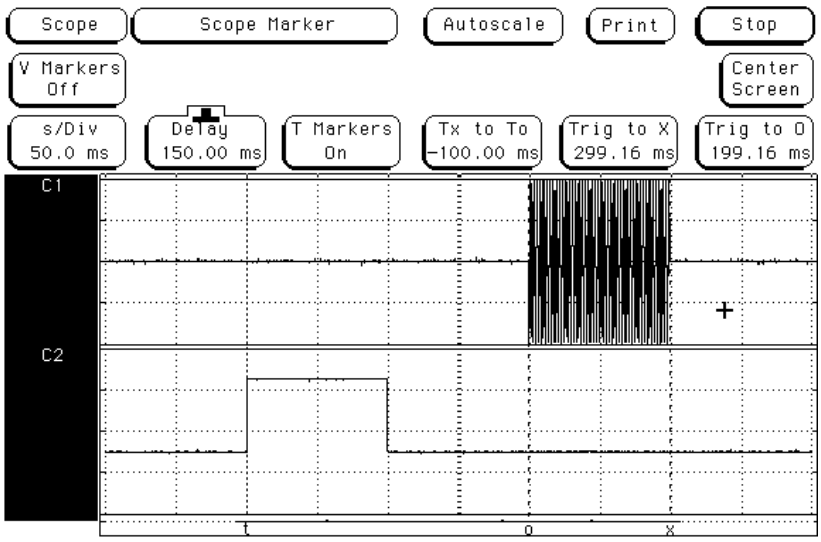


Abbildung B.10: Einzelnes Frame mit Sinuswerten gefüllt - nur $\frac{1}{10}$ aller Frames sind mit Werten ungleich Null gefüllt. Unter dem Eintrag `Tx to To` ist ersichtlich, dass die Schwingung 100 ms dauert, also gerade ein Frame lange. Die Ausgabe startet ungefähr 200 ms (`Trig to O`)nach dem `commit` Befehl und endet dementsprechend bei zirca 300 ms. `C1` entspricht dem Audiosignal - `C2` einem Softwareflag.

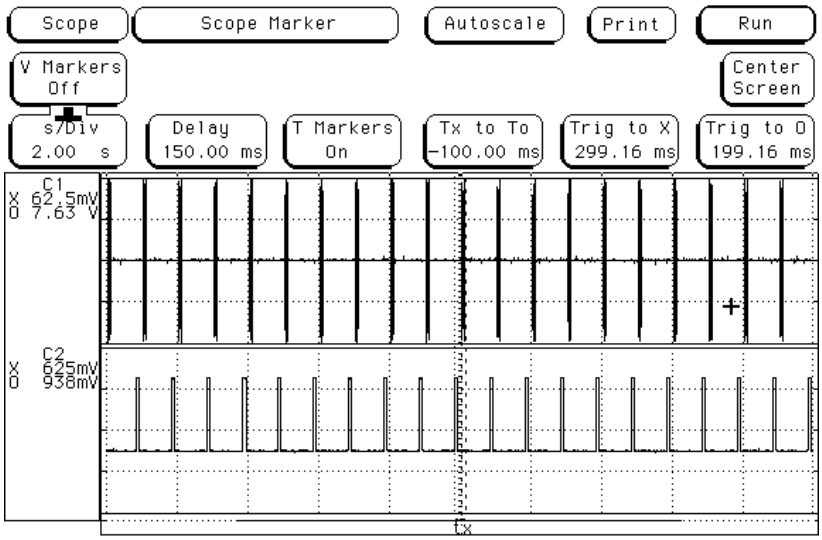


Abbildung B.11: Folge von mehreren Frames, welche jeweils mit der vorherigen Sinusschwingung gefüllt sind. `C1` entspricht dem Audiosignal - `C2` einem Softwareflag.

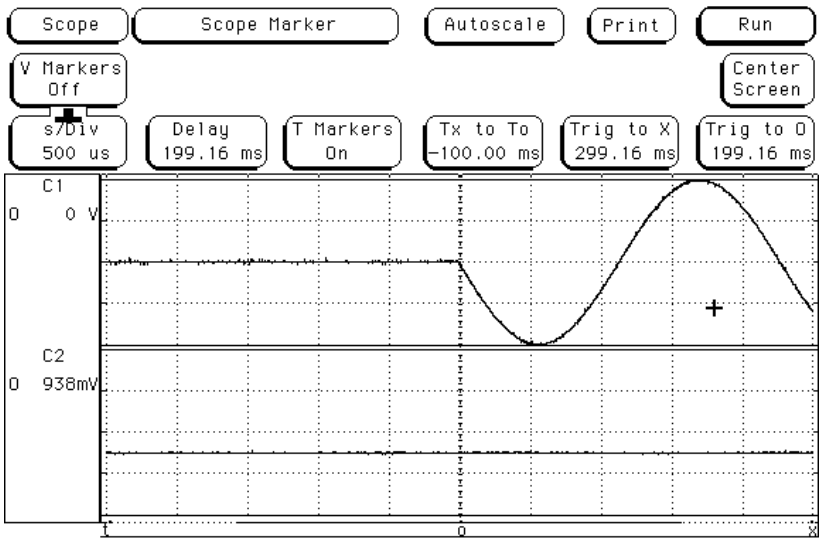


Abbildung B.12: Beginn der Sinusschwingung (Phase 0) bei 199.16 ms. `C1` entspricht dem Audiosignal - `C2` einem Softwareflag.

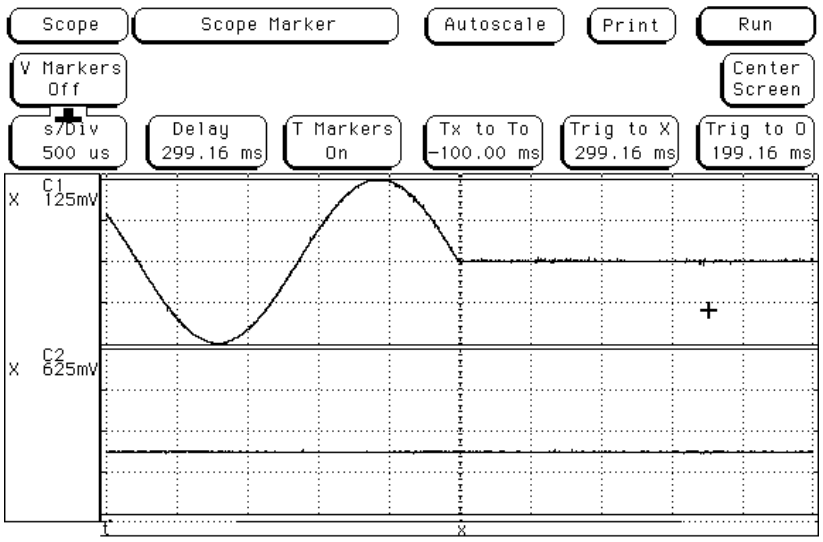


Abbildung B.13: Ende der Sinusschwingung bei 299.16 ms und in Phase. `C1` entspricht dem Audiosignal - `C2` einem Softwareflag.

B.2.2 Computerabhängige Verzögerungen der Audioausgabe

Die für den Demonstrator vorhandenen Computer hatten unterschiedliche Geschwindigkeiten (1000 MHz vs. 800 MHz). Eine zentrale Frage war jetzt, wie sich dieser Unterschied auf die Audioausgabe auswirkt, dh. wie gross das Delta zwischen den *Startzeitpunkten* des Softwarecodes im Userspace und der effektiven *Audioausgaben* an den Soundkarten jetzt ist.

Für das Messen wurde ein Logic Analyzer verwendet, da dieser die Möglichkeit bot, mehr als nur zwei "Kanäle" miteinander zu vergleichen. Das für die folgende Messung gewählte Setup verwendet eine Signalisation am Parallelport[2] und das schlussendlich auf den beiden Audiokanälen erscheinende Signal.

In der Abbildung B.14 wird für den ersten Computer (800 MHz) das Delay zwischen dem softwareseitigen `commit` Befehl und der erfolgten Audioausgabe verglichen. Aus dem Plot ist ersichtlich, dass im Schnitt dieser Computer 496.8 μs benötigt bis ein Signal am Ausgang erscheint.

In der Abbildung B.15 wird für den zweiten Computer (1000 MHz) das Delay zwischen dem softwareseitigen `commit` Befehl und der erfolgten Audioausgabe verglichen. Aus dem Plot ist ersichtlich, dass im Schnitt dieser Computer 396.0 μs benötigt bis ein Signal am Ausgang erscheint.

[2]Mittels eines `outb` Befehls und einer Zählvariablen können einzelne Bits auf dem Parallelport gesetzt oder gelöscht werden. Dadurch ist es möglich an einer bestimmten Stelle im Software Code Trigger-Signale an die Aussenwelt zu geben.

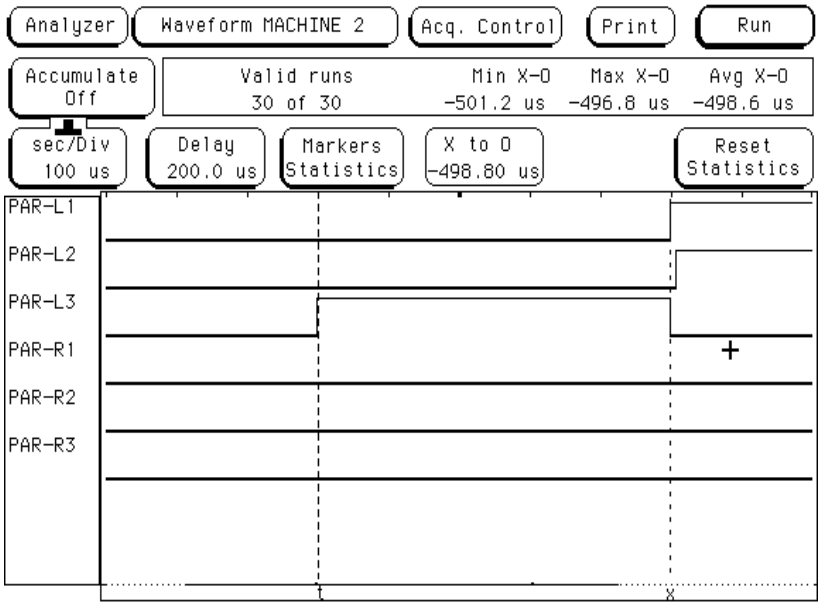


Abbildung B.14: Audioausgabe 800 MHz Computer: Im Schnitt braucht dieser Computer 496.8 μs bis ein Signal am Ausgang erscheint. Ersichtlich ist im Plot auch, dass der linke und der rechte Audiokanal nicht zur selber Zeit mit der Audiowiedergabe beginnen. Im Plot ist `PAR-L3` das Trigger-Signal der Software und `PAR-L1` der linke, bzw. `PAR-L2` der rechte Audiokanal.

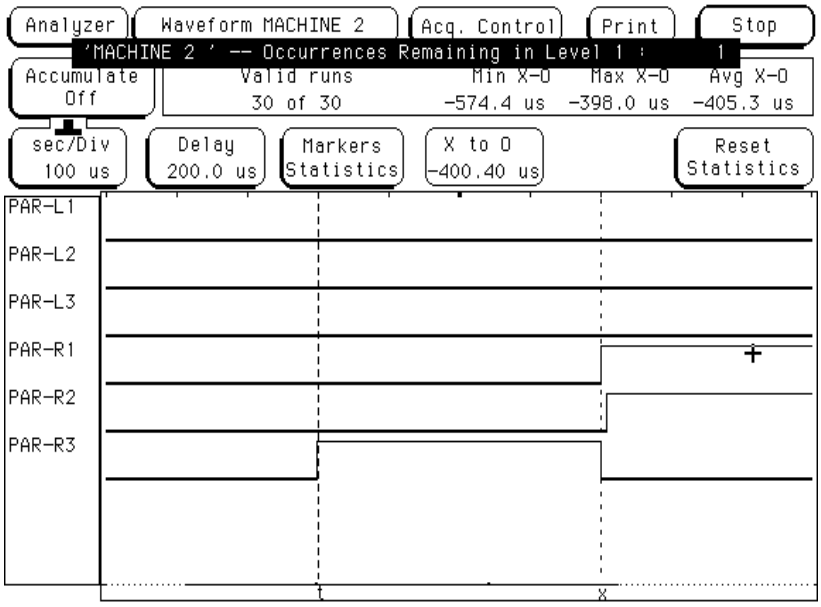


Abbildung B.15: Audioausgabe 1000 MHz Computer: Im Schnitt braucht dieser Computer 396.0 μs bis ein Signal am Ausgang erscheint. Ersichtlich ist im Plot auch, dass der linke und der rechte Audiokanal nicht zur selber Zeit mit der Audiowiedergabe beginnen. Im Plot ist `PAR-R3` das Trigger-Signal der Software und `PAR-R1` der linke, bzw. `PAR-R2` der rechte Audiokanal.

Wenn man beide PCs im direkten Vergleich zueinander aufzeichnen lässt, erhält man ein Bild wie in Abbildung B.16 dargestellt ist:

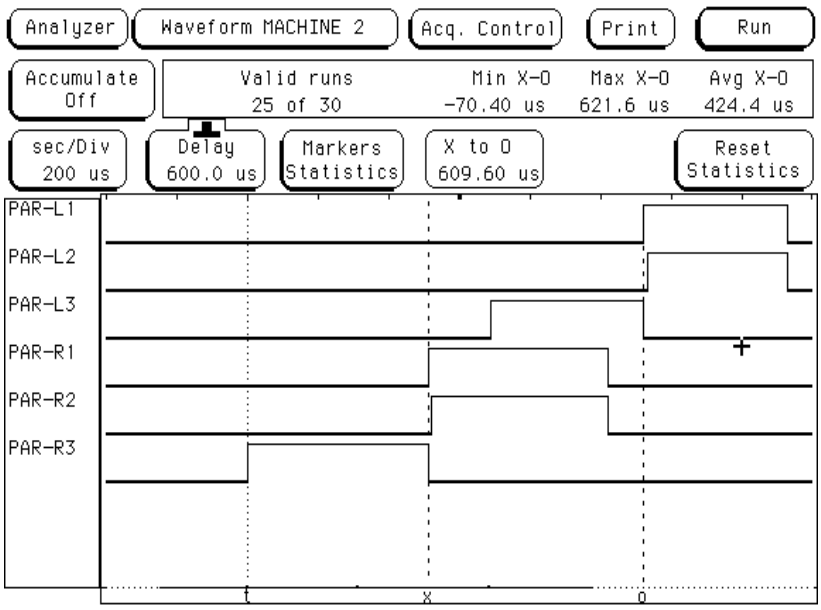


Abbildung B.16: unkorrigierte Computer im Vergleich: Deutlich ersichtlich ist, dass die beiden Computer trotz Synchronisation nicht genau zur selben Zeit ihr Audiosignal wiedergeben. Dies wird unter anderem durch folgende Ursachen hervorgerufen: Unterschiedliche Zeiten bis zur Audiowiedergabe (vgl. Abb. B.14 und Abb. B.15), kleine Abbweichungen der Clockobjekte und sicherlich auch aufgrund der Userspace Implementation.

Kurz nochmals die Resultate zusammengefasst:

- Delay zwischen Softwarebefehl und effektivem Ausspielen des 800MHz Rechner: 496.8 μs
- Delay zwischen Softwarebefehl und effektivem Ausspielen des 1000MHz Rechner: 396.0 μs
- Noch nicht korrigierte Computer im Vergleich: Durchschnittlich eine Verschiebung der Audioausgabe um 621.0 μs

B.2.3 Synchronisierte Computer

Die synchronisierten Computer (mit einem konstanten Korrekturfaktor des Abspielzeitpunktes modifiziert) spielen durchschnittlich innerhalb eines Fensters von 200μs ihre Audiodaten ab. Folgende Fotographie B.17 zeigt mehrere überlagerte Ausspielzeitpunkte. Insgesammt wurden 4 Messungen in diesem Bild überlagert; prinzipiell sollten die Flanken gleichzeitig auftreten, durch die Userspace-Implementation gibt es eine kleine, aber deutlich messbare Verschiebung. Ebenfalls der Userspace Implementation ist es zuzuschreiben, dass es zwischendurch Ausreisser gibt, d.h. in selten Fällen ergibt sich ein Delta von über einer 1ms bei der Audioausgabe. In der Abbildung entspricht `CHAN1` der Audioausgabe des ersten Computers, `CHAN2` der des zweiten.

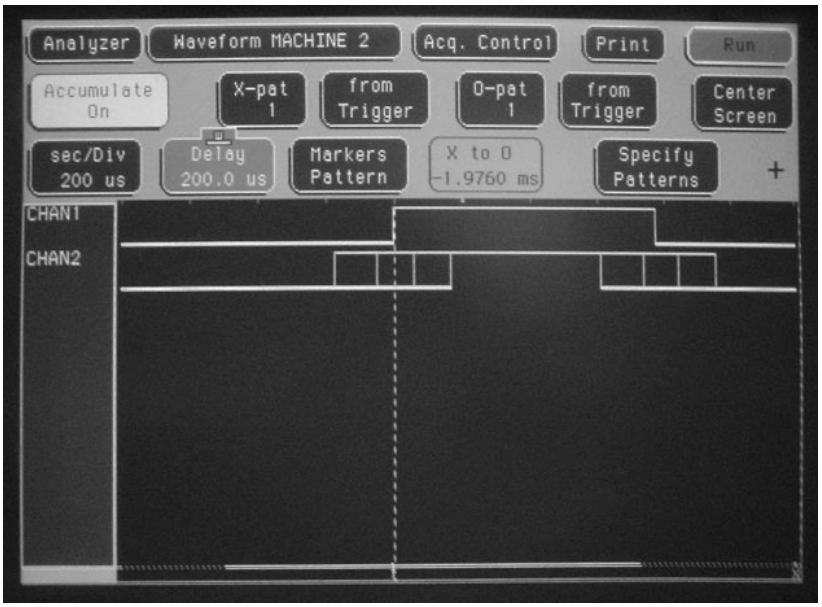


Abbildung B.17: Korrigierte Computer im Vergleich: Im Schnitt wird eine synchronisierte Ausgabe erreicht, welche eine Ungewissheit im 200μs aufweist.

B.2.4 Soundkartendrift

Wie schon kurz im Kapitel 2 (S.19) angeschnitten, driften die beiden Computer mit ihrer Audioausgabe auseinander, da die auf der Soundkarte befindlichen Oszillatoren eine gewisse Abweichung aufweisen. In der folgenden Testsequenz ist ein Drift nur schwer ersichtlich, jedoch nach einer gewissen Zeit klar hörbar: 43 Sekunden (430 Frames) nach dem synchronen Ausspielen beträgt der Drift schon 5 ms.

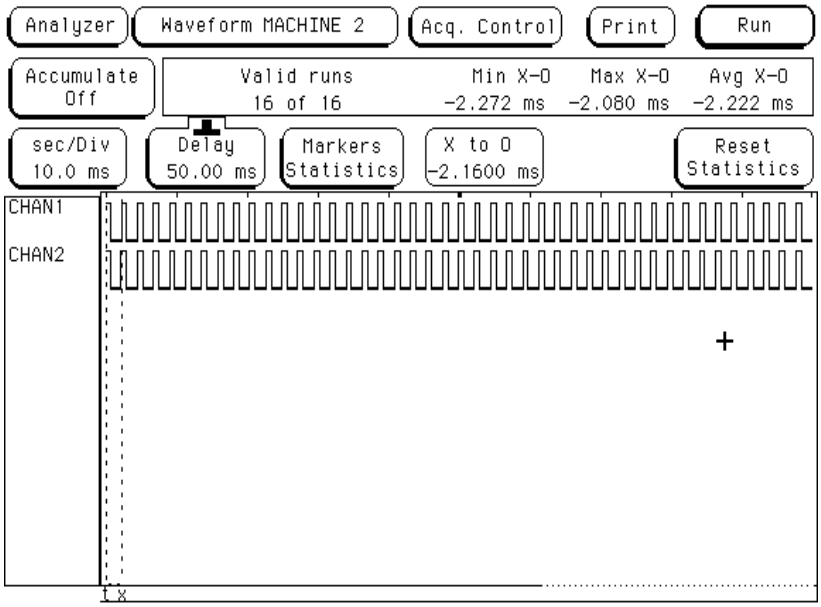


Abbildung B.18: Drift um 5 ms innerhalb der aufgezeigten Frames

Anhang C

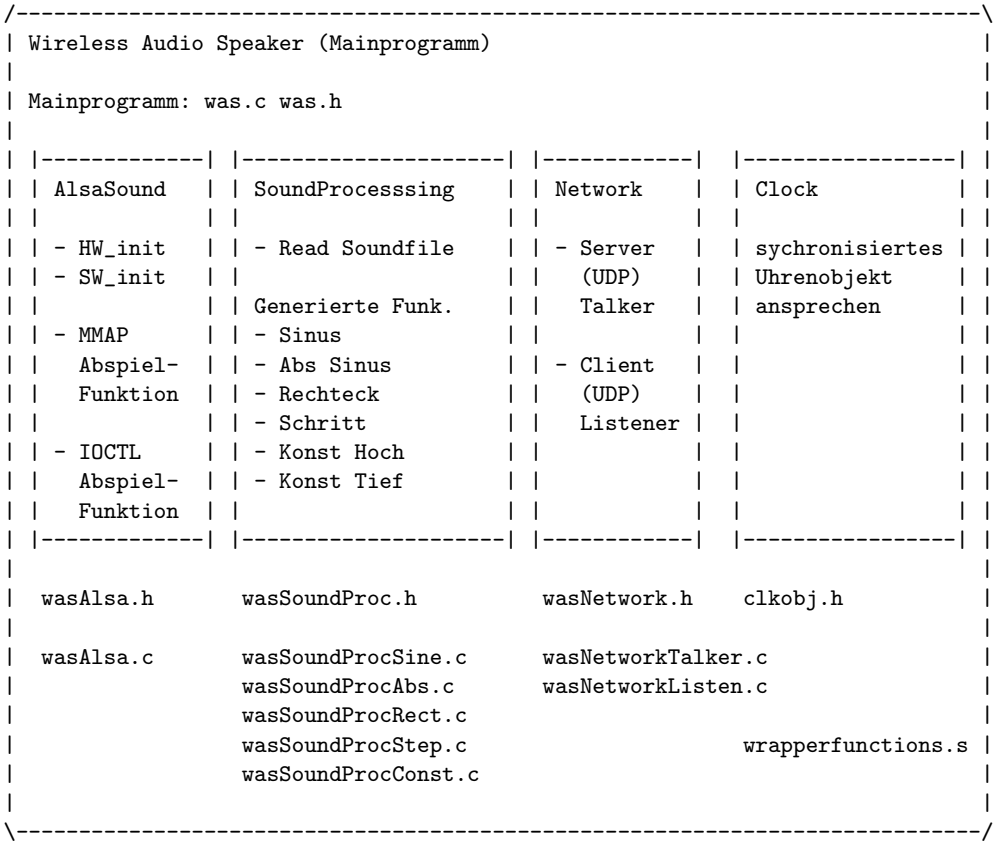
WAS Player im Detail

C.1 Überblick

Im Gegensatz zur Beschreibung für den Anwender im Abschnitt 3.2 möchten wir hier einen tieferen Einblick in den Sourcecode des WAS Players geben, um diesen gegebenenfalls in einem weiteren Projekt zu erweitern oder Codestücke davon zu verwenden.

C.1.1 Dateistruktur

Zur Errinnerung möchten wir die schon gezeigte Dateistruktur wiederholen.



C.1.2 Allgemeines zum Quellcode

In den folgenden Abschnitten sollen erwähnenswerte Teile vom Quellcode des Wireless Audio Speaker Players erläutert werden. Das Vorgehen wird bei allen Unterabschnitten dasselbe sein: Der Code wird von vorne her durchgegangen, wobei jeweils nur das wichtige zur Sprache kommt, in der Reihenfolge Strukturen, Variablen und anschliessend Funktionen. Ein entsprechendes Code-Segment ist jeweils nochmals angegeben um die Lesbarkeit zu verbessern.

Im Anhang sind als Übersicht alle hier vorgestellten Dateien aufgeführt, inklusive des Makefiles. Diverse Codestücke zur ALSA - Funktionalität wurden aus dem Musterbeispiel [13] abgeleitet.

C.2 was.h

Zu allererst möchten wir das zentrale Headerfile was.h betrachten.

Structs
Zu Beginn der Datei sind zwei *Enummerated Types* definiert, welche für die Auswertung der *Commandline* Parameter gebraucht werden:

```
typedef enum _was_snd_access {
    WAS_SND_ACCESS_MMAP_INTERLEAVED = 0,
    /** snd_pcm_readi/snd_pcm_writei access */
    WAS_SND_ACCESS_RW_INTERLEAVED,
    /** snd_pcm_readn/snd_pcm_writen access */
    WAS_SND_ACCESS_RW_NONINTERLEAVED,
    WAS_SND_ACCESS_LAST = WAS_SND_ACCESS_RW_NONINTERLEAVED
} was_snd_access_t;
```

Die hier definierten Zugriffstypen entsprechen dem Sinne derjenigen *Enummerated Types* aus der ALSA Userspace Library[1].

- WAS_SND_ACCESS_MMAP_INTERLEAVED: Entspricht einem *MMAP* Zugriff mit alternierenden[2] Kanälen.
- WAS_SND_ACCESS_RW_INTERLEAVED: Enstpricht einem Zugriff mit den IO Funktionen `readi` und `writei`, die Kanäle sind wiederrum alternierend.
- WAS_SND_ACCESS_RW_NONINTERLEAVED: Enstpricht einem Zugriff mit den IO Funktionen `readn` und `writen`, die Kanäle sind in diesem Fall nicht alternierend, dh. seriell aufeinanderfolgend.

```
typedef enum _was_generate_wave {
  WAS_GENERATE_SINE = 0,
  WAS_GENERATE_SINE_ABS,
  WAS_GENERATE_RECT,
  WAS_GENERATE_CONST_LOW,
  WAS_GENERATE_CONST_HIGH,
  WAS_GENERATE_STEP,
  WAS_GENERATE_NON,
  WAS_GENERATE_LAST = WAS_GENERATE_NON
} was_generate_wave_t;
```

Die oben definierten Typen entsprechen dem "Verhalten-Variablen", dass der WAS Player einnehmen soll, also welche Funktion generiert werden soll: Sinus, Absolutwert eines Sinus, Rechteck, konstant tief, konstant hoch, Schrittfunktion oder keine Generation eines Signals. Der letztgenannt Zustand wird für das Auslesen von .WAV-Dateien verwendet.

Variablen
Die folgenden sechs Variablen werden für die Initalisierung und für den Betrieb von ALSA benötigt.

```
snd_pcm_t *handle;
```

Ein Pointer, bzw. Handler, auf ein Objekt zur digitalen Audioausgabe, kurz PCM.

```
snd_pcm_hw_params_t *hwparams;
```

Ein ALSA Parameter-Objekt, welches die Hardware Parameter enthält und bei der Initialisierung der Soundkarte übergeben wird.

[1] Mehr dazu in der Webdokumentation http://www.alsa-project.org/alsa-doc/alsa-lib/pcm_8h-source.html oder [10]

[2] interleaved

```
snd_pcm_sw_params_t *swparams;
```

Entsprechend zum obigen HW-Parameter-Objekt das SW-Objekt.

```
snd_pcm_channel_area_t *areas;
```

Diesem komplexe Pointer verweist auf zwei Memorystellen mit der Gesammtgrösse von $areas = channel \cdot samples \cdot sample\_in\_bytes$, dh. ein Mehrkanal-Frame.

```
signed short *samples;
```

Entspricht der Anzahl Samples pro Frame, welches ausgespielt wird.

```
int chn;
```

Die Anzahl Kanäle.

C.3 was.c

Hier möchten wir kurz eine Einführung in das Main-Programm des Wireless Audio Speaker Player geben und die wichtigsten Programmstellen kurz anschneiden. Grundliegend gibt es im Main-Programm, d.h. im Player, die folgenden Phasen:

1. **Init:** Initialisierung der "Zustands-" und weiterer Variablen
2. **Parameter:** Auswertung der *Commandline*
3. **ALSA:** Setzen und Initialisierung der ALSA Umgebung
4. **Clock:** Auslesen des Clock-Objektes
5. **Netzwerk:** Aushandeln des Abspielpunktes
6. **Player:** Start des Players

Variablen
Generelle Variablen für den WAS Player und interne Variablen der Mainroutine:

```
// Error handling
int                 err;
// Help function called (yes equal 1)
int                 was_morehelp          = 0;

// "Boolean" MMAP Audio Output was called
int                 was_sound_mode_mmap   = 0;
// "Boolean" IOCTL Audio Output is called
int                 was_sound_mode_writei = 0;

// Commandline PCM Format Name
char*               was_pcm_format;

// Switch between multiple audio output methodes
was_snd_access_t    was_snd_access;
was_snd_access = WAS_SND_ACCESS_MMAP_INTERLEAVED;

// Output text of the audio output function
char                was_snd_access_output[50];

// Internal name of the generated function
```

```
was_generate_wave_t was_generate_function;
was_generate_function = WAS_GENERATE_SINE;

// Output text of the generated function
char                was_generate_output[50];

// play wave function called (yes equal 1)
int                 was_wav_file          = 0;
```

Variablen, welche für ein allfälliges Filehandling gebraucht werden:

```
FILE *in_file;       // Filehandle
char *wav_filename;  // Filename of the Wave File
```

Variablen für die Netwerkfunktionalität (mehr dazu auch in der Datei wasNetwork.h):

```
was_net_behaviour_t  was_net_behaviour  = WAS_NET_NEUTRAL;
char                *was_net_ip;   // IP   for Send/Recv
short int            was_net_port; // Port for Send/Recv
```

Variablen für das kreiierte Clock - Objekt und die Interaktion mit der Netzwerkfunktionalität des WAS Players. Mehr Informationen zum Clock-Objekt und die dort definierten Structs sind im Kapitel 2 zu entnehmen.

```
char              *was_clock_playtime_delta_string;  // Delta Parameter Text
long long          was_clock_playtime_delta = 0;     // Delta Parameter Value
unsigned long long was_clock_playtime = 0;           // Determined Playtime
unsigned long long was_clock_playtime_transport = TRANSFER_TIME;

struct clkval      clock_values;                     // Complex Clock Object used for
                                                     // communication (clock <-> player)
unsigned long long our_clock_nsec  = 0;              // Clock Object -> nanoseconds
unsigned long long our_clock_ticks = 0;              // Clock Object -> cpu ticks
```

Die Konstante `TRANSFER_TIME` soll dabei den Delay der Paketübertragung des WAS Players kompensieren und ist in wasNetwork.h definiert.

Variablen für die ALSA Funktionalität (auch kurz schon im Headerfile was.h auf Seite 63 und in wasAlsa.h auf Seite 70 erläutert.).

```
snd_pcm_t *handle;                  // PCM handle
snd_pcm_hw_params_t *hwparams;      // HW Parameters Soundcard
snd_pcm_sw_params_t *swparams;      // SW Parameters Soundcard

/* access format switch between MMAP and IOCTL */
/*    for choosing the right ALSA Function     */
snd_pcm_access_t pcm_access = SND_PCM_ACCESS_MMAP_INTERLEAVED;

snd_pcm_channel_area_t *areas;      // ALSA Frame
signed short *samples;              // Samples
int chn;                            // Channels
```

Zum Debuggen gibts eine Möglichkeit einzelne Bits auf dem Parallelport zu setzen oder löschen:

```
#ifdef DEBUG_PARPORT_OUT
     int retval = ioperm(MYPARPORT,MYPARPORT_BYTES,MYMODE);
#endif
```

Die vorher schon kurz erwähnten "Verhaltensvariablen" werden von der Kommandozeilenauswertung gesetzt und weiter unten durch die `switch`es verarbeitet.

Main-Funktion
Die Kommandozeile wird mit dem *Package getopt* und der Funktion `getopt_long` ausgewertet und anschliessend beginnt die Zuweisung der Werte in "Zustands-" oder "Prozessierungs"-Variablen:

```
if ((c = getopt_long(argc, argv, "h:tl:ip:mw:F:R:C:sarckSf:d:",
                     long_option, NULL)) < 0){
   switch (c) {
      case 't':
         was_net_behaviour  = WAS_NET_TALKER;
         break;
      case 'i':
         was_net_ip   = strdup(optarg);
         break;
      case 'm':
         pcm_access = SND_PCM_ACCESS_MMAP_INTERLEAVED;
         was_snd_access = WAS_SND_ACCESS_MMAP_INTERLEAVED;
         +was_sound_mode_mmap;
         break;
      ...
   }
}
```

Nach der Auswertung der Kommandozeile und der Initialisierung der Variablen, beginnt im Main-Programm der konkrete und auf ALSA basierende Player:
Memory Allocation und Alsa Initialisierung:

```
snd_pcm_hw_params_alloca(&hwparams);
snd_pcm_sw_params_alloca(&swparams);

snd_output_stdio_attach(&output, stdout, 0);
snd_pcm_open(&handle, device, SND_PCM_STREAM_PLAYBACK, 0);

set_hwparams(handle, hwparams, pcm_access);
set_swparams(handle, swparams);

samples = malloc((period_size * channels * snd_pcm_format_width(pcm_format)) / 8);
areas = calloc(channels, sizeof(snd_pcm_channel_area_t));

for(chn = 0; chn < channels; chn++){
   areas[chn].addr  = samples;
   areas[chn].first = chn * 16;
   areas[chn].step  = channels * 16;
}
```

Abfragen der Zeit der Clock-Objekte; wie schon erwähnt wird beim Server der *TSC*, beim Client das erzeugte Uhrenobjekt abgefragt:

```
if( was_net_behaviour == WAS_NET_TALKER){

  getrawtime(0, &clock_values);
  ticks2nsec(&clock_values.tsc, &our_clock_nsec, clock_values.cpu_khz);

  fprintf(stdout,"Value of the Wireless Audio Speaker Clock in nSec: \%Lu \n",\
         our_clock_nsec);
}
if(was_net_behaviour == WAS_NET_LISTENER){

  getrawtime(0, &clock_values);
  readtime(0, &our_clock_ticks);
  ticks2nsec(&our_clock_ticks, &our_clock_nsec, clock_values.cpu_khz);

  fprintf(stdout,"Value of the Wireless Audio Speaker Clock in nSec: \%Lu \n",\
          our_clock_nsec);
}
```

Der Server bestimmt einen gemeinsamen Abspielzeitpunkt und teilt diesen per UDP-Datengramm dem Client mit:

```
if( was_net_behaviour == WAS_NET_TALKER) {

   // set local times at server
   packet_t serverPacket;

   serverPacket.server_clock_nsec = our_clock_nsec;
   serverPacket.client_clock_nsec = (unsigned long long) 0;
   serverPacket.point_of_playing  = was_clock_playtime_transport +
         (unsigned long long)(our_clock_nsec +  was_clock_playtime_delta);

   was_clock_playtime = was_clock_playtime_transport + our_clock_nsec;

   fprintf(stdout,"The server WAS Clock is: \%Ld \n", our_clock_nsec);
   fprintf(stdout,"Wished Playtime in \%Ld nSec! \n",was_clock_playtime);

   // send packet to the client
   Talker(was_net_ip, &was_net_port, &serverPacket);

   printf("sendToHost:   \%s \n", was_net_ip);
   printf("serverPacket: \%Ld \%Ld \%Ld \n", serverPacket.server_clock_nsec, \
           serverPacket.client_clock_nsec, serverPacket.point_of_playing);
}
else if( was_net_behaviour == WAS_NET_LISTENER )
{

   //set local times at client
   packet_t clientPacket;

   clientPacket.server_clock_nsec = (unsigned long long)0;
   clientPacket.client_clock_nsec = our_clock_nsec;
   clientPacket.point_of_playing  = (unsigned long long)0;

   fprintf(stdout,"The client WAS Clock is: \%Ld \n", our_clock_nsec);
```

```
   // wait till packet from server arrives
   Listener(was_net_ip, &was_net_port, &clientPacket);

   clientPacket.client_clock_nsec = our_clock_nsec;
   was_clock_playtime = clientPacket.point_of_playing;

   printf("receivedFromHost: \%s \n", was_net_ip);
   printf("clientPacket: \%Ld \%Ld \%Ld \n", clientPacket.server_clock_nsec, \
           clientPacket.client_clock_nsec, clientPacket.point_of_playing);
}
```

Es wird die aktuelle Uhrzeit *gepollt* bis der Zeitpunkt erreicht ist, der als gemeinsamer Startzeitpunkt proklamiert wurde:

```
if( was_net_behaviour == WAS_NET_TALKER){
     do {
        getrawtime(0, &clock_values);
        ticks2nsec(&clock_values.tsc, &our_clock_nsec, clock_values.cpu_khz);
    } while(our_clock_nsec < was_clock_playtime);
}
if(was_net_behaviour == WAS_NET_LISTENER){

    do {
        getrawtime(0, &clock_values);
        readtime(0, &our_clock_ticks);
        ticks2nsec(&our_clock_ticks, &our_clock_nsec, clock_values.cpu_khz);
    } while(our_clock_nsec < was_clock_playtime);
}

if(was_snd_access == WAS_SND_ACCESS_MMAP_INTERLEAVED)
{
   switch ((int)was_generate_function) {
       case WAS_GENERATE_SINE:
            err = direct_loop(handle, samples, areas, generate_sine);
            break;
       ...
       case WAS_GENERATE_NON:
            if(was_wav_file){
               in_file = fopen(wav_filename,"rb"); /* Open the input file */
               if (in_file == NULL) {            /* Test for error */
                   fprintf(stderr, "Error: Unable to input file \%s !\n", \
                           wav_filename);
                   exit(8);
               }

               err = direct_loop_wav(handle, samples, areas, in_file);

               fclose(in_file);
               break;
            }

    }
...
}
```

Anmerkung zum letzten `switch`: Der Aufruf der zu generierenden Waves geschieht mit den Funktion direct_loop, bzw. write_loop. Dieser Funktion wird als letzter Parameter ein Prozedure-Name mit der "generierenden" Funktion übergeben[3].

C.4 wasAlsa.h

Nun möchten wir das Headerfile wasAlsa.h betrachten.
FIXME: PCM ... Digital Audio Interface

Includes
Notwendig für das Verwenden von der ALSA Library:

```
#define ALSA_PCM_NEW_HW_PARAMS_API
#define ALSA_PCM_NEW_SW_PARAMS_API
#include <alsa/asoundlib.h>
```

Wie dem deklarativen Text der `defines` schon zu entnehmen ist, wurde die API zur Einstellung der ALSA Hardware- und Softwareparameter mehrmals geändert und in der aktuellen Version der ALSA-Lib[4] coexistieren zwei unterschiedliche APIs. Durch das Setzen der beiden obigen `defines` kann auf mehr und klarer struktierierte Funktionen zurückgegriffen werden[5]. Die unterste Zeile wird für das Includieren der ALSA Headers verwendet.
Zum *Messen* und *Debuggen* wurde in WAS Code eingefügt, mit welchem Bits auf dem Parallelport gesetzt und gelöscht werden können:

```
#include <sys/io.h>
#define MYPARPORT 0x378
#define MYPARPORT_BYTES 3
#define MYMODE 0x777
```

Setzt Konstanten für das Freischalten des Parallelports mit `ioperm` und bei Gelingen kann man die einzelnen Bits eines Bytes mit dem Befehl `outb` ansprechen.

Variablen
Deklaratione von ALSA-spezifische Variablen:

```
char *device;
```

Enthält den "Namen" des Ausgabegeräts in ALSA-Terminologie; in der Regel wird dieser Name durch das Plug-And-Play System ermittelt (plughw:0,0"). Falls mehrere Soundkarten im System eingebaut wurden, kann hier explizit das Device bestimmt werden,

```
snd_pcm_access_t pcm_access;
```

Definiert das Zugriffsformat mit welchem ALSA das PCM ansprechen soll. Möglichkeiten dazu sind *read* und *write* Functionen welche mit IO Calls arbeiten, und welche es in einer *Interleaved* und einer *Non-Interleaved* Version gibt. Neben den IO-Calls sind auch Funktionen vorgesehen worden, welche

[3] Im Prinzip wird hier der Funktion direct_loop aus wasAlsa.c eine Funktion aus wasProcSound.* als Parameter weitergereicht.
[4] ALSA Userspace Library [10]
[5] Mehr dazu im Dokument der ALSA-Lib: http://www.alsa-project.org/alsa-doc/alsa-lib/pcm_8h-source.html

mit Ringbuffern und MMAP[6] arbeiten; hier gibt es neben der *Interleaved* agierenden Version, auch eine *Non-Interleaved* sowie noch einen komplexeren Typ[7].

```
snd_pcm_format_t pcm_format;
```

Digitaler Sound (PCM) kann abhängig von der Hostarchitektur und den gewählten Parameter unterschiedlich codiert werden. ALSA kann mit ziemlich vielen Formaten umgehen, dabei muss aber diese explizit gesetzt werden. In einem grössten Teil der Fälle ist das Format 16 Bit Little Endian, die weiteren Formate können der ALSA Dokumentation[8] entnommen werden.

```
unsigned int rate;
```

Samplerate mit der ein Soundstream erzeugt werden soll, bzw. aufgenommen wurde.

```
unsigned int channels;
```

Anzahl Kanäle für die Audioausgabe

```
unsigned int buffer_time;
```

Grösse des zirkulären Buffers in μs bei der Verwendung von MMAP ALSA Funktionen.

```
unsigned int period_time;
```

FIXME: Grösse der Periode?

```
double freq;
```

Frequenz der Sinusfunktion.

```
snd_pcm_sframes_t buffer_size;
```

Buffergrösse, welche mit dem vorgegebenen Hardware- und Software Setup von ALSA alloziert und verwendet kann. Zuerst wird dieser Wert in μs ermittelt und dann auf Samples umgerechnet.

```
snd_pcm_sframes_t period_size;
```

Periodengrösse, welche mit dem vorgegebenen Hardware- und Software Setup von ALSA alloziert und verwendet kann. Zuerst wird dieser Wert in μs ermittelt und dann auf Samples umgerechnet.

```
snd_output_t *output;
```

Handler zur Abstraktion des Output-Devices.

Für das Messen und Testen ist es eigentlich sinnvoll nur jedes zehnte SoundFrame auszuspielen, bzw. abzuspielen. Als Zähler dafür wird ein Counter benötigt:

```
int stepnummer;
```

[6] Memory Mapped

[7] Einen guten Überblick der verschiedenen Zugriffsarten und deren Funktionieren bietet die ALSA Dokumentation unter http://www.alsa-project.org/alsa-doc/alsa-lib/pcm.html#alsa_transfers

[8] http://www.alsa-project.org/alsa-doc/alsa-lib/group__p_c_m.html#a70

Funktionen
Um die Deklarationen lesbarer zu machen, wurden die Schlusselwörter, welche vorangestellt waren, weggelassen.

```
int set_hwparams(snd_pcm_t *handle,
                 snd_pcm_hw_params_t *params,
                 snd_pcm_access_t was_access);
```

Setzt die gewünschten Hardware Parameter der Soundkarte:

- `snd_pcm_t *handle`: Hardware Device
- `snd_pcm_hw_params_t *params`: Gewünschte Parameter
- `snd_pcm_access_t was_access`: Zugriffsart auf die Soundkarte

```
int set_swparams(snd_pcm_t *handle,
                 snd_pcm_sw_params_t *swparams);
```

Setzt die gewünschten Software Parameter der Soundkarte:

- `snd_pcm_t *handle`: Hardware Device
- `snd_pcm_sw_params_t *swparams`: Gewünschte Parameter

```
int xrun_recovery(snd_pcm_t *handle, int err);
```

Erkennt einen *Underrun* oder *Suspend* des zirkulären Buffers und versucht diese aufzulösen.

Der folgende Ausschnitt entspricht einer abstrakten Funktion, welcher der Audioausgabe übergeben wird:

```
void samplefunction(const snd_pcm_channel_area_t *areas,
                    snd_pcm_uframes_t offset,
                    int count,
                    double *_phase
);
```

Abspielen MMAP:

```
int direct_loop(snd_pcm_t *handle,
                signed short *samples,
                snd_pcm_channel_area_t *areas,
                void samplefunction(const snd_pcm_channel_area_t *areas,
                                    snd_pcm_uframes_t offset,
                                    int count,
                                    double *_phase)
);
```

Abspielen MMAP File:

```
int direct_loop_wav(snd_pcm_t *handle,
                    signed short *samples,
                    snd_pcm_channel_area_t *areas,
                    FILE * in_file
);
```

Abspielen IO Read/Write:

```
int write_loop(snd_pcm_t *handle,
               signed short *samples,
               snd_pcm_channel_area_t *areas,
               void samplefunction(const snd_pcm_channel_area_t *areas,
                                   snd_pcm_uframes_t offset,
                                   int count,
                                   double *_phase)
 );
```

Abspielen IO Read/Write File:

```
int write_loop_wav(snd_pcm_t *handle,
                   signed short *samples,
                   snd_pcm_channel_area_t *areas,
                   FILE * in_file
 );
```

Debug und Messfunktionen:

```
void debug_start_peak(void);
```

Erzeugt einen Peak

```
void debug_stop_peak(void);
```

Nimmt den Peak wieder zurück

C.5 wasAlsa.c

Variablen
Default Initialisierung der wichtigsten Variablen, welche für ALSA gebraucht werden; die vom User eingegebenen Werte überschreiben diese Werte je nach Gebrauch und Bedarf.

```
char *device = "plughw:0,0";                                    /* playback device */
snd_pcm_access_t pcm_access = SND_PCM_ACCESS_MMAP_INTERLEAVED; /* access format */
snd_pcm_format_t pcm_format = SND_PCM_FORMAT_S16;              /* sample format */
unsigned int rate = 44100;                                     /* stream rate */
unsigned int channels = 2;                     /* count of channels */
unsigned int buffer_time = 300000;             /* ring buffer length in us */
unsigned int period_time = 100000;             /* period time in us */
double freq = 440;                             /* sinusoidal wave frequency in Hz */

snd_pcm_sframes_t buffer_size;
snd_pcm_sframes_t period_size;
snd_output_t *output = NULL;
```

Funktionen
Zu allererst möchten wir die Hardware Initialisierungsfunktionen vorstellen:

```
int set_hwparams(snd_pcm_t *handle,
                 snd_pcm_hw_params_t *params,
                 snd_pcm_access_t pcm_access)
{
    /* choose all parameters */
```

```
    err = snd_pcm_hw_params_any(handle, params);

    /* set the interleaved read/write format */
    err = snd_pcm_hw_params_set_access(handle, params, pcm_access);

    /* set the sample format */
    err = snd_pcm_hw_params_set_format(handle, params, pcm_format);

    /* set the count of channels */
    err = snd_pcm_hw_params_set_channels(handle, params, channels);

    /* set the stream rate */
    rrate = rate;
    err = snd_pcm_hw_params_set_rate_near(handle, params, &rrate, 0);

    /* set the buffer time */
    err = snd_pcm_hw_params_set_buffer_time_near(handle, params, &buffer_time, &dir);
    err = snd_pcm_hw_params_get_buffer_size(params, &buffer_size);

    /* set the period time */
    err = snd_pcm_hw_params_set_period_time_near(handle, params, &period_time, &dir);
    err = snd_pcm_hw_params_get_period_size(params, &period_size, &dir);

    /* write the parameters to device */
    err = snd_pcm_hw_params(handle, params);
}
```

Als nächstes die Initalisierungsfunktionen der Software Parameter:

```
int set_swparams(snd_pcm_t *handle,
                 snd_pcm_sw_params_t *swparams)
{
    /* get the current swparams */
    err = snd_pcm_sw_params_current(handle, swparams);

    /* start the transfer when the buffer is full */
    err = snd_pcm_sw_params_set_start_threshold(handle, swparams, buffer_size);

    /* allow the transfer when at least period_size samples can be processed */
    err = snd_pcm_sw_params_set_avail_min(handle, swparams, period_size);

    /* align all transfers to 1 sample */
    err = snd_pcm_sw_params_set_xfer_align(handle, swparams, 1);

    /* write the parameters to the playback device */
    err = snd_pcm_sw_params(handle, swparams);
}
```

Wie im Headerfile schon kurz angetönt entdeckt die folgende Funktion die Zustände *underrun* und *suspend* und versucht diese aufzulösen.

```
int xrun_recovery(snd_pcm_t *handle, int err)
```

Die nun folgenden Funktionen werden gebraucht um ein erzeugtes Audio-Frame abzuspielen, dabei wird zwischen den MMAP Abspielfunktionen und den Abspielfunktionen basierend auf IO-Calls

unterschieden. Die leichten Abweichung beim Abspielen eines .WAV-Files werden hier nicht explizit betrachtet; im Wesentlichen unterscheiden sich diese Abspielarten nur in den einfliessenden Parametern, das Vorgehen ist aber dasselbe.

MMAP Abspielfunktion: Im folgenden Extrakt soll das generelle Vorgehen bei der Verwendung dieser Methode erläutert werden:

```
int direct_loop(snd_pcm_t *handle,
                signed short *samples,
                snd_pcm_channel_area_t *areas,
                void samplefunction(const snd_pcm_channel_area_t *areas,
                                    snd_pcm_uframes_t offset,
                                    int count,
                                    double *_phase)
    )
{
    while (1) {
```

Überprüfe den Füllstand des Buffers

```
        avail = snd_pcm_avail_update(handle);
```

Alloziere im Buffer einen freien Speicherplatz der Grösse `frames` und gib diesen Speicherplatz an den Pointer `my_areas`.

```
        err = snd_pcm_mmap_begin(handle, &my_areas, &offset, &frames);
```

Fülle den freien Speicherplatz mit der *Samplefunktion*[9].

```
        samplefunction(my_areas, offset, frames, &phase);
```

Spiel das Sample ab, d.h. aktiviere das Verschieben der Pointer[10].

```
        commitres = snd_pcm_mmap_commit(handle, offset, frames);
    }
}

int direct_loop_wav(snd_pcm_t *handle,
                    signed short *samples,
                    snd_pcm_channel_area_t *areas,
                    FILE * in_file
    );
```

IO-Calls Abspielfunktion: Im folgenden Extrakt soll das generelle Vorgehen bei der Verwendung dieser Methode erläutert werden:

```
int write_loop(snd_pcm_t *handle,
               signed short *samples,
               snd_pcm_channel_area_t *areas,
               void samplefunction(const snd_pcm_channel_area_t *areas,
                                   snd_pcm_uframes_t offset,
```

[9] Genau an dieser Stelle wird der entsprechende Funktionsname eingesetzt. D.h. *samplefunction* ist nur ein Platzhalter für eine der Funktionen aus wasAlsaProc*.

[10] Ein Kopieren der eben erzeugten Samples wird nicht ausgeführt - was bei MMAP ja auch gewünscht ist.

```
                                    int count,
                                    double *_phase)
    )
{
    while (1) {
```

Erzeuge Samples

```
        samplefunction(areas, 0, period_size, &phase);
```

Verschiebe die Pointers nach und nach und spiele mit der Funktion `snd_pcm_writei()` die Samples aus.

```
        ptr = samples;
        cptr = period_size;
        while (cptr > 0) {

            err = snd_pcm_writei(handle, ptr, cptr);

            ptr += err * channels;
            cptr -= err;
        }
    }
}

int write_loop_wav(snd_pcm_t *handle,
                   signed short *samples,
                   snd_pcm_channel_area_t *areas,
                   FILE * in_file
    )
```

C.6 wasNetwork.h

Variablen
Nun möchten wir das Headerfile wasAlsaNetwork.h betrachten, welches Variablen und Funktionen für die Netzwerkfähigkeit des WAS Players enthält.

Includes
Die folgenden Includes stellen die Netzwerkfunktionalität zur Verfügug:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
```

Defines
Die Übertragung des Paketes mit der gemeinsamen Abspielzeit benötigt eine gewisse Zeit; da diese "unbekannt"[11] ist, wird ein Wert von 10ms als eine realistische obere Schranke gesetzt.

```
#define TRANSFER_TIME 10000000  /* 10 ms shold be enough for */
                                /*the transfer Server -> Client */
```

[11] Eine Möglichkeit wäre es die Zeit per *RTT - Round-Trip-Time* zu bestimmen.

Structs
Nachfolgend das UDP-Paket welches die aktuelle Uhrzeit des Servers, die aktuelle Zeit des Clients und schlussendlich den gemeinsamen Abspielzeitpunkt enthält.

```
typedef struct {
  unsigned long long server_clock_nsec;
  unsigned long long client_clock_nsec;
  unsigned long long point_of_playing;
} packet_t ;
```

Enumerated Types
Verhalten der Netwerkfunktionalität; Server, Client oder Standalone

```
typedef enum _was_net_behaviour {
        WAS_NET_TALKER = 0,
        WAS_NET_LISTENER,
        WAS_NET_NEUTRAL,
        WAS_NET_LAST = WAS_NET_NEUTRAL
} was_net_behaviour_t;
```

Funktionen
Funktionen zum Senden oder Empfangen eines oben im Struct definierten Paketes via UDP/IP an einen beliebig anderen Host. Dabei wurde die Funktion so ausgelegt, dass ein beliebiger Struct gesendet werden kann.

```
int Talker(char *sendToHostIP,
           short int *sendToHostPort,
           packet_t *serverPacket);
int Listener(char *receivedFromHostIP,
             short int *receivedFromHostPort,
             packet_t *clientPacket);
```

C.7 wasNetworkTalker.c

Der *Talker* ist ein simple Umsetzung des klassischen Ansatzes *socket - bind - send*. Da die Implementierung leicht verständlich ist wird an dieser Stelle auf den Sourcecode (Seite **??**) verwiesen. Hier verwendete Codestücke stammen zum Teil aus [11] und aus [14].

C.8 wasNetworkListener.c

Der *Listener* ist ein simple Umstetzung des klassischen Ansatzes *socket - bind - receive*. Dabei agiert der *Listener* im blockierenden Modus, dh. das Programm wird erst dann fortgesetzt, wenn ein entsprechendes *datagram* empfangen wurde. Da die Implementierung leicht verständlich ist wird an dieser Stelle auf den Anhang (Seite **??**) verwiesen.

C.9 wasSoundProc.h

Im Headerfile wasSoundProc.h werden die unterschiedlichen “Wave”-Generierungsarten zusammengefasst.

Includes
Die mathematischen Bibliotheken werden für die Erzeugung der Sinus-Funktion und der daraus abgeleiteten Formen verwendet.

```
#include <math.h>
```

Variablen
Wird für die Funktion `generate_step` verwendet: `STEPATPROCENT` gibt das Sample in Prozent an, bei der die Schrittfunktion von *low* auf *high* springt.

```
extern int STEPATPROCENT;
```

Funktionen
Die nachfolgenden Funktionen besitzen allesammt die selber Parmameterliste, so dass die Funktionen gegeneinander ausgetauscht[12] werden können. Die Parameter setzen sich wiefolgt zusammen:

- *const snd_pcm_channel_area_t *areas*: Ein Pointer auf ein Mehrkanal - Frame, welches auf einen freien Speicherplatz der Grösse $Kanle \cdot Samples \cdot Sample_in_Bytes$ verweist.
- *snd_pcm_uframes_t offset*: Offset, falls im obengenannten Frame nicht bei 0 angefangen werden soll zu schreiben.
- *int count*: Anzahl Samples für einen Kanal.
- *double *_phase*: Phasenwinkel als Ein- und Rückgabewert, so dass eine Funktion ohne Sprünge im nächsten Frame fortgesetzt werden kann.

```
void generate_sine(const snd_pcm_channel_area_t *areas,
                   snd_pcm_uframes_t offset,
                   int count, double *_phase
);
void generate_sine_abs(const snd_pcm_channel_area_t *areas,
                       snd_pcm_uframes_t offset,
                       int count, double *_phase
);
void generate_rect(const snd_pcm_channel_area_t *areas,
                   snd_pcm_uframes_t offset,
                   int count, double *_phase
);
void generate_step(const snd_pcm_channel_area_t *areas,
                   snd_pcm_uframes_t offset,
                   int count, double *_phase
);
void generate_const_high(const snd_pcm_channel_area_t *areas,
                         snd_pcm_uframes_t offset,
                         int count, double *_phase
);
void generate_const_low(const snd_pcm_channel_area_t *areas,
                        snd_pcm_uframes_t offset,
                        int count, double *_phase
);
```

[12] Genau das wird bei dem Aufruf im Main-Programm gemacht, vgl. entsprechender Kommentar dort.

```
void read_wav(const snd_pcm_channel_area_t *areas,
              snd_pcm_uframes_t offset, int count,
              double *_phase, FILE *in_file
);
```

C.10 wasSoundProcSine.c

Exemplarisch möchten wir hier den Füllvorgang des Mehrkanal-Frames mit den Sinuswerten beschreiben, die weiteren nachfolgenden Funktionen sind nach demselben Muster erzeugt.

Funktion

```
void generate_sine(const snd_pcm_channel_area_t *areas,
                   snd_pcm_uframes_t offset,
                   int count,
                   double *_phase)
{
    double phase = *_phase;
```

Ein Phasewinkel kann als Parameter mitgegeben werden. Somit kann der Sinus auch mit einer anderen Phase beginnen.

```
    double max_phase = 1.0 / freq;
```

Die Phase kann in einem Schritt nicht mehr als obiger Wert “drehen”.

```
    double step = 1.0 / (double)rate;
```

Der Phasen - Schritt (konkrete Verschiebung der Phase).

```
    double res;
```

Das Result der Sinuswertberechnung.

```
    signed short *samples[channels];
```

Der konkrete Sample-Wert.

```
    int steps[channels];
```

Berücksichtigung mehrerer Kanäle.

```
    int chn, ires;
```

Zwei temporäre Variablen.

```
    /* verify and prepare the contents of areas */
    for (chn = 0; chn < channels; chn++) {
        if ((areas[chn].first \% 8) != 0) {
            printf("areas[\%i].first == \%i, aborting...\n", chn, areas[chn].first);
            exit(EXIT_FAILURE);
        }
        samples[chn] = (signed short *)(((unsigned char *)areas[chn].addr) + \
                                                    (areas[chn].first / 8));
        if ((areas[chn].step \% 16) != 0) {
```

```
            printf("areas[\%i].step == \%i, aborting...\n", chn, areas[chn].step);
            exit(EXIT_FAILURE);
        }
        steps[chn] = areas[chn].step / 16;
        samples[chn] += offset * steps[chn];
    }
```

Ab hier werden die Werte berechnet und das Frame ausgefüllt, der *stepnummer* - Zähler wird wie schon erwähnt für die Begrenzung der hör- und messbaren Werte verwendet:

```
    /* fill the channel areas */
    while (count-- > 0) {
       if((stepnummer % 10)  == 0){
            res = sin((phase * 2 * M_PI) / max_phase - M_PI) * 32767;
            ires = res;
       }
       else{
            ires = 0;
       }
       for (chn = 0; chn < channels; chn++) {
           *samples[chn] = ires;
           samples[chn] += steps[chn];
       }
       phase += step;
       if (phase >= max_phase)
           phase -= max_phase;
   }
   stepnummer++;
   *_phase = phase;
}
```

C.11 wasSoundProcAbs.c

Diese Funktion ist analog zur Sinus-Funktion aufgebaut, mit der folgenden Ausnahme:
Funktion

```
...
if((stepnummer % 10)  == 0){
    res = fabs(sin((phase * 2 * M_PI) / max_phase - M_PI)) * 32767;
    ires = res;
}
...
```

C.12 wasSoundProcRect.c

Diese Funktion ist analog zur Sinus-Funktion aufgebaut, mit der folgenden Ausnahme:
Funktion

```
...
if((stepnummer % 10)  == 0){
   res = sin((phase * 2 * M_PI) / max_phase - M_PI) * 32767;
   if(res < 0) {
```

```
        ires = -32767;
    }
    else {
        ires = 32767;
    }
}
...
```

C.13 wasSoundProcStep.c

Diese Funktion ist analog zur Sinus-Funktion aufgebaut, mit der folgenden Ausnahme:
Funktion

```
...
const int orgSamples = count;

/* STEP looks like

<-- PERIODE -->
           ---
----------|   |
          ^
          |
       STEPAT (Defines the place of the switching sample
              (in procent of samples(alias count))
*/

int stepat = (orgSamples * (STEPATPROCENT/100));

...
   if((stepnummer % 10)  == 0){
      if(count > stepat) {
        ires = 0;
      }
      else{
        ires = 32767;
      }
   }
...
```

C.14 wasSoundProcConst.c

Diese Funktion ist analog zur Sinus-Funktion aufgebaut, mit der folgenden Ausnahme:
Funktion
Const High:

```
...
ires = 32767;
...
```

Const Low:

```
...
ires = 0;
...
```

C.15 wasSoundProcRead.c

Diese Einlesefunktion ist wiederrum analog zur Sinus-Funktion aufgebaut, wurde jedoch um ein Filehandling erweitert. Somit muss der gewünschte Filehandler mitgegeben und in der Funktion ausgewertet werden.
Funktion

```
void read_wav(const snd_pcm_channel_area_t *areas,
              snd_pcm_uframes_t offset,
              int count,
              double *_phase,
              FILE *in_file)
{
...
int        ReadSize;
short int  ReadData;
...
   for (chn = 0; chn < channels; chn++) {

       ReadSize = fread((char *)&ReadData, 1, sizeof(ReadData),in_file);
       if (ReadSize != sizeof(ReadData)){
          fprintf(stderr,"Unable to read ReadData! \n");
          exit(8);
       }

       *samples[chn] = ReadData;
       samples[chn] += steps[chn];
  }
...
```

C.16 clkobj.h

Die Funktionen dieses ASM-Files wurden schon im Abschnitt 2 erklärt.

C.17 wrapperfunctions.s

Die Funktionen dieses Header-Files wurden schon im Abschnitt 2 erklärt.

C.18 Driftkompensation

Zu den beiden vorgeschlagenen Driftkompensationen (S. 19) möchten wir noch die entsprechenden Codestellen angeben:

1. Eine solche *Resynchronisation* müsste jeweils vor dem Ausspielen der digitalen Samples erfolgen. Ein möglicher Ansatz wäre die Synchronisation des Abspiel-Aufrufes `snd_pcm_mmap_commit(handle, offset, frames)` für die MMAP Abspielen, bzw. `snd_pcm_writei(handle, ptr, cptr)` für

die IO-Write Variante. Beide Abspielfunktionen befinden sich in der Datei wasAlsa.c (Kap. C.5, S. 73).

2. Die angesprochene *Dezimation* und *Interpolation* wird am Besten auf den erzeugten digitalen Sounddaten der `generate_...` Funktionen in den wasSoundProc*.c-Dateien angewendet. Eventuell genügt schon ein duplizieren oder weglassen des letzten Samples eines solchen Frames um das gewünschte Verhalten zu erreichen.

Anhang D

Checklisten

D.1 Uhrensynchronisation

Die Abschnitte D.1.1 und D.1.2 sollen einem Benutzer des Uhren- bzw. Parallelport-Programms helfen, die notwendigen Schritte zur erfolgreichen Bedienung dieser beiden Teile zeigen. Diese Schritt-für-Schritt-Anleitungen beziehen sich auf unsere, unter *Linux*, arbeitenden Rechner.

D.1.1 Checkliste: Synchronisation

- ins Verzeichnis wechseln, wo sich die zu kompilierenden Dateien befinden
- `make clean` ausführen
- `make` ausführen
- *Clockobject*-Modul laden:
  auf beiden Rechnern `insmod clkobj.o` ausführen (als *root*)
- auf dem Client *sync_client* starten:
  `sync_client [ [IP-Adresse des Servers] [Portnummer (9879)]`

  `[Name des Logfiles] ]`
- auf dem Server *sync_server* starten:
  `sync_server [ [IP-Adresse des Clients] [Portnummer (9879)]`

  `[Nachrichtenintervall (Einheit 10ms)]`

  `[Gesamte Anzahl Nachrichten] ]`

Das clkobj-Modul kann mit `rmmod clkobj` entladen werden, mit `lsmod` sieht man welche Module geladen sind.

D.1.2 Checkliste: Parallelport

- spezielles Parallelport-Kabel anschliessen! (Falls nicht angeschlossen, stürzt der Rechner ab. Vergleiche dazu die Funktion `tscpp_read()` in *tscpp.c*)
- ins Verzeichnis wechseln, wo sich die zu kompilierenden Dateien befinden
- Falls nicht schon vorher ausgeführt: `make clean` und `make` ausführen
- die Module *lp*, *parport_pc*, *parport* entladen (als *root*):
  `rmmod lp`
  `rmmod parport_pc`
  `rmmod parport`
- TSC-Modul laden:
  `./load_tscpp` ausführen (als *root*)
- auf dem Client *parallelport_client* starten:
  `parallelport_client [ [Device (/dev/tsc_latch)]`

  `[IP-Adresse des Servers]`

  `[Portnummer (nicht 9879!)]`

  `[Name des Logfiles] ]`
- auf dem Server *parallelport_server* starten:
  `parallelport_server [ [Device (/dev/tsc_trigger)]`

  `[IP-Adresse des Clients]`

  `[Portnummer (nicht 9879!)]`

  `[Messintervall (Einheit 10ms)]`

  `[Gesamte Anzahl Nachrichten] ]`

Das tscpp-Modul kann mit `rmmod tscpp` entladen werden.

D.2 Player

Im Folgenden möchten wir Checklisten für das Compilieren und die Ausführung der erstellten Applikationen geben. Im Wesentlichen ist das ein relativ einfacher

Vorgang, aus Erfahrung geht in der Regel geht immer etwas vergessen. Die Versionsnummer hinter den zu installierenden Programme und Bibliotheken entsprechen der in dieser Semesterarbeit verwendeten Version.

D.2.1 Checkliste: Wireless Audio Speaker Commandline

Checkliste für die Compilierung und das Ausführen des Wireless Audio Speaker (was) Commandline Programmes.

D.2.1.1 Compilieren

Folgende Programme bzw. Bibliotheken müssen installiert sein:

- □ gcc (Version 3.2)
- □ alsa (Library - Version 0.9.2)
- □ alsa (Entwicklung - Version 0.9.2)
- □ nasm (Version 0.98)

Anmerkung: Alsa-Pakete sind nicht von SuSE selber zusammengestellt, aber für SuSE 8.1 kompiliert worden. Die Pakete befinden sich auf der beigelegten CD.
Compilieren mit:

- `CVS checkout WAS`
- `cd WAS`
- `gmake clean`
- `gmake was`

D.2.1.2 Ausführen

Der Wireless Audio Speaker per Commandline wird mit ./was und folgenden Optionen gestartet:

```
Usage: was [-h] [[-t|-l] [-i=IP] [-p=Port]]
           [-m|-w] [-F=Format] [-R=Samplerate] [-C=Channels]
           [-s|-a|-r|-c|-k|-S|-f [FILE]]

-h,--help       help

-t,--talker     acts as an server
-l,--listener   acts as a client

-i,--ip         use the following IP
-p,--port       use the following Port

-m,--mmap       use mmaped audio functions
-w,--writei     use ioctl audio functions

-F,--format     use the following ALSA PCM format (q.v. the documentation)
-R,--samplerate use the following samplerate
-C,--channels   use the following channels (1 = mono, 2 = stereo)

-s,--sinus      generates a sine audio signal
-a,--abssinus   generates an abs sine audio signal
-r,--rect       generates a rectangular audio signal [sine in the background]
-S,--step       generates a step audio signal
-c,--const      generates a constant high audio signal
-k,--konst      generates a constant low audio signal
-i,--impulse    generates an impuls audio signal

-f,--file       play the following .WAV file

-d,--delta      start the audio playout at the client nSecs later
```

D.2.2 Checkliste: WaveChecker

D.2.2.1 Download

- CVS checkout WaveChecker

- cd Wavechecker

- ./WaveChecker.pl TEST.WAV

D.2.2.2 Ausführen

Zum Ausführen des WaveChecker Scriptes müessen neben Perl die folgenden Perl Klassen vorhanden sein (im ursprünglichen CVS Checkouts lokal vorhanden):

- □ Audio::Wav (mit Audio::Wav::Read)

Gestartet wird das WaveChecker - Skript mit ./WaveChecker.pl TEST.WAV

D.2.3 Checkliste: Wireless Stereo Speaker Frontend

Checkliste für die Compilierung und das Ausführen des Wireless Stereo Speaker GUIs.

D.2.3.1 Compilieren

Folgende Programme bzw. Libraries müssen installiert sein:

- □ gcc (Version 3.2)
- □ g++ (Version 3.2)
- □ qt (Library - Version 3.0.5 [libqt, libqt-mt])
- □ qt (Entwicklung - Version 3.0.5 [uic, moc, qmake])

Compilieren mit:

- `CVS checkout Frontend`
- `cd Frontend`
- `qmake WirelessStereoSpeaker.pro` (! qmake nicht gmake !)
- `gmake`

D.2.3.2 Ausführen

Zum Ausführen des Wireless Stereo Speaker Frontends müssen folgende Programme installiert sein:

- □ was (wireless audio speaker - commandline)

Unter Tools gibt es eine Reihe von optionalen Konverter-, bzw. Toolsaufrufen, für welche folgende Programm forteilhaft installiert werden sollten.

- □ xmms (Version ...)
- □ perl (Version 5.8)
- □ file (Version 3.37)
- □ mpg123 (Version 0.59s-mh4)

Forteilhaft für die Entwicklung waren die Integrierten Entwicklungs Umgebungen

- □ qt-designer (Version 2 oder 3 - [für Entwicklung QT-GUIs])
- □ kdevelop (Version 2.1.3 - [KDE Entwicklungsumgebung])

Das Wireless Stereo Speaker Frontend wird mit ./WirelessStereoSpeaker gestartet.

D.2.4 Checkliste: Messungen Oszilloskop

- ☐ parport_was kompilieren, wenn noch nicht vorhanden:
  - ☐ `gmake clean`
  - ☐ `gmake parport`
- ☐ Kanal 1 des Oszilloskopes mit Soundkarte verbinden (Masse und wahlweise linker oder rechter Audio-Kanal)
- ☐ Kanal 2 des Oszilloskopes mit der Masse und dem LSB des ParallelPort-Debug-Kabels
- ☐ `su`[1]
- ☐ `./parport_was`

D.2.5 Checkliste: Messungen Logic Analyzer

- ☐ was kompilieren mit
  - ☐ `gmake clean`
  - ☐ `gmake parport`
- ☐ Kanal 1 des Oszilloskopes mit dem "Schnitt-Stellenübersetzers" des ParallelPort-Debug-Kabels des Computer 1 verbinden.
- ☐ Kanal 2 des Logic Analyzers mit den Soundkarten des Computers 1 und 2 verbinden (Masse, jeweils linker und rechter Audio-Kanal).
- ☐ Kanal 3 des Oszilloskopes mit dem"Schnitt-Stellenübersetzers" des ParallelPort-Debug-Kabels des Computer 1 verbinden.
- ☐ `su`[2]
- ☐ `./parport_was`

[1] IO-Permissions werden auf einfache Weise nur dem Root zur Verfügung gestellt.
[2] IO-Permissions werden auf einfache Weise nur dem Root zur Verfügung gestellt.

Anhang E

Sourcecodes

In dieser Version wurde der Anhang mit dem Sourcecode entfernt. Ein vollständige Version inklusive des Sourcecodes befindet sich in der im Umschlag eingelegten CD.

Synchronisationsalgorithmus
clkobj.c
clkobj.h
load_tscpp
Makefile
parallelport_analyse.m
parallelport_client.c
parallelport_server.c
sync_analyse.m
sync_client.c
sync_server.c
tscpp.c
tscpp.h
tsc_common.c
tsc_common.h
tsc_log.c
tsc_log.h
tsc_types.h
unpconf.h
unp.h
wrapperfunctions.s

Wireless Audio Speaker: Player
was.c
was.h
wasAlsa.c
wasAlsa.h

wasNetwork.h
wasNetworkListen.c
wasNetworkTalker.c
wasSoundProc.h
wasSoundProcAbs.c
wasSoundProcConst.c
wasSoundProcRead.c
wasSoundProcRect.c
wasSoundProcSine.c
wasSoundProcStep.c
Makefile

Wireless Stereo Speaker: Frontend
main.cpp
wss.cpp
wss.h

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Philipp Blum and Men Muheim. *On the Performance of Clock Synchronisation: Algorithms for a Distributed Commodity Audio System.* 2003.

[2] W. Richard Stevens. *UNIX Network Programming, Volume 1.* Prentice Hall PTR, 1998.

[3] Andreas Moser. *Linux Kernel für hochpräzise Uhrensynchronisation.* Studienarbeit, 2002.

[4] Eric Schreiber and Daniel Sigg. *Clock Synchronization for Wireless LAN.* Diplomarbeit, 2002.

[5] ALSA Developpers. *Advanced Linux Sound Architecture (ALSA).* WWW page, 2003.
`http://www.alsa-project.org`.

[6] OSS Developers und 4front Technologies. Open Sound System - Kommerziell. WWW page, 2003.
`http://www.4front-tech.com/`.

[7] OSS Developers und 4front Technologies. Open Sound System. WWW page, 2003.
`http://www.linux.org.uk/OSS/`.

[8] JACK Developer. *JACK.* WWW page, 2003.
`http://jackit.sourceforge.net/`.

[9] Dave Phillips. *Sound & MIDI Software For Linux.* WWW page, 2003.
`http://www.linuxsound.at/one-page.html`.

[10] ALSA Developers. *ALSA Libraries and Interfaces.* WWW page, 2003.
`http://www.alsa-project.org/alsa-doc/alsa-lib/`.

[11] Mark Mitchell, Jeffrey Oldham and Alex Samuel. *Advanced Linux Programming.* newriders, 1st edition 01 edition, 2001.
`http://www.advancedlinuxprogramming.com/download`.

[12] Trolltech. QT Toolkit. WWW page, 2003.
`http://www.trolltech.com/products/qt/`.

[13] ALSA Developers. *ALSA Libraries: PCM Example*. WWW page, 2003.
`http://www.alsa-project.org/alsa-doc/alsa-lib/_2test_2pcm_8c-example.html#example_test_pcm`.

[14] Brian "Beej" Hall. *Beej's Guide to Network Programming - Using Internet Sockets*, 2001.