Audioverarbeitung in eingebetteten Systemen

Janik Hasche HAW HAmburg
janik.hasche@haw-hamburg.de

13. September 2017

Zusammenfassung

Dieser Artikel ist an Einsteiger im Bereich der Audioverarbeitung über Software in eingebetteten System gerichtet und liefert eine theoretische Einführung sowie eine Anleitung zur Umsetzung von digitaler Audioverarbeitung in einer Embedded-Linux-Umgebung. Damit der Leser für die Probleme, die die digitale Audioverarbeitung insbesondere bei eingebetteten Systemen mit begrenzen Ressourcen mit sich bringt, sensibilisiert ist und alle Konfigurationsparameter sinnvoll für seinen Anwendungsfall nutzen kann, wird zuerst auf die grundsätzliche Funktionsweise von digitaler Audioverarbeitung eingegangen. Aufbauend darauf wird am Beispiel des Beaglebone Black und dem ALSA-Treiber die Einrichtung eines Audiointerfaces, die Konfiguration des Treibers, Wiedergabe und Aufnahme von Audio bis hin zur Implementierung einer Full-Duplex-Audioanwendung erklärt. Am Beispiel der Full-Duplex-Anwendung wird die Performance des Beaglebone-Black untersucht, damit der Leser die Möglichkeiten und Grenzen von Audioverarbeitung mit einem solchen System besser einschätzen kann. So schafft dieser Artikel eine Wissensgrundlage zur digitalen Audioverarbeitung, mit der weiterführende Arbeiten, zum Beispiel das Erkennen von bestimmten Signalen (Klatschen, Pfeifen), Implementieren von Audioeffekten(Echo, Phaser) und vieles mehr möglich gemacht werden.

I. Einleitung

In einer Zeit, in der eingebettete Systeme immer leistungsfähiger und präsenter werden, übernehmen sie auch immer mehr Aufgaben im Bereich der digitalen Audiosignalverarbeitung. Es braucht heute keine aufwändige analoge Technik oder große Computersysteme mehr, um Audio in Echtzeit zu verarbeiten. So können zum Beispiel Gitarristen auf kleine digitale Effektgeräte zurückgreifen, anstatt zu jedem Gig eine große Sammlung von analogen Effektpedalen mitzuschleppen. Doch auch außerhalb der Musikbranche gewinnt die Audioverarbeitung in eingebetteten System eine immer größere Bedeutung: Angefangen bei Lichtschaltern, welche sich per Klatschen aktivieren lassen bis hin zu intelligenten Produkten wie Amazon Alexa, die Befehle per Spracherkennung entgegennehmen. Doch wie funktioniert digitale Audioverarbeitung und wie setzt man sie auf eingebetteten Systemen um? Wie nutzt man die begrenzten Ressourcen am besten? Dieser Artikel beschäftigt sich mit diesen Fragen und liefert am Beispiel des Beaglebone Black einen Einstieg in das Thema.

II. Einführung in die Digitale Audioverarbeitung

In diesem Kapitel wird die grundsätzliche Funktionsweise von digitaler Audioverarbeitung erklärt. Um ein gutes Verständnis für die Konfigurationsparameter, die in einer Audioanwendung wichtig sind aufzubauen, wird hier von der Beschaffenheit eines analogen Audiosignals über die Verarbeitung mittels Sampling bis hin zur internen Verarbeitung mit Puffern alles beschrieben, was wichtig ist, um je nach Anwendungsfall die richtigen Parameter wählen zu können.

i. Das Analoge Audiosignal

Bevor Audiosignale Digital verarbeitet werden können, müssen sie erst in analoger Form vorhanden sein. Am Beispiel einer Kette von einem Mikrophon, einem Verstärker und einem Lautsprecher lässt sich vereinfacht darstellen, wie analoge Audiosignale funktionieren: Geräusche und Töne liegen als Luftdruckschwankungen(Schallwellen) mit bestimmten Frequenzen und Verläufen vor und können mit einem Mikrophon aufgefangen und in ein analoges, elektrisches Signal umgewandelt werden. Dieses elektrische Signal lässt sich als Spannung über Zeit betrachten(siehe Bild 1), die die Schallwellen, also Schwingungen darstellt. Wird dieses elektrische Signal verstärkt an einen Lautsprecher weitergegeben, wird dieser in Bewegungen versetzt, welche das Signal wieder als Luftdruckschwankungen, also für das menschliche Gehör wahrnehmbare Schallwellen abbildet.[1]

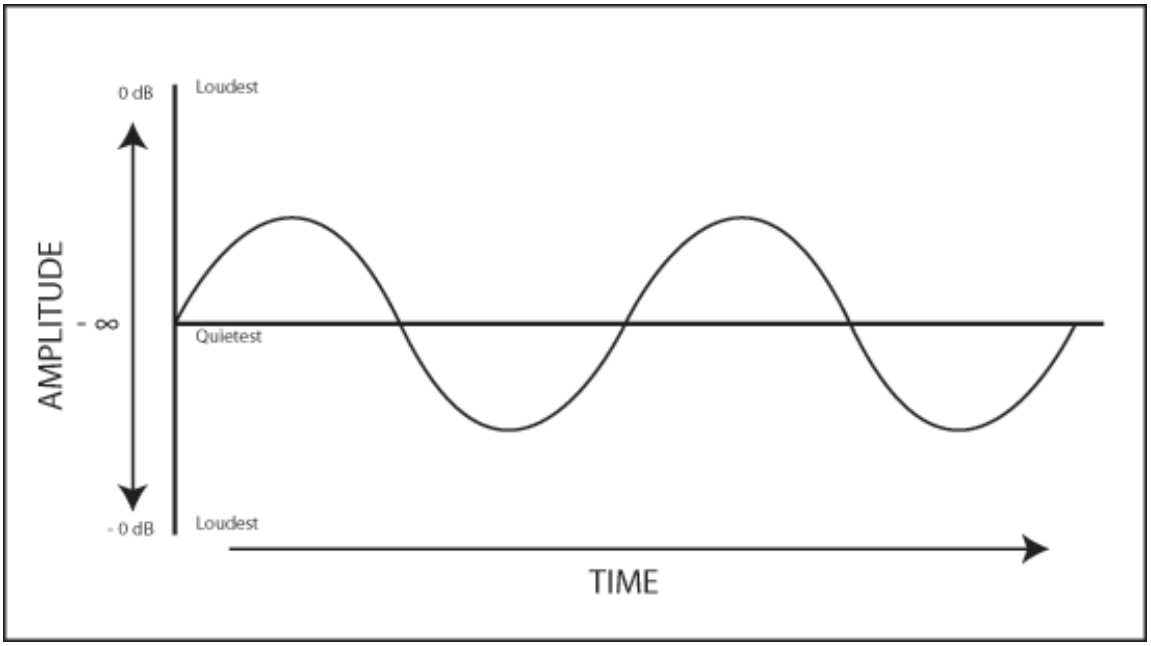

Abbildung 1: *Audiosignal*

Im folgenden werden die Parameter des elektrischen Audiosignals und ihre Beschränkungen beschrieben:

i.1 Amplitude

Die Amplitude des Signals bildet die Lautstärke bzw. den Schalldruck auf die Spannung ab. So wird die maximale Lautstärke durch die maximale Spannung, die das System verarbeiten kann, begrenzt.[1] So ist das Signal gestört, wenn es die maximale Lautstärke bzw. Spannung überschreitet. Dieser Effekt ist allgemein bekannt durch die analoge Verzerrung bei elektrischen Gitarren, im Normalfall ist dieser Effekt bei der Verarbeitung jedoch unerwünscht.

i.2 Frequenzbereich

Die Frequenz der Schwingungen stellt die Tonhöhe dar, je höher die Frequenz, desto höher wird der Ton wahrgenommen. Die maximale Frequenz ist durch mechanische Einschränkungen in der

Elektronik begrenzt, die wie ein Tiefpassfilter wirkt: Bis zu einer bestimmten Grenzfrequenz wird nahezu das vollständige Signal übertragen, bei höheren Frequenzen wird das Signal immer weiter gedämpft(siehe Bild 2). Da das Menschliche Gehör keine Frequenzen über 20 kHz wahrnehmen kann, ist dieser Effekt bei hohen Grenzfrequenzen zu vernachlässigen, bei Verwendung minderwertiger Hardware, zum Beispiel einem Mikrophon mit einer Grenzfrequenz von 8 kHz, wird das Signal jedoch unklar und "dumpf"wahrgenommen.[1]

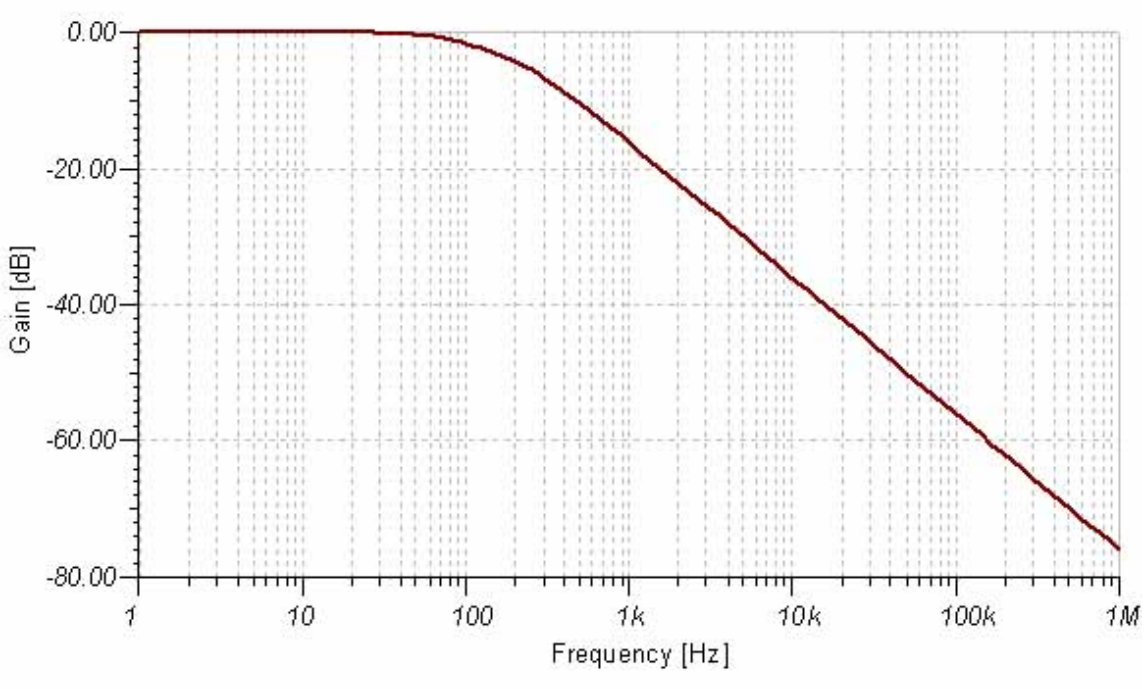

Abbildung 2: *Tiefpass-Filter*

i.3 Rauschen

Jedes analoge Audiosignal enthält ein Grundrauschen, welches ab einer gewissen Lautstärke störend wirkt, das Signal wird als unklar wahrgenommen. Dieser Effekt kann jedoch vernachlässigt werden, wenn das Audiosignal eine hohe Lautstärke im Verhältnis zum Grundrauschen hat(Signal-Rausch-Verhältnis).[1]

ii. Sampling: Wandlung des analogen Audiosignals in digitale Werte

Um ein analoges Audiosignal in digitale Daten zu überführen, ist zunächst ein Analog/Digital-Wandler notwendig, welcher die analogen Spannungswerte in digitale Zahlenwerte umwandelt. Dieser ist unter anderem in Audiointerfaces vorhanden. Um ein analoges Audiosignal aufzunehmen(z.B. auf eine SD-Karte), müssen die Zahlenwerte(*Samples*) vom Analog/Digital-Wandler in regelmäßigen Abständen(*Abtastrate*) abgenommen werden.Diesen Vorgang nennt man *Sampling*.[2]

ii.1 Abtastrate

Die Abtastrate beschreibt, wie viele Samples pro Sekunde vom Analog/Digital-Wandler abgenommen werden und wird gewöhnlich in Hertz oder Kilo-Hertz angegeben. Je höher die Abtastrate, desto höher die Frequenzen, die dargestellt werden können. Laut *Nyquist-Theorem* sollte die Abtastrate f_a mindestens doppelt so hoch sein wie die höchste aufzunehmende Frequenz[2] f_{max}:

$$f_a \geq 2 \times f_{max} \tag{1}$$

Im Musikbereich, zum Beispiel bei Audio-CDs wird häufig eine Abtastrate von $44,1kHz$ genutzt, dies beinhaltet $2 \times 22,05kHz$, ist also ausreichend, um den Frequenzbereich des menschlichen Gehörs darzustellen. Da mit steigender Abtastrate aber der Rechenaufwand bei der Verarbeitung sowie die nötige Speichergröße zum speichern des Signals steigt, können für Einsatzgebiete mit niedrigeren Qualitätsansprüchen auch geringere Abtastraten sinnvoll sein. So können zur Übertragung von Sprache schon Abtastraten von $8kHz$ bis $16kHz$ ausreichen, um brauchbare Ergebnisse zu erzielen und Rechenleistung sowie Bandbreite in der Übertragung zu einzusparen.[2]

ii.2 Bittiefe

Die Bittiefe spiegelt die Genauigkeit der digitalen Samples wieder, welche die analogen Spannungswerte des Audiosignals abbilden. Bei kleinen Bittiefen werden die analogen Samples nur sehr ungenau und mit größeren Sprüngen abgebildet, was sich in einem unklaren Klang widerspiegelt. Mit steigender Bittiefe wird die Abbildung jedoch immer exakter, der Klang wird also deutlicher(siehe Bild 3). Ein weiterer positiver Effekt, der mit einer höheren Bittiefe einhergeht, ist der größere Dynamikumfang: Da leisere Passagen nicht den vollen Wertebereich ausnutzen, haben sie auch eine geringere Bittiefe, also einen unklaren Klang. Ist die allgemeine Bittiefe hoch gewählt, fällt dieser Effekt nicht mehr so drastisch aus, so dass auch leisere Passagen noch deutlich klingen.

Im Musikbereich werden meist 16 Bit (CD-Qualität) oder im professionellen Bereich sogar 24 Bit Bittiefe genutzt. Für Anwendungen wie eine Sprachübertragung werden aber auch teilweise 8 Bit Bittiefe genutzt, zum Beispiel bei der ISDN-Telefonie.[2]

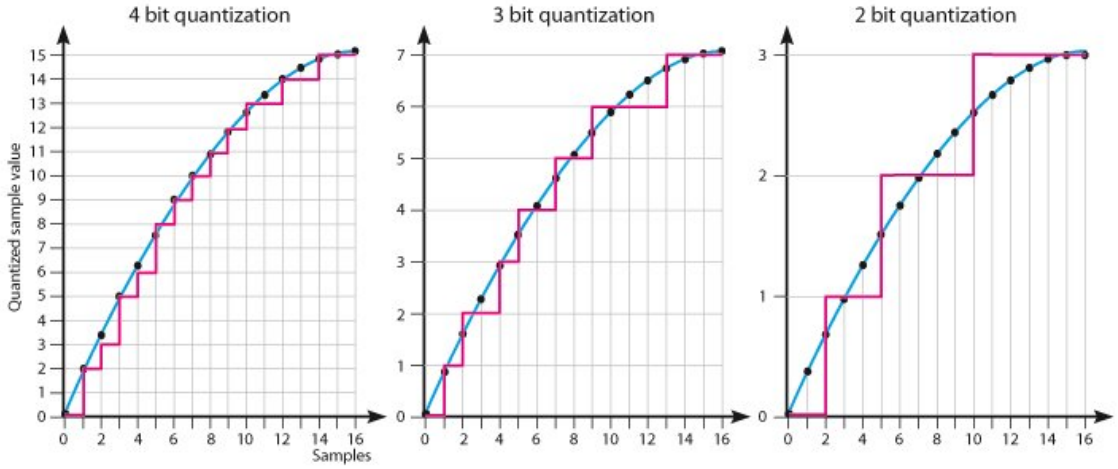

Abbildung 3: *Sampling mit verschiedenen Bittiefen*

iii. Buffering

Damit ein Audiosignal korrekt aufgenommen bzw. wiedergeben werden kann, darf es keine Aussetzer haben. So muss bei der Wiedergabe mit einer Abtastrate von $44,1kHz$ ca. alle $22\mu s$ eine neues Sample, also ein neuer Wert für den Digital/Analog-Wandler des Audiointerfaces verfügbar sein. Kann die Software in dieser Zeit keinen neuen Wert nachliefern, kommt es zu Knacken und Aussetzern im Audiosignal. Es ist für die Software schwer, diese Anforderungen einzuhalten. Das gilt insbesondere für Hardware mit begrenzen Ressourcen und Betriebsystemen ohne Echtzeit-Unterstützung, wie die meisten Linux-Distributionen, welche oft bei eingebetteten Systemen wie dem Beaglebone Black verwendet werden. Deshalb wird mit Puffern gearbeitet, welche es ermöglichen, dem Audiointerface gleich mehrere Samples auf einmal zu übergeben. So muss die Software bei einer Puffergröße von 512 Samples nur noch ca. alle $11,3ms$ neue Werte an das Audiointerface liefern. Mit größerer Puffergröße verringert sich die CPU-Auslastung und die Wahrscheinlichkeit, das die Software bzw. die CPU durch ungünstiges Scheduling nicht schnell genug neue Werte liefern kann, sinkt. Diese Sicherheit und Performance von einer größeren Puffergröße P geht jedoch auch mit einer größeren Latenz t_{Latenz} einher:

$$t_{Latenz} = P \times \frac{1}{f_{Abtast}} \tag{2}$$

Die Latenz beschreibt die Verzögerungszeit, die von der Verarbeitung bzw. Berechnung des Signal bis zur Ausgabe vergeht. Diese Zeit verdoppelt sich, wenn ein Signal im *Full-Duplex*-Modus verarbeitet wird, also wenn ein Signal aufgenommen, verarbeitet und wiedergeben wird, da sich hier die Latenzen von der Aufnahme und der Wiedergabe addieren.

Ab ca. $12ms$ ist für den Menschen eine Verzögerung wahrnehmbar. Dies kann je nach Anwendungsfall mehr oder weniger störend sein. Im Musikbereich, zum Beispiel bei der Klangerzeugung für MIDI-Keyboards oder bei der Verarbeitung von Gitarren-Effekten ist es sinnvoll, mit kleinen Latenzen zu arbeiten und in CPU-Leistung zu investieren. In anderen fällen, zum Beispiel bei der Klanganalyse zur Erkennung von Geräuschen zur Steuerung von Schaltern für eine Hausautomation können größere Latenzzeiten ausreichen, um noch genug Rechenleistung für andere Aufgaben bereitzustellen.

In der Praxis ist es sinnvoll, für den Anwendungsfall angemessene Puffergrößen zu wählen und durch testen von verschiedenen Einstellungen den richtigen Kompromiss zwischen Sicherheit, Latenz und Performance zu finden.[3]

III. Audioverarbeitung mit dem Beaglebone Black

In diesem Kapitel wird anhand von Programmbeispielen in C erklärt, wie man unter Linux mithilfe des ALSA-Treibers und eines USB-Audiointerfaces am Beaglebone Black Audiosignale verarbeitet. Hierfür wird zunächst auf die Inbetriebnahme von Audiointerfaces auf den Beaglebone Black eingegangen, bevor über konkrete Programmausschnitte die Implementierung von Konfiguration, Wiedergabe und Aufnahme bis hin zu einer Full-Duplex-Anwendung erklärt wird. Die Programmierbeispiele basieren auf [4], sind aber für die Arbeit angepasst worden.

i. Inbetriebnahme des Audiointerfaces

Die meisten Class-Compliant USB-Audiointerfaces sind kompatibel mit dem Beaglebone Black, da sie generell nach dem Plug-and-Play-Prinzip ohne externe Treiber mit Linux-Systemen funktionieren. In folgenden Beispiel wird ein Speedlink Vigo Interface verwendet, da es von mehreren Quellen als gute Low-Budget-Lösung für den Raspberry PI empfohlen wird.

Nach dem Bootvorgang des Beaglebone Black mit dem Speedlink Vigo, sollte er mit dem Befehl `aplay -L` als Audiointerface gelistet werden:

```
default:CARD=Device
    Generic USB Audio Device, USB Audio
    Default Audio Device
```

Der Name des Interfaces (`default:CARD=Device`) wird später für die Konfiguration der Audioanwendung benötigt.

ii. Programmbeispiel: Konfiguration des Audiointerfaces

Bevor die eigentliche Konfiguration des ALSA-Treibers beginnt, ist es sinnvoll, die zu setzenden Parameter in Variablen anzugeben, um sie später einfach anpassen zu können. Im folgenden werden die Werte für die Anzahl der Kanäle(`nchannels`), die Puffergröße(`buffer_size`), die Samplingrate(`sample_rate`), die Bittiefe(`bits`), sowie die Anzahl Perioden, in die der Puffer geteilt wird(`fragments`). Außerdem wird angegeben, welches Interface als Eingang(`snd_device_in`) und welches als Ausgang(`snd_device_out`) genutzt werden soll.

```
int             nchannels = 1;
int             buffer_size = 1881;
unsigned int    sample_rate = 44100;
int             bits = 16;
unsigned int    fragments = 2;

char const      *snd_device_in  = "default:CARD=CODEC";
char const      *snd_device_out = "default:CARD=CODEC";
```

Um den Alsa-Treiber nutzen zu können, muss er importiert werden:

```
#include <alsa/asoundlib.h>
```

Folgende Funktion kann genutzt werden, um das Sounddevice zu konfigurieren:

```
int configure_alsa_audio(snd_pcm_t *device, int channels){
    snd_pcm_hw_params_t *hw_params;
    int                 err;
    unsigned int        tmp;
    snd_pcm_uframes_t   frames;
    int frame_size;

    if ((err = snd_pcm_hw_params_malloc(&hw_params)) < 0) {
        fprintf (stderr, "cannot allocate parameter structure (%s)\n",
            snd_strerror(err));
        return 1;
    }

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

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

```
    if ((err = snd_pcm_hw_params_set_format(device, hw_params,
        SND_PCM_FORMAT_S16_LE)) < 0) {
        fprintf (stderr, "cannot set sample format: %s\n", snd_strerror(err));
        return 1;
    }

    tmp = sample_rate;
    if ((err = snd_pcm_hw_params_set_rate_near(device, hw_params, &tmp, 0)) < 0)
        {
        fprintf (stderr, "cannot set sample rate: %s\n", snd_strerror(err));
        return 1;
    }

    if (tmp != sample_rate) {
        fprintf(stderr, "Could not set requested sample rate, asked for %d got %d
            \n", sample_rate, tmp);
        sample_rate = tmp;
    }

    if ((err = snd_pcm_hw_params_set_channels(device, hw_params, channels)) < 0)
        {
        fprintf (stderr, "cannot set channel count: %s\n", snd_strerror(err));
        return 1;
    }

        if ((err = snd_pcm_hw_params_set_periods_near(device, hw_params, &
            fragments, 0)) < 0) {
                fprintf(stderr, "Error setting # fragments to %d: %s\n",
                    fragments, snd_strerror(err));
                return 1;
    }

        frame_size = channels * (bits / 8);
    frames = buffer_size / frame_size * fragments;

    if ((err = snd_pcm_hw_params_set_buffer_size_near(device, hw_params, &frames)
        ) < 0) {
        fprintf(stderr, "Error setting buffer_size %d frames: %s\n", frames,
            snd_strerror(err));
        return 1;
    }

    if (buffer_size != frames * frame_size / fragments) {
        fprintf(stderr, "Could not set requested buffer size, asked for %d got %d
            \n", buffer_size, frames * frame_size / fragments);
    }

        if ((err = snd_pcm_hw_params(device, hw_params)) < 0) {
                fprintf(stderr, "Error setting HW params: %s\n", snd_strerror(err
                    ));
                return 1;
        }
    return 0;
}
```

Mit `snd_pcm_hw_params_malloc` und `snd_pcm_hw_params_any` wird Speicher für die Struktur `hw_params` alloziert und initialisiert. Mit `snd_pcm_hw_params_set_access` wird der Zugriffmodus auf den Puffer konfiguriert. Ist der Zugriffsmodus, wie im Beispiel, auf `SND_PCM_ACCESS_RW_INTERLEAVED` konfiguriert, wird später bei der Aufnahme bzw. Wiedergabe `snd_pcm_readi` bzw. `snd_pcm_writei` zum lesen bzw. schreiben des Puffers genutzt. Ist der Zu-

griffsmodus hingegen auf SND_PCM_ACCESS_RW_NONINTERLEAVED konfiguriert, werden snd_pcm_readn bzw. snd_pcm_writen genutzt. Um das Sampleformat zu setzen, wird snd_pcm_hw_params_set_format genutzt, welches das in unserem Falle auf das oft genutzte Format 16 Bit Little-Endian(SND_PCM_FORMAT_S16_LE) konfiguriert, die Samplingrate wird mit snd_pcm_hw_params_set_rate_near gesetzt. Die Anzahl der Kanäle sowie Perioden, in die der Puffer geteilt wird werden mit snd_pcm_hw_params_set_channels und snd_pcm_hw_params_set_periods_near gesetzt. Folgende Rechnung berechnet die frames, die benötigt werden, um die gewünschte Puffergröße(buffer_size) setzen zu können:

```
frame_size = channels * (bits / 8);
frames = buffer_size / frame_size * fragments;
```

Mit snd_pcm_hw_params_set_buffer_size_near wird die Puffergröße auf die nächste kompatible Größe gesetzt. Je nach Audiointerface sind nur sehr wenige Puffergrößen konfigurierbar, im Rahmen der Entwicklung der Programmierbeispiele konnte mit der Speedlink Vigo nur eine Puffergröße von 1881 konfiguriert werden. Abschließend werden die Einstellungen mit snd_pcm_hw_params gesetzt.

Um die Konfiguration bzw. die Anwendung im Allgemeinen auszuführen, wird eine main-Funktion benötigt:

```
int main (int argc, char *argv[]){
        int err;
        snd_pcm_t               *playback_handle;
        snd_pcm_t               *capture_handle;

        //open playback handle
        if ((err = snd_pcm_open(&playback_handle, snd_device_out,
            SND_PCM_STREAM_PLAYBACK, 0)) < 0) {
                fprintf(stderr, "cannot open output audio device %s: %s\n",
                    snd_device_in, snd_strerror(err));
                exit(1);
        }
        //open capture handle
        if ((err = snd_pcm_open(&capture_handle, snd_device_in,
            SND_PCM_STREAM_CAPTURE, 0)) < 0) {
                fprintf(stderr, "cannot open input audio device %s: %s\n",
                    snd_device_out, snd_strerror(err));
                exit(1);
        }

        //configure in and out to nchannels(mono)
        configure_alsa_audio(playback_handle,  nchannels);
        configure_alsa_audio(capture_handle, nchannels);
```

Hier wird das Interface mit snd_pcm_open einmal als Playback-Handle(SND_PCM_STREAM_PLAYBACK) für die Wiedergabe und einmal als Capture-Handle(SND_PCM_STREAM_CAPTURE) für die Aufnahme geöffnet. Mit dem Aufruf von der vorher definierten Funktion configure_alsa_audio für beide Handles werden die gewünschten Parameter konfiguriert.

Um einen Fehlerfreien Ablauf zu gewährleisten, müssen vor der Aufnahme bzw. Wiedergabe für beide Handles mit snd_pcm_drop gestoppt und mit snd_pcm_prepare auf ihren Initialzustand vorbereitet werden:

```
snd_pcm_drop(capture_handle);
snd_pcm_drop(playback_handle);
snd_pcm_prepare(capture_handle);
snd_pcm_prepare(playback_handle);
```

iii. Wiedergabe von Audiosignalen

Als Beispiel für die Wiedergabe wird ein Sinussignal verwendet, welches in einem Array gespeichert ist:

```
sine_frames = 940;
short sine[sine_frames] ={32768,34375,35979,37575,39160,40729,42279,43807,...
```

Um das Beispiel zu vereinfachen, ist die Größe des Arrays an die Puffergröße angepasst:

$$Frames = \frac{Buffersize}{Periods} = \frac{1881}{2} = 940 \tag{3}$$

Für die Wiedergabe wird die Funktion `snd_pcm_writei` in einer Endlosschleife aufgerufen. Sie blockiert solange, bis die gewünschte Anzahl von Samples(`sine_frames`) aus dem Array(`sine`) auf den Puffer des Audiointerfaces(`playback_handle`) geschrieben werden konnten. Als Rückgabewert wird die Zahl der geschrieben Samples zurückgegeben(`outframes`). Unterscheidet sich diese von der gewünschten Anzahl von Samples(`sine_frames`), liegt ein Fehler vor.

```
boolean running = true;
int outframes;

while(running){
    /* Hier wird meist der Inhalt des Ausgangspuffers berechnet */

    outframes = snd_pcm_writei(playback_handle, sine, sine_frames);
    if (outframes != sine_frames){
        fprintf(stderr, "Short write to playback device!\n");
        break;
    }
}
```

Wird der Inhalt des Ausgangspuffers dynamisch während des Programmablaufs berechnet, ist es für eine klare Wiedergabe ohne Aussetzer wichtig, dass diese Berechnungen nicht zu lange dauern. Im Falle einer Ausgabe von 940 Samples bei $44,1kHz$ dürfen zwischen den einzelnen Aufrufen von `snd_pcm_writei` nicht mehr als $22,32ms$ vergehen:

$$t_{delay} = \frac{1}{f_{samplingrate}} * Buffersize = \frac{1}{44,1kHz} * 940 = 22,32ms \tag{4}$$

iv. Aufnahme von Audiosignalen

Die Aufnahme von Audiosignalen läuft ähnlich ab wie die Wiedergabe. In einer Endlosschleife wird mit der Funktion `snd_pcm_readi` ein Puffer(`input_buffer`) mit einer bestimmten Anzahl von Samples(`input_frames`) gefüllt:

```
boolean running = true;
int inframes;
input_frames = 940;
short input_buffer[input_frames];

while(running){
    inframes = snd_pcm_readi(capture_handle, input_buffer, input_frames);
    if (inframes != input_frames){
        fprintf(stderr, "Short read from capture device");
        break;
    }
```

```
    /* Hier wird meist der Inhalt des Eingangspuffers verarbeitet */
}
```

Auch hier ist es wichtig, dass zwischen den Aufrufen von `snd_pcm_readi` nicht zu viel Zeit vergeht. Der gefüllte Eingabepuffer kann nach dem Lesen in eine Datei geschrieben oder in Echtzeit weiterverarbeitet werden, zum Beispiel mit einem Algorithmus zur Erkennung von Sprache.

v. Full-Duplex-Audiosignalverarbeitung

Bei der Full-Duplex-Audiosignalverarbeitung wird das Audiosignal in einer Endlosschleife abwechselnd gelesen und geschrieben. Zwischen Lesen und Schreiben wird der Puffer verarbeitet, in diesem Beispiel wird von jedem Sample die Wurzel gezogen und durch Multiplikation an den Wertebereich von 16 Bit angepasst:

```
boolean running = true;
int inframes;
int outframes;
input_frames = 940;
short buffer[input_frames];

while(running){
    inframes = snd_pcm_readi(capture_handle, buffer, input_frames);
    if (inframes != input_frames){
        fprintf(stderr, "Short read from capture device");
        break;
    }

    /* Verarbeitung des Eingangssignals:
       Berechnung der Wurzel von jedem Sample */
    for(int i = 0; i < 940; i++){
        if(buffer[i] >= 0){
            buffer[i] = ((int) sqrt(buffer[i])) * 255;
        }else{
            buffer[i] = ((int) sqrt(abs(buffer[i]))) * 255 * (-1);
        }
    }

    outframes = snd_pcm_writei(playback_handle, buffer, inframes);
    if (outframes != inframes){
        fprintf(stderr, "Short write to playback device!\n");
        break;
    }
}
```

Auch hier ist es wichtig, dass die Berechnung nicht länger dauert als die durch die Puffergröße gegebene Latenzzeit(in diesem Beispiel $22,32ms$, Berechnung im Kapitel "Wiedergabe von Audiosignalen"). Je länger die Berechnung dauert, desto größer ist die CPU-Auslastung, da anteilig mehr Zeit in die der Berechnung und weniger in das Warten auf die Ein-und-Ausgabe des Signals investiert wird. Durch Wählen einer größeren Puffergröße kann dieses Verhältnis geändert werden und so die CPU-Auslastung verringern.

IV. Performaceuntersuchung des Beaglebone Black bei Full-Duplex-Audioverarbeitung

In diesem Kapitel wird die Beispielanwendung aus Kapitel 3.4. untersucht. Es wird die Latenz und die CPU-Auslastung bei unterschiedlichen Konfigurationsparametern gemessen und ausgewertet, um ein besseres Bild über die Möglichkeiten und Grenzen der Audiosignalverarbeitung mit dem Beaglebone Black zu bekommen.

i. Versuchsaufbau

Zum Test der Beispielanwendung wurde ein Beaglebone Black mit einem "Speedlink" Vigo Audiointerface verwendet. Um die Funktion nachzuweisen und die Latenz zu messen, wurde ein PC mit Ableton Live 9 und ein "Behringer UCA202" Audiointerface genutzt. In die Eingänge beider Interfaces wird ein Sinussignal gegeben, zusätzlich wird auf einem zweiten Kanal des Audiointerfaces zum Messen das verarbeitete Signal vom Beaglebone Black gegeben(siehe Bild 4). Da beide Kanäle gleichzeitig aufgenommen werden, ist es möglich, aus der Aufnahme die Latenzzeit abzulesen.

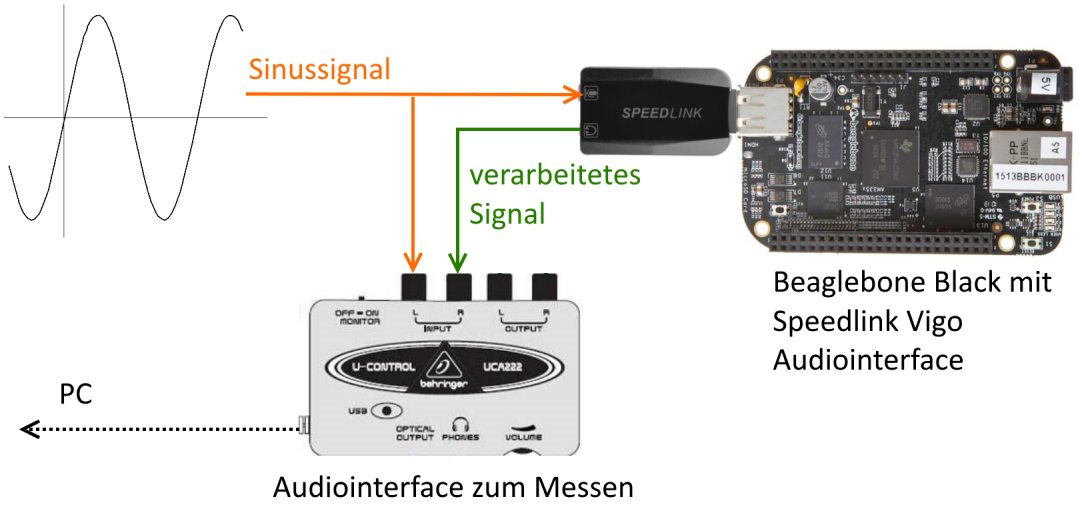

Abbildung 4: *Versuchsaufbau*

ii. Funktionsnachweis

In der Aufnahme(siehe Bild 5) ist das Eingangssignal unten zu sehen und das verarbeitete Signal oben. Man sieht einen deutlichen Unterschied in der Form des Signals, durch die Verarbeitung mit der Wurzelfunktion wird das Sinussignal gedehnt. Es ist auch eine Verzögerung zu erkennen, auf diese wird im folgenden Kapitel eingegangen.

iii. Latenzmessung

Um die Latenzzeit zu Messen, wurde eine Aufnahme gemacht, auf der am Anfang kein Eingangssignal vorhanden ist. Nachdem das Sinussignal gestartet wurde, ist auf der Aufnahme eine

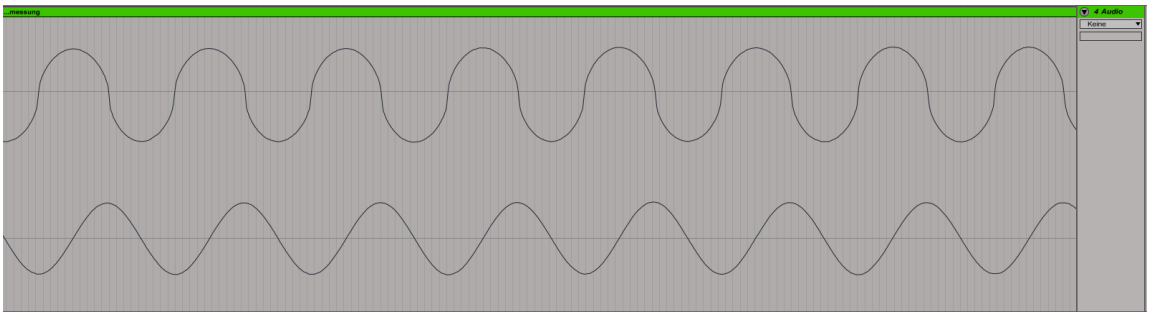

Abbildung 5: *Funktionsnachweis*

Verzögerung zu erkennen, die blau markiert wurde(siehe Bild 6). Die Zeit des blau markierten Bereichs beträgt laut Ableton $23,8ms$, was der errechneten Latenzzeit bei einem Puffer von 940 Samples($22,32ms$) ungefähr entspricht(siehe Kapitel 3.3).

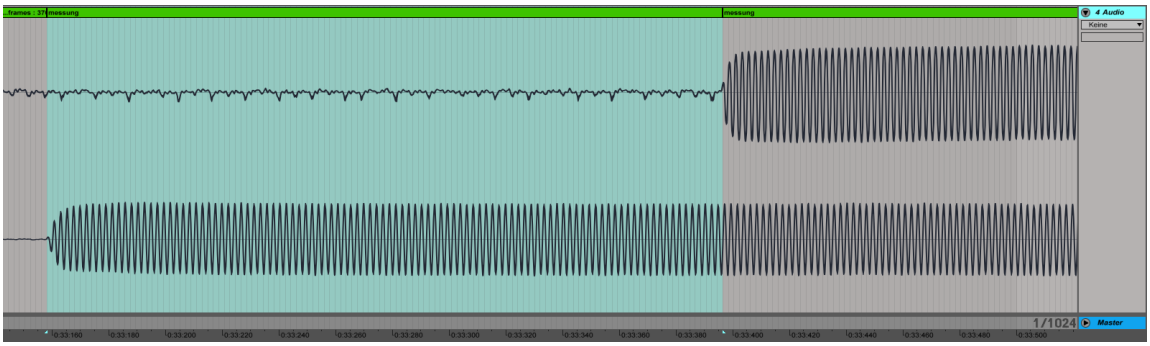

Abbildung 6: *Latenzmessung: Aufnahme*

iv. Performancemessung

Da mit dem Speedlink Vigo und dem Alsa-Treiber nur eine Puffergröße von 940 akzeptiert wird, konnte nur eine Performancemessung mit dieser stattfinden. Bei dem Test wurde mit dem Befehl `top` die CPU-Auslastung der Testanwendung(hier mit dem Namen `test`) untersucht(siehe Bild 7). Diese bewegte sich immer zwischen 95% und 99.9%, der Grund dafür konnte während der Untersuchung leider nicht gefunden werden. Um die Grenzen des Beaglebone Black bei der Audioverarbeitung sichtbar zu machen, wurde, neben der Wurzelberechnung, noch ein Stück Code zur gezielten Verzögerung zwischen Eingabe und Ausgabe eingefügt:

```
volatile long time_eater = 0;
for(time_eater = 0; time_eater < 96000; time_eater++) {}
```

Empirisch wurde ermittelt, das es bei Maximalwerten von `time_eater` zwischen 96000 und 97000 zu Buffer-Underruns bei der Ausgabe und somit zu Aussetzern des Ausgangssignals kommt. Auf die CPU-Auslastung hatte die zusätzliche For-Schleife keinen Einfluss.

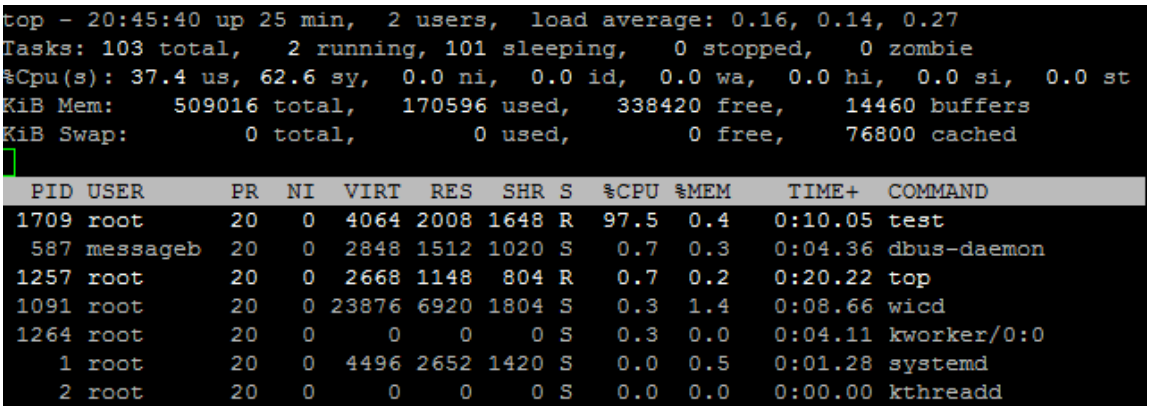

Abbildung 7: *Performancemessung mit Top*

V. Fazit

Im Rahmen dieser Arbeit wurde am Beispiel des Beaglebone gezeigt, wie Audioverarbeitung in eingebetteten Systemen mit einer Linux-Umgebung Implementiert werden kann. Dies lässt sich auch auf ähnliche Systeme wie den Raspberry PI übertragen. Es wurde ebenfalls auf die theoretischen Grundlagen der digitalen Audioverarbeitung eingegangen, sodass der Leser ein Verständnis für die Konfigurationsparameter einer Audioanwendung aufbauen kann. Somit schafft die Arbeit eine gute Grundlage für die Implementation von Audioanwendungen in eingebetteten Systemen. Die Untersuchung der Beispielanwendung wurde leider dadurch beschränkt, dass mit dem "Speedlink Vigo" nur eine Puffergröße(940) gewählt werden konnte. So konnten keine genaueren Untersuchungen zum Einfluss der Puffergröße auf Latenz und CPU-Auslastung gemacht werden. Der Test mit weiteren, hochwertigeren Audiointerfaces wären hier hilfreich gewesen. Trotzdem hat die Untersuchung gezeigt, dass es trotz der beschränkten Leistung möglich ist, im mittleren Latenzbereich(22ms) Audioverarbeitung mit aufwendigeren Berechnungen durchzuführen(im Beispiel eine Wurzelberechnung und eine For-Schleife mit 96000 Schritten).

VI. Quellen

- [1] Eschenbacher, Thomas: Grundlagen digitaler Audiobearbeitung(2017), URL: `http://kwave.sourceforge.net/doc/de/digital-audio-basics.html` (Stand:13.09.2017)
- [2] Robertson, Hal: Digital Audio Sampling(2010), URL `https://www.videomaker.com/article/c4/14524-digital-audio-sampling` (Stand:13.09.2017)
- [3] Connor, Dan: Dealing with Latency: The Audio Buffer(2007), URL: `http://thestereobus.com/2007/12/13/dealing-with-latency-the-audio-buffer/` (Stand:13.09.2017)
- [4] Clob, Alan: Full Duplex ALSA(2005), URL: `http://www.saunalahti.fi/~s7l/blog/2005/08/21/Full%20Duplex%20ALSA.html` (Stand:13.09.2017)

i. Bilder

Abbildung 1 blogs.uoregon.edu/uocinetech/files/2015/08/sine-wave-11jcwj1.png

Abbildung 2 www.ee.surrey.ac.uk/Projects/CAL/frequency-response/Activelowpass.jpg

Abbildung 3 i.stack.imgur.com/5JG8m.gif