

Jürgen Plate

Sound-Verarbeitung mit Perl

Supplement zum Buch:
Jürgen Plate
Der Perl-Programmierer
Erschienen im Hanser-Verlag

Inhaltsverzeichnis

1	Sounds bearbeiten	5
1.1	Akustische Signale	5
1.1.1	Grundlagen	5
1.1.2	Der Sound-Exchanger SoX	8
1.1.3	MP3-Player mpg123	9
1.1.4	Der Perl Audio Converter	9
1.2	Perl-Module speziell für Windows	11
1.3	Sound mit dem internen Lautsprecher	18
1.4	Perl und Sox	20
1.5	Weitere Sound-Module	23

1

Sounds bearbeiten

These are bagpipes. I understand the inventor of the bagpipes was inspired when he saw a man carrying an indignant, asthmatic pig under his arm. Unfortunately, the man-made sound never equalled the purity of the sound achieved by the pig.

Alfred Hitchcock

1.1 Akustische Signale

Wie Sie wahrscheinlich wissen, werden akustische Signale (Töne, Musik) wie alles andere in digitaler Form im Computer gespeichert. Wenn die Klänge nicht direkt per Programm erzeugt werden, werden sie meist aus einer anderen, analogen Quelle in den Computer „eingelassen“. Ich überspringe den rein elektrischen Teil aus Mikrofon, Verstärker usw. und wende mich gleich der Digitalisierung zu. Das elektrische Signal wird in eine computerge-rechte Form gebracht; es erfolgt eine Analog-zu-Digital-Wandlung. Dabei wird aus einem kontinuierlichen analogen Signal mit unendlich vielen Zwischenwerten eine kleine Menge diskreter Punkte erzeugt. Auch wenn eine analoge Schwingung beispielsweise nur eine Sekunde dauert, setzt sich diese doch aus unendlich vielen Zeitpunkten zusammen. Die Digitalisierung ändert daher eine wichtige Eigenschaft des analogen Signals, indem sie es aus einem kontinuierlichen und unendlichen in ein diskretes und endliches verwandelt.

1.1.1 Grundlagen

Wie läuft so etwas technisch ab? Der Schlüsselbegriff der modernen Digitaltechnik heißt „Sampling“ (sample = Beispiel, Muster). Ein digitales Signal entsteht, indem von einem analogen Signal Samples genommen werden. Diese Abtastung geschieht mit hoher Geschwindigkeit in regelmäßigen Abständen. Der Abstand jedes Samples von Null ist der Digitalwert, der als reiner Zahlenwert gespeichert wird. Mit einer für Soundadapter üblichen Sampling-Rate von 48 kHz sind das 48000 gespeicherte Werte pro Sekunde. Ein zweites Kriterium ist die Auflösung eines Samples, d. h. in wie viele Stufen der Spannungsbereich zwischen Null und der maximalen Amplitude aufgeteilt wird. Das digitalisierte Signal entspricht somit nur teilweise dem analogen Original.

Die Digitalisierung führt auch zu einer Reihe von störenden Nebenwirkungen, die durch besondere Maßnahmen beseitigt werden müssen. Sie treten bei der Wiedergabe, der

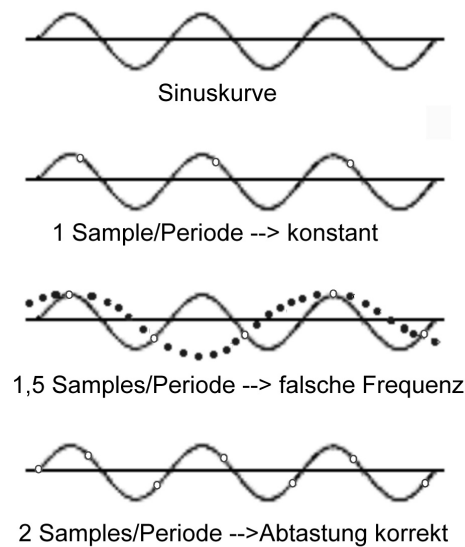


Bild 1.1: Sinuskurve mit verschiedenen Abtastraten

Digital-zu-Analog-Wandlung, auf. Angenommen, man hätte pro Welle einer Schwingung nur ein einziges Sample genommen. Dann gibt es bei der D/A-Wandlung mehrere Möglichkeiten, aus diesem eine analoge Schwingung zu rekonstruieren. Da man dann nicht weiss, wo die negative Halbwelle liegt, passt z. B. auch eine Welle mit halber Frequenz zu diesem Wert. Man benötigt also mindestens zwei Werte, um eine Schwingung aus Wellenberg und Wellental richtig vermessen zu können. Bild 1.1 illustriert diese Problematik.

Daraus erklärt sich auch die bei CD-Spielern verbreitete Sampling-Rate von 44,1 kHz. Da der Mensch Töne bis etwa 20 kHz hören kann, müssen diese mit der doppelten Frequenz abgetastet werden, damit sie eindeutig analysierbar sind. Man kann beweisen, dass bei Erfüllung der beiden folgenden Forderungen keine Informationsverluste entstehen, wenn das digitale Signal wieder in ein analoges zurückverwandelt wird:

- Die Sampling-Rate muss mindestens doppelt so groß sein wie die höchste zu verarbeitende Frequenz.
- Das Signal darf keine Frequenzen enthalten, die über der halben Sampling-Rate liegen.

Beide Forderungen folgen aus zwei wichtigen Forschungsergebnissen der Nachrichtentechnik und Informationstheorie. Harry Nyquist formulierte 1928 das Nyquist-Kriterium, nach der man eine Sampling-Frequenz finden kann, mit der es möglich ist, ein digitalisiertes Signal verlustfrei wieder zurückzuverwandeln. Der amerikanische Mathematiker Claude E. Shannon stellte 1949 das Abtasttheorem auf. Es gibt an, wie viele Werte man für eine verlustfreie Rekonstruktion des Ausgangssignals benötigt. Aus technischer Sicht muss man unbedingt sicherstellen, dass keine höheren, möglicherweise außerhalb des Hörbereichs liegenden Frequenzen in dem zu digitalisierenden Signal enthalten sind. Andernfalls treten hörbare Störungen auf, die man als „Aliasing“ (von lat. alias = anders) bezeichnet. Daher wird am Eingang eines A/D-Wandlers ein Tiefpassfilter eingesetzt, der alle unerwünschten Frequenzen unterdrückt. Die moderne CD-Technik erleichtert dies noch durch Oversampling. Dabei wird das Eingangssignal mit einer sehr hohen Frequenz abgetastet (bis zu 6 MHz), die die Filterung verbessert, wonach dann die erhaltenen Werte auf die eigentliche Sampling-Frequenz (z. B. 48 kHz) heruntergerechnet werden.

Zur Wiedergabe wird umgekehrt ein Digital-Analog-Umsetzer verwendet, der die digitalisierten Wellenformen wieder in analoge Spannungsschwankungen umsetzt. Danach kann über einen Verstärker und Lautsprecher die Umsetzung in akustische Schwingungen erfolgen. Die „Lücken“ eines digitalen Signals müssen wieder durch einen Filter beseitigt werden. Auch hier hilft Oversampling bei der Unterdrückung von Störungen, u. a. bei der Reduzierung von Quantisierungsrauschen, das während des Digitalisierungsprozesses entsteht.

Die Manipulation der Abtastrate ermöglicht aber auch typische Sampler-Effekte. Wird etwa das Sample bei der Wiedergabe mit einer anderen Abtastrate als bei der Aufnahme abgespielt, ändert sich die Tonhöhe.

Die Wortbreite oder Bitauflösung bestimmt, mit welcher Genauigkeit das Signal beschrieben wird. Allgemein üblich ist heutzutage eine Auflösung von 16 Bit. Das Signal kann also Werte zwischen -32767 und +32767 annehmen. 16 Bit entspricht schon HiFi; für ein Telefonsignal genügen schon acht Bit.¹

Es gibt im Prinzip drei Arten, digitale Signale zu betrachten:

- Als Folge von Abtastwerten (Zahlen, *sound.raw*), die z. B. mit dem Hexdump-Programm angesehen werden können,
- als gezeichnetes Signal auf dem Display oder Oszilloskop oder
- als rekonstruiertes und damit hörbares Signal (z. B. `play sound.wav`).

Bei der Wiedergabe von Sound sind sowohl die Abtastrate als auch das Format der einzelnen Abtastwerte für die korrekte Wiedergabe entscheidend. Tools zur Aufnahme, Bearbeitung und Wiedergabe von Sounds gibt es auf jeder Rechnerplattform haufenweise – sie sind auch nicht unser Thema. Suchen Sie sich einfach ein Tool aus, mit dem Sie gut zurechtkommen, und machen Sie ruhig einige Probeaufnahmen.

Da Computer ihre Daten grundsätzlich in Bytes anordnen, gibt es bei Auflösungen größer acht Bit noch das Problem der Byte-Reihenfolge (byteorder). Auf Maschinen mit beispielsweise Intel-Architektur kommt das untere Byte zuerst in die Datei (big endian), danach das höherwertige. Bei Motorola-, Sun- und anderen Architekturen ist es genau umgekehrt (little endian). Überträgt man nun ein Audio-File von einer Sun auf einen Linux-Rechner und versucht, es zu rekonstruieren, dann klingt das relativ gruselig. Das Vertauschen der Byte-Reihenfolge nennt man swappen. Daher haben viele Programme zum Abspielen von Audio-Files die Option, das Signal vor dem Abspielen zu Swappen.

Genau wie bei Grafikdateien gibt es eine Vielzahl verschiedener standardisierter Audio-File-Formate, die meistens (aber nicht immer) an der Datei-Extension erkennbar sind. Die wichtigsten Formate sind:

raw	gar kein Format, die Abtastwerte stehen einfach hintereinander
wav	Microsoft-Format, Mono-Stereo, verschiedene Abtastraten möglich
au	SUN Audio, alles möglich, default 8 kHz Abtastung, Mono
aiff	alles möglich
mp3	hohe Kompression (eigentlich MPEG 2.5 Audio Layer III)
ogg	(Ogg) Vorbis – patentfreie Alternative zu MP3

Daneben gibt es noch zahllose andere Formate. Um ein Audiofile eines Formats in ein anderes umzuwandeln, verwendet man Konvertierungsprogramme. Das Mächtigste dieser Programme ist das Programm `sox`, das auch zum Abspielen verwendet werden kann.

¹Daher rühren auch die 64'000 bps Datenrate beim ISDN-Telefon: Auflösung 8 Bit x 8 kHz Abtastrate.

Voraussetzung für das Anhören der Sounds ist natürlich, dass der Rechner über Soundkarte und Boxen verfügt. Das mit `sox` gelieferte Tool `play` erkennt Audioformate an der Extension und versucht, diese abzuspielen. Wenn es nicht klappt, erhält man eine Fehlerdiagnose. Zum Abhören von rohen Audio-Files (`.raw`), bei denen bekannt ist, welche Abtastrate sie haben, verwendet man beispielsweise `sox -t raw -r rate`.

Als „Header“ bezeichnet man einen Informationsblock zu Beginn eines Audio-Files. Die meisten Formate haben solche Header, in welchen die Metadaten des Files verzeichnet sind, beispielsweise die Abtastrate, die Bitbreite, das Sample-Format etc.). Die meisten Header sind binär gespeichert und daher nur mit geeigneter Software lesbar.

Die in den folgenden Abschnitten nur kurz beschriebenen Werkzeuge kennen eine schier unübersehbare Zahl von Möglichkeiten und Optionen, deren vollständige Behandlung den Rahmen dieses Buchs sprengen würde.² Sie dienen an dieser Stelle auch nur als „Mittel zum Zweck“, weshalb ich mich auf die Optionen beschränke, die in der jeweiligen Anwendung notwendig sind.

1.1.2 Der Sound-Exchanger SoX

In der Basisversion handelt es sich beim `SOund-eXchanger sox`, um ein reines Kommandozeilen-Utility, das vom Benutzer den exzessiven Einsatz von Kommandozeilenparametern fordert. Dafür hat das Programm einiges zu bieten: Es liest, schreibt und konvertiert faktisch alle existierenden Audioformate, seien sie auch noch so exotisch. Außer der bloßen Konvertierung inklusive Anpassung der Sampling-Rate und Quadro-, Mono- oder Stereoumwandlung kann `sox` den Files auch noch jeweils einen Effekt hinzurechnen, z. B. Highpass-, Lowpass- oder Bandpass-Filter, Echo, Hall, Chorus, Flanger und Vibrato etc. Nicht zuletzt handelt es sich um Freeware, die noch dazu mit dem kompletten Source-Code geliefert wird.

Da `sox` rein kommandozeilenorientiert arbeitet, ist es ebenso flexibel wie unkomfortabel. Beigelegte Shell-Skripte wie `play` und `record` erleichtern unter Linux den Einstieg; zudem greifen viele grafische Programme auf `sox` als Backend zurück, da die Programmierer nicht einsehen, warum sie das Rad noch einmal erfinden sollten (was ich auch mit Perl so machen werde). Außerdem ist `sox` das „Schweizer Taschenmesser“ unter den Soundbearbeitungsprogrammen. Auch unter Windows kann man z. B. mit wenigen Zeilen einer Batch-Datei alle Dateien eines Verzeichnisses bearbeiten. Einfacher geht's oft nicht.

Die Homepage von `sox` finden Sie unter <http://sox.sourceforge.net/>. Dort finden Sie neben vielen Infos auch die Programmversionen für Linux, Windows und MacOS zum Herunterladen. Das Programm läuft problemlos unter Linux und anderen Unix-Systemen, ebenso unter Windows XP auf der Kommandozeile. Bei Windows muss man gegebenenfalls dem Pfad zu `sox` mit in die PATH-Variable aufnehmen. Wer das Programm dann häufig und gerne verwendet, sollte auch mal an eine Spende (donation) an die Entwickler denken.³

Für den Anfang will ich nur drei kleine Beispiele geben, später binde ich dann Aufrufe von `sox` in die Perl-Programme ein. Um eine 10 Sekunden lange Musiksequenz mit Einblenden (fade-in) und Ausblenden (fade-out) zu erstellen, kann man folgendermaßen vorgehen:

```
sox starwars.wav out.wav fade 0:0.5 0:10 0:0.5
```

Das Fading dauert jeweils 0,5 Sekunden (Format: „hh:mm:ss.frac“, hier: „0:0.5“ für 0,5 s). Der Aufruf speichert die ersten 10 Sekunden der Melodie mit Ein- und Ausblendung in `out.wav`. Will man nicht einblenden, aber ausblenden, setzt man 0 für den ersten Wert ein.

²Allein das Programm `sox` böte genügend Stoff für ein eigenes Buch.

³Nicht nur dran denken, sondern Kohle abdrücken, verstanden?

Wenn Sie aber ein Musikstück von beispielsweise 12 Minuten Länge haben, aber gerne die bei 4 m 15 s beginnende Passage verwenden möchten, zerstückeln Sie zunächst die Ausgabedatei. Sie schneiden aus *starwars.wav* ab 4:15 die darauf folgenden 30 Sekunden heraus und wenden darauf dann den obigen Befehl an:

```
sox starwars.wav klein.wav trim 4:15 30
sox klein.wav out.wav fade 0:0.5 0:10 0:0.5
```

1.1.3 MP3-Player mpg123

mpg123 ist ein freier MPEG-1-Audio-Spieler für die Kommandozeile. Die unterstützten Formate sind die MPEG-1-Layer 1, 2 und 3 (Letzteres ist bekannt als MP3). Sein Name begründet sich in der Unterstützung dieser drei Formate. Er gehört zu den verbreitetsten MP3-Playern, und viele weitere Projekte nutzen dessen Code. Das Programm kann nicht nur als kommandozeilenbasierter Abspieler für Musik dienen, sondern durch seine Optionen auch als Dekomprimierstufe für andere Programme genutzt werden.

Das Programm läuft auf diversen Unix-Betriebssystemen. Offiziell unterstützt werden Linux, FreeBSD, SUN Solaris und andere. Seit einiger Zeit werden auch MacOS X und Windows (Cygwin) unterstützt – man benötigt einige DLL-Dateien aus dem Cygwin-Paket. Zu finden sind die mpg123-Dateien auf der Homepage <http://www.mpg123.de/>.

Das Programm unterstützt verschiedene Geräte zur Ausgabe des Tons. Außerdem kann es die decodierten Daten auch auf die Standardausgabe ausgeben. So können die Daten von einem anderen Programm weiterverarbeitet werden. Die Daten können auch als WAV in eine Datei oder roh auf die Standardausgabe geschrieben werden. mpg123 nimmt auch während des Abspielens/Decodierens Tastaturbefehle an. Es gibt noch eine weitere Steuerungsschnittstelle über die Standardeingabe, mit der die Musikausgabe gesteuert werden kann. Einige Bedienoberflächen nutzen diese Funktionalität. Ein Aufruf zum Abspielen könnte beispielsweise lauten:

```
# Musikdatei abspielen
mpg123 --audio-device /dev/dsp --verbose musik.mp3

# oder eine Jukebox basteln:
ls *.mp3 > mp3list
mpg123 --audio-device /dev/dsp --verbose --random --list ./mp3list
```

Statt der Option „-audio-device“ kann man auch nur „-a“ verwenden. Um Hilfe zu erhalten, ruft man das Programm mit den Optionen „-help“ oder „-longhelp“ auf.

mpg321 ist eine Alternative zu mpg123. Es greift es auf die *libmad* zurück. Der größte Vorteil von mpg321 besteht darin, dass man mit *-o* das gewünschte Audio-Device angeben kann (z. B. *oss*, *alsa*, *arts* oder *esd*).

1.1.4 Der Perl Audio Converter

Der Perl Audio Converter (PAC) ist ein Tool zum Umwandeln von Audiodateien und zur Extraktion von Audiodaten aus Videodateiformaten. Außerdem können mit dem Programm CDs gerippt und in vielen Formaten abgespeichert werden. Unterstützt werden eine Vielzahl auch recht exotischer Formate; die Bedienung erfolgt über die Kommandozeile (unter KDE ist auch eine Einbindung in Konqueror, Dolphin und Amarok möglich). Der PAC kennt die Formate AAC, AC3, AIFF, APE, AU, AVR, BONK, CAF, CDR, FAP, FLA, FLAC, IRCAM, LA, LPAC, M4A, MAT, MAT4, MAT5, MME, MP2, MP3, MP4, MPC, MPP, NIST, OFR, OFS, OGG, PAC, PAF, PVE, RA, RAM, RAW, SD2, SF, SHN, SMP, SND, SPX, TTA, VOC, W64, WAV, WMA, und WV. Audiodateien lassen sich aus den Videoformaten

RM, RV, ASF, DivX, MPG, MKV, MPEG, AVI, MOV, OGM, QT, VCD, SVCD, M4V, NSV, NUV, PSP, SMK, VOB, FLV, und WMV extrahieren.

Manchmal ist das Programm noch nicht als installierbares Paket in den Linux-Distributionen vorhanden und auch nicht mittels `apt-get install pacpl` einfach ins System einzubinden. In diesem Fall müssen Sie sich von der PAC-Homepage <http://pacpl.sourceforge.net/> das entsprechende Installationspaket herunterladen und installieren. Wenn auch da nichts Passendes zu finden ist, bleibt noch das Herunterladen und Compilieren des Quellpakets. Erschrecken Sie übrigens nicht – meist werden noch über Hundert Bibliotheken mit installiert, die alle mit der Soundcodierung zu tun haben. Eventuell benötigt das Tool auch noch einige bisher nicht installierte Perl-Module.

Nach der Installation können Sie fröhlich loskonvertieren. Die Syntax fällt denkbar einfach aus. Alle Optionen offenbart die Software, wenn Sie einfach `pacpl` ohne Parameter in der Shell aufrufen. Der Grundbefehl zur Konvertierung lautet:

```
pacpl --to <Format> <Optionen> [Datei(en)/Verzeichnis(se)]
```

Die Decodierungs- und Codierungsmöglichkeiten sowie Informationen zum Auslesen und Erstellen von Tags lassen sich mit `pacpl --formats` aufrufen. Sie erhalten eine Tabelle mit den verarbeitbaren Formaten (inklusive Dateierendungen), den eingesetzten Encodern und Decodern, und für welche Formate das Auslesen und Erstellen von Tags möglich ist. Zu den einzelnen Encodern lassen sich (je nach gewähltem Format) spezielle Einstellungen zu Qualität, Bitrate, Anzahl der Kanäle, Frequenz etc. machen. Eine komplette Übersicht über die sehr umfangreichen Optionen liefert die Manualpage des Programms. Gebräuchliche Optionen des Programms zeigt Tabelle 1.1:

Tabelle 1.1: Einige Optionen des Perl Audio Converters

Option	Funktion
<code>-t <Format>, --to <Format></code>	gewünschtes Ausgabeformat (gleichzeitig neue Dateierendung)
<code>-r, --recursive</code>	rekursive Umwandlung der Dateien auch in Unterverzeichnissen
<code>-p, --preserve</code>	existierende Ordnerstruktur wird auch für zu erstellende Ordner verwendet
<code>-o <Format>, --only <Format></code>	nur das angegebene Format wird umgewandelt
<code>-k, --keep</code>	bereits im gewünschten Format vorliegende Dateien werden übersprungen bzw. direkt verschoben
<code>--dryrun</code>	Testdurchlauf ohne tatsächliche Veränderung an den Dateien, sehr sinnvoll gerade bei potenziell gefährlichen Optionen!
<code>--normalize</code>	Standard-Normalisierung der Audiodateien
<code>--nopts <Option></code>	Normalisierung unter Angabe von Optionen
<code>--outdir <Dir></code>	Ausgabeverzeichnis, in das die Dateien geschrieben/verschoben werden
<code>--outfile <Name></code>	Ausgabename; Standard ist Beibehaltung des alten Namens mit neuer Dateierendung

Der folgende Aufruf konvertiert die im Verzeichnis `/home/sound` vorhandenen Audiodateien als `.ogg`-Dateien in das Verzeichnis `/home/oggs`.

```
pacpl --to ogg -p /home/sounds --outdir /home/oggs
```

Besonders interessant ist der Schalter `-r`, über den Sie die Inhalte ganzer Verzeichnisbäume unter Bewahrung der originalen Verzeichnisstruktur in das gewünschte Format konvertieren. Zum Beispiel:

```
pacpl --to mp3 --dir=/tmp/ogg
pacpl --to ogg --recursive --dir=/tmp/mp3 --outdir=/tmp/ogg
pacpl --to wav *.mp3
pacpl --to ogg -p -r --dir=/tmp/wav/ --outdir=/tmp/ogg
```

Sie können das Programm auch zum CD-Rippen verwenden. Als Default-Device verwendet der PAC `/dev/cdrom`. Bei Bedarf ändern Sie das mit der Option `--cdrom`. Als weitere Angaben benötigt PAC das Ausgabeformat und die zu wandelnden Tracks. Die Option `all` konvertiert die ganze CD. Ist das Perl-Modul `CDDB_get` installiert, kann man über den Schalter `--cdinfo` Informationen von CDDB abzurufen. Im folgenden Beispiel rippt der erste Befehl alle Tracks der eingelegten CD ins `.flac`-Format, der zweite die Tracks 1, 3, 9 und 15 als `mp3`-Dateien. Der dritte Befehl führt zur Ausgabe aller CD-Titel nach dem Muster „Künstler-Album-(Tracknummer)-Titel.mp3“.

```
pacpl --rip all --to flac
pacpl --rip 1,3,9,15 --to mp3
pacpl --rip all --to mp3 --nscheme="%ar-%ab-(%tr)-%ti"
```

Tabelle 1.2 listet einige Optionen für das Rippen von CDs auf.

Tabelle 1.2: Einige Rip-Optionen des Perl Audio Converters

Option	Funktion
<code>--rip <Option></code>	alle (all) oder die durch Kommata getrennten Titel der CD rippen (z. B. „1,3,9,15“)
<code>--nocddb</code>	keine CDDB-Abfrage zum Erstellen der Tags (Abfrage ist Standard)
<code>--noinput</code>	interaktiven Dialog zur CDDB-Abfrage unterdrücken (Dialog ist Standard)
<code>--device <Name></code>	anderes Gerät als <code>/dev/cdrom</code> für das CD-Laufwerk angeben
<code>--cdinfo</code>	CDDB-Infos zur momentan eingelegten CD anzeigen
<code>--nscheme <string></code>	Angaben zur Titelausgabe festlegen. Möglich sind folgende Parameter, die statt des Standards <code>%ar-%ti</code> (Künstler - Titel;Endung;) verwendet werden können: <code>%ar</code> Künstler <code>%ti</code> Titel <code>%tr</code> Tracknummer <code>%yr</code> Erscheinungsjahr <code>%ab</code> Album

Soviel zu den Audio-Tools, die extern aus einem Perl-Programm heraus aufgerufen werden können (PAC ist ja selbst auch schon wieder ein Perl-Tool). Daneben gibt es noch etliche andere wie z. B. `lame`, `flac`, `wavpac`, die `vorbis-tools` usw., die Sie natürlich auch verwenden könnten. Bevor ich nun mit Perl den Tools auf den Pelz rücke, mal etwas nur für die Windows-User.

1.2 Perl-Module speziell für Windows

Für Windows gibt es zwei sehr nette Module, die sich zum Abspielen von Musik und auch zum Erzeugen von Tönen eignen. Unabhängig davon, können natürlich auch die

Windows-Programmierer auf den `sox` zurückgreifen und die Programme der folgenden Abschnitte einsetzen. Eingesetzt wird im Folgenden das Modul `Win32::Sound`. Es war wohl ursprünglich erdacht worden, um die Windows-Systemklänge abzuspielen, und kann daher nur mit `.wav`-Dateien umgehen. Der Funktionsumfang des Moduls ist recht überschaubar:

Win32::Sound::Play() spielt eine `.wav`-Datei oder einen Systemklang ab. Es kann entweder eine Datei angegeben werden oder der Name eines Klangsignals. Es existieren die folgenden Namen: „SystemDefault“, „SystemAsterisk“, „SystemExclamation“, „SystemExit“, „SystemHand“, „SystemQuestion“ und „SystemStart“. Kann der angegebene Klang nicht gefunden werden, wird der Default-Klang des Systems gespielt. Ohne Parameter stoppt die Funktion die Soundausgabe.

Als zweiter Parameter kann noch ein Flag-Wert angegeben werden, der aus den folgenden Konstanten kombiniert werden kann.

SND_ASYNC spielt den Sound asynchron ab. Die Funktion kehrt sofort zurück, während es weiterdudelt. Per Default wartet die Funktion, bis das Stück zu Ende gespielt wurde.

SND_LOOP spielt den Sound in einer Endlosschleife. Will man die Ausgabe stoppen können, muss **SND_ASYNC** ebenfalls angegeben werden.

SND_NODEFAULT veranlasst, dass die Funktion geräuschlos endet, wenn der Sound oder die Datei nicht vorhanden sind.

SND_NOSTOP ergibt einen Fehler, wenn beim Aufruf der Funktion noch ein Sound gespielt wird. Per Default wird die alte Wiedergabe abgebrochen und der neue Sound gespielt.

Win32::Sound::Stop() beendet die (asynchrone) Wiedergabe.

Win32::Sound::Volume() liefert oder setzt die Lautstärke. Beim Aufruf ohne Parameter wird die eingestellte Lautstärke zurückgegeben. Dies kann entweder im Listenkontext geschehen (Werte in Prozent für linken und rechten Kanal) oder im skalaren Kontext. Hier wird ein 32-Bit-Wert zurückgegeben, dessen obere 16 Bits dem rechten und dessen untere 16 Bits dem linken Kanal entsprechen. Im Fehlerfall wird `undef` zurückgegeben.

Beim Aufruf mit ein oder zwei Parametern wird die Lautstärke gesetzt. Ist nur ein Parameter vorhanden, werden linker und rechter Kanal auf diesen Wert gesetzt. Im anderen Fall kommt erst der Wert für den linken und dann der Wert für den rechten Kanal. Die Werte können entweder in Prozent angegeben werden (als String: Zahl und %-Zeichen dahinter) oder als 16-Bit-Integerwert: 0 bis 65535).

Win32::Sound::Format(filename) liefert Informationen über die angegebene `.wav`-Datei in Form einer Liste: Sampling-Rate in Hz, Bits/Sample (8 oder 16) und Anzahl der Kanäle (1 oder 2).

Win32::Sound::Devices() sammelt alle erreichbaren Informationen über die Sound-Devices. Die Funktion liefert eine Liste der Device-Namen.

Win32::Sound::DeviceInfo() hat als Parameter einen der Device-Namen, die von `Win32::Sound::Devices()` geliefert wurden. Die Funktion liefert für dieses Device einen Hash zurück, der weitere Informationen enthält. Für alle Devices sind die Schlüssel `manufacturer_id`, `product_id`, `name` und `driver_version` definiert. Für einzelne Devices gibt es dann noch weitere Schlüssel (siehe Manualpage).

Höchste Zeit für ein Beispiel! Das folgende Programm demonstriert die Anwendung der oben besprochenen Funktionen:

```

use strict;
use warnings;
use Win32::Sound;

# Datei mit Musik ...
my $file = 'pomp.wav';

# Systemklang ausgeben
Win32::Sound::Play('SystemStart');

# Erst einmal Device-Infos einsammeln ...
my @devices = Win32::Sound::Devices();
# ... und ausgeben
for my $dev (@devices)
{
    my %info = Win32::Sound::DeviceInfo($dev);
    print "$dev\n";
    print "    Hersteller-Id: $info{manufacturer_id}\n";
    print "    Produkt-Id:    $info{product_id}\n";
    print "    $info{name}, Version $info{driver_version}\n";
}

# Was ist mit der Datei los?
my ($hz, $bit, $chan) = Win32::Sound::Format($file);
print "Sample rate: $hz Hz, $bit Bits, $chan Kanale\n";

# Eingestellte Lautstaerke
my ($l, $r) = Win32::Sound::Volume();
print "Aktuelle Lautstaerke L: $l, R: $r\n";

# Voll aufdrehen und losschmettern
Win32::Sound::Volume('100%', '100%');
Win32::Sound::Play($file);

# eigentlich unnoetig
Win32::Sound::Stop();

```

Das Programm liefert zur Musik die folgende Ausgabe:

```

WAVE_MAPPER
  Hersteller-Id: 1
  Produkt-Id:   2
  Microsoft Soundmapper, Version 5.0
WAVEOUT0
  Hersteller-Id: 1
  Produkt-Id:   100
  Realtek HD Audio rear output, Version 5.10
WAVEIN0
  Hersteller-Id: 65535
  Produkt-Id:   65535
  USB Audio Device, Version 1.0
WAVEIN1
  Hersteller-Id: 1
  Produkt-Id:   101
  Realtek HD Audio rear input, Version 5.10
MIDI_MAPPER
  Hersteller-Id: 1
  Produkt-Id:   1
  Microsoft MIDI-Mapper, Version 5.0
MIDIOUT0
  Hersteller-Id: 1
  Produkt-Id:   102
  Microsoft GS Wavetable SW Synth, Version 5.10
MIXER0
  Hersteller-Id: 1
  Produkt-Id:   104

```

```

Realtek HD Audio rear output, Version 5.10
MIXER1
  Hersteller-Id: 1
  Produkt-Id: 104
  Realtek HD Audio rear input, Version 5.10
  Sample rate: 22050 Hz, 16 Bits, 2 Kanäle
  Aktuelle Lautstärke L: 39, R: 39

```

Das Modul enthält aber noch ein völlig anderes Paket für Audiodaten, das Package `WaveOut`. Es besitzt Methoden zum Laden und Speichern von Audiodaten, wobei ein bestimmter Bereich ausgewählt werden kann. Es kann also genau das, was ich weiter oben mit `sox` demonstriert habe. Damit aber nicht genug, kann man auch beliebige Klänge generieren und zur Soundkarte schicken. Das Package versorgt uns mit folgenden Funktionen:

new Win32::Sound::WaveOut () erzeugt ein neues `WaveOut`-Objekt. Ein Aufruf ohne Parameter stellt die Samplingrate auf 44,1 kHz, die Auflösung auf 16 Bit Stereo. Man kann aber beim Aufruf drei Parameter mitgeben: Der erste Parameter ist die Samplingrate in Hz, der zweite die Auflösung (8 oder 16) und der dritte legt fest, ob es sich um Mono (1) oder Stereo (2) handelt. Alternativ kann als Parameter der Dateiname einer `.wav`-Datei angegeben werden. Es wird dann diese Datei wie bei `Open` geladen. Sie lässt sich dann sofort abspielen.

GetErrorText (ERROR) liefert den Fehlertext zur angegebenen Fehlernummer.

Load () lädt den als Parameter übergebenen Datenpuffer in die Soundkarte. Das Format des Puffers hängt vom definierten Soundformat ab: Bei „8 Bit Mono“ besteht jedes Sample aus einem einzigen Byte, bei „16 Bit Stereo“ sind es dagegen vier Bytes (zwei Bytes = 16 Bit für jeden Kanal). Bei beispielsweise 44,1 kHz Samplingrate und 16 Bit Stereo enthält der Puffer für eine Sekunde Sound $44100 * 4 = 176400$ Bytes.

Unload () löscht die geladenen Daten aus der Soundkarte.

Open () öffnet die angegebene `.wav`-Datei.

Close () schließt die geöffnete `.wav`-Datei.

OpenDevice () öffnet das Sound-Device mit den bei `new ()` spezifizierten Parametern (nur nötig, wenn `CloseDevice ()` verwendet wurde).

CloseDevice () schließt das gerade geöffnete Sound-Device. Es kann dann mit dem Aufruf von `OpenDevice` wieder geöffnet werden.

Pause () stoppt die Sound-Wiedergabe. Eine Fortsetzung erfolgt mit `Restart ()`.

Restart () setzt die mit `Pause ()` angehaltene Wiedergabe fort.

Reset () stoppt die Sound-Wiedergabe. Eine Fortsetzung mit `Restart ()` erfolgt dann wieder von Anfang an.

Play () spielt eine geöffnete `.wav`-Datei ab. Optional können zwei Parameter angegeben werden, die festlegen, ab welcher Stelle und bis zu welcher Stelle die Datei abgespielt werden soll. Die Angabe erfolgt in Sample-Schritten, hängt also von der Sampling-Rate ab. Der Dateianfang ist dabei 0. Will man beim zweiten Parameter das Dateiende angeben, nimmt man -1. Die Wiedergabe erfolgt wie auch bei `Write ()` im Hintergrund; das Programm läuft also weiter, während der Sound erklingt.

Write () spielt den Sound ab, der vorher mit `Load ()` in die Soundkarte geladen wurde. Auch hier erfolgt die Wiedergabe im Hintergrund.

Volume () setzt die Lautstärke. Die Funktion arbeitet genauso wie die entsprechende Funktion in `Win32::Sound`.

Status () gibt 0 zurück, solange die Soundkarte noch abspielt, und 1, wenn das Abspielen beendet ist. Im Fehlerfall wird `undef` retourniert.

Position () gibt die Nummer des Samples zurück, das beim Aufruf der Funktion gerade abgespielt wird. So kann festgestellt werden, wie lange die Ausgabe schon dauert. Der Wert wird nach Ende der Ausgabe nicht auf 0 gesetzt, weshalb nach jeder Wiedergabe ein `Reset ()`-Aufruf notwendig ist, wenn diese Funktion verwendet werden soll.

Save () schreibt den als zweiten Parameter übergebenen Datenpuffer in eine `.wav`-Datei, deren Name als erster Parameter angegeben wird. Wird nur der Dateiname angegeben, wird der aktuell in die Soundkarte geladene Datenpuffer abgespeichert.

Die Spezifikationen lassen sich auch ändern, ohne ein neues `WaveOut`-Objekt zu erzeugen, indem man die Funktionen `OpenDevice ()` und `CloseDevice ()` einsetzt:

```
# Parameter der Soundkarte umstellen
$WAV->CloseDevice();
$WAV->{samplerate} = 8000; # 8 kHz
$WAV->{bits}      = 8;    # 8 Bit
$WAV->{channels}  = 1;    # Mono
$WAV->OpenDevice();
```

Man kann auf die oben gezeigte Weise auch jederzeit die eingestellten Werte abfragen.

Das folgende Beispiel demonstriert die meisten Möglichkeiten des Pakets. Die Funktion `tone` erzeugt einen Sinuston der angegebenen Frequenz und Dauer. Wenn, wie im Beispiel, 8 Bit Stereo verwendet wird, müssen jeweils immer zwei Bytes je Sample gespeichert werden, was in der `pack ()`-Funktion erledigt wird. Im Hauptprogramm wird dann zunächst ein 880-Hz-Ton von einer Sekunde Dauer erzeugt, abgespielt und in einer Datei gespeichert. Danach wird das nervigste Jingle der deutschen Fernsehwerbung abgespielt und schließlich ein kontinuierlich ansteigender Ton zwischen 200 Hz und 4 kHz erzeugt.

```
use strict;
use warnings;
use Win32::Sound;

# WaveOut-Objekt anlegen Samplingrate: 44100, 8 Bit, 2 Kanäle
my $wavsound = new Win32::Sound::WaveOut(44100, 8, 2);

# halbvoll aufdrehen
$wavsound->Volume('50%');

# 880-Hz-Ton, 1 s Dauer "bauen"
my $dat = tone(880,1);
$wavsound->Load($dat);          # Rohdaten --> wav
$wavsound->Write();             # anhoeren
1 while(! $wavsound->Status()); # warten, bis fertig
$wavsound->Save("sinus.wav");   # in Datei schreiben
$wavsound->Unload();           # wegwerfen

# Jingle
$dat = tone(1244,0.2);
$dat .= tone(1244,0.2);
$dat .= tone(1244,0.2);
$dat .= tone(1760,0.2);
$dat .= tone(1244,0.2);
$wavsound->Load($dat);          # Rohdaten --> wav
$wavsound->Write();             # anhoeren
```

```

1 while(! $wavsound->Status()); # warten, bis fertig
$wavsound->Unload();           # wegwerfen

print "Moment bitte ...\n";
# 200 Hz - 4000 Hz ansteigender Ton
$dat = '';
my $k;
for ($k = 200; $k <= 4000; $k += 20)
{
    $dat .= tone($k,0.4);
}
$wavsound->Load($dat);         # Rohdaten --> wav
$wavsound->Write();           # anhören
1 while(! $wavsound->Status()); # warten, bis fertig
$wavsound->Unload();           # wegwerfen

sub tone # ($frequency, $duration)
{
    my ($freq, $duration)= @_;
    my $pi = 3.14159265358979;
    my $data = "";           # Speicher fuer die Sounddaten
    my $value = 0;           # aktueller Wert
    my $step = $freq/44100;  # Increment = Frequenz/Samplingrate

    # 44100 Samples/s generieren
    for my $i (1 .. $duration*44100)
    {
        # Sinuskurve berechnen, Wertebereich 0-255 fuer 8 Bit
        # Nulllinie bei 128
        my $v = int(sin($value/2.0*$pi)*128 + 128);
        # zweimal packen fuer linken und rechten Kanal
        $data .= pack("CC", $v & 255, $v & 255);
        $value += $step;
    }
    return($data);
}

```

Falls Sie lieber Noten anstelle von Frequenzen verwenden wollen, hilft die folgende Tabelle. Sie ordnet den Noten für jede Oktave die entsprechenden Frequenzen zu. Um eine Note „x“ der Oktave „y“ auszugeben, setzen Sie die Frequenz = \$FC{\$x}[\$y].

```

# Notentabelle (b --> h,  as --> #a)
%FC = (
    'a', [55,110,220,440,880,1760],
    'as', [58.270,116.541,233.082,466.164,932.328,1864.655],
    'b', [61.735,123.471,246.942,493.883,987.767,1975.533],
    'c', [65.406,130.813,261.626,523.251,1046.502,2093.005],
    'cs', [69.296,138.591,277.183,554.365,1108.731,2217.461],
    'd', [73.416,146.832,293.665,587.330,1174.659,2349.318],
    'ds', [77.782,155.563,311.127,622.254,1244.508,2489.016],
    'e', [82.407,164.814,329.628,659.255,1318.510,2637.020],
    'f', [87.307,174.614,349.228,698.456,1396.913,2793.826],
    'fs', [92.499,184.997,369.994,739.989,1479.978,2959.955],
    'g', [97.999,195.998,391.995,783.991,1567.982,3135.963],
    'gs', [103.826,207.652,415.305,830.609,1661.219,3322.438],
    'p', [0,0,0,0,0,0]
);

```

Noch ein abschließendes Beispiel, in dem das Sound-Modul mit Perl-Tk kombiniert wird. Die grafische Oberfläche (Bild 1.2) hat neben dem obligatorischen „EXIT“ vier Buttons. Für jeden Button gibt es eine ganz einfache Callback-Funktion:

play spielt den eingestellten Sound ab,

stop beendet die Ausgabe,

start timer sorgt dafür, dass der Sound „dog.wav“ alle 5 Sekunden abgespielt wird,

stop timer beendet die Geräuschbelästigung.



Bild 1.2: Soundsteuerung mit grafischer Oberfläche

```

use strict;
use warnings;
use Tk;
use Win32::Sound;

# Musikdatei aus der Kommandozeile
my $sound = $ARGV[0];

# Delay-Zeit Timer im ms
my $delaytime = 5000;

my $timer_id;

# GUI-Fenster erzeugen
my $mw = new MainWindow();

# Buttonzs auf dem GUI-Fenster platzieren
my %row1 = (-side=>'left', -pady=>10, -padx=>10);
my %row2 = (-side=>'bottom', -pady=>10, -padx=>10);

my $b5=$mw->Button(-text => "EXIT",
                  -command => sub {exit})->pack(%row2);
my $b1=$mw->Button(-text => "play",
                  -command => \&start_sound)->pack(%row1);
my $b2=$mw->Button(-text => "stop",
                  -command => \&stop_sound)->pack(%row1);
my $b3=$mw->Button(-text => "start timer",
                  -command => \&timer_on)->pack(%row1);
my $b4=$mw->Button(-text => "stop timer",
                  -command => \&timer_off)->pack(%row1);
MainLoop();

# Callback zum Abspielen der Sounddatei
sub start_sound()
{
    Win32::Sound::Volume('100%');
    Win32::Sound::Play($sound, SND_ASYNC);
}

# Callback macht "wuff"
sub wuff()
{
    Win32::Sound::Volume('100%');
    Win32::Sound::Play("dog.wav", SND_ASYNC);
}

```

```

# Ton aus
sub stop_sound()
{ Win32::Sound::Stop(); }

# Timer starten
sub timer_on()
{
  if (! defined $timer_id)
  { $timer_id = $mw->repeat($delaytime, \&wuff); }
}

# Timer beenden
sub timer_off()
{
  if (defined $timer_id)
  {
    $timer_id->cancel;
    undef($timer_id);
  }
}

```

Ein lustiger Effekt ergibt sich, wenn Sie eine längere Datei abspielen und gleichzeitig den Timer starten. Dann macht es nach fünf Sekunden „wuff“ und Stille kehrt ein.

Wenn Sie statt des Timers eine Dateiauswahl hinzufügen, haben Sie schon einen einfachen .wav-Player mit GUI.

Soviel zur Windows-Variante der .wav-Bearbeitung. Unter Linux und Unix bietet das Modul `Audio::Wav fast` die gleiche Funktionalität, weshalb an dieser Stelle nur auf die entsprechenden Manualpages verwiesen wird.

Zum Konvertieren von Sounddateien im kleinen Rahmen kann das `Audio::SndFile`-Modul eingesetzt werden. Es gibt noch etliche weitere Sound-Module, die zum Teil aber nur sogenannte Wrapper um externe Programme herum darstellen.

1.3 Sound mit dem internen Lautsprecher

Ja, Sie haben richtig gelesen – jeder PC besitzt noch ein Relikt aus den frühen Zeiten, einen internen Lautsprecher. Manchmal ist er schon zu einem kleinen Piezo-Teil degeneriert, aber es gibt ihn noch. Bei Servern, die sich in 19-Zoll-Gehäusen befinden und nur in entsprechenden Schaltschränken wohl fühlen, ist er auch die einzige Möglichkeit, akustisch Aufmerksamkeit zu erregen. Meist nimmt man seine Existenz erst wahr, wenn aufgrund eines Hardwarefehlers der Bildschirm dunkel bleibt und aus dem PC-Gehäuse hilfloses Gepiepse dringt.

Das Modul `Audio::Beep` erlaubt die Nutzung des Lautsprechers auf etwas intelligentere Art und Weise. Er kann natürlich keine Musik spielen, aber zumindest Töne von sich geben.⁴ Das Modul bietet sogar neben der funktionalen eine objektorientierte Schnittstelle.

Als Funktion bietet das Modul nur `beep()`, eine Funktion mit zwei Parametern: zuerst die Frequenz in Hertz und dann die Dauer in Millisekunden. Das war’s dann auch schon. Im folgenden Beispiel wird der Jingle von oben noch einmal aufgegriffen:

```

use strict;
use warnings;
use Audio::Beep;

beep (1244, 220);

```

⁴Im PC-Mittelalter gab es für Windows 3.1 einen Sound-Treiber für den internen Lautsprecher, der sogar .wav-Dateien abspielen konnten – HiFi war das aber wirklich nicht.

```
beep (1244, 220);
beep (1244, 220);
beep (1760, 220);
beep (1244, 220);
```

Ruft man die Funktion ohne Parameter auf, werden die Default-Werte 440 Hz und 100 ms verwendet. Zufällige Geräusche kann man mit den folgenden beiden Zeilen produzieren:

```
use Audio::Beep;
beep(50 + rand 1000, rand 300) while 1;
```

Bei den Objektmethoden geht es nicht ganz so spartanisch zu. Die Methode `new()` erzeugt ein neues `Audio::Beeper`-Objekt, das die folgenden Optionen besitzt:

player erlaubt die Bindung an einen bestimmten Audio-Player.

rest legt die Pause zwischen zwei Tönen (in ms) fest.

Diese beiden Optionen sind auch auf Methoden abgebildet und erlauben auch später noch deren Einstellung.

Die Methode `play()` ermöglicht sogar einen Hauch von Musik, denn sie hat als Parameter einen String, der keine Frequenzen, sondern Noten nach der Lilypond-Syntax enthält (siehe <http://lilypond.org>). Wenn nichts anderes angegeben wurde, wird die mittlere Oktave mit Viertelnoten verwendet. Jede Notenangabe folgt der Struktur:

```
[Note] [Vorzeichen] [Oktave] [Dauer] [Punkte]
```

Die Note wird durch einen Buchstaben aus der Menge [c d e f g a b] repräsentiert. Hinzu kommt noch das „r“ für eine Pause. Als Vorzeichen dienen das „#“ („c#“ wird zu cis) und das „s“ für das „b“.

Ein Apostroph (‘) hebt den Ton um eine Oktave an, ein Komma senkt ihn um eine Oktave.

Die Dauer wird durch eine Zahl repräsentiert, die 1 entspricht einer ganzen Note; höhere Zahlen machen die Dauer entsprechend kürzer.

Durch Hinzufügen von Punkten kann die Hälfte der definierten Länge zur Dauer hinzugefügt werden, z. B. ist „d4.“ ein D, das $1/4 + 1/8$ Note dauert.

Mit dem Spezialbefehl `\bpm` kann die Grundgeschwindigkeit (beats per minute) eingestellt werden, z. B. `\bpm120`.

Das folgende Beispiel erzeugt einige leicht melodische Geräusche, aber erwarten Sie nicht zuviel.

```
use strict;
use warnings;
use Audio::Beep;

my $beeper = Audio::Beep->new();

my $music = "\bpm144 c d e f g a b c2. r4 c b a g f e d c1 ";
$beeper->play($music);

$music = qq~g' f bes' c8 f d4 c8 f d4 bes c g f2
          g' f bes' c8 f d4 c8 f d4 bes c g f2~;
$beeper->play($music);
```

Wie schon erwähnt, ist `Audio::Beep` für viele Server die einzige Möglichkeit, sich bemerkbar zu machen, da in der Regel Tastatur und Bildschirm fehlen.

1.4 Perl und Sox

In diesem Abschnitt geht es um die Kopplung von Perl und dem Universal-Tool `sox`. Insbesondere unter Linux und Unix wird es gern eingesetzt, um die eigentliche Arbeit zu verrichten, und das Perl-Programm drumrum macht die Bedienung leichter und erspart das Nachschlagen von zahllosen Optionen. Das erste Beispiel demonstriert die Vorgehensweise. Und weil es so schön ist, nehme ich wieder das Jingle von oben. Zur Abwechslung mache ich die Töne etwas tiefer und erzeuge wieder eine `.wav`-Datei.

Das Programm läuft auch unter Window (so wie auch alle folgenden), lediglich der Pfad zum Programm `sox` muss entsprechend angegeben werden. Im Listing finden Sie beide Varianten:

```
use strict;
use warnings;

my $SOX = '/usr/local/bin/sox';      # Linux
# my $SOX = 'C:\Programme\sox\sox.exe';  # Windows

# die fuenf Toene erzeugen
system("$SOX -U -r 8000 -n -t raw - synth 0.2 sine 622 gain -3 > t.ul");
system("$SOX -U -r 8000 -n -t raw - synth 0.2 sine 622 gain -3 >> t.ul");
system("$SOX -U -r 8000 -n -t raw - synth 0.2 sine 622 gain -3 >> t.ul");
system("$SOX -U -r 8000 -n -t raw - synth 0.2 sine 880 gain -3 >> t.ul");
system("$SOX -U -r 8000 -n -t raw - synth 0.2 sine 622 gain -3 >> t.ul");

# Raw-Sound als .wav-Datei speichern
system("$SOX -c 1 -r 8000 t.ul jingle.wav");
# Raw-Dateien entsorgen
unlink ("t.ul");

# Laerm machen
system("$SOX jingle.wav -d");
```

Wie Sie bemerkt haben, kann `sox` das Erzeugnis auch gleich ausgeben. Das nachgestellte „-d“ sorgt für die Ausgabe auf dem Standard-Sounddevice.

Einen Nachteil gegenüber dem Windows-Programm vom vorletzten Abschnitt ist, dass der Sound nicht im Hintergrund abgespielt wird, sondern man warten muss, bis der `system()`-Aufruf beendet ist.

Dem kann aber abgeholfen werden, wenn man die Sound-Ausgabe auf einen Kindprozess verlagert. Der Kindprozess spielt die Sounddatei ab, während der Elternprozess weiterläuft. Mittels `kill()` kann man die Soundwiedergabe auch vom Elternprozess aus abbrechen.

```
use strict;
use warnings;

my $SOX = '/usr/local/bin/sox';

my $pid = playsound("jingle.wav"); # Sound spielen
waitpid($pid, 0);                 # auf das Ende warten

sub playsound
{
    my $file = shift;
    my $child_pid;
    $child_pid = fork;
    if ($child_pid > 0)
    { # Parent
        return $child_pid;
    }
}
```

```

}
elseif ($schild_pid == 0)
{ # Child
my @play = ($SOX, $file, "-d");
# Kind ueberlagert sich mit sox
exec(@play);
exit(0); # wenn exec nicht ging
}
else
{ die "Cannot fork: $!\n"; }
}

```

Neben vielen Effekten beherrscht `sox` auch das Synthetisieren von Sounds. Das folgende Programm erzeugt für Messzwecke eine `.wav`-Datei mit verschiedenen Formen von Rauschsignalen (sogenanntes weißes Rauschen). Es werden zunächst bis zu 32 Dateien mit Rauschsignalen erzeugt, bei denen Dauer, Mittenfrequenz und Bandbreite innerhalb vorgegebenen Grenzen zufällig sind. Per Kommandozeile lassen sich die Anzahl der Dateien und jeweils die Ober- und Untergrenzen von Dauer, Mittenfrequenz und Bandbreite wählen.

```

use strict;
use warnings;

# my $SOX = '/usr/local/bin/sox';      # Linux
my $SOX = 'C:\Programme\sox\sox.exe';  # Windows

# Temporaerverzeichnis angeben
my $tmp_dir = 'tmpnoise';
mkdir($tmp_dir) if(! -e $tmp_dir);

# Hilfe ausgeben, falls gewünscht
usage() if($ARGV[0] =~ /[h?]/);

# Dateinamen von der Kommandozeile holen
my $outfile = shift;
unless (defined $outfile)
{
    print "Keine Ausgabedatei angegeben\n";
    usage();
}

# Defaultwerte einstellen
my $segs = shift || 32;          # Anzahl Dateien (max. 32)
my $min_dur = shift || 0.25;    # Minstdauer
my $max_dur = shift || 1.0;     # Maximaldauer
my $min_center = shift || 500;  # Mittenfrequenz Untergrenze
my $max_center = shift || 8000; # Mittenfrequenz Obergrenze
my $min_bw = shift || 100;     # Bandbreite Untergrenze
my $max_bw = shift || 1000;    # Bandbreite Obergrenze

# Speicher für die Namen der erzeugten Dateien
my $filelist = '';

# Segmente erzeugen
for my $i (1..$segs)
{
    # Zufallswerte erzeugen
    my $dur = rand($max_dur) + $min_dur;
    my $center = int(rand($max_center - $min_center)) + $min_center;
    my $bw = int(rand($max_bw - $min_bw)) + $min_bw;
    # Dateinamen erzeugen ...
    my $segfile = "$tmp_dir/temp$i.wav";
    # ... und in der Liste speichern
    $filelist .= "$segfile ";
    # Sounddatei generieren
}

```

```

system("$SOX -n $segfile synth $dur noise bandpass $center $bw");
}

# Alle Segmente zusammenmischen
system("$SOX -m $filelist $outfile");

sub usage
{
    print "Usage: $0 <outfile> <segs (<=32)> <min dur> <max dur> " .
          "<min cent> <max cent> <min bw> <max bw>\n";
    exit;
}

```

Übrigens liefert das Programm `soxi` oder auch der Aufruf

```
sox sounddatei -n stat stats
```

alle erdenklichen Informationen über die Sounddatei, wie das folgende Listing zeigt. Man kann diese Infos verwenden, um Sounds automatisch an bestimmte Gegebenheiten anzupassen. So könnten man beispielsweise durch Auswerten der maximalen Amplitude mehrere Dateien auf etwa gleiche Lautstärke bringen.

```

$ sox pomp.wav -n stat stats

Samples read:          3712848
Length (seconds):     84.191565
Scaled by:            2147483647.0
Maximum amplitude:    0.842896
Minimum amplitude:    -0.805786
Midline amplitude:    0.018555
Mean norm:            0.157652
Mean amplitude:       -0.000039
RMS amplitude:        0.207150
Maximum delta:        1.454651
Minimum delta:        0.000000
Mean delta:           0.178555
RMS delta:            0.234321
Rough frequency:      3969
Volume adjustment:    1.186

          Overall      Left      Right
DC offset  -0.000039 -0.000039 -0.000038
Min level  -0.805786 -0.805786 -0.801483
Max level   0.842896  0.842896  0.806793
Pk lev dB   -1.48     -1.48     -1.86
RMS lev dB  -13.67    -13.35    -14.03
RMS Pk dB   -7.17     -7.17     -7.70
RMS Tr dB   -143.98  -143.98   -140.73
Crest factor    -      3.92     4.06
Flat factor    0.00     0.00     0.00
Pk count       2        2         2
Bit-depth      16/16    16/16    16/16
Num samples    1.86M
Length s       84.192
Scale max      1.000000
Window s       0.050

```

Beim Seefunk wird einem Notruf eine 30 Sekunden dauernde Alarmsequenz vorangestellt, damit die nachfolgende Meldung auch die nötige Aufmerksamkeit erlangt. Diese Alarmsequenz besteht aus zwei Tönen von 1300 und 2100 Hz, die sich mit einer Rate von 4 Hz abwechseln. Das folgende Programm erzeugt eine .wav-Datei mit dieser Alarmsequenz.

Die Töne werden mit einer Rate von 8 kHz in zwei temporäre Dateien im Raw-Format generiert. Sie dauern jeweils 0,25 Sekunden, was der Wechselfrequenz von 4 Hz entspricht. Diese Töne werden nun mehrfach hintereinander in eine weitere Datei kopiert. Das funktioniert einwandfrei, sodass die Dateien nur Sound-Samples und keinerlei Verwaltungsinfo enthalten. Zum Schluss generiert das Programm dann eine .wav-Datei aus diesen Daten.

```
use strict;
use warnings;

my $SOX = '/usr/local/bin/sox';          # Linux
# my $SOX = 'C:\Programme\sox\sox.exe';  # Windows

# die beiden Toene erzeugen
system("$SOX -U -r 8000 -n -t raw - synth 0.25 sine 1300 gain -3 > t.ul");
system("$SOX -U -r 8000 -n -t raw - synth 0.25 sine 2100 gain -3 >> t.ul");

# Datei t.ul 60 mal auf alert.ul kopieren
my $buffer;
open (OUT, '>', "alert.ul");
for my $i (1 .. 60)
{
    open (IN, "t.ul");
    print OUT $buffer while
        (read (IN, $buffer, 16384));
}
close(OUT);
close(IN);

# Raw-Sound als .wav-Datei speichern
system("$SOX -c 1 -r 8000 alert.ul alert.wav");
# Raw-Dateien entsorgen
unlink ("t.ul");
unlink ("alert.ul");
```

Auch wenn ich bisher nur an der Oberfläche von `sox` gekratzt habe, ist wohl deutlich geworden, wie man mit diesem Programm arbeitet. Die Perl-Unterstützung sorgt dafür, dass man einerseits mit den vielen Parametern klarkommt, indem man die komplizierten Aufrufe von `sox` in Perl-Programmen versteckt. Andererseits lassen sich die Parameter von `sox` mit Perl-Programmen berechnen und vorbereiten. Damit bietet sich die Möglichkeit, eine komfortable – gegebenenfalls auch grafische – Benutzerschnittstelle zu diesem Programm zu schaffen.

1.5 Weitere Sound-Module

Ich will nun nur noch zwei Module kurz anreißen. Das erste Modul beschäftigt sich mit MP3-Musikdateien: `MP3::Info` erlaubt das Auslesen und Verändern der MP3-Infos, die mit in den Dateien gespeichert sind. Ich lasse einfach das folgende Beispiel für sich selbst sprechen. Die Funktionen `get_mp3tag()` und `get_mp3info()` liefern alle in der Datei gespeicherten Infos. Ergebnisvariable ist jeweils eine Hash-Referenz:

```
use strict;
use warnings;
use MP3::Info;

my $file = 'PompAndCircumstance.mp3';

my $tag = get_mp3tag($file) or die "No TAG info";

foreach my $key (keys(%{$tag}))
```

```

    { printf("%-15s %s\n", $key, $tag->{$key}); }

my $info = get_mp3info($file);
print "\n";

foreach my $key(keys(%{$info}))
    { printf("%-15s %s\n", $key, $info->{$key}); }

```

Nach dem Programmlauf erhalte ich eine lange Liste mit Informationen:

```

YEAR
ARTIST          no artist
COMMENT
TITLE           PompAndCircumstance
ALBUM          no title
GENRE          Unknown
TRACKNUM       3
TAGVERSION     ID3v1.1 / ID3v2.3.0

SIZE           9703601
OFFSET        1715
MS            316.708333333281
STEREO        1
SECS          404.316708333333
PADDING       0
LAME          HASH(0x191eb94)
MM            6
COPYRIGHT     0
SS            44
LAYER         3
MODE          0
FREQUENCY     44.1
VBR           0
TIME          06:44
FRAMES        15477
BITRATE       192
VERSION       1
FRAME_LENGTH  626

```

Die Funktion `set_mp3tag()` erlaubt das Ändern der Taginformation, wobei man auf das 30-Byte-Limit bei allen Tags achten muss. Ich kann beispielsweise das oben als „unbekannt“ gemeldete Aufnahmedatum setzen:

```

$tag->{YEAR} = '2001';
set_mp3tag($file, $tag);

```

Analog lassen sich alle Informationen verändern oder mittels `remove_mp3tag()` komplett entfernen. Weitere Einzelheiten entnehmen Sie bitte den Manualpages des Moduls.

Das Modul `Audio::DSP` ermöglicht einen direkten Zugriff auf die Soundkarte. Er stützt sich direkt auf die OSS-API und ist dementsprechend hardwarenah. Mit diesem Modul kann man nicht nur direkt Samples in die Soundkarte transferieren, sondern auch Daten abrufen, also Sounds aufnehmen. Da die Soundkarte im Grunde nichts weiter als ein Analog-Digital-Wandler ist, eröffnen sich auch andere Möglichkeiten der Nutzung. Solange das Eingangssignal im Aussteuerungsbereich bleibt, kann man beliebige analoge Signale erfassen.

Die folgende Funktion würde von der Soundkarte Daten mit einer Samplingrate von 8 kHz und einer Auflösung von 8 Bit Mono einlesen. Da der Puffer 80000 Bytes groß ist, dauert die Aufnahme 10 Sekunden, bis der Puffer voll ist. Die Daten werden dann zur weiteren Verarbeitung an eine (fiktive) Funktion `store()` übergeben).

```

sub record
{
  my $dsp = new Audio::DSP (
    buffer    => 80000, # Puffergroesse
    channels => 1,      # Kanale (1 oder 2)
    format   => 8,      # Aufloesung (8 oder 16 Bit)
    rate     => 8000,   # Samplingrate
  );

  $dsp->init() or die $dsp->errstr();
  $dsp->read() or die $dsp->errstr();
  store($dsp->data());
  $dsp->clear();
}

```

Mit der Methode `dsp_write()` lassen sich die aufgezeichneten Daten wiedergeben. Für die Berechnung der Puffergröße kann man nach der folgenden Formel vorgehen:

$$\text{Groesse} = \frac{\text{Kanale} \cdot \text{Format} \cdot \text{Samplingrate}}{8}$$

Selbstverständlich kann man `dsp_read()` auch mehrmals hintereinander aufrufen. Bei jedem Aufruf werden die Daten an den Puffer im Speicher angehängt. Der Parameter `buffer` legt lediglich fest, wie viele Daten bei einem Aufruf der Methode gesampled werden.

Weitere Methoden unterstützen sogar noch hardwarenäheres Lesen und Schreiben der Soundkarte, beispielsweise für Harddisk-Recording. Die Daten liegen im raw-Format vor und können mit den schon bekannten Tools weiterbearbeitet werden. Mit diesem Ausflug zu einem hardwarenahen Modul will ich auch das Sound-Kapitel schließen.

Auf der Netzmafia-Webseite finden Sie noch eine Sammlung kleiner Sounddateien für Signalzwecke und ein Referenzhandbuch für den `sox`.

Auf der Webseite der Firma Werma, die Signalsysteme herstellt, können Sie sich alle möglichen (offiziellen) Signaltöne anhören und herunterladen (links den Menüpunkt „Signaltöne“ wählen). Da hören Sie dann auch den Unterschied der Alarmsignale aus aller Welt: <http://www.werma.de>.