Floating-Point-Zahlen in digitaler Audioanwendung

Facharbeit von Daniel Mack

SAE Berlin, AEDS603

4. Dezember 2004

Inhaltsverzeichnis

1. Einleitung

Diese Facharbeit beschäftigt sich mit der Verwendung von Floating-Point-Zahlen (einer speziellen Form der Zahlendarstellung in Computersystemen) in der digitalen Audiowelt - ein Thema, das in der klassischen Audio-Fachliteratur noch weitgehend unbeleuchtet ist.

Es wird zunächst ein Überblick über die Bereiche vermittelt, in denen diese Technologie bereits eingesetzt wird.

Vorteile und Probleme werden aufgezeigt, die durch die Verwendung von Floating-Point entstehen können - auch und gerade im Vergleich zum diskret ganzzahligen Zahlenformat.

Im weiteren Verlauf wird anhand einiger theoretischer Skizzen gezeigt, wie die zukünftige Weiterentwicklung dieser Technologie - speziell neue Wandlerkonzepte – aussehen könnte.

Anhand kurzer, in der Programmiersprache C geschriebenen Programme, können die gewonnenen theoretischen Erkenntnisse anschaulich in der Praxis demonstriert und überprüft werden.

Zum Verständnis der Thematik Floating-Point ist vorab ein etwas längerer theoretischer Teil notwendig, der aus einem Standard des IEEE-Konsortiums zitiert.

2. Zahlenformate

2.1 Das Binärsystem

Um Zahlen von Computersystemen les- und interpretierbar zu machen, wurde in den Anfangszeiten der digitalen Datenverarbeitung ein System entwickelt, das Zahlen mithilfe zweier digitaler Zustände (1 und 0) darstellen kann. Dieses System, das Binär- bzw. Dualsystem genannt wird, kodiert pro Stelle i, die man auch "Bit" nennt, einen Wert von 2^i. Es wird verwendet, um Zahlenwerte darzustellen, die entweder als signed (mit Vorzeichen) oder als unsigned (ohne Vorzeichen) interpretiert werden und in Abhängigkeit ihrer Bittiefe folgende Wertebereiche abbilden können:

signed/unsigned	Bits	Wertebereich
signed	8	-128 bis 127
unsigned	8	0 bis 255
signed	16	-32768 bis 32767
unsigned	16	0 bis 65535
signed	32	-2147483648 bis 2147483647
unsigned	32	0 bis 4294967296
signed	n	-2^{n-1} bis $2^{n-1}-1$
unsigned	n	0 bis 2^n-1

Solche ganzzahligen Werte werden im Folgenden "Integerzahlen" genannt.

Um Werte zwischen zwei ganzen Zahlen darstellen zu können, kann das so genannte Fixed-Point-Binärsystem verwendet werden, das analog zur Dezimalschreibweise ein Komma in eine Binärzahl einfügt. Der Exponent der Zahl 2 wird dabei, wie im Dezimalsystem der Exponent der Zahl 10, negativ. Die erste Stelle, die nach diesem Komma steht (im Dezimalsystem hat sie eine Wertigkeit von 10^{-1} (also 0,1)), wertet im Binärsystem entsprechend 2^{-1} (also 1/2).

Die Zahl $0{,}011_{\text{binär}}$ lässt sich also beispielsweise ausdrücken als

$$0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0 + 0 + (1/4) + (1/8) = 0{,}375_{\text{dezimal}}$$

Bei der Kodierung der in diesem Beispiel verwendeten Zahl muss allerdings bekannt sein, an welcher Stelle im Bitstream das Komma zu setzen ist. Da sich diese Information nicht ohne Weiteres in einem Binärcode verpacken lässt, wurde mit Floating-Point ein Datenformat entwickelt, das darüber eine Aussage trifft und zudem noch eine Menge anderer Möglichkeiten mit sich bringt.

2.2 Grundlagen von Floating-Point-Zahlendarstellung

Konrad Zuse entwickelte im Jahr 1936 das Prinzip der Floating-Point-Arithmetik (zu deutsch auch Fließkommazahl, Gleitkommazahl oder binäre Gleitkommazahl genannt).

Diese basiert auf der Idee, Computerzahlen in der allgemeinen Exponentenschreibweise

$$a = m \cdot b^e$$

darzustellen, die Information also in folgende drei Teile aufzuspalten:

- Faktor m, die so genannte Mantisse, auch Signifikant genannt
- Basis b
- Exponent e

Alle drei Werte können negativ sein.

Nachfolgend ein paar Beispiele für die Darstellung von Zahlen in dieser technischen Exponentenschreibweise:

143.210.000	$= 1{,}4321 \cdot 10^8$
0,00000014321	$= 1{,}4321 \cdot 10^{-7}$
-143,21	$= -1{,}4321 \cdot 10^2$
1,4321	$= 1{,}4321 \cdot 10^0$

Die Einsparung von ausgeschriebenen Nachkommastellen oder endlos langen Ketten von Nullen am Ende einer Zahl, die sich in der Wissenschaft als äußerst praktisch erweist, kommt natürlich auch der elektronischen Datenverarbeitung zugute, da der zur Verfügung stehende Speicherplatz für die Speicherung der wesentlichen Komponenten genutzt werden kann.

Konsequenterweise sollte daher die Mantisse in einer normierten Form angegeben werden, die nur eine Stelle vor dem Komma aufweist, die außerdem ungleich Null ist. Das ist einfach möglich, indem der Exponent derart skaliert wird, dass die Mantisse bis zum höchstwertigen Bit gefüllt ist.

2.3 IEEE 754

Das in dieser Arbeit vorgestellte Floating-Point-System, eine spezielle Form der generischen Floating-Point-Darstellung, ist heute vom IEEE-Konsortium als Standard unter dem Namen

IEEE754-1985 mit dem Typ “single” festgeschrieben und zeichnet sich durch folgende Eigenschaften aus:

- Es werden 32 Bits zur Darstellung des gesamten Konstrukts verwendet
- Die Basis b ist auf 2 festgelegt
- Es werden 23 Bit für die Mantisse, 8 Bit für den Exponenten und ein Bit für das Vorzeichen der Mantisse verwendet
- Die Mantisse wird binär kodiert, es sind also nur die Ziffern 0 und 1 zulässig
- Es wird eine Standarddarstellung der Mantisse gewählt, durch die die Position des Kommas implizit an der ersten Stelle bekannt ist
- Der Exponent wird, um auch negativ sein zu können, mit einem Bias von 127 gespeichert (siehe unten)

Durch die Festlegung auf die Basis 2 und die binäre Kodierung der Mantisse kann man also von einer wählbaren Position des Kommas innerhalb der Zahl sprechen, eben von einem gleitenden Komma oder einem “floating point”.

Das Layout des verwendeten Datenwortes, also die Verwendung der 32 Bits, ist wie folgt definiert:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
S	Exponent								Mantisse																						

Das MSB („most significant bit“), Bit 31, zeigt eine negative Mantisse an, die Bits 30 – 23 werden zur Speicherung des Exponenten genutzt und die restlichen Bits (22 - 0) beinhalten den Wert der Mantisse.

Genau wie die Mantisse kann auch der Exponent negativ sein. Beim IEEE 754 wird dafür jedoch nicht eine Darstellung im Zweierkomplement gewählt, sondern die so genannte Biased-Darstellung, bei der zum eigentlichen Exponenten eine festgelegte Zahl, der Bias addiert wird. Beim IEEE 754-single beträgt dieser Bias 127 und bewirkt, dass vom eigentlich abgespeicherten Wert 127 subtrahiert werden muss, um den „echten“ Exponentenwert zu erhalten. Das hat den entscheidenden Vorteil, dass ein Vergleich zwischen zwei Floating-Point-Zahlen auf einen lexikographischen Vergleich des Exponenten und der Mantisse beschränkt bleibt, während eine echte Subtraktion wesentlich rechenintensiver wäre. Der Nachteil dieser Darstellung ist natürlich, dass bei der Addition zweier Biaswerte eine

Subtraktion des Bias vorgenommen werden muss.

Wird die Mantisse so gewählt, dass an der Stelle vor dem Komma eine 1 steht, so spricht IEEE 754 von einer normalisierten Floating-Point-Zahl, die dadurch spezifiziert ist, dass sie einen von Null verschiedenen Exponenten hat.

Ebenfalls eine Besonderheit stellt das “Hidden-Bit” dar, das von der Tatsache Gebrauch macht, dass die erste Stelle einer normalisierten Mantisse nie Null sein kann und daher im Binärsystem immer 1 ist. Daher muss diese Information nicht explizit gespeichert werden. Da mit dieser Regelung die Null nun nicht mehr direkt als Gleitkommazahl gespeichert werden kann, wird hierfür eine Spezialnotation gewählt.

Die folgenden Fälle definiert der IEEE Standard 754:

Exponent	Mantisse	Bedeutung
$111...111_{binär}$	$000...000_{binär}$	+/- unendlich („+/- inf“)
$111...111_{binär}$	$\neq 000...000_{binär}$	ungültig (“Not a number”, „NaN“)
$000...000_{binär}$	$000...000_{binär}$	+/- 0 (Null)
$000...000_{binär}$	$\neq 000...000_{binär}$	denormalisierte Gleitkommazahl
$000...001_{binär}$ bis $111...110_{binär}$	beliebig	normalisierte Gleitkommazahl

Der Wert einer Gleitkommazahl, die im Bereich einer “normalisierten” liegt, berechnet sich wie folgt:

$$a = (-1)^S \cdot (1,\text{Mantisse})_{binär} \cdot 2^{\text{Exponent}}$$

Ist eine Zahl zu klein, um in normalisierter Form mit dem kleinsten, von Null verschiedenen Exponenten gespeichert zu werden, so wird sie als so genannte “denormalisierte Zahl” gespeichert. Bei einer solchen Zahl wird von der Mantisse dann 1 subtrahiert und der Exponent auf den kleinsten möglichen Wert gesetzt. Der Sinn dieser Darstellung ist die Schließung der Lücke zwischen der kleinsten normalisiert darstellbaren zahlen und der Null. In diesem Fall gilt folgende Regel:

$$a = (-1)^S \cdot (0,\text{Mantisse})_{binär} \cdot 2^{-126}$$

Mit “Null” wird sowohl die “echte Null” bezeichnet als auch Werte nahe der Null, die zu klein

für eine andere Darstellung sind, also durch einen Unterlauf entstanden sind. Es wird zwischen „+Null“ und „-Null“ unterschieden, um das Vorzeichen eines Unterlaufs ausdrücken zu können.

Werte, die zu groß sind, um sie darstellen zu können, werden als “unendlich” bezeichnet, wobei auch hier zwischen “plus unendlich” und “minus unendlich” unterschieden wird.

Das hier vorgestellte Floating-Point-System kann also - neben der Null und “unendlich” - Zahlen im Bereich von $\pm 1{,}18 \cdot 10^{-38}$ bis $\pm 3{,}40 \cdot 10^{+38}$ abbilden.

Ein paar Beispiele sollen zeigen, wie die 32 Bits eines IEEE 754-single-Wertes zu “echten” Zahlen führen:

`0 00000000 00000000000000000000000`	0
`1 00000000 00000000000000000000000`	-0
`0 11111111 00000000000000000000000`	unendlich
`1 11111111 00000000000000000000000`	-unendlich
`0 10000000 00000000000000000000000`	$(-1)^0 \cdot 1{,}0_{\text{binär}} \cdot 2^{(128-127)} = 2$
`0 10000001 10100000000000000000000`	$(-1)^0 \cdot 1{,}101_{\text{binär}} \cdot 2^{(129-127)} = 2{,}5$
`1 10000001 10100000000000000000000`	$(-1)^1 \cdot 1{,}101_{\text{binär}} \cdot 2^{(129-127)} = -2{,}5$

Ein wesentlicher Fakt ist außerdem, dass durch “fließende” Positionierung des Kommas nicht in jedem Wertebereich die gleiche Auflösung erzielt werden kann. Beträgt die minimale Differenz zweier Floating-Point-Zahlen im untersten Bereich noch:

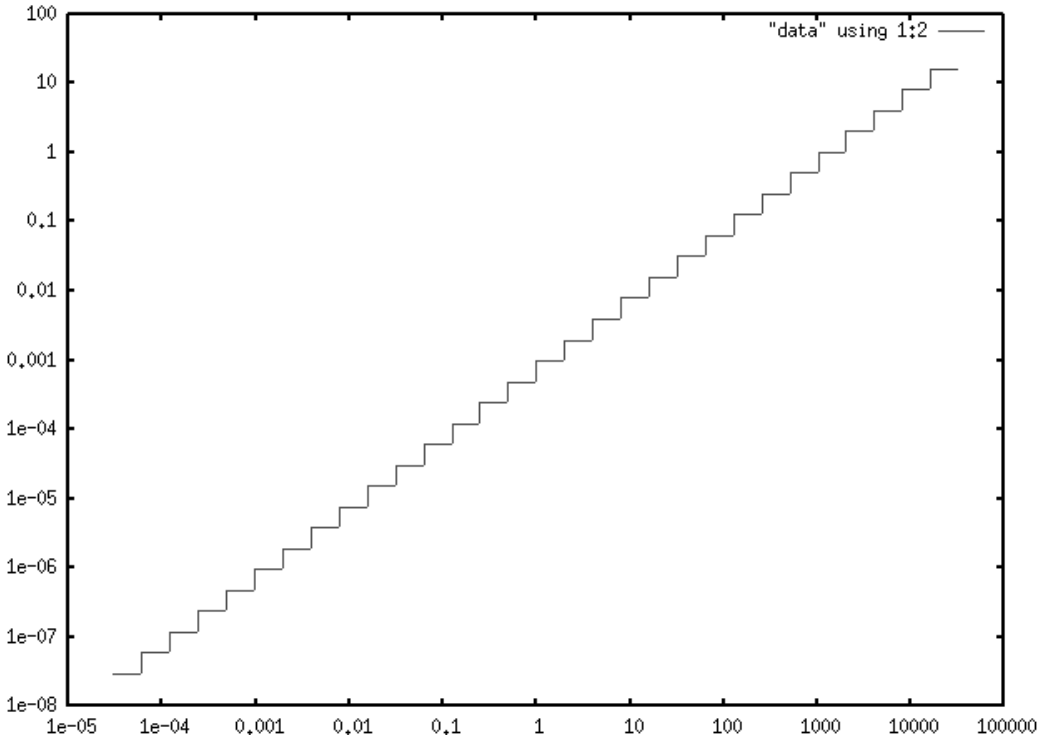
$$2^{-23} \cdot 2^{-127} \approx 5{,}877 \cdot 10^{-39}$$

so beträgt sie in den oberen Bereichen:

$$2^{-23} \cdot 2^{128} \approx 3{,}507 \cdot 10^{38}$$

Es findet also eine Anpassung der Auflösungsgenauigkeit in Abhängigkeit des Zahlenbereichs statt, der dargestellt werden soll. Das ist in allen Anwendungsbereichen sinnvoll, in denen davon ausgegangen werden kann, dass bei extrem großen Werten sehr geringe Gradienten in der Betrachtung keine Rolle mehr spielen.

Der folgende Graph zeigt am Beispiel einer Floating-Point-Zahl mit einem 4-Bit-Exponenten und einer 10-Bit-Mantisse die jeweils kleinste darstellbare Differenz auf der y-Achse sowie den Wertebereich der Floating-Point-Zahl auf der x-Achse. Man beachte die logarithmische Skalierung der Achsen:



Der Vollständigkeit halber sei an dieser Stelle noch erwähnt, dass der Typ “double” -- ebenfalls im IEEE754-Standard beschrieben -- sich ausschließlich in den folgenden Punkten vom Typ “single” unterscheidet:

- Die Wortbreite beträgt 64 Bit
- Für die Mantisse werden 52 Bit (+1 Sign-Bit) und für den Exponenten 11 Bit verwendet
- Der Exponentenbias ist auf 1023 festgelegt

Ein Ausdruck in diesem Format benötigt also im Vergleich zum Typ “single” doppelt soviel Speicherplatz und kann verwendet werden, um Zahlen mit etwa doppelter Genauigkeit anzugeben. Einige Computersysteme bieten außerdem den Typ „long double“ mit 80 Bit.

2.4 Integerarithmetik

Um zeigen zu können, in welchen Anwendungen Floating-Point Gebrauch finden kann, muss zunächst erklärt werden, wo Probleme mit klassischer, ganzzahliger Arithmetik auftreten können. Dazu sollen in diesem Kapitel drei Effekte betrachtet werden – das schnell erreichbare obere Ende der Zahlenskala, die Problematik der Rundungsfehler bei Divisionen und die unzureichende Möglichkeit, Nachkommastellen darstellen zu können.

Im Fall von diskreten Integerzahlen ergibt sich das Problem einer schnell erreichten

Obergrenze der darstellbaren Zahlen, die auch innerhalb eines Algorithmus, also z.B. in Zwischenergebnissen, nicht überschritten werden darf.

Wird beispielsweise die Zahl 255 ($11111111_{binär}$) mit 1 addiert, so ergibt sich 256, bzw. $100000000_{binär}$. Handelt es sich dabei um eine Integerzahl, die ohne Vorzeichen interpretiert wird, ergibt sich der Wert, in dem nur die untersten 8 Bit maskiert werden, der resultierende Wert ist also 0:

```
#include <stdint.h>
int main(void) {
        uint8_t i = 255 + 1;
        printf ("Ergebnis: %d\n", i);
        return 0;
}
```

```
Ergebnis: 0
```

Handelt es sich um Zahlen, die mit Vorzeichen interpretiert werden (signed), springt der Wert in den negativen Bereich, 127 + 1 wird also, mit 8 Bit Breite betrachtet, zum kleinsten in diesem Bereich darstellbaren Wert:

```
#include <stdint.h>

int main(void) {
        int8_t i = 127 + 1;
        printf ("Ergebnis: %d\n", i);
        return 0;
}
```

```
Ergebnis: -128
```

Bei anderen Bitbreiten tritt dieser Effekt an entsprechend anderen Stellen auf, die Maximalwerte sind in der Tabelle im Kapitel 2.1 aufgelistet.

Man spricht in einem solchen Fall, der das Ergebnis einer Berechnung unbrauchbar macht, von einem so genannten „Integerüberlauf“.

In der Anwendung mit A/D-Wandlern werden zu große Werte mit dem größtmöglichen Wert dargestellt, ein Effekt, der auch als “Clipping” bezeichnet wird (siehe Kapitel 4).

Ein weiteres Problem ergibt sich durch die Beschränkung auf ganze Zahlen, wodurch sich Divisionen nur ohne Restwert berechnen lassen, da die Darstellung von Zwischenwerten nicht möglich ist:

```
int main(void) {
        int i = 9 / 5;
        printf ("Ergebnis: %d\n", i);
        return 0;
}
```

```
Ergebnis: 1
```

In Integerarithmetik ist also 9 / 5 = 1.

Um diesen entscheidenden Nachteil kompensieren zu können, also an das Ergebnis des Restwerts zu kommen, wurde in der Integerarithmetik den Modulo-Operator eingeführt.

Will man eine feste Anzahl von Nachkommastellen ermöglichen, kann das beispielsweise erreicht werden, indem die Werte vor der Berechnung um eine Zehnerpotenz oder einen anderen Faktor skaliert werden und das Ergebnis entsprechend anders interpretiert wird. Beispielsweise kann man von vornherein festlegen, dass eine der beiden Zahlen eines Quotienten mit 10 multipliziert werden und das Ergebnis ebenfalls als einen um den Faktor 10 skalierten Wert ansehen. Somit würde die Berechnung wie folgt aussehen:

$$90 / 5 = 18$$

Bei der Arithmetik mit Integerzahlen muss also immer darauf geachtet werden, dass Zahlen nicht zu groß werden und man muss sich darüber im Klaren sein, in welchem Bereich Werte erwartet werden, um entsprechende Skalierungen bereits im Algorithmus zu verankern. Eine dynamische Anpassung des Wertebereichs ist mit Integern per se nicht möglich.

2.5 Floating-Point-Arithmetik

Arithmetik mit Floating-Point-Zahlen unterscheidet sich in einigen wesentlichen Details von der mit Integerzahlen. Diese lassen sich auf Rundungsfehler zurückführen und auf die Tatsache, dass bei Algorithmen, die gleichzeitig mit sehr großen und sehr kleinen Zahlen rechnen, Wertinformationen verloren gehen können.

Außerdem sind mit binärer Floating-Point-Logik nicht alle Zahlen, die in dezimaler Schreibweise einfach darzustellen sind, ausdrückbar, und umgekehrt. Die Dezimalzahl 0,1

beispielsweise kann in binärer Notation nicht exakt beschrieben werden - ebenso wie der Quotient 1/3 auch mit beliebig vielen Nachkommastellen nicht in dezimaler Schreibweise darstellbar ist. Die Approximation an $0{,}1_{dezimal}$ würde in der Binärlogik zu einer endlosen Rekursion des Musters $0{,}000110011001100..._{binär}$ führen.

Die Konsequenz daraus ist, dass ein Aufsummieren vieler dieser Werte zu einer Multiplikation des Fehlers führt. Das folgende C-Code-Beispiel verdeutlich dies:

```
int main(void) {
        float f = 0;
        int i;

        for (i=0; i<100; i++)
                f += 0.1;

        printf ("Ergebnis: %f\n", f);
        return 0;
}
```

Obwohl, wie einfach zu erkennen ist, 10.0 mathematisch korrekt wäre, liefert dieses Beispiel liefert als Ausgabe:

```
Ergebnis: 10.000002
```

Bei der Addition, Subtraktion, Multiplikation und Division von Floating-Point-Zahlen wird für gewöhnlich nach folgendem Schema vorgegangen:

- Beide Zahlen werden normalisiert
- Die Exponent der kleineren der beiden Zahlen wird so gewählt wie der Exponent der größeren Zahl
- Die Mantisse der kleineren Zahl wird entsprechend angepasst
- Die beiden Mantissen werden aufsummiert
- Ergebnis wird renormalisiert
- Das Sign-Bit wird entsprechend der Operation neu berechnet
- Der Exponent wird auf eventuell aufgetretene Unter- oder Überläufe geprüft und eventuell Spezialfälle, wie in Kapitel 2.3 beschrieben, angewandt

Durch die Anpassung der Mantisse tritt der Effekt auf, dass bei großen Differenzen der beiden Exponenten, also durch extreme Anpassungen der Mantissen, Werte verloren gehen

können.

Auch hierzu ein kurzes Code-Beispiel, das diesem Effekt anhand einer Addition demonstriert:

```
int main(void) {
        float f1 = 1.23456e10;
        float f2 = 0.1234e-10;

        printf ("Ergebnis: %.20f + %.20f = %.20f\n", f1, f2, f1+f2);
        return 0;
}
```

Hier lautet die Ausgabe nach der Kompilierung:

```
Ergebnis: 12345600000.00000000000000000000 + 0.00000000001233999968 =
12345600000.00000000000000000000
```

Neben der Tatsache, dass auch die Zahl $0{,}1234 \cdot 10^{-10}$ nicht exakt darstellbar ist, sieht man außerdem, dass bei der Addition, genauer gesagt bei der Mantissenanpassung, der zweite Wert verloren ging und im Ergebnis keine Rolle mehr spielt.

Werden allerdings zwei kleine Zahlen addiert und dieses Ergebnis erst zu einer größeren Zahl addiert, lässt sich dieser Effekt unter Umständen vermeiden:

```
int main(void) {
        float f1 = 1.23456e5;
        float f2 = 0.1234e-3;
        float f3 = 0;
        int i;

        f3 = f1;
        for (i=0; i<100; i++)
                f3 += f2;
        printf ("Ergebnis 1: %.20f\n", f3);

        f3 = 0;
        for (i=0; i<100; i++)
                f3 += f2;
        f3 += f1;
        printf ("Ergebnis 2: %.20f\n", f3);

        return 0;
}
```

Dieses Beispiel summiert im ersten Schritt 100 Mal die Zahl $1{,}23456 \cdot 10^{-3}$ auf die Zahl $1{,}23456 \cdot 10^{5}$ auf und gibt das Ergebnis aus. Im zweiten Schritt wird eine Summe über $100 \cdot 1{,}23456 \cdot 10^{-3}$ gebildet und diese Ergebnis erst ganz zum Schluss zu $1{,}23456 \cdot 10^{5}$ addiert. Die Ausgabe liefert:

```
Ergebnis 1: 123456.00000000000000000000
Ergebnis 2: 123456.01562500000000000000
```

Es spielt also im Gegensatz zur Rechnung mit Integerzahlen eine Rolle, in welcher Reihenfolge Floating-Point-Zahlen in einer Berechnung auftauchen; man spricht in der Mathematik von “nicht assoziativ”.

Es gilt generell für 3 Floating-Point-Zahlen a, b und c:

$$(a+b)+c \neq a+(b+c)$$

und

$$(a \cdot b) \cdot c \neq a \cdot (b \cdot c)$$

Desweiteren trifft das Distributivgesetz nicht zu:

$$a \cdot (b+c) \neq a \cdot b + a \cdot c$$

Diese Aussagen spielen insofern eine Rolle, als das bei der Arithmetik mit Floating-Point davon ausgegangen werden muss, dass zwei mathematisch identische Formeln ein jeweils anderes Ergebnis liefern können. Allerdings werden die Unterschiede beider Ergebnisse nicht so gravierend sein wie bei der Integerarithmetik, die im vorigen Kapitel beschrieben wurde.

2.6 Rechenprozessoren

Auf praktisch allen aktuellen Computerarchitekturen übernehmen dedizierte Subsysteme der CPUs die Arithmetikarbeit für intensive Rechenaufgaben.

Für Integerarithmetik wird dieses Subsystem ALU (arithmetic and logic unit) genannt und übernimmt die Aufgabe der Addition, Subtraktion, Multiplikation und Division von Integerzahlen sowie bitweise Operationen wie AND, OR, NOT und XOR.

Eine Einheit namens FPU (“floating point unit”) ist in CPUs, die Floating-Point nativ

unterstützten, dafür zuständig, sämtliche Rechenoperationen durchzuführen, die Floating-Point-Zahlen betreffen. Quasi alle derzeit verwendeten FPUs – bis auf ein paar exotische Ausnahmen wie zum Beispiel sehr frühe Versionen von Cray Großrechnern – implementieren Floating-Point-Arithmetik nach der vorgestellten IEEE-754-Spezifikation. Der Einsatz einer FPU entlastet den Hauptprozessor, der diese Aufgaben in vielen aufwendigen Einzelinstruktionen durchführen müsste.

Spezielle digitale Signalprozessoren (DSPs), die auf die Ausführung mathematischer Rechenoperationen spezialisiert sind, kommen heute in praktisch allen digitalen, autarken Soundanwendungen zum Einsatz. Diese Bausteine enthalten in der Regel viele “arithmetic and logic units” und sind daraufhin optimiert, viele rechenintensiven Aufgaben in wenigen Prozessortakten durchführen zu können. Darüber hinaus sind heutige DSPs oft parallel programmierbar, was sie in die Lage versetzt, mehrere Operationen gleichzeitig ausführen zu können.

Auch in modernen CPUs finden sich Erweiterungen, die helfen, aufwendige Operationen schneller durchführen zu können. Motorola entwickelte hierfür mit AltiVec eine Erweiterung, mit der sich Vektorrechnungen beschleunigen und einzelne Rechenoperationen auf bis zu acht Variablen gleichzeitig anwenden lassen, wodurch sich theoretisch durch geschickte Programmierung eine Performancesteigerung von Faktor acht erreichen lässt. AltiVec findet sich unter anderem in den G4- und G5-Prozessoren des Konsortiums aus Apple, Motorola und IBM. Die Hersteller der x86er-CPUs nennen ihre vergleichbare Erfindung SSE bzw. 3DNow!, welche in der Pentium-Serie ab der Version 3, in AMD-Prozessoren seit der Generation der K6-2/Athlon sowie in einigen Chips der Marken VIA und Transmeta zu finden sind.

Alle diese Erweiterungen herkömmlicher Desktop-CPUs lassen sich mit dem Begriff SIMD (“single instruction multiple data”) zusammenfassen und werden, so sie verfügbar sind, oft zur Kodierung und Dekodierung von Multimediadatenformaten wie MP3, JPEG oder MPEG2 verwendet.

3 Anwendung im Audiobereich

Von Floating-Point und dessen Vorteilen wird bereits seit Jahren in der digitalen Audiobearbeitung Gebrauch gemacht. In diesem Kapitel soll unter anderem am Beispiel von Implementierungen digitaler Filter gezeigt werden, an welchen Stellen die Verwendung von Floating-Point Sinn macht und welche Probleme entstehen können.

3.1 Summenbusse

Als erstes Beispiel dienen Mixingbusse in digitalen Mischpulten und HD-Recordingsystemen, die Signale aus verschiedenen Quellen aufsummieren und zu einem Signal zusammenfassen, das dann weiterverarbeitet wird.

Bei der Rechnung mit Integerzahlen treten bei dieser Anwendung, sofern die zu verarbeitenden Eingangssignale nicht alle ohne Pegelveränderung, also mit Unity-Gain in die Summe einfließen sollen, wie bereits in Kapitel 2.4 beschrieben, Rundungsfehler auf.

Als Beispiel sollen hier 5 Signale unter der Betrachtung eines einzelnen Samples dienen, die jeweils den Wert 12345 haben. Diese sollen mit jeweils -20dB in einen Summenbus einfließen, der dann wiederum um +6dB angehoben wird. Weiterhin wird extremerweise (um den Effekt zu verdeutlichen) angenommen, dass der Mixingbus intern mit nur 16 Bits arbeitet, die aufgezeigten Effekte sind selbstverständlich auch bei anderen Bittiefen nachzuweisen. Rein rechnerisch ergibt sich nun auf dem Bus das Signal

$$(12345 \cdot \text{-20dB}) \cdot 5 \cdot \text{+6dB} = (12345 / 10) \cdot 5 \cdot 2 = 12345$$

Um diese Berechnung mathematisch korrekt durchzuführen, muss bei der Division des Eingangssamples auf Kommazahlen zurückgegriffen werden, also als Ergebnis des Zwischenergebnisses “(12345/10)” 1234,5 als Wert gültig sein. Im Falle einer reinen Integerdivision kann hier allerdings, wie in Kapitel 2.4 erläutert, ein deutlicher Fehler ausgemacht werden. Das Ergebnis auf dem Summenbus ist dann $1234 \cdot 10 = 12340$, es sind also Informationen verloren gegangen. Noch eindeutiger wird dieser Effekt, wenn man von einer noch stärkeren Dämpfung des Eingangssignals und/oder einer noch größeren Verstärkung des Summensignals, die den Fehler multipliziert, ausgeht.

Alternativ hierzu kann der Algorithmus auch so umgestellt werden, dass die Division erst im letzten Schritt erfolgt:

$$(12345 / 10) \cdot 5 \cdot 2 = (12345 \cdot 10) / 10$$

Diese Vorgehensweise würde zwar den Rundungsfehler auch mit Integerarithmetik eliminieren, allerdings würde als Zwischenwert innerhalb der Berechnung aber 123450 auftauchen, ein Wert, der mit den angenommenen 16 Bits nicht mehr darstellbar ist.

Desweiteren muss bei Integermixingbussen bedacht werden, dass für ein einfaches Aufsummieren von Signalen stets ein ausreichender Headroom zur Verfügung steht, da pro Signaladdition ein weiteres Bit benötigt wird, um das Summensignal nicht ins Clipping zu bringen. Für eine Addition von 32 Eingangssignalen ist also schon ein Summenbus notwendig, der 4 Bit breiter ist als die Eingangssignale. Mit 8 Bit Headroom ließen sich maximal 256 Signale zusammenfassen. Im folgenden Kapitel wird sich zeigen, dass diese Grenze in anderen Szenarien deutlich schneller erreicht werden kann als in Summenbussen digitaler Mischpulte.

Mit Floating-Point-Arithmetik fallen die in diesem Kapitel bisher vorgestellten Effekte weniger ins Gewicht, da die zu erwartenden Rundungsfehler geringer sind nicht davon ausgegangen werden muss, dass ein Zwischenwert den Rahmen des mit Floating-Point-Zahlen darstellbaren Zahlenraums sprengt.

Allerdings können, wie in Kapitel 2.5 beschrieben, sehr kleine Werte gegenüber sehr großen bei der Berechnung verloren gehen, ein Effekt, der aber gerade in der Audiotechnik eher verschmerzbar ist, da für das menschliche Ohr sehr leise Signale von lauten Signalen meist psychoakustisch verdeckt werden.

In dieser Anwendung wird im Übrigen von Floating-Point reger Gebrauch gemacht – alle mir bekannten HD-Recordingsysteme (namentlich ProTools, Nuendo, Logic, SADiE, Pyramix) arbeiten intern mit auf Fließkommazahlen basierenden Bussen. Mehr Informationen hierzu finden sich im Kapitel 3.4.

3.2 Filter

Alle digitalen Effektprozessoren, Filter und Equalizer arbeiten im Kern mit Implementierungen digitaler Transformationen und Filter wie DFT („discrete fourier transformation“), FFT („fast fourier transformation“), FIR („finite impulse response filter“) oder IIR („infinite impulse response filter“).

In diesem Kapitel soll am Beispiel des einfachsten aller digitalen Filter, einem sogenannten 1-Pol-Filter, gezeigt werden, dass es selbst bei einfachsten Filterkonfigurationen sinnvoll bzw. nötig ist, auf Floating-Point-Implementierungen zurückzugreifen.

Da diese Systeme nicht Schwerpunkt dieser Facharbeit sind, soll hier nicht im Detail auf

deren Implementierung eingegangen werden, sondern lediglich die Grundarbeitsweise erläutert werden.

Fourier-Transformationen können unter anderem dazu verwendet werden, ein eingehendes Signal in seine Frequenzbestandteile zu zerlegen. Dazu wird das Signal stückweise in Fensterintervallen analysiert und, je nach Rechenaufwand, in ein mehr oder weniger genaues Frequenzspektrum zerlegt.

Der Algorithmus, mit dem eine diskrete Fourier-Transformation implementiert wird, ist in der folgenden Formel beschrieben:

$$\sum_{k=0}^{n-1} x_k e^{-(2\Pi i/n)jk}$$

Die nach dem Summenzeichen auftauchenden Variable x stellt für jeden Iterationszyklus der Summenbildung eine komplexe Zahl dar. Die Lösung dieser Formel würde eine Anzahl von n^2 Rechenoperationen erfordern. Man spricht in der Mathematik im Zusammenhang mit der von Bachmann erfundenen O-Notation von $O(n^2)$, der Lösungsaufwand steigt also zur Iterationstiefe quadratisch an. Das bedeutet, dass auch die Anzahl der zu addierenden Zahlen quadratisch wächst. Dies kann, wie in den vorigen Kapiteln hergeleitet, zu einem Überlauf führen, wenn mit Integern gerechnet wird. In diesem Fall müsste also sorgfältig geplant werden, in welcher Reihenfolge diese Summenbildung stattfinden soll und eventuelle Zwischenergebnisse quantisiert, also die bei Addition und Multiplikation entstandenen zusätzlichen Bits “weggerundet“ werden.

FIR-Filter werden sowohl zur Frequenzgang- und Phasenkorrektur als auch zur Simulation von Hall- und Echoräumen verwendet. Ein FIR-Filter besteht aus P Filtertaps h_k, mit denen die Samples x_k des eingehenden Signals anteilig gewichtet akkumuliert werden. Durch die fehlende Rekursion sind FIR-Filter in jedem Fall stabil und ihre Impulsantwort von fester Länge. Ausgehende Samples werden nach folgender Vorschrift berechnet:

$$\sum_{k=0}^{P-1} h_k \cdot x_{n-k}$$

Im Gegensatz zu FIR-Filtern sind **IIR-Filter** rekursiv definiert, es gibt also einen Rückkopplungspfad vom Ausgang des Filters zurück in den Filter. IIR-Filter sind nicht stateless, d.h. das Filterverhalten hängt zu jedem Zeitpunkt von der Historie des schon ausgewerteten Signals ab. Auch IIR-Filter lassen sich aber, trotz dieser Rekursion, als Summe von Produkten abbilden:

$$\sum c_n \cdot x$$

Die einzelnen Filterkoeffizienten müssen hierbei nicht fest sein, sondern sind Funktionen der Rückkopplungssignale. Klassischerweise wird dieser Filtertyp zur Phasen- und Frequenzgangkorrekturen sowie zur Klangsynthese verwendet.

Wie bereits erwähnt sollen die einzelnen Filter und Transformationen nicht Schwerpunkt dieser Arbeit sein, interessant in diesem Zusammenhang ist hier jedoch, dass alle ihre Implementationen als Summe von Produkten dargestellt werden können. Es ist somit offensichtlich, dass sich hier enorme Probleme mit komplexeren Filtern und vielen Taps (Koeffizienten) ergeben können.

Gerade bei der Implementierung mit Integern muss darauf geachtet werden, dass genügend Headroom zur Abbildung der Zwischenergebnisse verfügbar ist, um Overflows zu verhindern.

Nach jedem Rechenschritt müssen die bei jeder Addition (je 1 Bit) und jeder Multiplikation (Verdopplung der Bits) hinzugekommenen Bits durch Division, Bitshifting oder Skalierung wieder eliminiert werden.

Bei Floating-Point-Aithmetik sollten Operationen derart sortiert werden, dass Zischenergebnisse gleicher Größenordnung miteineander verrechnet werden.

Dieses Prinzip soll nun am Beispiel des sehr einfachen 6db/Oktave **Ein-Pol IIR Hoch- und Tiefpassfilters** demonstriert werden. Hierzu wurde ein einfaches C-Programm geschrieben, das nun Stück für Stück seziert wird. Das komplette Listing ist im Appendix B zu finden.

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

Diese Zeilen sind notwendig, um den Compiler anzuweisen, notwendige Include-Dateien, die Funktionsdeklarationen und Konstanten enthalten, die im Folgenden verwendet werden, einzubinden.

```
#define sample          int16_t
#define internal        int16_t
```

Das Programm wurde so geschrieben, dass sich durch die Änderung dieser beiden Zeilen der gesamte Algorithmus von Floating-Point auf Integers umstellen lässt. Im Fall einer Floatingpointimplementierung muss hier stattdessen stehen:

```
#define sample        float
#define internal      float
```

```
#define SCALE           (internal) ((1 << 15) - 1)
#define SAMPLERATE      8000            /* Hz */
#define N_SAMPLES       (SAMPLERATE/2)
#define CUTOFF          200.0           /* Hz */
#define FREQ1           4000.0          /* Hz */
#define FREQ2           10.0            /* Hz */
```

Das Präprozessormakro “SCALE” ist notwendig, um das Eingangssignal sowie die entstehenden Werte in einen Bereich zu bringen, der den Integerzahlenraum voll ausnutzt. Das ist notwendig, da die C-internen Funktion sin() Werte in Floating-Point im Bereich von 0-1 ausgibt.

“SAMPLERATE” gibt die Samplingrate an, mit der gerechnet werden soll und “N_SAMPLES”, für wieviele Samples die Berechnung stattfinden soll.

“CUTOFF” beschreibt die Frequenz, bei der der IIR-Filter einsetzen soll und “FREQ1” und “FREQ2” die Frequenzen der Sinussignale, aus denen das Eingangssignal entstehen soll.

Da es sich hierbei um einen Filter mit 6dB/Oktave handelt, müssen die FREQ1 und FREQ2 weit genug auseinander liegen und die CUTOFF-Frequenz entsprechend dazwischen gewählt werden.

```
double coeff_omega = (2*M_PI*CUTOFF)/(double)SAMPLERATE;
double freqsin(double freq, int i) {
        return sin((freq*M_PI)*((double)i/SAMPLERATE));
}
int main(void) {
        sample in[N_SAMPLES];
        sample hp_out[N_SAMPLES], lp_out[N_SAMPLES];
        int i;

        for (i=0; i<N_SAMPLES; i++)
           in[i] = (sample) (0.5 * (double) scale *
                 (freqsin(FREQ1, i) + freqsin(FREQ2, i)));
        calc_hp(N_SAMPLES, in, hp_out);
        calc_lp(N_SAMPLES, in, lp_out);
        for (i=0; i<N_SAMPLES; i++)
                printf ("%i %g %g %g\n", i,
                                (double) in[i],
                                (double) hp_out[i],
                                (double) lp_out[i]);
        return 0;
```

```
}
```

Im Mainloop des Programms wird nun zunächst ein Array mit N_SAMPLES Ausgangswerten erzeugt, indem zwei Sinussignale, die entsprechend der oben definierten Auflösung horizontal skaliert wurden, aufaddiert werden. Dieses Array wird nun an die beiden Funktionen calc_hp() und calc_lp() übergeben, die darauf die Highpass- und Lowpassfilter applizieren. Zum Schluss werden dann alle Ergebnisse ausgegeben.

```
void calc_hp(int n_samples, sample *in, sample *out) {
        internal acc = 0;
        internal coeff;
        int i;

        coeff = (internal)
            ((double)SCALE*((2.0+cos(coeff_omega))
            - sqrt(pow(2.0+cos(coeff_omega), 2.0) - 1.0)));
        for (i=0; i<n_samples; i++) {
            acc = (((coeff - scale) * (internal) in[i])
                  - (coeff * acc)) / (internal) SCALE;
            out[i] = -acc;
        }
}
```

Dies ist der Quellcode des Hochpassfilters, der zunächst seinen Koeffizienten nach folgender Formel berechnet:

$$coeff = 2 + \cos\left(\frac{2 * \pi * CUTOFF}{SAMPLERATE}\right) - \sqrt{2 + \cos\left(\left(\frac{2 * \pi * CUTOFF}{SAMPLERATE}\right) - 1\right)^2}$$

Dann wird für jedes Sample die folgende for()-Schleife durchlaufen und der Akkumulator, der anfangs auf Null gesetzt wurde, entsprechend der Implementierungsvorschrift für diesen Filter neu berechnet:

$$acc = (coeff - 1) \cdot input + coeff \cdot acc$$

Das Ergebnis jedes Schleifendurchlaufs wird abgespeichert und später von der main()-Routine ausgegeben. Entsprechend gilt für den Lowpassfilter:

```
void calc_lp(int n_samples, sample *in, sample *out) {
        internal acc = 0;
        internal coeff;
        int i;

        coeff = (internal)
            ((double) SCALE * ((2.0-cos(coeff_omega))
            - sqrt(pow(2.0-cos(coeff_omega), 2.0) - 1.0)));
```

```
        for (i=0; i<n_samples; i++) {
                acc = (((SCALE - coeff) * (internal) in[i])
                      + (coeff * acc)) / (internal) SCALE;
                out[i] = acc;
        }
}
```

Die Implementierung des Lowpass-Filters ist vergleichbar zu der des Highpasspendants, die Filter- und Koeffizientenformeln sind jedoch andere:

$$coeff = 2 - \cos\left(\frac{2 * \pi * CUTOFF}{SAMPLERATE}\right) - \sqrt{2 - \cos\left(\left(\frac{2 * \pi * CUTOFF}{SAMPLERATE}\right) - 1\right)^2}$$

$$acc = (1 - coeff) \cdot input + coeff \cdot acc$$

Die Casts (Typenumwandlungen) in diesem Code sind nur notwendig, um die Algorithmen sowohl mit Integern als auch mit Floats funktionieren zu lassen, sie werden daher nicht weiter erläutert.

Betrachtet man nun die beiden Stellen, an denen die Summenbildung stattfindet

```
acc = (((coeff-scale)*(internal)in[i])-(coeff*acc))/(internal)SCALE;
```

und

```
acc = (((SCALE-coeff)*(internal)in[i])+(coeff*acc))/(internal) SCALE;
```

so wird klar, dass hier es hier zu großen Werten kommen kann, sobald der Akkumulator, der bei jedem Zyklus mit dem Eingangssignal aufgeladen wird, den hinzukommenden Anteil nicht mehr in seinem Wertebereich abbilden kann.

Selbst bei diesen einfachen Filtern muß besondere Sorgfalt auf die Integerimplementation verwendet werden. Die entsprechende Floating-Point-Variante ist deutliche einfacher: hier kann als Skalierungsfaktor („SCALE“) der Wert 1,0 angesetzt werden, damit entfallen die bei Integerzahlen für die Overflowpräventionen nötigen Divisionen.

3.3 Oszillatoren

Bei der Simulation von Instrumenten sowie bei der virtuellen Klangsynthese spielen Oszillatoren zur Erzeugung von Signalen sowie LFO (low frequency oscillators) eine zentrale Rolle. Um einen breiten Bereich an Frequenzen erzeugen zu können und beispielsweise einen Sinus von 1000Hz mit einem LFO um wenige Prozente zu modulieren,

ist es notwendig, auch diese Werte sehr exakt angeben zu können und zur internen Verarbeitung Frequenzen in exakten Bruchteilen darstellen zu können.

Mit Integerarithmetik würde sich hierbei zwangsläufig das bereits bekannte Problem ergeben, dass man sich mittels Fixed-Point-Darstellung auf eine genau definierte Anzahl von Nachkommastellen einigen müsste und sich die Präzision an Zwischenwerten mit einer entsprechend sinkenden Obergrenze des darstellbaren Bereichs erkaufen würde.

Aus diesem Grund wird auch in dieser Anwendung von allen namhaften Herstellern Floating-Point eingesetzt, um diese Probleme zu umgehen.

3.4 Computerschnittstellen

Apple führte im März 2001 mit MacOSX eine komplett überarbeitete Version des Betriebssystems ein, das mit **CoreAudio** ein Audiosubsystem mitbringt, welches intern komplett auf Floating-Point basiert. Lediglich bei der Schnittstelle in Richtung der angeschlossenen Hardwarekomponenten oder beim Ausspielen in Dateien kann optional noch in Integerlogik gewandelt werden. Seit der Version 10.3 kann für die Durchführung dieser Operation auf die Erweiterungen der AudioUnits zurückgegriffen werden. Hierzu muss nur ein entsprechendes Konverterobjekt instantiiert und an der richtigen Stelle in die Prozesskette eingehängt werden.

DirectSound, die Microsoft-eigene Schnittstelle zur Soundverarbeitung hingegen, arbeitet in aller Regel nur mit Integern.

Das von Steinberg (mittlerweile Pinnacle) entwickelte, speziell auf niedrige Latenzen ausgelegte Interface namens „**ASIO**“ ist zur Anbindung von Soundhardware gedacht und unterstützt Sampleraten von 32kHz bis 96kHz bei Samplebreiten von 16, 24 und 32 Bit in Integernotation sowie 32-bit und 64-bit Floating-Point-Werte. Es ist mit diesem System also wie mit Apples CoreAudio möglich, Floating-Point-Daten an Hardwarekomponenten zu übergeben. Hiervon wird allerdings nach meinem Kenntnisstand noch nicht allzu viel Gebrauch gemacht.

Plugins, die für die **RTAS**-Schnittstelle von ProTools, für Steinbergs **VST**-Umgebung programmiert wurden oder die für die generische Verwendung unter MacOSX und Logic als **AudioUnits** implementiert wurden, werden ebenfalls mit Floating-Point-Werten angesprochen.

Die Schnittstelle **DXi,** die von TwelveToneSystems für deren Produkte Cakewalk Sonar und Cakewalk HomeStudio entwickelt wurde, verschafft sich in diesem Zusammenhang einen

eher zweifelhaften Ruhm als einzige Ausnahme. In diesem System sind Floating-Point-Zahlen nicht angedacht, es muss komplett in ganzzahliger Arithmetik gerechnet werden.

Die Wandlung in analoge Signale geschieht bei fast allen System bereits in der Treiberschicht, es sei denn, es wird Hardware angesprochen, die mit einem DSP ausgestattet ist. In diesen Fällen übernimmt dieser dann die Aufgabe und entlastet damit die CPU des Hostrechners.

Alle aktuellen Programme, die Audiodaten verarbeiten, sind mittlerweile bei der internen Datenverarbeitung auf die Verwendung von Floating-Points umgestiegen, da wie in den Kapiteln 3.1 und 3.2 bereits beschrieben die Vielzahl an Signalquellen schnell den Headroom des digitalen Mixers in Integerarithmetik sprengen würde. Die Hersteller der folgenden Applikationen werben mit der Verwendung von Floating-Points als internes Datenformat:

- ProTools
- Logic
- Nuendo
- Creamware
- Pyramix Virtual Studio
- SADiE
- WaveLab
- Fairlight Constellation/Station Plus/Satellite
- AVID
- Merging Tec.
- Native Instruments

ProTools spielt in diesem Zusammenhang mit den von Digidesign eingesetzten externen DSP-Racks der **TDM**-Systeme eine Sonderrolle, da die dort zum Einsatz kommenden DSPs von Motorola nur Integerarithmetik beherrschen. Aus diesem Grund muss vor der Übergabe an die externe Hardware in Integer gewandelt werden, obwohl als internes Datenformat Floating-Point verwendet wird.

3.5 Floating-Point in Multimedia-Datenformaten

Auch in einigen Datenformaten zur Speicherung von Multimediainformationen wird Floating-Point verwendet, um Umrechnungsfehler, die ansonsten bei der Kodierung und Dekodierung entstehen könnten, zu vermeiden.

Die folgende Tabelle gibt einen Überblick über häufig verwendete, nicht datenreduzierte Audioformate geben und zeigt, in welchen Zahlenformaten diese jeweils Audioinformationen ablegen können.

Format	8-bit Int	16-bit Int	24-bit Int	32-bit Int	32-bit FP	64-bit FP
Microsoft WAV format (little endian)	-	x	x	x	x	x
Apple/SGI AIFF format (big endian)	x	x	x	x	x	x
Sun/NeXT AU format (big endian)	x	x	x	x	x	x
RAW PCM data	x	x	x	x	x	x
Ensoniq PARIS file format	x	x	x	x	-	-
Amiga IFF / SVX8 / SV16 format	x	x	-	-	-	-
Sphere NIST format	x	x	x	x	-	-
VOC files	-	x	-	-	-	-
Berkeley/IRCAM/CARL	-	x	x	x	x	-
Sonic Foundry's 64 bit RIFF/WAV	-	x	x	x	x	x
Matlab (tm) / GNU Octave	-	x	-	x	x	x
Portable Voice Format	x	x	-	x	-	-
Audio Visual Research	x	x	-	-	-	-
MS WAVE mit WAVEFORMATEX	-	x	x	x	x	x

(FP = Floating-Point, Int = Integer)

Bei datenreduzierten Formaten gestaltet sich die Analyse etwas schwieriger, das diese auf Containerformaten basieren, die in der Regel intern alle Notationen zulassen. So ist es beispielsweise in Apples Quicktime-Codecs möglich, Floats direkt abzuspeichern.

Es gibt allerdings nach wie vor kein Medium für Endverbraucher, auf dem in standardisierter Form Samples in Floating-Points zum Einsatz kommen. Es bleibt abzuwarten, wie sich das in nächster Zukunft entwickeln wird.

4. Floating-Point A/D-Wandler

A/D-Wandler sind bekanntermaßen dafür zuständig, analoge Signale in einen digitalen Wert umzuwandeln. Eine optimale Aussteuerung des Eingangssignals ist essentiell, da ein zu schwaches Signal die Bittiefe des Wandlers nicht ausnutzt und dieser somit nicht mit seiner maximalen Qualität arbeiten kann. Ein zu stark ausgesteuertes Signal hingegen bringt den Wandler an die Grenze seines Wertebereichs und es wird ab der Clippinggrenze nur noch der Maximalwert ausgegeben, Waveformen werden hierbei abgeschnitten. Clipping ist gerade im Audiobereich ein Desaster und muss unbedingt verhindert werden.

Wird eine automatische Gainanpassung (automatic gain control - „AGC“, in Kompressoren und Limitern eingesetzer Funktionsblock) an der analogen Eingangsseite vorgenommen, so werden diese Effekte zwar kompensiert, es ist dann allerdings nachträglich nicht mehr möglich, die Dynamik des originären Signals zu bestimmen.

In diesem Kapitel wird die Idee vorgestellt, einen Wandler zu konstruieren, der von den vielen Vorteilen der Floating-Point-Technik Gebrauch macht.

Trotz der weiten Verbreitung von Floating-Point-Arithmetik in vielen Audioprozessoren und der Verwendung als internes Datenformat in Betriebssystemen und HD-Recordingtools gibt es nach wie vor keine A/D-Wandler auf dem Markt, die Datenströme mit solchen Zahlen nativ erzeugen.

Das im Folgenden vorgestellte Modell eines pipelined Floating-Point A/D-Wandlers, der eine IEEE754-Zahl als Ergebnis ausgibt, basiert auf einer Idee des Autors und soll als “proof-of-concept” verstanden werden. Es kann an dieser Stelle aufgrund des engen Rahmens, der in einer SAE-Facharbeit vorgegeben ist, nicht im Detail auf den Aufbau der einzelnen Komponenten eingegangen werden, sondern lediglich das Funktionsprinzip erklärt werden.

Weiterführende Informationen zu ähnlichen Ansätzen finden sich unter anderem in den Literaturhinweisen im Appendix A.

4.1 Das “Pipeline-Prinzip”

Neben den im SAE-Unterricht erläuterten Wandlerprinzipien „Delta-Sigma“ und „Flash“ existiert außerdem der Ansatz des Pipeline-Wandlers, dessen Funktionsweise hier kurz erläutert werden soll, da diese Technik einen zentralen Teil des im Folgenden beschriebenen Wandlers darstellt.

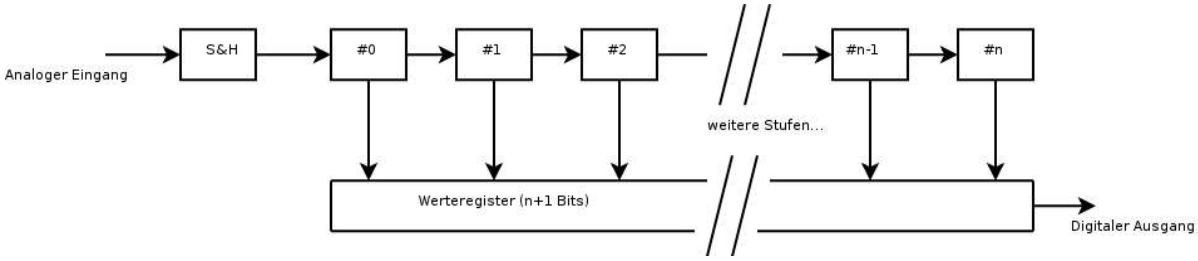
Das analoge Eingangssignal wird mittels einer Sample-and-Hold-Stufe für kurze Zeit

festgehalten und an die erste von n Pipeline-Stages übergeben, die jeweils einen analogen Eingang, einen analogen Ausgang sowie einen weiteren, digitalen Ausgang haben.

Die Logik in jeder dieser Stufen ist simpel: wird festgestellt, dass das eingehende Signal höher liegt als ein intern geeichter Referenzpegel, wird am digitalen Ausgang eine 1 als Wert signalisiert und am analogen Ausgang exakt die Hälfte des Eingangssignals angelegt. Ist dies nicht der Fall, wird das Eingangssignal unverändert an den analogen Ausgang weitergegeben und die Logikleitung auf Null gelegt.

Alle Logikpegel werden dann an ein so genanntes Latch übergeben, das, sobald die letzte Stufe ihren Pegel angelegt hat, den Wert ausgibt und die Sample-and-Holdstufe anweist, den nächsten Wandlerdurchlauf zu starten.

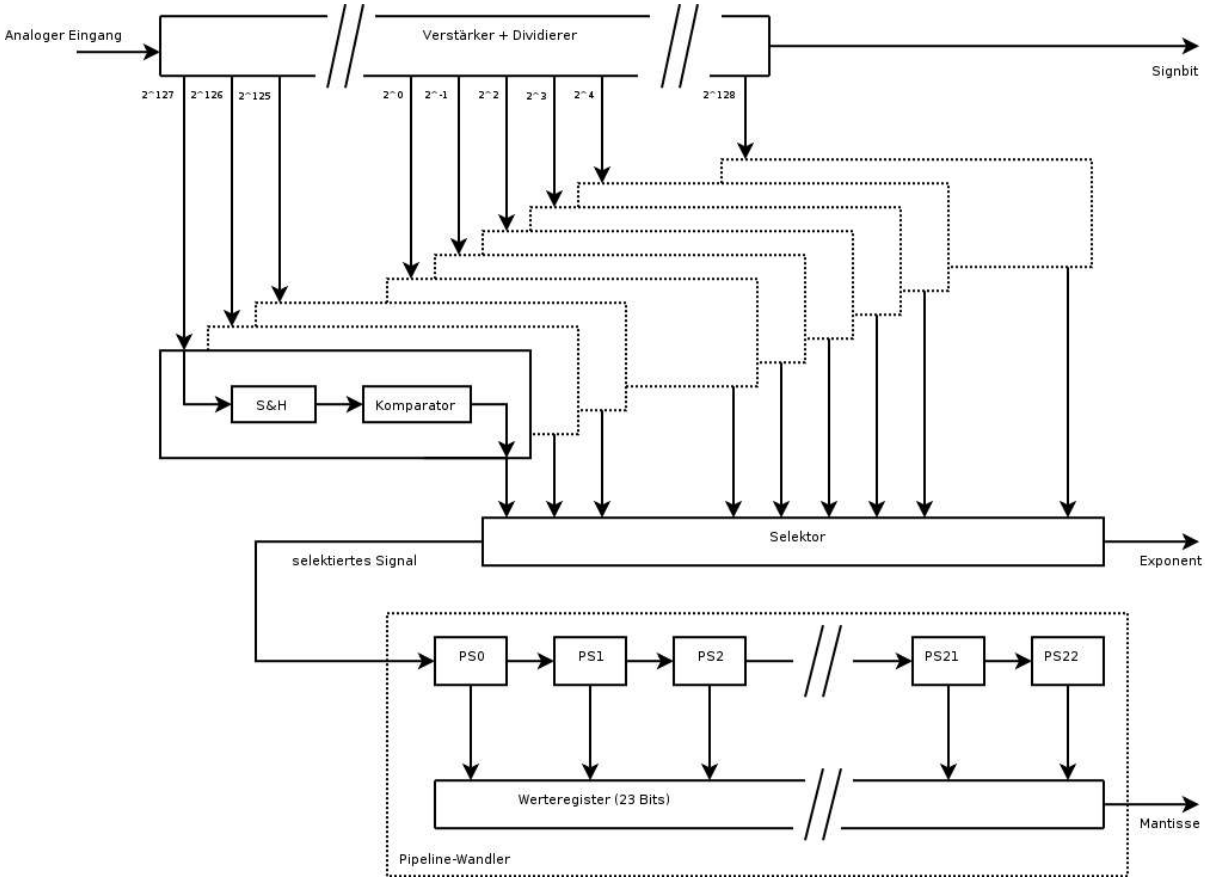
Im folgenden Blockschaltbild wurde die Synchronisations- und Clockinglogik bewusst unterschlagen, um die Technik klar erkennbar zu machen:



Der Vorteil dieses Aufbaus gegenüber einem Flashwandler ist, dass bei einer Auflösung von n Bits lediglich n Stufen benötigt werden, während beim Flashwandler hier 2^n Stufen nötig wären. Problematisch ist allerdings auch hier, dass eine Eichung des Referenzpegels und des Spannungsteilers in jeder Stufe stattfinden muss, was eine aufwendige und teure Herstellung mit sich bringt.

4.2 Blockschaltbild

Dieses Blockschaltbild, das den D/A-Wandler beschreibt, erhebt keinen Anspruch auf Vollständigkeit der technischen Details. Es fehlen auch hier Timing- und Clockcontrolleitungen, die für einen funktionierenden Aufbau zwingend erforderlich wären.



4.3 Funktionsweise

Der analoge Eingang wird oben links an einem Funktionsblock angelegt, der das Signal zunächst normalisiert, also im Falle eines negativen Signals die Polarität umkehrt, dann um verschiedene Faktoren (2^{127}-fach bis 1-fach) verstärkt und durch mehrere Quotienten (2^{1} bis 2^{128}) teilt und jeweils einen eigenen Ausgang besitzt, um alle 256 analogen Signale an weitere Blöcke zu verteilen. Außerdem wird die Polarität des Originalsignals durch eine digitale Leitung nach außen gereicht. Dieses Bit wird später als Sign-Bit in die Floating-Point-Zahl eingehen.

Jedes dieser Signale durchläuft dann eine Einheit, die aus einer Sample-and-Hold-Stufe und einem nachgeschalteten Komparator besteht, der erkennt, ob das Signal den jeweiligen, für diesen Block spezifischen Referenzpegel überschritten hat, also den nachgeschalteten Wandler zum Clipping bringen würde. Im Diagramm ist nur einer dieser Blöcke detailliert gezeichnet, in den gepunkteten Kästchen befindet sich jeweils dieselbe Logik nochmal.

All diese 256 Signale laufen dann in einer Einheit auf, die die erzeugten “Overloadbits”

auswertet. Das stärkste Signal, bei dem dieses Bit nicht gesetzt ist, wird zum Pipeline-Wandler weitergegeben und die Ordinalzahl des gewählten Signals als 8 Bit breites Signal ausgegeben. Diese Information wird später als Exponent in den Wert der Floating-Point-Zahl einfließen.

Im nächsten Schritt wird dann ein 23-bittiger Pipeline-Wandler verwendet, um das Signal, das sich jetzt auf jeden Fall knapp unterhalb des Referenzpegels befindet, den Wandler aber gut aussteuert, auszuwerten und in einem Werteregister zu speichern.

Eine weitere Einheit, die im Blockschaltbild nicht auftaucht, verwertet im finalen Schritt nun das Sign-Bit der Gainstufe als Sign-Bit der erzeugten Floating-Point-Zahl, den vom Selektor erzeugten Wert als Exponenten und die Ausgabe des Pipelinewandlers als Mantisse. Anschließend wird die Floating-Point-Zahl über eine beliebige Schnittstelle an ein Computersystem übergeben und dort als solche weiterverarbeitet.

Vereinfachend und zusammenfassend lässt sich das Prinzip also so beschreiben, dass ein Eingangssignal gleichzeitig teils unsinnig hoch verstärkt und extrem abgeschwächt wird, um dann eines dieser entstandenen Signale, das den nachgeschalteten Wandler weder unter- noch übersteuern wird, zu digitalisieren und auszugeben.

Das ist nicht zu vergleichen mit einer analogen Kompression, bei der Dynamik verloren geht. Der große Unterschied ist hierbei, dass zu jedem Sample der Verstärkungs- oder Abschwächungsfaktor in Form des Exponenten mit angegeben wird, sodass es der verarbeitenden Software überlassen bleibt, ein Sample als “zu klein” oder “zu groß” einzustufen. Pump- und Kompressionseffekte werden hierbei ausgeschlossen.

4.4 Diskussion

Der eindeutige Vorteil dieses Wandlers ist, dass es zumindest im Audiobereich kein Szenario geben dürfte, das das Gesamtkonstrukt zum digitalen Clipping bringt.

Geht man wie anfangs beschrieben von einem Wertebereich der Floating-Point-Zahlen von $1{,}18 \cdot 10^{-38}$ bis $3{,}40 \cdot 10^{+38}$ aus, lässt sich hiermit auch rein mathematisch der Dynamikumfang pro Polaritätsseite dieses Wandlers berechnen:

$$D = 20 \cdot \log\left(\frac{3{,}40 \cdot 10^{38}}{1{,}18 \cdot 10^{-38}}\right) \approx 1530 \ \ dB$$

Dieser Wert ist natürlich mit Vorsicht zu genießen und als rein theoretisch anzusehen, da man sich durch den Einsatz analoger Verstärkerstufen selbstverständlich ein erhebliches Grundrauschen erkauft, das sich negativ auf die Signal-to-Noise-Ration auswirkt und damit

auch den Dynamikbereich schmälert.

Außerdem muss davon ausgegangen werden, dass die verwendeten Verstärkerstufen nicht optimal linear arbeiten, was zumindest bei hohen Faktoren zu Verzerrungen im Frequenzband führt.

Man müsste in der Praxis demzufolge den Bereich des Exponenten auf einen für den entsprechenden Anwendungsfall sinnvollen Bereich beschränken.

Da in diesem Modell 256 analoge Stufen zum Einsatz kommen, deren Referenzpegel und Spannungsteiler erheblichen Einfluss auf das Messergebnis haben, wird auch hier einige Arbeit nötig sein, um das System zu kalibrieren.

Ein weiteres großes Problem bei der Herstellung des vorgestellten Modells ist der verwendete Pipeline-Wandler am Ende der Kette, der wie bereits beschrieben eine aufwendige Produktion mit sich bringt und ebenfalls kalibriert werden muss.

Die Gesamtlogik muss außerdem sehr schnell reagieren, was bei der Menge an Einzelkomponenten ein nicht unerheblicher Punkt bei der Realisierung werden dürfte.

Alternativ könnte natürlich ein anderes Wandler- oder Spannungsteilerprinzip eingesetzt werden, bisher gibt es jedoch noch keinen ernstzunehmenden Ansatz hierzu.

Nur um einen Eindruck zu vermitteln, von welch absurd hoher Dynamikumfang hier die Rede ist, stelle man sich vor, der Wandler wäre auf Studiorecordingpegel genormt. Eine einfache Berechnung zeigt, dass die Obergrenze dieses Wandlers dann bei ca. +/- $5{,}1 \cdot 10^{38}$ Volt liegen würde. Andererseits liegt die Eingangsempfindlichkeit des Wandlers im unteren Bereich bei einigen nano-Volt. Es wäre also problemlos möglich, neben hochohmigen Mikrophonausgängen auch gleichzeitig Blitzeinschläge zu samplen, ohne den Wertebereich anpassen zu müssen. Einige weitere Beispiele finden sich in Kapitel 4.5.

Im Bereich der Floating-Point A/D-Wandler wird derzeit weltweit an einigen wenigen Instituten wie z.B. dem MIT, der Stanford University in San Francisco und der Universität Lund in Schweden sowie in einigen Konzernen wie Nokia geforscht. Es gibt sinnvolle, zu dem in dieser Arbeit beschriebenen System alternative Ansätze und erste in Silizium gegossene Prototypen, allerdings noch kein wirklich serientaugliches Design.

Eine derartige Wandlertechnik könnte allerdings, wenn sie exakt und kostengünstig herzustellen ist, die digitale Audiowelt nochmals revolutionieren und durch den Ausschluss von Clipping eines der größten Probleme der digitalen Audioverarbeitung beseitigen.

Auch im Bereich des digitalen Films könnte man einen Quantensprung erreichen. Man stelle sich nur ein System vor, das pro Pixel eines extrem empfindlichen CCD-Chips und pro Frame eine dem Bild angepasste Helligkeitabsenkung vornehmen und Bilddaten im

Floating-Point-Format zur Bildverarbeitung schicken kann. Mit einer solchen Kamera könnte man, ohne einzelne Bildbereiche über- oder unterzubelichten, eine Glühbirne mit 100 Watt filmen, den Schriftzug auf ihr lesen und im Hintergrund noch den Raum erkennen, den sie beleuchtet.

Interessant ist eine solche Technik im Übrigen nicht nur für Multimediaanwendungen, sondern gerade auch für andere Wissenschaften, die sich mit Signalen sehr großer Dynamik befassen. Als Beispiel seien hier Seismographen, Wetterstationen, Crashtestmessgeräte, Flugzeugsteuerungen oder medizinische Messaufbauten genannt. Es ist also zu erwarten, dass Impulse zur Massentauglichkeit aus dieser Richtung gegeben werden.

Da mit ASIO und CoreAudio bereits Schnittstellen existieren, die Floating-Point an Hardwarekomponenten übergeben können, wäre es also ein leichtes, bestehende Software zur Zusammenarbeit mit diesem System zu bewegen.

4.5 Anwendungsbeispiele

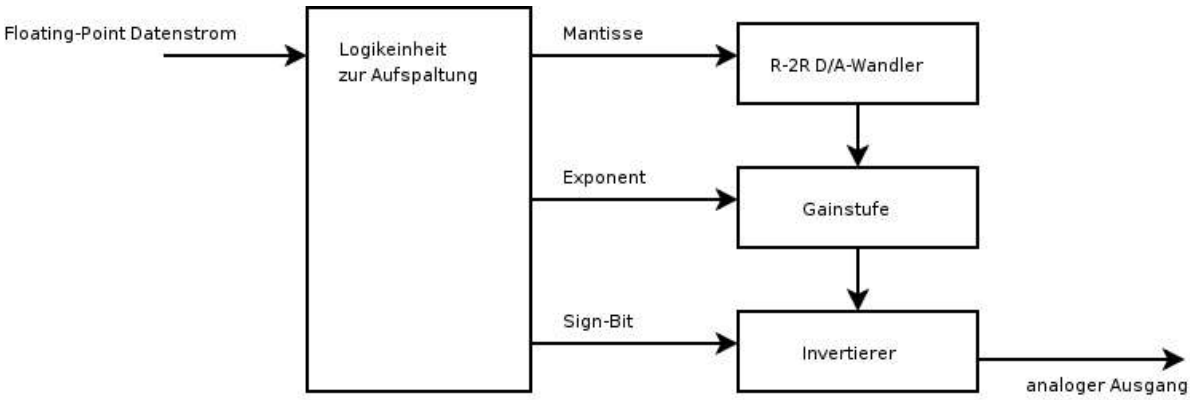
Der gewaltige Dynamikumfang, der mit diesem Wandler möglich wäre, bietet etliche Anwendungsmöglichkeiten. Ohne weitere Kompressoren, Limiter oder Verstärker wären je nach praktischer Umsetzung folgende Anwendungsszenarien denkbar:

- Abnahme von Mikrophonen und Instrumenten
- Direktes Sampling von Gitarren-Röhren-Verstärkern (sehr hochpegelig)
- Anschluss an Plattenspielersaphire ohne RIAA-Entzerrungsstufe
- Aussteuerung von niederpegeligen Bändchenmikrophonen

5. Floating-Point D/A-Wandler

Konsequenterweise muss an dieser Stelle natürlich auch noch über die Umkehrung der im vorigen Kapitel beschriebenen Idee nachgedacht werden, über den möglichen Aufbau eines digital-analog-Wandlers, der also Floating-Point als digitales Eingabeformat akzeptiert und einen analogen Pegel ausgibt. Auch das in diesem Kapitel vorgestellte Design basiert auf einer Idee des Autors. Dem Autor sind keine derartigen oder ähnlichen Implementierungen bekannt.

5.1 Blockschaltbild



5.2 Funktionsweise

Im Vergleich zum skizzierten A/D-Wandler hält sich die Komplexität dieses Aufbaus in engen Grenzen.

Zunächst gelangt der digitale Datenstrom, in dem Floating-Point-Zahlen kodiert sind, zu einer Logikeinheit, die die Aufspaltung in Mantisse, Exponent und Sign-Bit vornimmt und den Exponenten entsprechend seines Bias anpasst.

Die Mantisseninformation wird dann einen herkömmlichen R-2R-Wandler übergeben, der daraus ein analoges Signal erzeugt, welches in einer Gainstufe landet, die aus einer handelsüblichen, mit 8 Bit ansteuerbaren, programmierbaren Verstärkerstufe besteht. Der Wert der Verstärkung bezieht dieser Verstärker aus dem von der Logikeinheit separierten Exponentenwert.

Am Ende der Kette steht ein Invertierungsglied, das in Abhängigkeit vom Zustand des Sign-Bits das erzeugte Signal umkehrt. Dieser Block kann mittels eines simples Operationsverstärkers aufgebaut werden.

5.3 Diskussion

Um es vorweg zu nehmen: die Realisierung eines FP-D/A-Wandlers macht deutlich weniger Sinn als die eines FP-A/D-Wandlers, da bei der Erzeugung eines Ausgangssignals der Pegel ja bekannt ist und zur Not noch auf digitaler Seite angepasst werden kann. Diese Idee sei hier nur der Vollständigkeit halber angesprochen.

Der Vorteil bei einem solchen Wandler wäre jedoch ebenfalls ein gewaltiger Dynamikumfang, theoretisch handelt es sich wie bereits oben beschrieben Prinzip um mehr als 1500dB.

Wie gesagt – ein rein theoretischer Wert, denn man muss bedenken, dass allein, wenn man von einem Homerecordingpegel von 0.316 Volt ausgeht, ein Dynamikzuwachs von +100dB schon einen Ausgangspegel von einigen kV bedeuten würde.

Allerdings dürfte auch hier ein Clipping ausgeschlossen sein, vorausgesetzt, die Verstärkereinheit arbeitet zuverlässig in allen genutzten Aussteuerbereichen. Der Nutzbereich dieses Wandlers wäre nicht mehr durch die Bittiefe des Nachgeschalteten Wandlers allein bedingt sondern letztendlich durch die maximale Auslenkung der Lautsprechermembran und natürlich durch die Spannungsversorgung, mit der die Verstärkereinheit arbeiten kann. Dieses Limit wird den erwähnten theoretischen Dynamikumfang natürlich erheblich relativieren.

Probleme wird es beim Aufbau dieses Wandler an ähnlichen Stellen geben wie bei dessen A/D-Pendant: Es wird problematisch werden, einen Verstärker zu finden, der in allen Frequenzbereichen bei einer solch hohen Dynamik zuverlässig arbeitet, Rauschen und ein Dynamikverlust durch den minimierten Rauschabstand werden auch hier die Folge sein.

6. Fazit

Floating-Point wird in der digitalen Audiowelt bereits lange verwendet und ist zumindest als internes Datenformat längst akzeptiert.

Ich hoffe, mit dieser Arbeit einen Einblick in die Funktionsweise vermittelt zu haben und ein wenig zum Verständnis applikationsinterner Prozessabläufe beigetragen zu haben.

Es darf mit Spannung verfolgt werden, wann die erste marktreife Hardware für derartige D/A – A/D-Wandlung verfügbar sein wird und ob in nächster Zukunft auch Verbrauchermedien entwickelt werden, die dieses Datenformat speichern können.

Es bleibt abzuwarten, ob Floating-Point seinen Siegeszug in der digitalen Audioanwendung fortsetzen wird und ob der Schritt in Richtung Wandlertechnik eines Tages ernsthaft angegangen wird. Meines Erachtens könnte die konsequente Verwendung dieser Technologie von der Digitalisierung bis hin zur analogen Ausspielung in nächster Zukunft die verhängnisvollsten Probleme im Bereich der digitalen Audioverarbeitung relativieren.

Appendix A – Quellenangaben

A.1 Literatur

- 754-1985 IEEE Standard for Binary Floating-Point Arithmetic, ISBN: 1-5593-7653-8
- Floating point, Wikipedia encyclopedia, http://en.wikipedia.org/wiki/Floating_point
- “Floating Point Analog-to-Digital Converter“, Johan Piper, ISSN: 1402-8662, no. 47, 19.11.2004, University of Lund, Sweden
- “A 15-bit Pipelined Floating-Point A/D Converter”, Dwight U. Thompson und Bruce A. Wooley, Standford University
- “Delta-Sigma Data Coverters . Theory, design and simulation”, Steven R. Norsworthy, Richard Schreier, Garbor C. Temes, 1997, ISBN 0-7803-1045-4
- “Meaning of ARITHMETIC AND LOGIC UNIT” http://www.hyperdictionary.com/computing/arithmetic+and+logic+unit
- One pole LP and HP – http://www.musicdsp.org/archive.php?classid=3
- Fast Fourier transform, Wikipedia encyclopedia, http://en.wikipedia.org/wiki/Fast_Fourier_transform
- Discrete Fourier transform, Wikipedia encyclopedia, http://en.wikipedia.org/wiki/Discrete_Fourier_transform
- Finite impulse response filter, Wikipedia encyclopedia, http://en.wikipedia.org/wiki/Finite_impulse_response
- Infinite impulse response filter, Wikipedia encyclopedia, http://en.wikipedia.org/wiki/Infinite_impulse_response

A.2 Bilder

Alle in dieser Arbeit verwendeten Bilder, Diagramme sowie Programmbeispiele sind vom Autor selbst erstellt.

Appendix B – Quelltext

B.1 1-Pol-HP/LP-IIR-Filter

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

#define sample          int16_t
#define internal        int16_t
//#define sample        float
//#define internal      float
#define SCALE           (internal) ((1 << 15) - 1)


#define SAMPLERATE      8000            /* Hz */
#define N_SAMPLES       (SAMPLERATE/2)
#define CUTOFF          200.0           /* Hz */
#define FREQ1           4000.0          /* Hz */
#define FREQ2           10.0            /* Hz */

double coeff_omega = (2*M_PI*CUTOFF)/(double)SAMPLERATE;

double freqsin(double freq, int i) {
        return sin((freq*M_PI)*((double)i/SAMPLERATE));
}

void calc_hp(int n_samples, sample *in, sample *out) {
        internal acc = 0;
        internal coeff;
        int i;

        coeff = (internal)
            ((double) SCALE * ((2.0+cos(coeff_omega))
            - sqrt(pow(2.0+cos(coeff_omega), 2.0) - 1.0)));

        for (i=0; i<n_samples; i++) {
                acc = (((coeff-SCALE)*(internal) in[i])-(coeff*acc))
                        / (internal) SCALE;
                out[i] = -acc;
        }
}

void calc_lp(int n_samples, sample *in, sample *out) {
        internal acc = 0;
        internal coeff;
```

```
        int i;

        coeff = (internal)
            ((double)SCALE*((2.0-cos(coeff_omega))
            -sqrt(pow(2.0-cos(coeff_omega), 2.0) - 1.0)));

        for (i=0; i<n_samples; i++) {
                acc = (((SCALE-coeff)*(internal)in[i])
                  + (coeff*acc))/(internal)SCALE;
                out[i] = acc;
        }
}


int main(void) {
        sample in[N_SAMPLES];
        sample hp_out[N_SAMPLES], lp_out[N_SAMPLES];
        int i;

        for (i=0; i<N_SAMPLES; i++)
                in[i] = (sample) (0.5 * (double) SCALE *
                        (freqsin(FREQ1, i) + freqsin(FREQ2, i)));
        calc_hp(N_SAMPLES, in, hp_out);
        calc_lp(N_SAMPLES, in, lp_out);

        for (i=0; i<N_SAMPLES; i++)
                printf ("%i %g %g %g\n", i,
                                (double) in[i],
                                (double) hp_out[i],
                                (double) lp_out[i]);

        return 0;
}
```

Hiermit erkläre ich, dass die vorliegende Arbeit von mir selbständig erarbeitet wurde und alle übernommenen Textstellen anderer Autoren entsprechend gekennzeichnet sind.

_____ _____

Datum Unterschrift