1.5 Einführung und Zahlensysteme/Darstellung gebrochener Zahlen

1.5.1 Situation

Manchmal möchte man in Programmen mit Kommazahlen rechnen.

- In der Mathematik
- Im der Wirtschaft, im kaufmännischen Bereich (Geldbeträge)
- In der Physik und der Technik (physikalische Größen wie f oder R)

In den einzelnen Bereichen werden dabei unterschiedliche Genauigkeits-Ansprüche gestellt:

- Mathematik: Exaktheit
- Wirtschaft: gleichbleibende absolute Genauigkeit (z.B. auf 1ct genau)
- Physik: gleichbleibende relative Genauigkeit (z.B. bei Widerständen: $\pm 0,1\%$ vom Nennwert)

Auf diese unterschiedlichen Ansprüche kann und muss man bei der Programmierung eingehen. Für jeden dieser Ansprüche gibt es eine besondere Lösung.

1.5.2 Darstellung als Bruch

Eine Kommazahl kann man als Bruch darstellen: $Zahl = \frac{Zaehler}{Nenner}$. Mit dieser Darstellung kann man exakt rechnen. Allerdings ist die Arithmetik etwas kompliziert:

$$\frac{z_1}{n_1} + \frac{z_2}{n_2} = \frac{z_1 \cdot n_2 + z_2 \cdot n_1}{n_1 \cdot n_2} usw.$$

Diese Darstellung kann in den meisten Programmiersprachen ohne Problem erstellt werden, ist aber in der Regel nicht eingebaut[1].

1.5.3 Darstellung als Festkommazahl

Bei Festkommazahlen wird jede Zahl mit einer festen Anzahl von Nachkommastellen gespeichert. So kann man bei Geldbeträgen (in vielen Währungen) von zwei Nachkommastellen ausgehen. Die Zahl $5,99$ wird dann so wie die ganze Zahl 599 gespeichert, eine Zahl $0,99$ so wie die ganze Zahl 99. Addiert man die beiden Beträge, erhält man $599 + 99 = 698$. Bei der Ausgabe werden wieder die beiden Nachkommastellen wirksam, aus der Zahl 698 wird wieder $6,98$.

Man könnte also sagen, man rechnet die ganze Zeit in Cent statt in Euro, lediglich die Ein- und Ausgabe findet komfortablerweise in Euro mit zwei Nachkommastellen statt.

Bei Festkommazahlen ändert sich also nur die *Bedeutung* der gespeicherten Zahl, die Rechenregeln für Addition und Subtraktion bleiben dagegen gleich (bei Multiplikation und Division muss man um einen Faktor korrigieren).

1.5.4 Kommazahlen im Dualsystem

Für das Rechnen mit Kommazahlen im Dualsystem muss man sich zuerst überlegen, wie Kommazahlen im Stellenwertsystem überhaupt funktionieren:

- Kommazahlen im Dezimalsystem: $421,735 = 4 \cdot 100 + 2 \cdot 10 + 1 \cdot 1 + 7 \cdot (1/10) + 3 \cdot (1/100) + 5 \cdot (1/1000)$ Die Stellenwerte sind: 1000,100,10,1, nach dem Komma: 1/10,1/100,1/1000 usw.
- Kommazahlen im Dualsystem: $1011,101_{(2)} = 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 + 1 \cdot (1/2) + 0 \cdot (1/4) + 1 \cdot (1/8) = 11,625$ Die Stellenwerte sind: 8,4,2,1, nach dem Komma: 0,5, 0,25, 0,125 usw.

[1] Außer in speziellen Mathematikpaketen und Mathematiksprachen

1.5.5 Probleme mit Dezimalbrüchen im Dualsystem

Auch Kommazahlen kann man vom Dezimalsystem ins Dualsystem umrechnen:

- $9,5 = 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 1 \cdot (1/2) = 1001,1_{(2)}$
- $5,875 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 1 \cdot (1/2) + 1 \cdot (1/4) + 1 \cdot (1/8) = 101,111_{(2)}$
- $6,8 = 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 + 1 \cdot (1/2) + 1 \cdot (1/4) + 0 \cdot (1/8) + 0 \cdot (1/16) + 1 \cdot (1/32) + ... = 110,110011001100...._{(2)} = 110,\overline{1100}_{(2)}$

Bei vielen Zahlen, die im Dezimalsystem endlich sind, ist im Dualsystem keine Exaktheit möglich. Das liegt daran, dass fünftel, zehntel, $1/20$, $1/40$, $1/50$ usw. im Dualsystem unendliche periodische Zahlen sind. Zwei Abhilfen sind üblich:

- Darstellung als Bruch (s.o.) in der Mathematik
- BCD-Darstellung im kaufmännischen Bereich

1.5.6 BCD-Zahlen (ganze Zahlen und Festkomma)

BCD-Zahlen sind Dezimalzahlen. Allerdings wird jede Ziffer einzeln für sich im natürlichen Binärcode codiert. Hier ein Beispiel für die Zahl 4711:

$$4711 = 0100\,0111\,0001\,0001_{(2)}$$

Hier sieht man die Stellenwerte:

```
+----+----+----+----+---+---+---+---+--+--+--+--+-+-+-+-+
|8000|4000|2000|1000|800|400|200|100|80|40|20|10|8|4|2|1|
+----+----+----+----+---+---+---+---+--+--+--+--+-+-+-+-+
|   0|   1|   0|   0|  0|  1|  1|  1| 0| 0| 0| 1|0|0|0|1|
+----+----+----+----+---+---+---+---+--+--+--+--+-+-+-+-+
|                  4|               7|          1|      1|
+----+----+----+----+---+---+---+---+--+--+--+--+-+-+-+-+
```

Kommazahlen lassen sich so ebenfalls im Dezimalzahlen mit Binärziffern darstellen:

$$1,234 = 0001,0010\,0011\,0100_{(2)}$$

Hier sind wieder die Stellenwerte:

```
+-+-+-+-+---+---+---+---+----+----+----+----+-----+-----+-----+-----+
|8 4 2 1 0,8 0,4 0,2 0,1 0,08 0,04 0,02 0,01 0,008 0,004 0,002 0,001|
+-+-+-+-+---+---+---+---+----+----+----+----+-----+-----+-----+-----+
|0 0 0 1 0   0   1   0   0    0    1    1    0     1     0     0     |
+-+-+-+-+---+---+---+---+----+----+----+----+-----+-----+-----+-----+
|       1|              2|                   3|                       4|
+-+-+-+-+---+---+---+---+----+----+----+----+-----+-----+-----+-----+
```

Man erhält eine Reihe von Vorteilen:

- Exaktheit im Dezimalsystem gleich der Exaktheit beim schriftlichen Rechnen von Hand
- Einfache Umwandlung vom Dezimalsystem nach BCD und zurück, selbst mit einfachen TTL-ICs (SSI) möglich (z.B. TTL 7441, Kodierschalter, 7-Segment-Anzeigen mit integriertem BCD-Dekoder)

Daraus ergeben sich einige spezielle Anwendungen:

- kaufmännische Software (z.B. Programmiersprache COBOL)
- Taschenrechner (einfache Hardware)
- Automatisierungs-, Digitaltechnik ohne Prozessoren/Controller
- Spezielle Befehle in PC-Prozessoren (z.B. `DAA`, `DAS`)

1.5.7 Gleitkommazahlen

Beim Taschenrechner werden sehr große und sehr kleine Zahlen häufig im Gleitkommaformat dargestellt, und das aus gutem Grund:

- 1234567890 Ω — schwer abzuschätzen: Wie viele Ohm?
- $1,234567890 \cdot 10^9$ Ω = 1,234 GΩ — leicht abzuschätzen Die Zahl besteht aus Mantisse (1,234567890) und Exponent (9).
- 0,0015 m — wie viele Millimeter?
- $1,5 \cdot 10^3$ m — 1,5 Millimeter

Eine Gleitkommazahl besteht aus Mantisse (im ersten Beispiel 1,234567890) und Exponent (im ersten Beispiel 9), wobei beide positiv oder negativ sein können.

Die höchste Genauigkeit wird erreicht, wenn eine Gleitkommazahl *normalisiert* ist; dann steht eine Ziffer zwischen eins und neun vor dem Komma:

- $0,00000005 \cdot 10^7$ — ungünstig
- $5,29869876 \cdot 10^0$ — gleiche Zahl, viel genauer

Durch die Normalisierung wird die Mantisse am besten ausgenutzt. Im Dezimalsystem funktioniert Normalisierung so: Man schiebt die Mantisse so lange nach links (zum Komma hin), bis vor dem Komma keine Null mehr steht. Bei jedem Schritt muss man dabei den Exponenten um eins verringern:

- $0,00247 \cdot 10^4$ – sechs Stellen für die Mantisse erforderlich
- $0,0247 \cdot 10^3$ – fünf Stellen für die Mantisse erforderlich
- $0,247 \cdot 10^2$ – vier Stellenfür die Mantisse erforderlich
- $2,47 \cdot 10^1$ – drei Stellen für die Mantisse erforderlich

Bei einer Zahl mit großer Mantisse (≥ 10) geht man andersherum vor: Man schiebt die Mantisse so lange nach rechts, bis sie einstellig ist und erhöht bei jedem Schritt den Exponenten um eins.

- $615,8 \cdot 10^0$ – drei Stellen vor dem Komma
- $61,58 \cdot 10^1$ – zwei Stellen vor dem Komma
- $6,158 \cdot 10^2$ – eine Stelle vor dem Komma

Im Computer kommen Gleitkommazahlen meistens im Dualsystem vor:

- Dualsystem: Ziffern=0,1; Basis=2
- Beispiel: $1,01_{(2)} \cdot 2^{101_{(2)}} = 1,25 \cdot 2^5 = 1,25 \cdot 32 = 40$

Gleitkommazahlen haben einige Vorteile:

- betragsmäßig sehr große Zahlen sind darstellbar
- betragsmäßig sehr kleine Zahlen ebenfalls
- annähernd gleichbleibende relative Genauigkeit (z.B. 0,0005

Demgegenüber stehen auch Nachteile:

- Der Test auf Gleichheit zweier mathematisch gleicher Zahlen kann fehlschlagen:
  - 1000.0 verglichen mit $1000.0/3.0 \cdot 3.0$ kann beliebige Ergebnisse hervorrufen

– 1234567890 und 1234567894 können als Gleitkommazahlen für den Computer gleich sein (sie werden als float-Variable unter dem gleichen Bitmuster gespeichert).

Gleitkommavariable sollten deshalb niemals zur Programmsteuerung benutzt werden:

```
1 for(a=1234567890.0; a<1234567990.0; ++a)
2     printf("100mal"); // wirklich?
3 a=1234567890.0;
4 b=1234567894.0;
5 if(a==b)
6     printf("dieser Fall kann nie eintreten!\n"); // denkt man...
```

- Berechnungen mit Gleitkommazahlen sind vergleichsweise aufwändig und damit langsam.

1.5.8 IEEE-754

Das IEEE-Format ist ein Industriestandard für Gleitkommazahlen. Einige Formate entspringen der IEEE-754 (float, double, long double), andere wurden entsprechend dem IEEE-Standard von bestimmten Firmen eingebracht (INTEL: extended, SUN: extended double).

Das Format hat folgenden Aufbau:

$$Zahlenwert = (-1)^S \cdot 1, mmm \cdot 2^{eee-Offset}$$

Das Zeichen S steht für das Vorzeichenbit. Ist es eins, ist die Zahl negativ, sonst ist sie positv. *mmm* steht für die Bits der Mantisse, *eee* für die Bits des Exponenten. Der Offset beträgt beim Single-Format 127[2].

Aber warum ist die erste Ziffer der Mantisse mit eins festgelegt? Das liegt an der Normalisierung. Sie wird (s. o.) vorgenommen, um die Mantisse möglichst gut auszunutzen. Die Normalisierung hat zur Folge, dass die erste Ziffer der Mantisse nicht null ist; denn wenn sie null wäre, könnte man den Exponenten solange um eins verringern und die Mantisse mal zwei nehmen, bis die erste Ziffer eins ist:

- $0,0011 \cdot 2^6$
- $0,011 \cdot 2^5$
- $0,11 \cdot 2^4$
- $1,1 \cdot 2^3$

Im Dualsystem ist jetzt ein Trick möglich: Wenn sie nicht null ist, dann ist sie eins. Und wenn man das weiß, dann braucht man *diese* Eins nicht eigens zu speichern[3].

Einige Werte sind für spezielle Ergebnisse reserviert (E=Exponent, M=Mantisse):

- $E = 11111..111$, $M \neq 0$: `NAN` (z.B. Ergebnis von 0/0)
- $E = 11111..111$, $M = 0$: `INF` (z.B. Ergebnis von 3/0)
- $E = 00000..000$: betragsmäßig sehr kleine Zahlen; der Zahlenwert ist nicht normalisiert, damit man z.B. auch die Zahl null darstellen kann:

$$Zahlenwert = (-1)^S \cdot 0, mmmmmmm \cdot 2^{1-Offset}$$

Damit hat die Null auch hier ein Bitmuster aus lauter Nullbits.

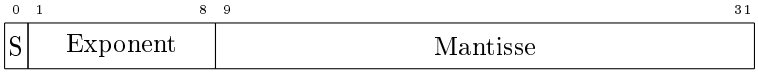
[2] Nicht 128, wie man denken sollte, aber so wollten es die IEEE-Leute eben.

[3] Bei den beiden längsten Formaten nach IEEE-754 (INTEL extended und SUN quadruple) wird dieser Trick übrigens nicht benutzt. Dort sind genug Bits für die Mantisse vorhanden; die Mantisse lautet dann $m, mmmmm$

1.5.9 Formate nach IEEE-754

Folgende Formate gibt es nach dieser Norm:

a) single-Format: 32 Bit (C/C++: float)

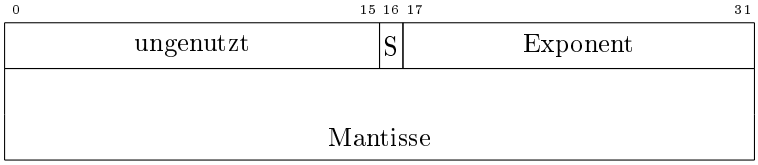


- 1 Bit Vorzeichen
- 8 Bit Exponent (Offset: 127)
- 23 Bit Mantisse (höchstwertiges Bit der Mantisse ist 1, wird nicht gespeichert)

b) double-Format: 64 Bit (C/C++: double)

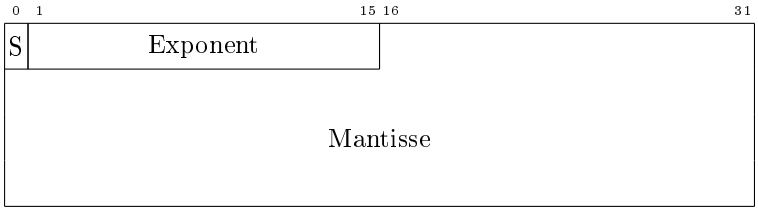
- 1 Bit Vorzeichen
- 11 Bit Exponent (Offset: 1023)
- 52 Bit Mantisse (höchstwertiges Bit der Mantisse ist 1, wird nicht gespeichert)

c) INTEL extended-Format: 96 Bit (C/C++: long double)



- 16 Bit ungenutzt
- 1 Bit Vorzeichen
- 15 Bit Exponent
- 64 Bit Mantisse (das höchstwertige Bit *wird* gespeichert)

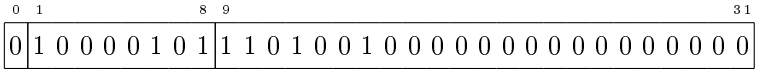
d) SUN quadruple-Format: 128 Bit (C/C++: long double)



- 1 Bit Vorzeichen
- 15 Bit Exponent
- 112 Bit Mantisse (das höchstwertige Bit *wird* gespeichert)

1.5.10 Beispiele zu IEEE-754

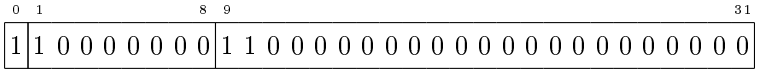
Im ersten Beispiel soll ein zu einem Bitmuster der Wert berechnet werden:



- `S=0`, also ist die Zahl positiv.
- Exponenten-Bits: $eee = 10000101_{(2)} = 128 + 4 + 1 = 133$
- Exponent: $E = eee - OS = 133 - 127 = 6$
- Mantissen-Bits: $mmm = 11010010000000000000000$
- Mantisse: $M = 1,1101001_{(2)} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{128} = 1,8203125$
- Wert: $W = +1,8203125 \cdot 2^6 = 116,5$

Ein zweites Beispiel soll zeigen, wie die Zahl $W = -3,5$ im single-Format nach IEEE-754 abgelegt wird:

- Die Zahl ist negativ, also ist das Vorzeichen-Bit $S = 1$.
- $|W| = 3,5 = 3\frac{1}{2} = 1 \cdot 2 + 1 \cdot 1 + 1 \cdot \frac{1}{2} = 11,1_{(2)}$
- Im Gleitkommaformat, ohne Normalisierung: $11,1_{(2)} \cdot 2^0$
- Im Gleitkommaformat, mit Normalisierung: $1,11_{(2)} \cdot 2^1$
- Mantisse: $M = 1,11$
- Zu speichernde Mantissen-Bits: $mmm = 11000000000000000000000$
- Exponent: $E = 1$
- Zu speichernde Exponenten-Bits: $eee = E + OS = 1 + 127 = 128 = 10000000_{(2)}$
- Gesamtzahl:



Man sieht, dass allein das Umwandeln von Gleitkommazahlen in ein bestimmtes Format sehr aufwändig ist. Genauso ist es beim Rechnen mit Gleitkommazahlen. Zwar macht das im Computer die CPU, entweder per Hardware (bei ausgewachsenen Rechnern) oder per Programm (bei Mikrocontrollern), so dass man als Programmierer selten damit zu tun hat. Aber es kostet die CPU viel Zeit. Daher benutzt man Gleitkommazahlen immer nur dann, wenn es von der Aufgabenstellung her nicht anders geht.