

2 Elementare Begriffe der Informatik

Einige für die Informatik zentrale Begriffe wie Nachricht und Information sowie Datentypen und Datenstrukturen werden in diesem Kapitel in kurzer und knapper Form eingeführt. Naturgemäß ist die Verarbeitung von (binär codierten) Zahlen eine der wichtigsten Aufgaben eines Rechners. Durch die Konvertierung von Dezimalzahlen und die endliche Menge der in einem Rechner darstellbaren Zahlen ergeben sich unter Umständen große Fehler. Deshalb wird in diesem Kapitel auch auf Zahlensysteme und die Darstellung von Zahlen in einem Rechner eingegangen.

2.1 Nachricht und Information

Bedingt durch die enormen Leistungssteigerungen in der Halbleitertechnologie erfolgt heute praktisch die gesamte Nachrichtenübertragung und Informationsverarbeitung in digitaler Form, ganz gleich ob es sich um Musik, Bilder, Messwerte, Personendaten o. Ä. handelt. Die digitale Nachrichtenübertragung erfolgt im Wesentlichen durch die kontinuierliche Übertragung von zwei diskreten Spannungswerten. Diesen werden z. B. die Werte 1 und 0 bzw. wahr und falsch zugeordnet.

Die kleinste mögliche Einheit der Information wird als Bit (Kurzwort aus *binary digit*) bezeichnet, da nur eine binäre Entscheidung zwischen zwei Möglichkeiten vorgenommen werden kann (hohe Spannung/niedrige Spannung, 1/0, ja/nein...). Aus praktischen Gründen wird eine Gruppe von zumeist 8 Bit zu einem Byte zusammengefasst (früher wurden auch Gruppen von 5, 6, und 7 Bit als Byte bezeichnet). Mit zwei Bits können vier und mit acht Bits, also einem Byte, bereits $2^8 = 256$ Entscheidungsmöglichkeiten codiert werden. Allgemein gilt, dass mit n Bits 2^n verschiedene Möglichkeiten codiert werden können.

Abbildung 2.1 zeigt schematisch einen Spannungsverlauf über der Zeit, welcher letztendlich die physikalische Grundlage einer Nachrichtenübertragung darstellt. Um die in der Nachricht enthaltene Information zu gewinnen, muss bekannt sein, wie diese Bitfolge zu interpretieren ist. Der dargestellte Signalverlauf soll exemplarisch dafür

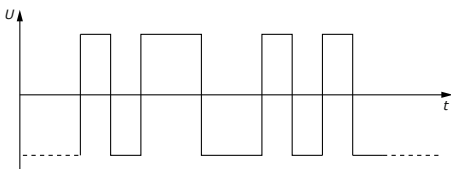


Abb. 2.1: Spannungsverlauf über der Zeit

verwendet werden, um diesen Zusammenhang zu veranschaulichen. Die vollständige Interpretation der Nachricht, nämlich die Übertragung des Buchstabens „Y“ über eine serielle RS232-Schnittstelle, zeigt Abbildung 2.2.

Zunächst muss bekannt sein, wie hoch der Spannungspegel ist, z. B. $\pm 12\text{ V}$. Den beiden Spannungspegeln wird eine 1 bzw. 0 zugeordnet. Bei der Interpretation wird zwischen positiver und negativer Logik unterschieden. Die positive Logik, die mit Ausnahme dieses Beispiels in diesem Buch verwendet wird, interpretiert eine 1 als logisch wahr und eine 0 als logisch unwahr. In der negativen Logik, die bei der Kommunikation zwischen Geräten aus schaltungstechnischen Gründen häufig bevorzugt wird, repräsentiert die 0 logisch wahr und die 1 logisch unwahr.

Darüber hinaus muss bekannt sein, mit welcher Frequenz das Signal gesendet wird, d. h. welche Zeitdauer ein Bit der Bitfolge aufweist. Im Beispiel ergibt sich bei einer Dauer von $104\text{ }\mu\text{s}$ eine Übertragungsrate von 9600 Bit/s . Schließlich muss auch bekannt sein, was übertragen wird, Zahlen, Zeichen, Musikdaten etc., wie diese gegebenenfalls in ein Übertragungsprotokoll eingebunden worden sind und wann die Übertragung beginnt und endet, wofür hier ein Start- und ein Stopp-Bit verwendet werden.

Eine Auswertung des Signalverlaufs kann vorgenommen werden, indem die Werte der acht Datenbits gewichtet werden (s. Abschn. 2.2) und dem resultierenden Zahlenwert das entsprechende Zeichen einer Code-Tabelle zugeordnet wird (s. Abschn. 2.3.3). In diesem Beispiel wird mit der Zahl 89 der Buchstabe „Y“ codiert.

Erst die Kenntnis darüber, wie eine Nachricht zu interpretieren ist, führt zur Information. Die in Programmiersprachen verwendeten Datentypen und Datenstrukturen stellen somit vor allem eine Interpretationsvorschrift für Bits bzw. Bytes dar. Eine weiter gehende Einführung in die beiden fundamentalen Begriffe Nachricht und Information der Informationstheorie gibt beispielsweise H. Ernst [3].

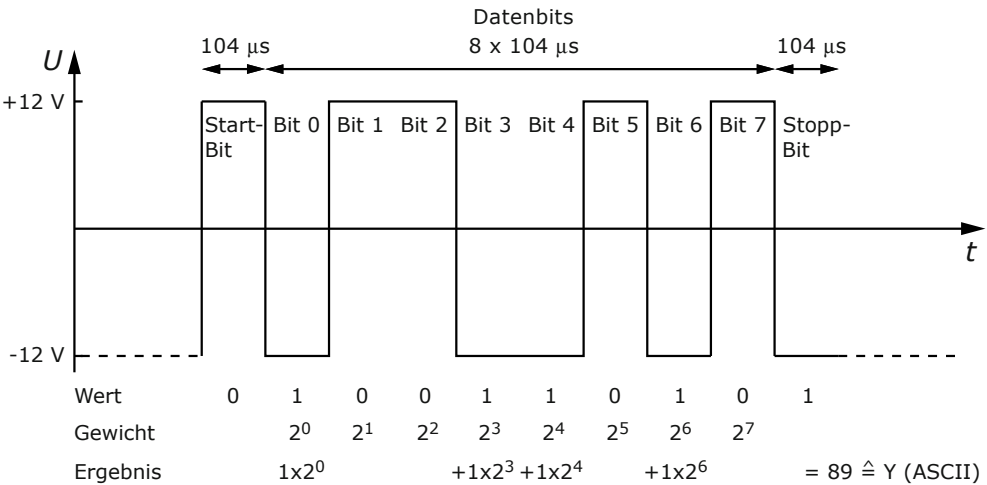


Abb. 2.2: Übertragung des ASCII-Zeichens Y über eine RS232-Schnittstelle

2.2 Zahlen und Zahlensysteme

Die Menge der natürlichen Zahlen $\mathbb{N} = \{1, 2, 3, \dots\}$ ist die elementarste Zahlenmenge, da Zahlen zunächst nur zum Zählen verwendet wurden, um beispielsweise Besitz zu ermitteln oder den Ablauf der Zeit zu bestimmen. Den meisten Zahlensystemen liegen die Basiszahlen 5, 10 und 20 zugrunde. Dies hat vor allem biologische Ursachen, da es zum Abzählen naheliegend ist, Gliedmaßen wie Finger und Zehen zur Hilfe zu nehmen. Zum Abzählen war die Null nicht erforderlich und die Zahl Null wurde erst im 13. Jahrhundert – gegen erbitterte Widerstände – in Europa eingeführt [4].

Mittlerweile wird die Zahl Null den natürlichen Zahlen $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$ zugeordnet. Diese Zuordnung wird u. a. in DIN 5473 (DIN = Deutsche Industrie-Norm) festgelegt. Sie ist die Ursache für einen häufig zu beobachtenden Fehler bei der Software-Entwicklung, da der Wert und die Position einer natürlichen Zahl nicht mehr übereinstimmen. So ist nun das erste Element der natürlichen Zahlen die Null, das zweite Element die Eins etc.

► Mathematische Symbole

Im Folgenden werden einige wenige, in der Mathematik übliche Symbole verwendet:

$\{a, b, c, \dots\}$ Menge mit den Elementen $a, b, c, \dots$

$\{a \mid \text{mit } \dots\}$ Menge der Elemente a mit der Eigenschaft $\dots$

$A \subset B$ A ist eine echte Teilmenge von B

$a \in X$ a ist Element der Menge X

$\sum_{i=1}^n a_i = a_1 + a_2 + a_3 + \dots + a_n$ ◀

Stellenwertsystem

Die Darstellung von Zahlen ist auf vielerlei Weisen möglich, üblich ist die Verwendung eines Stellenwertsystems (Gl. 2.1). Beim täglichen Umgang mit Zahlen wird stillschweigend vorausgesetzt, dass eine Zahl in einem Stellenwertsystem zur Basis 10, also als Dezimalzahl, dargestellt wird. Die Schreibweise 5394 ist dabei eine Abkürzung für

$$5394 = 5 \cdot 10^3 + 3 \cdot 10^2 + 9 \cdot 10^1 + 4 \cdot 10^0.$$

In allgemeiner Form kann eine Zahl in einem Stellenwertsystem wie folgt dargestellt werden:

$$N = a_m \cdot B^m + a_{m-1} \cdot B^{m-1} + \dots + a_1 \cdot B^1 + a_0 \cdot B^0 \quad (2.1)$$

$$= \sum_{i=0}^m a_i \cdot B^i \quad \text{mit: } 0 \leq a_i \leq B - 1. \quad (2.2)$$

Dabei ist B die Basis oder Grundzahl des Stellenwertsystems und die Koeffizienten a_i werden mit dem Wert der jeweiligen Stelle gewichtet, wobei die einzelnen Koeffizienten die Werte aus der Menge der Ziffern annehmen können. Im Dezimalsystem ist die Basis $B = 10$ und die Menge der Ziffern $\{0, \dots, 9\}$.

Rechenregeln für Dualzahlen

Da in einem Rechner zwei Zustände sehr einfach codiert werden können, ist es sinnvoll anstelle der Zahlendarstellung im Dezimalsystem eine binäre Zahlendarstellung zu verwenden. Im den meisten Fällen erfolgt die Codierung als Dualzahl im Stellenwertsystem mit der Basis $B = 2$ und der Menge der Ziffern $\{0, 1\}$. Dabei ist es vorteilhaft, dass nur wenige Rechenregeln bekannt sein müssen, weil Rechenschaltungen dann einfach in Hardware realisiert werden können (vgl. Kap. 3). Diese Rechenregeln unterscheiden sich natürlich nicht von den bekannten Regeln für Dezimalzahlen (Tab. 2.1).

Addition	Multiplikation	Tab. 2.1: Rechenregeln für Dualzahlen
$0 + 0 = 0$	$0 \cdot 0 = 0$	
$0 + 1 = 1$	$0 \cdot 1 = 0$	
$1 + 1 = 10$	$1 \cdot 1 = 1$	

In einem Beispiel soll die Addition von zwei Dualzahlen veranschaulicht werden:

	1 0 1 1 0 0 1 0	Summand a
+	0 1 0 1 1 0 1 0	Summand b
	1 1 1 1 1	Übertrag
	1 0 0 0 0 1 1 0 0	Summe

In Abschnitt 2.3.1 wird gezeigt, wie die Subtraktion von zwei Dualzahlen mit Hilfe des Zweierkomplements auf eine Addition zurückgeführt werden kann. Auch die Multiplikation von Dualzahlen erfolgt nach den gleichen Regeln wie die Multiplikation von Dezimalzahlen:

1 1 0 1 · 1 1 1 0	Multiplikand · Multiplikator
1 1 0 1 0 0 0	
1 1 0 1 0 0	
1 1 0 1 0	
0 0 0 0	
1 0 1 1 0 1 1 0	Produkt

Das Beispiel verdeutlicht unter anderem, dass bei einer Multiplikation der Multiplikator 10 das Bitmuster des Multiplikanden im Stellenwertsystem um eine Stelle nach links verschiebt (analog dazu verschiebt bei der Division der Divisor 10 das Bitmuster des Dividenden im Stellenwertsystem um eine Stelle nach rechts). Die Multiplikation von Dualzahlen lässt sich damit in einem Rechenwerk durch Verschiebungen und fortgesetzte Additionen realisieren.

Somit ist es prinzipiell möglich, Subtraktion, Multiplikation und Division (die im weiteren Verlauf nicht näher betrachtet werden soll) auf eine Addition zurückzuführen, wodurch der Entwurf eines Rechenwerks für Dualzahlen erheblich vereinfacht werden kann.

Konvertierung von Zahlen

Ein Nachteil der Zahlendarstellung im Dualsystem ist die relativ schlechte Lesbarkeit, da die Anzahl der Stellen im Vergleich zum Dezimalsystem wegen $10^3 \approx 2^{10}$ ca. 3 bis 4 mal so groß wird. Deshalb werden auch Zahlensysteme mit der Grundzahl 16 (Hexadezimal) und – eher selten – mit der Grundzahl 8 (Oktal) verwendet, weil die Konvertierung zwischen diesen Zahlensystemen sehr einfach vorgenommen werden kann. Tabelle 2.2 zeigt die ersten 16 natürlichen Zahlen für diese Zahlensysteme. Im Hexadezimalsystem werden dabei für die fehlenden Ziffern die Buchstaben A bis F verwendet.

dezimal	hexa- dezimal	oktal	dual
0	0	0	0 0 0 0
1	1	1	0 0 0 1
2	2	2	0 0 1 0
3	3	3	0 0 1 1
4	4	4	0 1 0 0
5	5	5	0 1 0 1
6	6	6	0 1 1 0
7	7	7	0 1 1 1
8	8	10	1 0 0 0
9	9	11	1 0 0 1
10	A	12	1 0 1 0
11	B	13	1 0 1 1
12	C	14	1 1 0 0
13	D	15	1 1 0 1
14	E	16	1 1 1 0
15	F	17	1 1 1 1

Tab. 2.2: Darstellung von Zahlen in unterschiedlichen Zahlensystemen

Die Konvertierung einer Zahl mit einer beliebigen Basis in eine Dezimalzahl kann mit Gleichung 2.1 vorgenommen werden. Beispielsweise ergibt sich für die Konvertierung einer Dualzahl:

$$11001101_2 = 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 205_{10}$$

und für die Konvertierung einer Hexadezimalzahl:

$$11111111_2 = FF_{16} = 15 \cdot 16^1 + 15 \cdot 16^0 = 255_{10}.$$

Anhand der Darstellung einer Zahl ist die zugrunde liegende Basis nicht unbedingt ersichtlich. Deshalb ist es unter Umständen sinnvoll, diese als Index bei der Zahlendarstellung zu verwenden. Besonders einfach ist die Umwandlung von Dualzahlen in Zahlen zur Basis 8 und 16 möglich (wegen $2^3 = 8$ und $2^4 = 16$), da jeweils Gruppen von 3 bzw. 4 Ziffern einer Dualzahl unabhängig voneinander konvertiert werden können.

Beispielsweise kann die Dualzahl 101010111100_2 ausführlich im Stellenwertsystem dargestellt werden (s. Gl. 2.1). Anschließend wird aus jeder Gruppe die größtmögliche Potenz von 2 ausgeklammert, die in eine Potenz von 8 umgewandelt werden kann. Für jede Zifferngruppe ergibt sich damit der Stellenwert zur Basis 8. Innerhalb eines Klammerausdrucks können nur die Zahlen 0 bis $111_2 = 7$ auftreten, also die Ziffern des Oktalsystems.

$$2748_{10} =$$

$$101010111100_2 =$$

$$\begin{aligned} & 1 \cdot 2^{11} + 0 \cdot 2^{10} + 1 \cdot 2^9 + 0 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = \\ & (1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)2^9 + (0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)2^6 + (1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)2^3 + (1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)2^0 = \\ & (1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)8^3 + (0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)8^2 + (1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)8^1 + (1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)8^0 \end{aligned}$$

Anhand dieser Darstellung wird deutlich, dass jeweils 3 Bit einer Dualzahl unabhängig voneinander betrachtet werden können:

$$\underbrace{1\ 0\ 1}_5 \underbrace{0\ 1\ 0}_2 \underbrace{1\ 1\ 1}_7 \underbrace{1\ 0\ 0}_4 = 5274_8 = 5 \cdot 8^3 + 2 \cdot 8^2 + 7 \cdot 8^1 + 4 \cdot 8^0 = 2748_{10}.$$

In analoger Weise kann eine Dualzahl in das Hexadezimalsystem umgewandelt werden, wobei nun jeweils 4 Bit zu einer Gruppe zusammengefasst werden:

$$\underbrace{1\ 0\ 1\ 0}_A \underbrace{1\ 0\ 1\ 1}_B \underbrace{1\ 1\ 0\ 0}_C = ABC_{16} = 10 \cdot 16^2 + 11 \cdot 16^1 + 12 \cdot 16^0 = 2748_{10}.$$

Die Umwandlung einer Dezimalzahl in eine Dualzahl kann u. a. durch eine fortgesetzte Division erfolgen. Dabei wird die Dezimalzahl durch die Grundzahl 2 dividiert. Es ergibt sich entweder ein Rest von 1 oder ein Rest von 0. Dieser Rest wird notiert. Anschließend wird das Ergebnis wieder durch 2 dividiert und wieder der Rest notiert. Dieses Verfahren wird fortgesetzt, bis eine 1 verbleibt, die ein letztes Mal dividiert wird, so dass eine 0 mit Rest 1 bleibt. Dieser Rest ist das erste, höchstwertige Bit der Dualzahl, welches auch als MSB (*most significant bit*) bezeichnet wird. Der zuerst ermittelte Rest ergibt das letzte, niedrigstwertige Bit der Dualzahl, welches auch als LSB (*least significant bit*) bezeichnet wird. Im folgenden Beispiel wird die Zahl 205_{10} in eine Dualzahl umgewandelt:

Damit ergibt sich $205_{10} = 11001101_2$. Im Prinzip eignet sich dieses Verfahren zur Konvertierung von Zahlen beliebiger Zahlensysteme, was durch die Gewöhnung an das Dezimalsystem aber möglicherweise erschwert wird, z. B.: $12_8 : 2_8 = 5_8$.

$$\begin{array}{rcl} 205 : 2 & = & 102 \text{ Rest } 1 \text{ LSB} \\ 102 : 2 & = & 51 \text{ Rest } 0 \\ 51 : 2 & = & 25 \text{ Rest } 1 \\ 25 : 2 & = & 12 \text{ Rest } 1 \\ 12 : 2 & = & 6 \text{ Rest } 0 \\ 6 : 2 & = & 3 \text{ Rest } 0 \\ 3 : 2 & = & 1 \text{ Rest } 1 \\ 1 : 2 & = & 0 \text{ Rest } 1 \text{ MSB} \end{array}$$

Zahlenmengen

Die Menge der natürlichen Zahlen $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$ ist für die Lösung vieler Gleichungen nicht ausreichend. In der Mathematik werden die Zahlenbereiche schrittweise erweitert, um jeweils mehr Gleichungen lösen zu können. Mit natürlichen Zahlen ist nur die Addition und Multiplikation immer ausführbar. Gleichungen wie $x + 2 = 1$ erfordern den Übergang zu den ganzen Zahlen

$$\mathbb{Z} = \{0, \pm 1, \pm 2, \pm 3, \dots\}. \quad (2.3)$$

Der Quotient von zwei ganzen Zahlen ist aber nicht in jedem Fall wieder eine ganze Zahl, wie das Beispiel $2x = 3$ zeigt. Erst die Erweiterung des Zahlenbereichs durch die rationalen Zahlen

$$\mathbb{Q} = \left\{ \frac{m}{n} \mid m, n \in \mathbb{Z}, n \neq 0 \right\} \quad (2.4)$$

ermöglicht immer die Division. Rationale Zahlen sind dadurch gekennzeichnet, dass sie entweder eine endliche Anzahl von Nachpunktstellen aufweisen, wie $1/2$ oder $3/5$, oder – gegebenenfalls nach einer Vorperiode – eine periodische Ziffernfolge, wie $5/6$ oder $1/17$.

Um Gleichungen der Art $x^2 = 2$ lösen zu können, ist die Erweiterung des Zahlenbereichs der rationalen Zahlen durch die irrationalen Zahlen erforderlich, wie z. B. $\sqrt{2}$, π . Die Menge der rationalen und irrationalen Zahlen ergibt die Menge der reellen Zahlen $\mathbb{R}$.

Eine letzte Erweiterung des Zahlenbereichs ist bedingt durch Gleichungen der Art $x^2 + 1 = 0$, die in der Menge der reellen Zahlen nicht gelöst werden können. Dafür ist der Übergang zu komplexen Zahlen notwendig

$$\mathbb{C} = \{a + i b \mid a, b \in \mathbb{R}\}, \quad (2.5)$$

die aus einem Real- und Imaginärteil zusammengesetzt werden, wobei „i“ die imaginäre Einheit ist.

Aus der schrittweisen Erweiterung des Zahlenbereichs resultiert die Enthaltenseinsbeziehung

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}, \quad (2.6)$$

die besagt, dass der jeweils umfassendere Zahlenbereich alle Elemente des weniger umfassenden Zahlenbereichs enthält. Die Darstellung von Zahlen mit Hilfe der Riemannschen Zahlenkugel ermöglicht es, zahlreiche mathematische Operationen anschaulich vorzunehmen [4].

Die Abbildung von Zahlenmengen in einem Rechner ist mit großen Einschränkungen verbunden. Wenn für die Darstellung der natürlichen Zahlen 1 Byte verwendet wird, können aus der unendlich großen Menge der natürlichen Zahlen $\mathbb{N}$ nur 256 Zahlen dargestellt werden. Für die Darstellung von reellen Zahlen steht nur eine im

mathematischen Sinne vernachlässigbar kleine Teilmenge der rationalen Zahlen zur Verfügung. Dies führt zum einen zu Bereichsüberschreitungen und zum anderen zu Rundungsfehlern, da die Menge der in einem Rechner darstellbaren Zahlen auf der Zahlengeraden viel mehr Lücken als Zahlen aufweist. Tatsächlich ergeben sich in der Praxis kaum Beschränkungen, wenn die Besonderheiten der numerischen Mathematik beachtet werden. Diese werden an einigen Beispielen im folgenden Abschnitt veranschaulicht.

2.3 Einfache und zusammengesetzte Datentypen

In allen Programmiersprachen werden unterschiedliche Datentypen verwendet. Dabei ist zwischen elementaren bzw. primitiven Datentypen und zusammengesetzten Datentypen, die auch als Datenstruktur bezeichnet werden, zu unterscheiden. Durch die Definition eines Datentyps wird festgelegt, wie ein Bitmuster zu interpretieren ist. Praktisch alle Programmiersprachen stellen zumindest vier Datentypen zur Verfügung:

- Logische Daten (*Boolean*) (1 Byte),
1 Bit wäre für die Codierung ausreichend, in der Regel ist 1 Byte aber die kleinste adressierbare Speichereinheit
- Zeichen (*Character*) (1 Byte)
- Ganzzahlen (*Integer*) (1, 2, 4 und 8 Byte),
für die Darstellung der natürlichen und ganzen Zahlen
- Gleitpunktzahlen (*Floating Point Numbers*) (4 und 8 Byte, zum Teil auch 10 Byte),
für die Darstellung der reellen Zahlen.

Dabei variieren die Bezeichnungen für die Datentypen und die Ausführungsformen unterscheiden sich geringfügig voneinander. Beispielsweise werden für die Darstellung von Zeichen in Java 16 Bit verwendet, in LabVIEW ist ein elementarer Datentyp für Zeichen gar nicht implementiert, da Zeichen bzw. Zeichenketten in der Datenstruktur *String* verwaltet werden und in C gibt es keinen Typ für logische Daten, dort werden logische Funktionen mit dem Datentyp für Zeichen realisiert. Die zugrunde liegenden Datenformate sind aber in der Regel vergleichbar.

Neben diesen vier Datentypen werden in Abhängigkeit von der jeweiligen Programmiersprache viele weitere Datentypen verwendet. Tabelle B.2 im Anhang gibt eine Übersicht über die Datentypen der Entwicklungsumgebung LabVIEW. Ihre Verwendungsmöglichkeiten werden in den folgenden Kapiteln aufgezeigt. Bei der Einführung von Datentypen ist es häufig üblich, auch die auf die Daten zulässigen Operationen einzuführen. Im Zusammenhang mit LabVIEW ist diese Form der Darstellung sicher nicht sinnvoll, da das Konzept darin besteht, dem Entwickler in Form von Bibliotheken möglichst viele Operationen bzw. Funktionen zur Verfügung zu stellen, so dass für einen Datentyp häufig einige hundert Funktionen zur Auswahl stehen.

In den folgenden Abschnitten werden zunächst die Datentypen für Ganzzahlen und Gleitpunktzahlen eingeführt und an Beispielen wird ihre Beschränkung bei der Darstellung und Verarbeitung aufgezeigt.

2.3.1 Ganze Zahlen

Vorzeichenlose Ganzzahlen

Für die Darstellung der natürlichen Zahlen werden üblicherweise 1, 2, 4 oder 8 Byte verwendet. Dabei ist die Bezeichnung nicht einheitlich. Die Datentypen werden als Byte, Wort, Doppelwort und Quad-Wort aber auch als Byte, Halbwort, Wort und Doppelwort bezeichnet. Unabhängig von der Bezeichnung ist das Datenformat aber immer gleich. Wie in Abbildung 2.3a dargestellt, beträgt die Wortbreite 8, 16, 32 und 64 Bit. Für die Darstellung dieser vorzeichenlosen Datentypen werden in der visuellen Programmiersprache LabVIEW graphische Symbole verwendet (Abb. 2.3b), die zudem farblich codiert sind. Für Ganzzahlen wird die Farbe Blau verwendet. Tabelle 2.3 gibt eine Übersicht über den Wertebereich von vorzeichenlosen Ganzzahlen (*Unsigned Integer*) in Abhängigkeit von der jeweiligen Wortbreite.

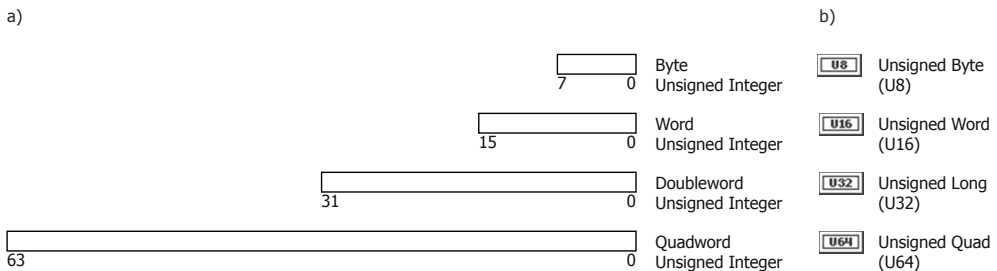


Abb. 2.3: a) Wortbreite von vorzeichenlosen Ganzzahlen, b) Darstellung in LabVIEW

Üblich ist die Darstellung von vorzeichenlosen Ganzzahlen im bereits vorgestellten Stellenwertsystem (s. Gl. 2.1, Tab. 2.2). Für eine 3-Bit-Zahl ergibt sich damit die Zuordnung von Dezimal- zu Dualzahlen zu:

dezimal	dual
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

Mit 3 Bit können also $2^3 = 8$ Dezimalzahlen codiert werden, verallgemeinert bedeutet dies, dass mit n Bit 2^n Zahlen codiert werden können. Prinzipiell kann die Zuordnung von Dezimalzahlen zu einem Bitmuster willkürlich vorgenommen werden. Je nach Anwendungsfall können die Dezimalzahlen in ihrer Reihenfolge vertauscht werden und es ist auch möglich, den Bitmustern andere Zahlenwerte zuzuordnen.

Vorzeichenbehaftete Ganzzahlen

Negative Ganzzahlen können in der bisher eingeführten Form nicht dargestellt werden. Um negative und positive Zahlen zu codieren, ist es erforderlich, ein Bit für die Codierung des Vorzeichens vorzusehen. Abbildung 2.4a gibt eine Übersicht über das Format vorzeichenbehafteter Ganzzahlen (*Signed Integer*) und Abbildung 2.4b zeigt die entsprechenden graphischen Symbole in LabVIEW. Die Wertebereiche für vorzeichenbehaftete Ganzzahlen sind in Tabelle 2.3 aufgeführt.

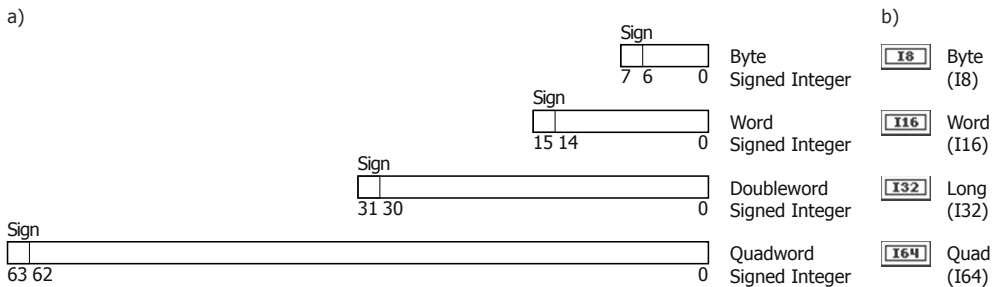


Abb. 2.4: a) Wortbreite von vorzeichenbehafteten Ganzzahlen, b) Darstellung in LabVIEW

Für die Darstellung vorzeichenbehafteter Ganzzahlen ist es zunächst naheliegend, die so genannte Vorzeichendarstellung zu verwenden, bei der im MSB die Information über das Vorzeichen abgelegt wird:

dezimal	binär
+0	0 0 0
+1	0 0 1
+2	0 1 0
+3	0 1 1
-0	1 0 0
-1	1 0 1
-2	1 1 0
-3	1 1 1

Bei dieser Codierung ergeben sich jedoch zwei Nachteile. Die Zahl Null ist zweimal vorhanden und es ist nicht möglich, positive und negative Zahlen ohne weitere Maßnahmen zu addieren. Diese Nachteile können vermieden werden, wenn für die Darstellung vorzeichenbehafteter Ganzzahlen das so genannte Zweierkomplement verwendet wird. Auch dabei wird das höchstwertige Bit für die Codierung des Vorzeichens verwendet; mit 1 wird eine negative und mit 0 eine positive Zahl gekennzeichnet. Das Ziel bei der Bildung des Zweierkomplements ist es, die Subtraktion zweier Zahlen $A - B$ auf eine Addition $A + (-B)$ zurückzuführen, da dann keine unterschiedlichen Schaltungen für die Addition und Subtraktion konstruiert werden müssen und sich der Schaltungsaufwand für die Realisierung eines Rechenwerks reduzieren lässt (s. Abschn. 3.2).

Um von einer positiven Zahl das Zweierkomplement zu bilden, wird zunächst das Einerkomplement gebildet, das heißt, alle Bits werden invertiert und anschließend wird

eine 1 addiert. Dieses Vorgehen wird im folgenden Beispiel veranschaulicht, indem von der Zahl 104_{10} das Zweierkomplement gebildet wird:

$$\begin{array}{rcl}
 104_{10} & = & 0\ 1101000 \\
 \text{Einerkomplement} & & 1\ 0010111 \\
 & + & 1 \\
 \text{Zweierkomplement} & = & 1\ 0011000 = -104_{10}
 \end{array}$$

Auf umgekehrtem Weg lässt sich auf analoge Weise der Betrag einer negativen Zahl ermitteln:

$$\begin{array}{rcl}
 \text{Zahl im} & & \\
 \text{Zweierkomplement } -104_{10} & = & 1\ 0011000 \\
 \text{Einerkomplement} & & 0\ 1100111 \\
 & + & 1 \\
 \text{Betrag} & = & 0\ 1101000 = 104_{10}
 \end{array}$$

Nach der Bildung des Zweierkomplements können auch vorzeichenbehaftete Ganzzahlen ohne weitere Maßnahmen addiert werden. In einem Beispiel soll $104_{10} - 99_{10} = 104_{10} + (-99_{10})$ berechnet werden. Für die Zahl $99_{10} = 01100011$ ergibt sich dabei das Zweierkomplement zu $-99_{10} = 10011101$ und die Addition liefert:

$$\begin{array}{r}
 0\ 1101000 \\
 + 1\ 0011101 \\
 \hline
 0\ 0000101 = 5_{10}.
 \end{array}$$

Dabei ist zu beachten, dass bei der Addition von Zahlen im Zweierkomplement immer eine feste Wortbreite, hier von 8 Bit, vorausgesetzt werden muss, was dazu führt, dass entstehende Überträge nicht berücksichtigt werden sondern, wie auch in diesem Beispiel, wegfallen. In einem weiteren Beispiel liefert die Addition von $99_{10} + (-104_{10})$

$$\begin{array}{r}
 0\ 1100011 \\
 + 1\ 0011000 \\
 \hline
 1\ 1111011 = -5_{10}.
 \end{array}$$

Die Überprüfung des Ergebnisses kann zum Beispiel dadurch erfolgen, dass von der negativen Zahl durch die Bildung des Zweierkomplements der Betrag ermittelt wird (s. o.).

Wertebereiche von Ganzzahlen

Bei einer Wortbreite n ergibt sich für vorzeichenlose Ganzzahlen ein Wertebereich von $0 \dots 2^{n-1}$ und für vorzeichenbehaftete Ganzzahlen von $-2^{n-1} \dots 2^{n-1} - 1$. Eine Übersicht über die Wertebereiche von Ganzzahlen mit den üblicherweise verwendeten Wortbreiten von 1, 2, 3 und 4 Byte gibt Tabelle 2.3.

Bereichsüberschreitungen bei der Verarbeitung von Ganzzahlen

Die begrenzte Wortbreite von Ganzzahlen kann bei der Addition oder Subtraktion zu einem Über- bzw. Unterlauf führen, der im Rechner aber wegen der begrenzten

Tab. 2.3: Wertebereiche von vorzeichenlosen und vorzeichenbehafteten Ganzzahlen

Wortbreite/Bit	Dualzahl	Zahl im Zweierkomplement
8	$2^0 \dots 2^8 - 1 = 0 \dots 255$	$-2^7 \dots 2^7 - 1 = -128 \dots 127$
16	$2^0 \dots 2^{16} - 1 = 0 \dots 65535$	$-2^{15} \dots 2^{15} - 1 = -32768 \dots 32767$
32	$2^0 \dots 2^{32} - 1$	$-2^{31} \dots 2^{31} - 1$
64	$2^0 \dots 2^{64} - 1$	$-2^{63} \dots 2^{63} - 1$

Wortbreite nicht berücksichtigt werden kann. Beispielsweise ergibt sich für eine 4-Bit-Dualzahl:

Überlauf:

$$\begin{array}{r} \boxed{1\,1\,1\,1} \\ + \\ \hline 1\,\boxed{0\,0\,0\,0} \end{array} \rightsquigarrow \boxed{0\,0\,0\,0}$$

Unterlauf:

$$\begin{array}{r} \boxed{0\,0\,0\,0} \\ - \\ \hline \dots 1\,\boxed{1\,1\,1\,1} \end{array} \rightsquigarrow \boxed{1\,1\,1\,1}$$

Nach der Verarbeitung von Ganzzahlen werden diese im Allgemeinen als Dezimalzahl angezeigt. Als Resultat ergibt sich dann:

	4-Bit-Dualzahl	4-Bit-Zahl im Zweierkomplement
Überlauf	$15 + 1 = 0$	$7 + 1 = -8$
Unterlauf	$0 - 1 = 15$	$-8 - 1 = 7$

► **Anmerkung**

Bereichsüberschreitungen oder Rundungsfehler führen zu (Un-)Gleichungen wie

$$127 + 1 = -128 \quad \text{oder} \quad 0.2 \cdot 10 \neq 2.$$

Das kleine Symbol eines Taschenrechners soll darauf hinweisen, dass diese (Un-)Gleichungen nicht im mathematischen Sinne gültig sind, sondern ausschließlich durch die Verwendung eines Rechners bedingt sind. ◀

Veranschaulichen lässt sich ein Über- bzw. Unterlauf, indem die zur Verfügung stehenden ganzen Zahlen auf einem Zahlenkreis angeordnet werden. Abbildung 2.5a zeigt eine 4-Bit-Dualzahl und Abbildung 2.5b eine Zahl in Zweierkomplementdarstellung. Wird der Zahlenkreis im Uhrzeigersinn durchlaufen, indem die Zahlen inkrementiert werden, ergibt sich ein Überlauf beim Übergang von 15 auf 0 bzw. beim Übergang von 7 auf -8 . Wird der Zahlenkreis gegen den Uhrzeigersinn durchlaufen, indem die Zahlen dekrementiert werden, ergibt sich ein Unterlauf beim Übergang von 0 auf 15 bzw.

beim Übergang von -8 auf 7 . Bei Rechenschaltungen werden Über- und Unterlauf in der Regel ausgewertet. Bei Programmiersprachen ist dagegen keine Auswertung vorgesehen und es ist die Aufgabe des Software-Entwicklers, einen Über- oder Unterlauf zu vermeiden.

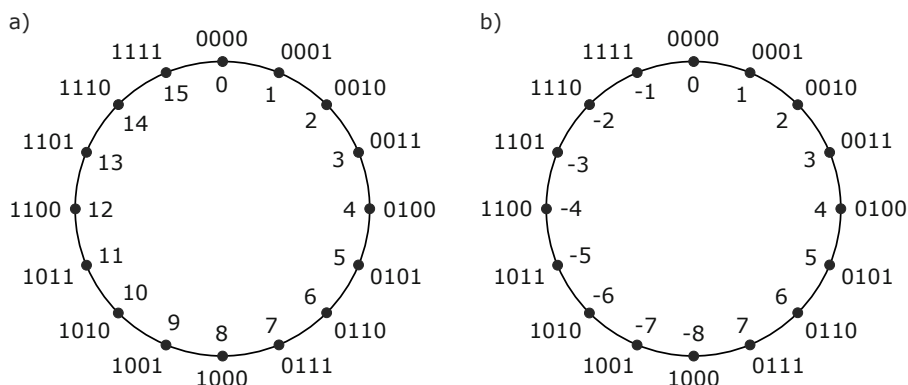


Abb. 2.5: Zahlenkreis, a) für eine Dualzahl und b) für eine Zahl im Zweierkomplement

In verallgemeinerter Form zeigt Tabelle 2.4 das Ergebnis für einen Über- bzw. Unterlauf, wobei n die Wortbreite der Ganzzahl angibt.

Tab. 2.4: Über- und Unterlauf bei vorzeichenlosen und vorzeichenbehafteten Ganzzahlen

	Dualzahl	Zahl im Zweierkomplement
Überlauf	$(2^n - 1) + 1 = 0$	$(2^{n-1} - 1) + 1 = -2^{n-1}$
Unterlauf	$0 - 1 = (2^n - 1)$	$-2^{n-1} - 1 = (2^{n-1} - 1)$

2.3.2 Gleitpunktzahlen

Um auch sehr große und sehr kleine reelle Zahlen mit hoher Genauigkeit darstellen zu können, wird im Allgemeinen eine Gleitpunktdarstellung bzw. halblogarithmische Darstellung verwendet. Diese ist direkt vergleichbar mit der wissenschaftlichen Notation eines Taschenrechners, bei der eine Gleitpunktzahl (*Floating Point Number*) F in der Form

$$F = \pm M \cdot B^{\pm E} \quad (2.7)$$

dargestellt wird. Dabei ist M die vorzeichenbehaftete Mantisse (*Mantissa* oder *Significand* für die Beschreibung der signifikanten Stellen der Gleitpunktzahl), E der vor-

zeichenbehaftete Exponent und B die Basis bzw. Grundzahl. Dabei bestimmt die Mantisse die Genauigkeit und der Exponent den Wertebereich der Zahl. Die Darstellung einer Zahl ist damit nicht mehr eindeutig. Die Dezimalzahl 72.4375 kann beispielsweise in den Formen

$$72.4375 \times 10^0 = 72437.5 \times 10^{-3} = 7.24375 \times 10^1$$

dargestellt werden und die Darstellung dieser Zahl als Gleitpunktzahl mit der Basis 2 kann in der Form

$$\begin{aligned} 1001000.0111 \cdot 2^0 &= (1 \cdot 2^6 + 1 \cdot 2^3 + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4}) 2^0 = \\ 1.0010000111 \cdot 2^6 &= (1 \cdot 2^0 + 1 \cdot 2^{-3} + 1 \cdot 2^{-8} + 1 \cdot 2^{-9} + 1 \cdot 2^{-10}) 2^6 = \\ 0.10010000111 \cdot 2^7 &= (1 \cdot 2^{-1} + 1 \cdot 2^{-4} + 1 \cdot 2^{-9} + 1 \cdot 2^{-10} + 1 \cdot 2^{-11}) 2^7 \end{aligned}$$

erfolgen. Die Erhöhung des Exponenten um 1 hat dabei eine Verschiebung des Dezimaltrennzeichens um eine Stelle nach links zur Folge und bei einer Erniedrigung des Exponenten um 1 verschiebt sich das Dezimaltrennzeichen dementsprechend um eine Stelle nach rechts. Bei einer Dualzahl kann die Verschiebung des Dezimaltrennzeichens so vorgenommen werden, dass vor dem Dezimaltrennzeichen eine 0 steht. Diese Form wird als normalisierte Darstellung bezeichnet. Eine Verschiebung, die dazu führt, dass vor dem Dezimaltrennzeichen eine 1 steht, wird dagegen als normierte Darstellung bezeichnet. Letztere wird im Allgemeinen bei der Darstellung von Gleitpunktzahlen im Rechner verwendet.

Darstellung von Gleitpunktzahlen im Rechner

Sowohl bei der Hardware-Entwicklung [5] als auch bei Programmiersprachen wird im Allgemeinen der Standard 754 des IEEE (*Institute of Electrical and Electronic Engineers*) verwendet, der das Datenformat für Gleitpunktzahlen mit einfacher (32 Bit), doppelter (64 Bit) und erweiterter Genauigkeit (≥ 80 Bit) spezifiziert (Abb. 2.6a). Die Darstellung von Gleitpunktzahlen in LabVIEW (mit der Farbe Orange) zeigt Abbildung 2.6b. Im Folgenden werden für die rechnerinternen Variablen Kleinbuchstaben verwendet, s für das Vorzeichen der Mantisse, f für die Nachpunktstellen der Mantisse und e für den Exponenten.

Auch bei Gleitpunktzahlen ist die Menge der darstellbaren Zahlen durch die endliche Wortbreite im Rechner begrenzt. Bei einer Wortbreite von beispielsweise 32 Bit können $\approx 4 \cdot 10^9$ Zahlen dargestellt werden, wobei ein Teil der Bitmuster für „besondere“ Zahlen wie 0, ∞ und so genannte Nichtzahlen reserviert wird (s. u.).

Die binäre Gleitpunktdarstellung ermöglicht zudem nur die Darstellung einer kleinen Teilmenge der reellen Zahlen, da sich der Wert der Nachpunktstellen der Mantisse nur durch das Aufsummieren von Kombinationen der Zahlen

$$\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16} \dots = 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4} \dots$$

zusammensetzen lässt. Somit können reelle Zahlen (beispielsweise die Dezimalzahlen 0.1, 0.2, 0.3, 0.4) nur näherungsweise dargestellt werden und die Zahlengerade weist

erhebliche Lücken auf. Der Abschnitt der Zahlengeraden zwischen zwei benachbarten Gleitpunktzahlen enthält eine unendlich große Menge reeller Zahlen. Dies hat zur Folge, dass das Ergebnis einer mathematischen Operation im Allgemeinen keine Gleitpunktzahl ist und daher auf die nächste Gleitpunktzahl gerundet werden muss.

Im Prinzip ist die Approximation von reellen Zahlen durch Gleitpunktzahlen für praktische Anwendungen ausreichend, da sich durch die verwendeten Datenformate ≈ 7 bzw. ≈ 16 signifikante Dezimalstellen ergeben. Erst durch die Anwendung mathematischer Operationen, wie zum Beispiel fortgesetzte Additionen oder trigonometrischer Funktionen, wie zum Beispiel $\sin x$, bei denen das Argument x weit außerhalb des Grundintervalls liegt, d. h. $x \gg 2\pi$, können Rundungsfehler zu unbrauchbaren Ergebnissen führen.

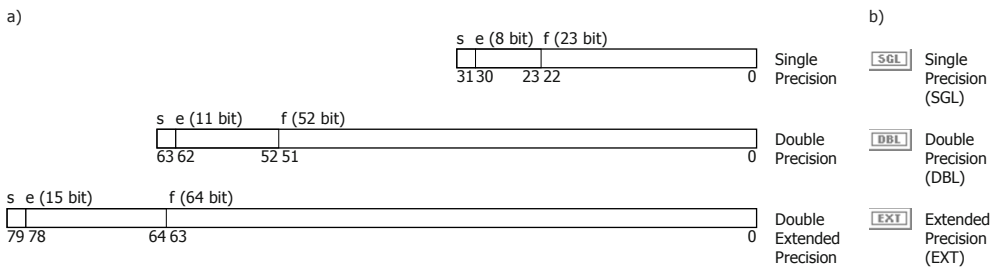


Abb. 2.6: a) Wortbreite von vorzeichenlosen Ganzzahlen, b) Darstellung in LabVIEW

Vorzeichen

Für die Codierung des Vorzeichens (*Sign*) s wird ein Bit benötigt. Mit $(-1)^s$ ergibt sich für eine 0 ein positives und für eine 1 ein negatives Vorzeichen.

Exponent

Um auch Zahlen < 1 darstellen zu können, ist ein vorzeichenbehafteter Exponent erforderlich, der durch eine Darstellung des Exponenten im Zweierkomplement realisiert werden könnte. Aus praktischen Gründen (s. u.) wird der Exponent E dagegen mit einem Versatz (*Bias*) b versehen, um die Null in die Mitte des zur Verfügung stehenden Zahlenbereichs zu verschieben. Der *Bias* b ergibt sich in Abhängigkeit von der Wortbreite p des Exponenten zu

$$b = 2^{p-1} - 1, \quad (2.8)$$

Beispielsweise wird $b = 127$ für einen Exponenten mit 8 Bit und $b = 1023$ für einen Exponenten mit 11 Bit Breite. Die rechnerinterne Darstellung des Exponenten e wird über

$$e = E + b \quad (2.9)$$

festgelegt. Für einen Exponenten mit einer Wortbreite von 8 Bit ergibt sich dann ein Wertebereich von $E_{min} = -127$ bis $E_{max} = 128$ und für einen Exponenten mit einer Wortbreite von 11 Bit von $E_{min} = -1023$ bis $E_{max} = 1024$. Der größte und kleinste Wert des Exponenten wird dabei jeweils für „besondere“ Zahlen reserviert (s. Tab. 2.6).

Da bei der Darstellung einer Gleitpunktzahl im Rechner erst der Exponent und anschließend die Mantisse abgelegt wird (vgl. Abb. 2.6) und bei beiden das MSB jeweils links und das LSB rechts angeordnet ist, können für Gleitpunktzahlen die gleichen Vergleichsoperationen wie für Ganzzahlen genutzt werden, wenn der Exponent mit einem *Bias* versehen wird.

Mantisse

Die Mantisse M wird immer in normalisierter Form dargestellt. Das heißt, der Punkt als Trennzeichen wird in der Mantisse verschoben, bis vor dem Punkt eine 1 steht. Die Mantisse M lässt sich damit in der Form

$$M = m_0 \cdot 2^0 + m_1 \cdot 2^{-1} + m_2 \cdot 2^{-2} + \dots \quad (2.10)$$

$$= 1 + m_1 \cdot 2^{-1} + m_2 \cdot 2^{-2} + \dots \quad (2.11)$$

$$= 1 + \sum_{i=1}^q m_i \cdot 2^{-i} \quad (2.12)$$

$$= 1.f \quad (2.13)$$

darstellen, wobei q die Wortbreite des *Fractional Part* f ist und für $m_i \in \{0, 1\}$ gilt. Durch die Normalisierung ist $m_0 = 1$. f steht für die Nachpunktstellen der Mantisse, für die $0 \leq f < 1$ gilt.

Da durch die Normalisierung immer eine 1 vor dem Dezimaltrennzeichen steht, wird sie nicht gespeichert (Ausnahme: erweiterte Genauigkeit), aber intern berücksichtigt, so dass die Genauigkeit der Mantisse immer um 1 Bit größer ist als f . Diese 1 wird auch als *Hidden Bit* (verstecktes Bit) bezeichnet.

► Anmerkung: Mantisse

Der Begriff Mantisse wird in der Mathematik für die Bezeichnung der Nachpunktstellen der Werte einer Logarithmentafel verwendet. In der Informatik hat es sich eingebürgert, den Begriff Mantisse auch im Zusammenhang mit Gleitpunktzahlen zu verwenden. Daher wird er auch hier verwendet. Dabei sollte beachtet werden, dass die Definition der Mantisse in der Literatur nicht einheitlich vorgenommen wird – so wie auch viele andere Begriffe in der Informatik nicht eindeutig und präzise verwendet werden. Die Darstellung der Mantisse erfolgt hier in der Form $M = 1.f$, das heißt, die 1 vor dem Dezimaltrennzeichen ist Bestandteil der Mantisse. ◀

Wert einer Gleitpunktzahl

Der Wert w einer Gleitpunktzahl kann über

$$w = (-1)^s \cdot (1.f) \cdot 2^{e-b} \quad (2.14)$$

ermittelt werden. Beispielsweise ergibt sich damit für die Dezimalzahl 1.5_{10} bei einer Darstellung im Rechner als Gleitpunktzahl mit einfacher Genauigkeit das Bitmuster:

$$\begin{array}{ccc} s & e & f \\ 1.5_{10} = & 0 & 01111111 \, 100000000000000000000000 \end{array}$$

und die Auswertung des Musters mit Gleichung 2.14 ergibt:

$$1.5_{10} = (-1)^0 \cdot (1 + 2^{-1}) \cdot 2^{127-127}.$$

Ein weiteres Beispiel zeigt das Bitmuster für die Dezimalzahl -2308.140625_{10} :

$$\begin{array}{ccc} s & e & f \\ -2308.140625_{10} = & 1 & 10001010 \, 00100000100001001000000, \end{array}$$

welches analog zum vorigen Beispiel ausgewertet werden kann:

$$-2308.140625_{10} = (-1)^1 \cdot (1 + 2^{-3} + 2^{-9} + 2^{-14} + 2^{-17}) \cdot 2^{138-127}.$$

Für Gleitpunktzahlen einfacher, doppelter und erweiterter Genauigkeit gibt Tabelle 2.5 eine Übersicht über die Wertebereiche und die Anzahl der signifikanten Stellen. In Tabelle 2.6 werden die Zahlenbereiche aufgeführt, die für „besondere“ Zahlen reserviert sind.

Tab. 2.5: Wertebereiche und signifikante Stellen von Gleitpunktzahlen

Datenformat	Wertebereich		Signifikante Stellen	
	binär	dezimal	binär	dezimal
einfach (32 Bit)	$2^{-126} \dots 2^{127}$	$\approx 10^{\pm 38}$	$23 + 1$	≈ 7
doppelt (64 Bit)	$2^{-1022} \dots 2^{1023}$	$\approx 10^{\pm 308}$	$52 + 1$	≈ 16
erweitert (80 Bit)	$2^{-16382} \dots 2^{16383}$	$\approx 10^{\pm 4932}$	64	≈ 19

Tab. 2.6: Besondere Zahlenbereiche von Gleitpunktzahlen

<i>Biased Exponent</i>	<i>Fractional Part</i>	Interpretation
$e = 0$	$f = 0$	± 0
$e = 2^p - 1$	$f = 0$	$\pm \infty$
$e = 2^p - 1$	$f \neq 0$	NaN (<i>Not a Number</i> (Nichtzahl))
$e = 0$	$f \neq 0$	nicht normalisierte Zahlen

Bei der normalisierten Darstellung ist es nicht möglich, die Zahl 0 darzustellen, da immer das *Hidden Bit* berücksichtigt werden muss. Die Codierung der Zahl 0 wird

dadurch realisiert, dass sowohl der Wert des *Biased Exponent* als auch der Wert des *Fractional Part* Null sind (s. Tab. 2.6). In Abhängigkeit vom Wert des Vorzeichenbits wird diese als $+0$ bzw. -0 interpretiert. Ein Bereichsüberlauf ist dadurch gekennzeichnet, dass der Wert des *Fractional Part* Null ist und der *Biased Exponent* den maximal möglichen Wert aufweist. In Abhängigkeit vom Wert des Vorzeichenbits wird eine Bitkombination dieser Art als $+\infty$ bzw. als $-\infty$ interpretiert. Bereichsüberschreitungen werden beispielsweise durch Operationen wie $1/0$ oder $\log 0$ verursacht.

Unzulässige Operationen wie $\log(-1)$ oder $\sqrt{-1}$ führen zu so genannten Nichtzahlen (*Not a Number, NaN*), für die die Bitkombinationen reserviert sind, bei denen der *Biased Exponent* den Maximalwert aufweist und der *Fractional Part* $f \neq 0$ ist. Ein Unterlauf ergibt sich, wenn nach einer Berechnung keine Normalisierung durchgeführt werden kann, weil der Wertebereich des Exponenten überschritten wird. Dann steht vor dem Trennzeichen eine 0. Das Bitmuster einer nicht normalisierten Zahl ist dementsprechend dadurch gekennzeichnet, dass $e = 0$ und $f \neq 0$ ist. Nach dem Auftreten eines Unterlaufs wird im Allgemeinen mit 0 weiter gerechnet.

Beispiele für die Addition von Gleitpunktzahlen mit vereinfachtem Datenformat

Um die Grenzen bei der Verarbeitung von Gleitpunktzahlen zu verdeutlichen, soll in den beiden folgenden Beispielen ein vereinfachtes Datenformat mit einem Bit für das Vorzeichen, 3 Bit für den Exponenten und 4 Bit für den *Fractional Part* der Mantisse verwendet werden (Abb. 2.7). Um das Verständnis zu erleichtern, wird der Exponent in den Beispielen *nicht* mit einem *Bias* versehen.

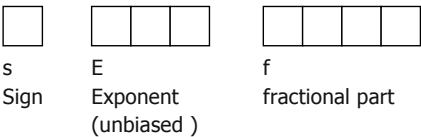


Abb. 2.7: Vereinfachter Datentyp mit Vorzeichenbit, 3 Bit-Exponent und 4 Bit *Fractional Part*

Im Beispiel nach Abbildung 2.8 sollen die beiden Zahlen 78_{10} und 34_{10} addiert werden. Abbildung 2.8a zeigt die mathematisch korrekte Lösung und Abbildung 2.8b die Addition der beiden Zahlen im vereinfachten Datenformat.

Um die beiden Dezimalzahlen addieren zu können (Abb. 2.8a), müssen sie zunächst in Dualzahlen umgewandelt werden. Anschließend werden sie in die normierte Darstellung gebracht, so dass vor dem Dezimalpunkt nur noch eine 1 steht. Um die beiden Zahlen (d. h. die Mantissen) addieren zu können, ist der Angleich der Exponenten erforderlich. Dies erfolgt, indem der Exponent der kleineren Zahl an den Exponenten der größeren Zahl angeglichen wird. In diesem Beispiel muss also der Exponent der Zahl 34 um 1 erhöht werden, was zur Folge hat, dass der Dezimalpunkt der Mantisse um eine Stelle nach links verschoben wird. Danach können die beiden Mantissen addiert werden.

Die Addition der beiden Zahlen im vereinfachten Datenformat hat zwei Rundungsfehler zur Folge (Abb. 2.8b), die durch die begrenzte Wortbreite des *Fractional Part* verursacht werden. Für den *Fractional Part* der Zahl 78_{10} ergibt sich nach der Normierung 0011100 ein Rundungsfehler. Weil die Wortbreite des *Fractional Part* 4 Bit

Dezimalzahl	Dualzahl	Normierte Darstellung	Exponentenangleich und Addition																
a) mathematisch																			
78	1001110.0	$1.0011100 \cdot 2^6$	$1.0011100 \cdot 2^6$																
+ 34	0100010.0	$1.0001000 \cdot 2^5$	$+ 0.1000100 \cdot 2^6$																
112			$1.1100000 \cdot 2^6 = 112$																
b) im vereinfachten Datenformat																			
	$s \quad E \quad f$		$s \quad E \quad f$																
	<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr></table>	0	1	1	0	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	0	0	1	1	<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr></table>	0	1	1	0				
0	1	1	0																
0	0	1	1																
0	1	1	0																
	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	0	1	0	1	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	1	$+ \quad$ <table border="1"><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr></table> <table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	1	0	1	0	0	0
0	1	0	1																
0	0	0	1																
0	1	1	0																
1	0	0	0																
			<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr></table> <table border="1"><tr><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>  = 108	0	1	1	0	1	0	1	1								
0	1	1	0																
1	0	1	1																

Abb. 2.8: Beispiel für die Addition von zwei Gleitpunktzahlen

beträgt, werden bei der Normierung auch nur die vier höchstwertigen Bits berücksichtigt:

$$0011100 \rightsquigarrow \boxed{0} \boxed{0} \boxed{1} \boxed{1} 100 \rightsquigarrow \boxed{0} \boxed{0} \boxed{1} \boxed{1}.$$

Im vereinfachten Datenformat wird die Zahl 34_{10} zunächst korrekt dargestellt. Ein Rundungsfehler ergibt sich durch den Exponentenangleich vor der Addition. Der Exponent muss dafür um 1 erhöht werden und dementsprechend wird der *Fractional Part* um eine Stelle nach rechts verschoben. Dadurch entfällt das LSB und auf der linken Seite des *Fractional Part* erscheint das *Hidden Bit* der Mantisse:

$$(1).\boxed{0} \boxed{0} \boxed{0} \boxed{1} \cdot 2^1 \rightsquigarrow \boxed{1} \boxed{0} \boxed{0} \boxed{0} 1 \rightsquigarrow \boxed{1} \boxed{0} \boxed{0} \boxed{0}.$$

Diese beiden Fehler führen dazu, dass sich nach der Umwandlung in eine Dezimalzahl des Ergebnis 108_{10} ergibt.

Prinzipiell können sich durch den Exponentenangleich Fehler ergeben und zu Gleichungen der Art

$$a + b = a \quad \text{für:} \quad a \neq 0, b \neq 0, \quad (2.15)$$

führen. Im Beispiel nach Abbildung 2.9 soll dies verdeutlicht werden, indem zur Zahl 34_{10} eine 1_{10} addiert wird.

Abbildung 2.9a zeigt wieder die mathematisch korrekte Lösung und Abbildung 2.9b die Addition im vereinfachten Datenformat. Das Vorgehen zur mathematisch korrekten Addition entspricht dem vorigen Beispiel, ebenso wie die normierte Darstellung der beiden Zahlen. Analog zum vorigen Beispiel wird die Zahl 1_{10} im vereinfachten Datenformat zunächst korrekt dargestellt. Es entsteht aber wieder ein Rundungsfehler beim Exponentenangleich. Der Exponent muss um 5 erhöht werden. Dementsprechend

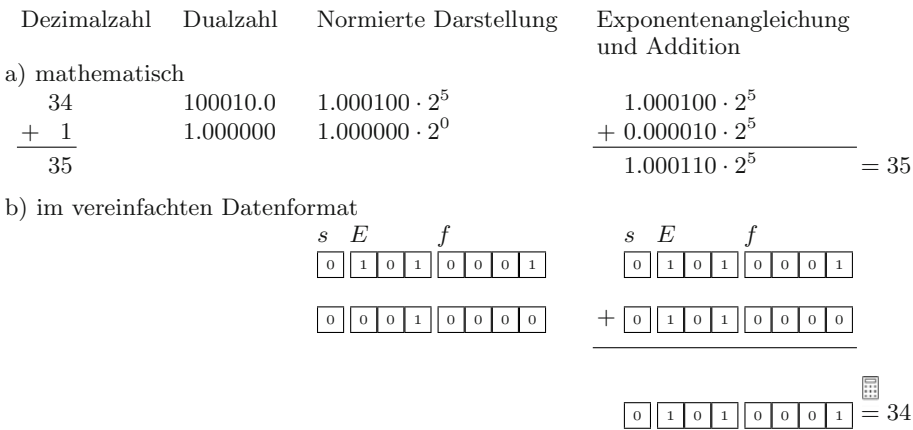


Abb. 2.9: Beispiel für die Addition von zwei Gleitpunktzahlen unterschiedlicher Größenordnung

wird der *Fractional Part* um fünf Stellen nach rechts verschoben. Dadurch entfällt das LSB und und auch das *Hidden Bit* der Mantisse wird nach rechts aus dem *Fractional Part* heraus geschoben:

(1).

0	0	0	1
---	---	---	---

 · 2⁵ ∼→

0	0	0	0	1
---	---	---	---	---

 ∼→

0	0	0	0
---	---	---	---

.

Als Folge der Verarbeitung von Gleitpunktzahlen ist das Distributivgesetz und das Assoziativgesetz der Algebra nicht mehr gültig, da sich die Reihenfolge, in der Zahlen addiert oder multipliziert werden, auf das Ergebnis der Berechnung auswirkt:

$$a \cdot (b + c) \neq (a \cdot b) + (a \cdot c) \tag{2.16}$$

und

$$a + (b + c) \neq (a + b) + c. \tag{2.17}$$

In Analogie zum vorigen Beispiel wird die Addition von drei Zahlen die Ergebnisse

$$1 + (1 + 34) = 34 \tag{2.18}$$

$$(1 + 1) + 34 = 36 \tag{2.19}$$

zur Folge haben, je nachdem, in welcher Reihenfolge sie addiert werden.

Beispiel: Harmonische Reihe

Dass die Reihenfolge der Addition mehrerer Zahlen das Ergebnis einer Berechnung beeinflussen kann, soll schließlich am Beispiel der Harmonischen Reihe gezeigt werden:

$$H = \sum_{i=1}^{\infty} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \cdots = \infty. \quad (2.20)$$

Da das Assoziativgesetz bei der numerischen Berechnung nicht mehr gültig ist, gilt:

$$\sum_{i=1}^n \frac{1}{i} \neq \sum_{i=n}^1 \frac{1}{i}. \quad (2.21)$$

Zudem ist es bei der Berechnung natürlich nicht möglich, eine unendliche Anzahl von Iterationen durchzuführen. Mit $n = 2^{31} - 1$, der maximal möglichen Anzahl von Iterationen in LabVIEW, ergibt sich bei einfacher Genauigkeit (*Single Precision*, SGL):

$$H_{SGL} = \sum_{i=1}^n \frac{1}{i} = 15.4036 \dots \quad \text{und} \quad H_{SGL} = \sum_{i=n}^1 \frac{1}{i} = 18.8079 \dots \quad (2.22)$$

Die große Diskrepanz der beiden Ergebnisse wird durch den Datentyp SGL mit einfacher Genauigkeit verursacht. Bereits nach $2^{21} - 1$ Iterationen ändert sich das Resultat bei aufsteigender Summierung nicht mehr. Das heißt, von $2^{31} - 1$ Iterationen tragen nur ca. 0.1% zum Ergebnis bei. Die Berechnung der harmonischen Reihe mit doppelter Genauigkeit (*Double Precision*, DBL) ergibt:

$$H_{DBL} = \sum_{i=1}^n \frac{1}{i} = 22.0647782623180788 \quad (2.23)$$

und

$$H_{DBL} = \sum_{i=0}^{n-1} \frac{1}{n-i} = 22.0647782620208481. \quad (2.24)$$

Die Berechnung führt praktisch nicht zu einer Verbesserung, denn obwohl sich die beiden Ergebnisse ähnlicher geworden sind und Rundungsfehler nun erst in der zehnten Nachpunktstelle auftreten, weichen die Ergebnisse durch die endliche Zahl der Iterationen trotzdem erheblich von der mathematisch korrekten Lösung ab.

Auslöschung

Ohne die Tatsache zu berücksichtigen, dass die Ergebnisse nach Gleichung 2.23 und 2.24 mathematisch nicht korrekt sind, lässt sich an ihnen ein weiteres Problem der numerischen Mathematik veranschaulichen. Wenn zwei annähernd gleich große Zahlen subtrahiert werden, verbleiben unter Umständen nur die mit Rundungsfehlern behafteten Stellen, ein Effekt, der auch als Auslöschung bezeichnet wird. Die Subtraktion der Ergebnisse nach den Gleichungen 2.23 und 2.24 ergibt:

$$\begin{array}{r} 22.0647782623180788 \\ -22.0647782620208481 \\ \hline 0.0000000002972307 \end{array}$$

Das Resultat enthält keine relevanten Stellen mehr und ist damit unbrauchbar. In ähnlicher Weise kann die Auslöschung von relevanten Stellen in der Messtechnik zu Fehlinterpretationen führen. In der Messtechnik wird praktisch jede physikalische Größe in eine analoge Spannung umgewandelt und anschließend digitalisiert. Das heißt, ein Messbereich von beispielsweise 0 V bis 10 V wird auf eine Menge diskreter Werte abgebildet, typischerweise auf 8 Bit, 12 Bit oder 16 Bit. Bedingt durch Rauschen und andere Störungen, die das Messsignal überlagern, sind die niedrigstwertigen Bits nicht nutzbar. Bei einem – weniger guten – 16 Bit-ADC (Analog-Digital-Konverter) können davon die vier (oder mehr) niedrigstwertigen Bits betroffen sein, so dass sich nach einer Subtraktion von zwei Messwerten beispielsweise

$$\begin{array}{r} 1010\ 1010\ 1010\ 1100 \\ -1010\ 1010\ 1010\ 0011 \\ \hline 0000\ 0000\ 0000\ 1001 \end{array}$$

ergeben kann. Das Resultat ist eine durch Rauschen generierte Zufallszahl.

Vergleiche von Gleitpunktzahlen

Abschließend soll darauf hingewiesen werden, dass bei der Software-Entwicklung auch der Vergleich von zwei Gleitpunktzahlen zu unerwünschten Ergebnissen führen kann, da sich nur wenige reelle Zahlen als Gleitpunktzahl darstellen lassen, beispielsweise gilt:

$$0.2 \cdot 10 \neq 2. \quad (2.25)$$

Vergleiche von Gleitpunktzahlen sollten daher niemals in der Form

$$a = b$$

sondern immer in der Form

$$a \leq b \quad \text{oder} \quad a \geq b$$

erfolgen. An einigen wenigen Beispielen sollten in diesem Abschnitt die Besonderheiten von Gleitpunktzahlen aufgezeigt werden. Eine tiefer gehende Analyse und weiterführende Literaturhinweise finden sich beispielsweise bei Donald E. Knuth [6].

2.3.3 Zeichen

Neben der Verarbeitung von Zahlen ist die Verarbeitung von Zeichen (*Character*) wie Buchstaben, Ziffern und Satzzeichen eine Standardaufgabe. Die Probleme, die sich dabei durch sprachspezifische Zeichen wie Umlaute ergeben, können noch auf die Zeit zurückgeführt werden, als Rechenleistung und Speicherplatz begrenzt waren. Um Speicherplatz optimal zu nutzen, wurden zunächst nur wenige Bits für die Codierung von Zeichen (und einigen Steuerzeichen) verwendet, d. h. einer Dualzahl wird ein

Zeichen zugeordnet und diese Zuordnung in einer Code-Tabelle abgelegt. In Abhängigkeit vom Wert der Dualzahl wird dann z. B. auf dem Bildschirm oder Drucker das zugehörige Zeichen ausgegeben, beispielsweise $01011001_2 = 59_{16} = Y$.

Mit 7 Bits können 128 Zeichen codiert werden, was in etwa dem Umfang der Zeichenmenge einer Tastatur entspricht. Diese Codierung wird auch als ASCII-Code (American standard code for information interchange) bezeichnet. Eine Erweiterung des ASCII-Codes auf 8 Bit mit 256 Zeichen ist in ISO/IEC 8859 (ISO: *International Organization for Standardization*, IEC: *International Electrotechnical Commission*) genormt. Da auch 8 Bit nicht für die vollständige Codierung von sprachspezifischen Zeichen ausreichen, enthält ISO/IEC 8859 insgesamt 16 Erweiterungen, z. B. ISO/IEC 8859-1 mit der Erweiterung *Latin-1* für westeuropäische Sprachen. Bei diesen Erweiterungen sind die ersten 128 Zeichen (0 ... 127) immer gleich, während die Zeichen 128 ... 255 für sprachspezifische Zeichen verwendet werden. Da bekannt sein muss, welche Erweiterung des Zeichensatzes verwendet worden ist, ergeben sich bei der Zeichendarstellung häufig Kompatibilitätsprobleme.

Zudem werden nicht bei allen Implementierungen die Normen vollständig berücksichtigt. So wird beispielsweise unter Windows ein Teil der Steuerzeichen durch andere Zeichen, wie z. B. €, ersetzt. Diese als PC-ANSI (ANSI: *American National Standards Institute*) bezeichnete Code-Tabelle zeigt Abbildung 9.18 im Zusammenhang mit einem Programmbeispiel.

Um die Kompatibilitätsprobleme zwischen unterschiedlichen Zeichensätzen zu vermeiden, werden in Unicode 16 Bit und in UCS 32 Bit (*universal character set*) für die Codierung von Zeichen verwendet. Insbesondere Letzterer bietet die Möglichkeit, alle bekannten Schriftzeichen eindeutig zu codieren. Beiden Erweiterungen ist gemeinsam, dass die ersten 128 Zeichen (0 ... 127) den ASCII-Code und die Zeichen 128 ... 255 die ASCII-Erweiterung *Latin-1* nach ISO/IEC 8859-1 enthalten, um die Abwärtskompatibilität sicher zu stellen. Eine Übersicht über die Codierung von Schriftzeichensätzen geben u. a. Rechenberg und Pomberger [7].

2.3.4 Boolesche Daten

Den einfachsten Datentyp stellen die booleschen Daten (*Boolean*) dar. Sie enthalten nur zwei Wahrheitswerte bzw. logische Werte {wahr, falsch} deren Darstellung unter Verwendung der positiven Logik im Allgemeinen mit den Symbolen

wahr: TRUE, T, 1, high, H,
falsch: FALSE, F, 0, low, L

vorgenommen wird. Die Verknüpfung der booleschen Werte mit logischen Funktionen (UND, ODER, NICHT) führt zur Booleschen Algebra (s. Kap. 3).

In LabVIEW wird die Darstellung boolescher Daten durch ein graphisches Element (in der Farbe Grün) mit der Beschriftung „TF“ (für *True/False*) vorgenommen (Abb. 2.10).



Abb. 2.10: Darstellung von booleschen Daten in LabVIEW

Boolesche Daten und ihre Verarbeitung sind von fundamentaler Bedeutung, da die Informationsverarbeitung unabhängig von den verwendeten Datentypen und -strukturen in einem Rechner immer auf der booleschen Algebra beruht, die die Basis für die schaltungstechnische Realisierung von Schaltnetzen und Schaltwerken ist. Deshalb werden diese Aspekte ausführlich in Kapitel 3 behandelt.

2.3.5 Datenstrukturen

Die bisher eingeführten elementaren Datentypen sind bei der Programmentwicklung im Allgemeinen nicht ausreichend. Häufig ist es erforderlich, aus den elementaren Datentypen komplexere Datenstrukturen zusammenzusetzen. Vor allem bestimmt die Wahl einer geeigneten Datenstruktur die Effizienz des Algorithmus. Den meisten Programmiersprachen gemeinsam sind die Datenstrukturen Zeichenkette, Datenfeld und Verbund.

Zeichenketten (Strings)

Um Texte verarbeiten und darstellen zu können, wird eine Reihung von Zeichen (*Character*) benötigt, die als Zeichenkette (*String*) bezeichnet wird. Diese ist im Prinzip nichts anderes als ein eindimensionales Datenfeld (s.u.), bei dem im Allgemeinen spezifiziert werden muss, wie viele Zeichen die Zeichenkette maximal enthalten kann.

In LabVIEW steht der elementare Datentyp (*Character*) nicht zur Verfügung, sondern nur die Datenstruktur *String*, weil die Größe einer Zeichenkette dynamisch verwaltet wird, so dass eine komfortable Programmierung ermöglicht wird. In LabVIEW erfolgt die Darstellung von *Strings* durch ein graphisches Element (in der Farbe Rosa) mit der Beschriftung „abc“ (Abb. 2.11). *Strings* werden im Zusammenhang mit der Entwicklungsumgebung LabVIEW in Abschnitt 8.3 ausführlich behandelt.



Abb. 2.11: Darstellung einer Zeichenkette (*String*) in LabVIEW

Datenfelder (Arrays)

Genauso wie in der Mathematik werden auch in der Informatik zusammengesetzte Datenstrukturen benötigt, um zum Beispiel Folgen, Vektoren oder Matrizen bearbeiten zu können. Diese Datenstruktur wird auch als Datenfeld *Array* bezeichnet. *Arrays* enthalten Daten eines Typs und können eine oder mehrere Dimensionen aufweisen.

Die übliche Schreibweise in der Mathematik kennzeichnet einzelne Elemente durch einen Index, zum Beispiel $N = n_1, n_2, n_3 \dots$. In der Informatik wird der Index eines Elementes meistens in ein Klammerpaar [...] eingefügt. Dabei ist zu beachten, dass die Indizierung bei Null beginnt. Für die Indices eines *Array* mit sechs Elementen ergibt sich also: $n[0] \dots n[5]$.

Grundsätzlich ist es notwendig, *Arrays* zunächst zu deklarieren. Um ein Element bearbeiten zu können, muss es möglich sein, dieses zu indizieren und ihm einen Wert zuweisen zu können:

- `n: array [0..5] of boolean`
deklariert ein leeres *Array* `n` mit sechs Elementen vom Datentyp *Boolesch*
- `n[2]`
indiziert das dritte Element des *Arrays* mit dem Index 2
- `n[3] ← TRUE`
weist dem vierten Element des *Arrays* mit dem Index 3 den Wert *TRUE* zu

Die Schreibweisen für diese Aufgaben werden so oder in ähnlicher Form in den meisten Programmiersprachen verwendet. Vergleichbar zur dynamischen Verwaltung von *Strings* können in LabVIEW auch *Arrays* dynamisch verwaltet werden. Bei der Deklaration ist es dann nicht zwingend erforderlich, die Größe des *Arrays* zu spezifizieren, wodurch die Programmentwicklung erheblich vereinfacht werden kann.

Abbildung 2.12 zeigt einige Beispiele für die graphischen Symbole eines *Arrays* in LabVIEW. Das Klammerpaar `[...]` innerhalb der Symbole deutet dabei in Anlehnung an textbasierte Programmiersprachen die Datenstruktur eines *Array* an. Im Zusammenhang mit der Entwicklungsumgebung LabVIEW wird die Deklaration und die Bearbeitung von *Arrays* in Abschnitt 9.1 eingehend behandelt.

a) 

b) 

c) 

Abb. 2.12: Darstellung eines *Array* in LabVIEW, a) leeres *Array*, b) *Array* mit booleschen Daten und c) *Array* mit numerischen Daten doppelter Genauigkeit

Verbund (Cluster)

Während in *Arrays* nur Daten eines Typs abgelegt werden können, bietet ein *Cluster* die Möglichkeit, Daten unterschiedlichen Typs zu verwalten. Ein Verbund wird in LabVIEW als *Cluster*, in C als *Struct* und in Pascal als *Record* bezeichnet. Diese Datenstruktur wird zum Beispiel erforderlich, wenn Kundendaten, mit Angaben wie z. B. Name, Adresse, Geburtsdatum und Kundennummer oder Messergebnisse, mit Angaben wie z. B. Messwerten und Messbedingungen, gespeichert werden müssen.

Das graphische Symbol für ein *Cluster* in LabVIEW ist in Abbildung 2.13 dargestellt. Bei der strukturierten Datenflussprogrammierung in LabVIEW ermöglicht diese Datenstruktur zudem, mehrere Datenflüsse in einer Verbindungsleitung zusammenzufassen, d. h. gewissermaßen zu bündeln, um die Lesbarkeit des Quellcodes zu gewährleisten. *Cluster* werden im Zusammenhang mit der Entwicklungsumgebung LabVIEW in Abschnitt 9.2 ausführlich behandelt.



Abb. 2.13: Darstellung eines *Cluster* in LabVIEW

Weitere Datenstrukturen

Je nach Programmiersprache steht eine Vielzahl weiterer Datenstrukturen zur Verfügung. Einige der wichtigsten sind:

- Warteschlange (*Queue*)
- Stapel (*Stack*)
- Liste (*List*)
- verkettete Liste (*Linked List*)
- Graph (*Graph*)
- Baum (*Tree*)

Im Zusammenhang mit der Entwicklungsumgebung LabVIEW werden diese Datenstrukturen nicht näher betrachtet; eine Einführung ist aber u. a. bei H. Ernst zu finden [3]. Abschließend soll angemerkt werden, dass diese Datenstrukturen grundsätzlich immer mit Hilfe der Datenstruktur *Array* realisiert werden können. In erster Näherung ist dies darauf zurückzuführen, dass letztendlich alle Datenstrukturen im Hauptspeicher eine Rechner's abgelegt werden müssen, dessen lineare Struktur mit einem *Array* vergleichbar ist. Eine differenziertere Betrachtung findet sich bei P. Pepper [8].

Im folgenden Kapitel werden zunächst die allen Datentypen und -strukturen zugrunde liegenden booleschen Daten und ihre logische Verknüpfung sowie Schaltnetze eingeführt.

Handbuch für die Programmierung mit LabVIEW
mit Studentenversion LabVIEW 2009

Mütterlein, B.

2009, XVIII, 517 S., Softcover

ISBN: 978-3-8274-2337-5