

Programmiertechnik 1

Unit 3: Informationsdarstellung

Ablauf

- Universalität binärer Daten
- Abtasttheorem
- ganze Zahlen, Binary Coded Decimals
- 1er Komplement
- 2er Komplement
- Gleitkommaformate
- ASCII, EBCDIC
- Unicode

Datenrepräsentation

- Daten
 - Aus Zeichen zusammengesetzte – Symbole
 - Zeichenmenge – Alphabet
 - Beispiele:
 - { 0, 1, 2, 3, ..., 9 } Alphabet der Dezimalziffern
 - { a, b, c, ..., A, B, C, ... } Alphabet der Buchstaben
 - {Frühling, Sommer, Herbst, Winter} Alphabet der Jahreszeiten
 - Frage: Welches ist das kleinste Alphabet?
 - Binärzahlen, bits – binary digits
 - Symbole aller denkbaren Alphabete lassen sich durch Gruppen von Binärzeichen ausdrücken.
 - Beispiel Kleinbuchstaben: $26 < 2^5$
 - a → 00000
 - b → 00001
 - c → 00010
 - ...

Morsecode
(Codierungen verschiedener Länge):

- a → . -
- b → - ...
- c → - . - .
- d → - . -
- e → .

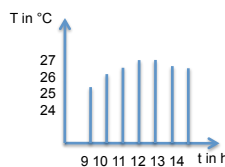
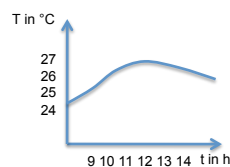
Andreas Polze

Unit 3: Informationsdarstellung, Programmiertechnik 1

3

Universalität binärer Daten

- Zuordnungsvorschrift zwischen Alphabeten → Code
- Codierung eines Alphabets durch Binärzahlen → Binärcodierung
- Stetige Größen:
 - Rasterung, Diskretisierung
 - Bei genügend großem Wertebereich bleibt Fehler unter Wahrnehmungsgrenze
 - alles, was zählbar / meßbar ist, kann codiert werden
 - Alles, was codiert werden kann, kann der Rechner verarbeiten
 - Codierung im Rechner ist immer binär
- Wieviele Binärzeichen sind für die Darstellung eines Zeichens aus dem Alphabet nötig?



9h	25°C
10h	25,5°C
11h	26°C
12h	27°C
13h	26,5°C
14h	26°C

Andreas Polze

Unit 3: Informationsdarstellung, Programmiertechnik 1

4

Nyquist-Shannon-Abtasttheorem

- Grundlegendes Theorem der Nachrichtentechnik
- Theorie der maximalen Kanalkapazität
- Das Abtasttheorem besagt:
 - ein kontinuierliches, bandbegrenztes Signal
 - mit einer Minimalfrequenz von 0 Hz und einer Maximalfrequenz f_{\max}
 - Tastet man dieses Signale mit einer Frequenz größer als $2 * f_{\max}$ gleichförmig ab, so lässt sich aus dem zeitdiskreten Signal das Ursprungssignal rekonstruieren
 - Exakt (ohne Informationsverlust) – mit unendlich großem Aufwand (d.h. unendlich viele Abtastpunkten)
 - Beliebig genau approximiert – mit endlichem Aufwand
- Höhere Abtastfrequenz liefert keine zusätzlichen Informationen
 - Der Aufwand für Verarbeitung, Speicherung und Übertragung steigt jedoch
 - Analoger Tiefpassfilter u.U. einfacher

Zahlendarstellungen

- Darstellung als Text
 - Üblich, wenn überhaupt keine Berechnung stattfinden soll
 - „Die Regionalbahn 18671 fährt um 15:00 in Golm ab.“
 - Die Zahlen 18671, 15 und 0 werden als Textzeichen repräsentiert
 - Speicheraufwand: 1 Byte pro Dezimalziffer
- Variante: BCD (binary coded decimal)
 - Kodierung von 10 Ziffern: 4 Bit pro Ziffer
 - 2 Ziffern pro Byte
- Beispiel: 18671 kodiert als 0001 1000 0110 0111 0001 (Hex: 01 86 71)
 - effizienter als Textdarstellung (0,5B pro Ziffer)
 - leicht in Dezimaldarstellung umrechenbar
 - arithmetische Operationen aufwändiger in Hardware realisierbar

Dezimale Zahlendarstellung

- Binary-Coded Decimal (BCD) kann 0 bis 9 in 4 binären Bits darstellen.
- Die Arithmetik basiert auf Modulo-10-Rechnung.
 - Da 4 bit = $2^4 = 16$ sind, werden bei der Addition Zahlen, die größer als 9 sind durch Addition von 6 justiert: 6 (= 0110), ($16-10 = 6$).
 - Kodierung des Vorzeichens: 1010 steht für "+" und 1011 für "-"
- Beispiel: $4739 + 1281 = 6020$

```

– in BCD-Code:   0100 0111 0011 1001           = 4739
                  + 0001 0010 1000 0001       = 1281
Zwischenresultat: (0101 1001 1011 1010)
– Die Werte müssen angepasst werden indem man durchlaufend 0110 addiert:
0101 1001 1011 1010
                        1 0110
0101 1001 1100 0000
                        1 0110
0101 1010 0010 0000
                        1 0110
0110 0000 0010 0000 = 6020
6  0  2  0

```

Ganze Zahlen

- Binärdarstellung
- Ziel: Ausschöpfung aller Bitkombinationen
 - Mit N Bits sollen 2^N Zahlen repräsentiert werden
 - üblich: Zahlen von $0 .. 2^N-1$ (nichtnegative Zahlen, *unsigned*)
 - Zahlen von $-2^{N-1} .. 2^{N-1}-1$ (ganze Zahlen, *signed*)
 - Darstellung reeller Zahlen wird später diskutiert
- Binärdarstellung: Bitfolgen werden im Binärsystem interpretiert
 - Bitfolge aus Null-Bits stellt die kleinste Zahl dar (0)
 - Bitfolge aus Eins-Bits stellt die größte Zahl dar (2^N-1)

Speicherung von Mehrbyte-Zahlen

- Bitpositionen innerhalb eines Bytes fest in Hardware verdrahtet
 - etwa: Bits 0 .. 7
- Bitpositionen in Mehrbytezahlen abhängig von Prozessorarchitektur
- Little-endian: geringstwertiges Byte „zuerst“ im Speicher (auf kleinster Adresse)
 - Beispiel: Intel x86
- Big-endian: geringstwertiges Byte „zuletzt im Speicher“
 - Beispiel: Motorola/IBM PowerPC
- Middle-endian:
 - Beispiel: DEC PDP-11

0 1 2 3 4 5 6 7

7 6 5 4 3 2 1 0

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7

18.03.2012

2012/03/13

Arithmetische Operationen

- Addition analog dem Dezimalsystem
 - $$\begin{array}{r} 10010 \\ +100111 \\ \hline 111001 \end{array}$$
- Allgemeiner:

Rechenoperationen auf Basis von Assoziativ- und Distributivgesetzen

 - Beispiel: Multiplikation
$$a_2a_1a_0 * b_2b_1b_0$$
 - $(4a_2+2a_1+a_0) * (4b_2+2b_1+b_0) =$

$$\begin{array}{l} 16a_2b_2 \\ +8(a_2b_1+a_1b_2) \\ +4(a_2b_0+a_1b_1+a_0b_2) \\ +2(a_1b_0+a_0b_1) \\ +a_0b_0 \end{array}$$

Überlauf

- Darstellbar sind Zahlen von $0.. 2^N-1$
- Überlauf: Ergebnis ist größer als 2^N
- Beispiel: $11+8$, 4-Bit-Zahlen


```

        1 0 1 1
      + 1 0 0 0
      -----
      1 0 0 1 1
      
```

 - Ergebnis wieder in 4 Bit: $11+8 \Rightarrow 3$
- Prozessor erkennt den Überlauf, zeigt ihn in „carry“-Bit an
 - abhängig von Programmiersprache wird carry-Bit verworfen oder führt zu einer Programmausnahme (exception)
- Verwerfen des Carry-Bits: Alle Operationen werden modulo 2^N ausgeführt

Vorzeichenbehaftete Zahlen

Drei Repräsentationen:

- Vorzeichen und Wert (sign and magnitude):



- 1'er-Komplement
 - $\overline{N} = (2^n-1) - N$ (Negation aller bits)
- 2'er Komplement
 - $\overline{N}^* = 2^n - N = (2^n-1) - N + 1 = \overline{N} + 1$ (Negation plus Eins)
 - Eindeutige Darstellung des Wertes 0
- Alle Rechner verwenden heute Zweier-Komplementdarstellung

Sign and Magnitude

- Repräsentation der Zahl durch zusätzliches Vorzeichenbit (0: Zahl ist nichtnegativ, 1: Zahl ist negativ)
 - bei N Bits stehen N-1 Bits für die Zahl zur Verfügung
 - Beispiel (4 Bit)
 - 0000 = +0, 0100 = +4, 1000 = -0, 1100 = -4
- Darstellbare Zahlen: $-(2^{N-1}-1) \dots 2^{N-1}-1$ (insgesamt 2^N-1 Zahlen)
 - negative Zahlen erkennt man am vordersten Bit
- Problem: Rechenoperationen sind kompliziert durchzuführen (Addition, Subtraktion: 4 verschiedene Fälle)
- Problem: zwei verschiedene Darstellungen der Zahl 0

Einer-Komplement

- negative Zahlen entstehen durch bitweise Negation aus den positiven Zahlen ($-x = \bar{x}$)
 - Beispiel (5 Bit) 01001 = 9, 10110 = -9
- Darstellbare Zahlen: $-2^{N-1}-1 \dots 2^{N-1}-1$ (insgesamt 2^N-1 Zahlen)
 - negative Zahlen erkennt man am vordersten Bit
 - gleiche Probleme wie bei Vorzeichendarstellung
- Arithmetische Operationen sind schwierig, zwei Darstellungen der 0 (000000 = +0, 111111 = -0)

Zweier-Komplement

- Negative Zahl durch Subtraktion von 2^N ($-x = 2^N - x$)
- 2^N ist außerhalb des darstellbaren Bereichs; $2^N \equiv 0 \pmod{2^N}$
 - Alternative Bildungsregel: $-x = \bar{x} + 1$
 - Beispiel (4 Bit): $0011 = 3$, $-3 = \overline{0011} + 1 = 1100 + 1 = 1101$
 - negative Zahlen erkennt man am vordersten Bit
- Darstellbare Zahlen: $-2^{N-1} \dots 2^{N-1}-1$ (insgesamt 2^N Zahlen)
- Beispiel: 8 Bit: darstellbar sind die Zahlen von $-128 \dots 127$
- Addition negativer Zahlen: entsprechend der Addition vorzeichenloser Zahlen
 - sowohl Überlauf als auch Unterlauf möglich
 - Subtraktion: Addition des negierten Werts ($a-b = a+(-b)$)
- Problem: kleinste negative Zahl lässt sich nicht negieren

Basiswert-Darstellung

- Darstellung negativer Zahlen durch Basiswert (bias)
- Bits werden zunächst als nichtnegative Zahl interpretiert
 - danach wird ein Versatz V subtrahiert
 - darstellbarer Bereich: $-V \dots 2^N - V - 1$ (insgesamt 2^N Zahlen)
- Beispiel (4b, Versatz 7): $0000 = -7$, $0111 = 0$, $1000 = 1$, $1111 = 8$
- Problem: Vorzeichen der Zahl nicht leicht erkennbar
 - Problem: 0 ist nicht durch eine Bitfolge von Nullbits repräsentiert
- Wird verwendet. als Teil der Gleitkommadarstellung

Binäre Zahlendarstellung

Bitmuster $b_3b_2b_1b_0$	Vorzeichen und Wert	Wert	
		1'er Komplement	2'er Komplement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	0	+0	+0
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

Andreas Polze

Unit 3: Informationsdarstellung, Programmiertechnik 1

17

Übliche Wertebereiche

- Größe von Datentypen abhängig von Programmiersprache
 - C: $\text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \leq \text{sizeof}(\text{long long})$
 - Beispiel: gcc, Linux, x86: short=2B, int=4B, long long=8B
 - Java: plattformunabhängig: byte=1B, short=2B, int=4B, long=8B
 - C Datentypen:
 - char (n = 8) -128..127
 - short (n = 16) -32768..32767
 - int, long (n = 32) $-2^{31}..2^{31}-1$
-2147483648.. 2147483647
 - long long (n = 64) $-2^{63}..2^{63}-1$
-9223372036854775808.. 9223372036854775807
- Datentypen als programmiersprachliches Konzept verbergen die zugrundeliegende Rechnerarchitektur
→ Portabilität

Andreas Polze

Unit 3: Informationsdarstellung, Programmiertechnik 1

18

Große Ganze Zahlen

- Idee: Aufhebung des Wertebereichs durch Verwendung einer variablen Anzahl Bytes
 - größerer Absolutwert \Rightarrow mehr Bytes
- Wertebereich nur durch Hauptspeicher eingeschränkt
- Beispiele: BigInteger in Java, long in Python
- arithmetische Operationen nicht durch Prozessor unterstützt

Reelle Zahlen

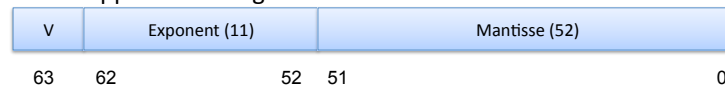
- überabzählbar unendlich viele reelle Zahlen von 0..1
 - \Rightarrow nur endlich viele Zahlen mit endlichem Speicher unterscheidbar
- Variante 1: Darstellung rationaler Zahlen durch Zähler und Nenner
 - etwa: Zähler ist ganze Zahl, Nenner ist natürliche Zahl
 - Problem: elementare Rechenoperationen führen leicht aus dem Bereich der darstellbaren Nenner heraus
- Variante 2: Darstellung mit festem Nenner (etwa: 100)
 - Festkommazahlen
 - arithmetische Operationen müssen auf nächste darstellbare Zahl runden
- Verbreitet in Finanzmathematik
 - Beispiel: Decimal in Java
- Variante 3: Gleitkommazahlen

Gleitkommazahlen

- Ziel: möglichst großes Intervall darstellbarer Zahlen, möglichst viele Nachkommastellen
 - aber nicht gleichzeitig: für „große“ Zahlen werden meist keine Nachkommastellen benötigt
 - tatsächliche Zahl wird durch Näherungswert ersetzt
- Gleitkommadarstellung heute meist nach IEEE-754 (IEC 60559:1989)
- Idee: feste Zahl signifikanter Stellen, mit Faktor skalierbar
 - negative Zahlen durch Vorzeichenbit repräsentiert
- IEEE-754 einfache Genauigkeit:



- IEEE-754 doppelte Genauigkeit:

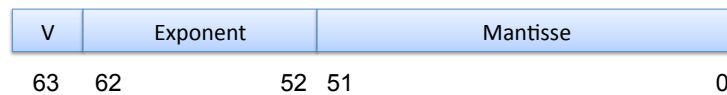


Gleitkommazahlen (contd.)

- jede Zahl hat 3 Bestandteile:
 - Vorzeichen V (1 Bit)
 - Mantisse M (signifikante Stellen)
 - Exponent E (für Faktor zur Basis 2), ganze Zahl in Binärdarstellung
- Dargestellte Zahl ist $(-1)^V * M * 2^E$
- Normalisierung: Durch Wahl von E kann man die Mantisse immer in die Form $1.a_1a_2a_3a_4\dots$ bringen
 - Zur Speicherung von M genügt es, nur die Nachkommastellen zu speichern (normalisierte Zahlen)
- Exponent: Bias-Darstellung
 - Bias-Darstellung erlaubt einfacheres Vergleichen von zwei Gleitkommazahlen
 - Spezialwerte für den Exponenten: minimaler Wert (alles 0) und maximaler Wert (alles 1)

Gleitkommazahlen (contd.)

- Übliche Formate: 32 Bit, 64 Bit
- 32-Bit-Darstellung (Java float, C float)
 - 1 Vorzeichenbit, 8 Bit Exponent (Bias 127), 23 Bit Mantisse
- Wertebereich normalisierter Zahlen
 - Mantisse immer $1 \leq m < 2$
 - negativ: $-(2 \cdot 2^{-23}) \times 2^{127} \dots -2^{-126}$
 - positiv: $2^{-126} \dots (2 \cdot 2^{-23}) \times 2^{127}$
- 64-Bit-Darstellung (Java double) (doppelte Genauigkeit)
 - 1 Vorzeichenbit, 11 Bit Exponent (Bias 1023), 52 Bit Mantisse



Gleitkommazahlen (contd.)

- Spezialfälle: Exponent 0000...000, 1111...111
- Zahl 0.0: Exponent 0, Mantisse 0
 - Vorzeichen 0: +0.0 (also: 0.0 wird durch 32/64 Nullbits dargestellt)
 - Vorzeichen 1: -0.0
- +/- Infinity: Exponent maximal (32-bit: FF16), Mantisse 0
 - „kanonische“ Rechenregeln für +/-∞
- Darstellung in Programmiersprachen uneinheitlich
 - Java: Float.POSITIVE_INFINITY
 - NaN (not-a-number): Exponent maximal, Mantisse ≠ 0
 - zur Anzeige von Bereichsverletzungen (etwa: Wurzel aus negativen Zahlen)
 - Berechnungen mit NaN als Eingabe liefern meist NaN als Ergebnis
- denormalisierte Zahlen: Exponent 0, Mantisse ≠ 0
 - füllen Lücke zwischen 0 und kleinster normalisierter Zahl
 - Interpretiert als $0.a_1a_2a_3a_4a_5\dots$

IEEE-754 - Betriebsmodi

- Rundungsmodus: Anpassung des Berechnungsergebnisses an „nächste“ darstellbare Zahl
 - towards zero
 - towards negative infinity
 - towards positive infinity
 - unbiased (zur nächstliegenden Zahl, bei Zahl in der Mitte zu der Zahl, deren letztes signifikantes Bit 0 ist)
- Fehlerbehandlung
 - Generierung von Ausnahmen
 - Generierung von Spezialwerten (infinity, NaN)
- „strikte“ Implementierung
 - Prozessorhardware weicht oft in Details von IEEE-754 ab
 - Javas Schlüsselwort `strictfp` erzwingt Konformität

Rundungseffekte

- Zahlen mit endlicher Darstellung im Dezimalsystem haben oft keine exakte Darstellung als Gleitkommazahl
 - Beispiel: $0,1 == 1/10$ hat periodische Darstellung im Binärsystem
0.00011001100110011...
- Ausnahmen: Brüche, deren Nenner eine Zweierpotenz ist
 - $0,25 == \frac{1}{4} == 0.012$
- Gleitkommadarstellung sucht nächstliegende darstellbare Zahl
 - Beispiel: 0.1, float

Vorzeichen	Exponent	Mantisse	Dezimalwert
0	111011	1,0011E+22	0,0999999940..
0	111011	1,0011E+22	0,100000002

gerundet auf 

Weitere Zahlendarstellungen

- komplexe Zahlen: Darstellung als Paar (Realteil, Imaginärteil)
- Intervallarithmetik: Darstellung einer Zahl als Intervall
 - Berücksichtigung von Rundungen in vorigen Operationen
- „Preliminary Discussion of the logical designs of an Electronic Computing Instrument“, Burks, Goldstine, von Neumann, 1946:
 - Gleitkommazahlen seine nicht nötig / zu aufwendig;
 - Sie dienten nur als Skalierungsfaktor um den Zahlenbereich der Maschine einzuhalten;
 - Diese Skalierung könne der geschulte Benutzer (Mathematiker) aber von Hand vorab durchführen

Plain Text

- Abstraktion: Text wird durch eine Folge von Symbolen (Buchstaben, Zahlen, Interpunktion) dargestellt
 - Verzicht auf Informationen über Schriftart, Schriftgröße, Schriftfarbe, Seitenstruktur
 - Eingeschränkte Unterstützung für Zeilen- und Absatzstruktur (Zeilenumbrüche, Tabulatorzeichen)
- Alternativen:
 - „rich text“: zuzüglich typographischer Informationen
 - strukturierter Text: zuzüglich editorielle/semantischer Informationen (Kapitelstruktur, Querverweise, Auszeichnung von Zitaten, ...)

Codes für Zeichen – historischer Werdegang

- EBCDIC (Extended Binary Coded Decimal Interchange Code)
 - Eingeführt durch IBM, Lochkarten
 - Nicht alle Bitkombinationen besetzt
 - Buchstaben und Ziffern liegen dicht
- ASCII (American Standard Code for Information Interchange)
 - 7-bit Code
 - Englisches Alphabet mit Steuerzeichen
 - Buchstaben und Ziffern liegen dicht
- ISO 8859
 - Erweiterung auf nationale Alphabete
- Unicode
 - Einheitlicher Code
 - Zwei-Byte-Code

ASCII

- American Standard Code for Information Interchange
 - 1963 erstmalig vorgeschlagen, 1968 normiert (ANSI X3.4-1968)
 - später auch (in Varianten) internationaler Standard
- ISO 646, CCITT International Alphabet #5
- international reference version, nationale Varianten
 - DIN 66003: @ vs. \$, [vs. Ä, \ vs. Ö,] vs. Ü, ...
- 7-Bit-Zeichensatz (128 Zeichen)
 - 32(34) Steuerzeichen, 96(94) druckbare Zeichen
 - heute immer in 8 Bit kodiert (MSB ist dann immer 0)
- systematisch strukturiert
 - Ziffern (0..9), Großbuchstaben (A..Z), Kleinbuchstaben (a..z) sind jeweils durchgehend nummeriert (48..57, 65..90, 97..122)
- Steuerzeichen: 0..31
 - üblich: 7 (BEL), 9 (TAB), 10 (NL), 13 (CR)

ASCII und ISO 8859

- American Standard Code for Information Interchange

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HAT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	XON	DC3	XOF	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

- Die Werte über 127 werden im ISO 8859 genutzt, um nationale Sonderzeichen darzustellen
 - Deutsche Umlaute, etc.

8-bit Zeichensätze

- Oft Erweiterungen von ASCII: weitere druckbare Zeichen im Bereich 128..255
 - ISO 8859: weitere Zeichen im Bereich 160..255 für Europa und den Nahen Osten
 - ISO-8859-1: Westeuropa (Latin-1)
 - ISO-8859-2: Mitteleuropa (Latin-2)
 - ISO-8859-3: Südeuropa (Latin-3)
 - ISO-8859-4: Nordeuropa (Latin-4)
 - ISO-8859-5: Kyrillisch
 - ISO-8859-6: Arabisch
 - ISO-8859-7: Griechisch
 - ISO-8859-8: Hebräisch
 - ISO-8859-9: Türkisch (Latin-5; Austausch der isländischen Zeichen mit türkischen)
 - ISO-8859-10: Nordisch (Latin-6; Latin 4 + Inuit, Nicht-Skolt Sami)
 - ISO-8859-11 (1999): Thai
 - ISO-8859-13: Ostseeraum (Latin-7)
 - ISO-8859-14: Keltisch (Latin-8)
 - ISO-8859-15: Westeuropa (Latin-9, Latin-1 ohne Bruchzeichen, plus €, Š, Ž, Ć, Ÿ)
 - ISO-8859-16: Europa (Latin-10, Verzicht auf Symbole zugunsten von Buchstaben)

Andere Zeichensätze

- Viele proprietäre 8-Bit-Zeichensätze:
 - IBM-Codepages (z.B. CP437)
 - Windows-Codepages (z.B. windows-1252)
 - Mac-Zeichensätze (z.B. mac-roman)
 - Auch EBCDIC; heute nur noch für Großrechner im Einsatz
- Mehrbytezeichensätze: ein oder zwei Bytes pro Zeichen
 - Chinesisch: Big5 (traditional Chinese), GB-2312 (simplified Chinese)
 - Japanisch: JIS 0208, JIS 0212
 - Koreanisch, Vietnamesisch
- Standards zur Kombination von Kodierungen: ISO-2022
 - z.B. iso-2022-jp (RFC 1554): kombiniert ASCII, JIS X 0208-1978, GB2312-1980, ...

Unicode

- Problem: Verschiedene Sprachen benutzen unterschiedliche Alphabete
- Lösungsidee:
 - Eindeutige Codierung für jedes Zeichen, unabhängig von der Plattform, dem Programm oder der Sprache
- Unicode als „Über“-Alphabet
- Ursprünglich 16 bit-Zeichensatz, heute 31 bit-Zeichensatz
 - Mit $17 \cdot 2^{16}$ Zeichen
- Organisation von Alphabeten in „Blocks“
- Steuerzeichen, z.B. für Schreibrichtungen, Ligaturen
- Aktuelle Version: Unicode 6.0

Unicode (contd.)

- Unicode 6.0: ISO/IEC 10646-2003 (einschließlich Amendments 1 bis 8)
 - 109242 graphische Zeichen
 - 142 Formatierungszeichen
 - 65 Steuerzeichen
 - 865081 reservierte Zeichen
- Zeichensatz wird UCS-4 genannt
 - UCS-2 ist eine Teilmenge mit < 65536 Zeichen

Beispiele für Unicode

Georgisch (10A0..10FF)

	10A	10B	10C	10D	10E	10F
0	Ⴀ	Ⴁ	Ⴂ	Ⴃ	Ⴄ	Ⴅ
1	Ⴆ	Ⴇ	Ⴈ	Ⴉ	Ⴊ	Ⴋ
2	Ⴌ	Ⴍ	Ⴎ	Ⴏ	Ⴐ	Ⴑ
3	Ⴒ	Ⴓ	Ⴔ	Ⴕ	Ⴖ	Ⴗ
4	Ⴘ	Ⴙ	Ⴚ	Ⴛ	Ⴜ	Ⴝ
5	Ⴞ	Ⴟ	Ⴀ	Ⴁ	Ⴂ	Ⴃ
6	Ⴄ	Ⴅ	Ⴆ	Ⴇ	Ⴈ	Ⴉ
7	Ⴊ	Ⴋ	Ⴌ	Ⴍ	Ⴎ	Ⴏ
8	Ⴐ	Ⴑ	Ⴒ	Ⴓ	Ⴔ	Ⴕ
9	Ⴗ	Ⴘ	Ⴙ	Ⴚ	Ⴛ	Ⴜ
A	Ⴝ	Ⴞ	Ⴟ	Ⴀ	Ⴁ	Ⴂ
B	Ⴃ	Ⴄ	Ⴅ	Ⴆ	Ⴇ	Ⴈ
C	Ⴉ	Ⴊ	Ⴋ	Ⴌ	Ⴍ	Ⴎ
D	Ⴏ	Ⴐ	Ⴑ	Ⴒ	Ⴓ	Ⴔ
E	Ⴖ	Ⴗ	Ⴘ	Ⴙ	Ⴚ	Ⴛ
F	Ⴜ	Ⴝ	Ⴞ	Ⴟ	Ⴀ	Ⴁ

Pfeilsymbole (2190..21FF)

	219	21A	21B	21C	21D	21E	21F
0	↔	↔	↔	↔	↔	↔	↔
1	↔	↔	↔	↔	↔	↔	↔
2	↔	↔	↔	↔	↔	↔	↔
3	↔	↔	↔	↔	↔	↔	↔
4	↔	↔	↔	↔	↔	↔	↔
5	↔	↔	↔	↔	↔	↔	↔
6	↔	↔	↔	↔	↔	↔	↔
7	↔	↔	↔	↔	↔	↔	↔
8	↔	↔	↔	↔	↔	↔	↔
9	↔	↔	↔	↔	↔	↔	↔
A	↔	↔	↔	↔	↔	↔	↔
B	↔	↔	↔	↔	↔	↔	↔
C	↔	↔	↔	↔	↔	↔	↔
D	↔	↔	↔	↔	↔	↔	↔
E	↔	↔	↔	↔	↔	↔	↔
F	↔	↔	↔	↔	↔	↔	↔

Unicode-Prinzipien

- **Universalität: Ein Zeichensatz für alle Schriftsysteme**
 - Effizienz: Einfach zu verarbeiten
 - Zeichen, nicht Glyphen
 - Semantik: Alle Zeichen haben klare Bedeutung
 - Reiner Text: Unicode-Zeichen stellen nur Text dar
 - Logische Ordnung: Reihenfolge im Speicher folgt der „logischen“ Ordnung
- **Vereinheitlichung (unification): Doppelte Zeichen in verschiedenen Schriftsystemen werden nur einmal in Unicode aufgenommen**
- **Dynamische Komponierung: Neue Zeichen können durch Komposition aus bestehenden entstehen**
- **Zeichenäquivalenz: Vorkomponierte Zeichen sind „äquivalent“ zu dekomponierten Zeichenfolgen**
- **Konvertierbarkeit: Unicode kann verlustfrei in andere Zeichensätze konvertiert werden (und zurück)**

Unicode-Zeichen

- **haben stabilen Code (code point)**
 - z.B. U+00DF
- **haben stabilen Zeichennamen**
 - z.B. LATIN SMALL LETTER SHARP S
- **Unicode-Standard enthält Demo-Glyphen**
 - z.B. „ß“
- **Unicode-Datenbank gibt Zeicheneigenschaften an**
 - z.B. „Letter, lower case“ (LI)

UTF-8

- Kodierung von Unicode in Bytes: mehrere Bytes pro Zeichen
 - Variante 1: jedes Zeichen benötigt mehrere Bytes (UTF-16, UTF-32)
- Probleme: Rückwärtskompatibilität, Byte Order
 - Variante 2: Jedes Zeichen benötigt eine variable Zahl von Bytes
 - UTF-8: 1 bis 4 Bytes pro Zeichen
- Zeichen <128: 1 Byte (äquivalent zu ASCII)
- Nullbytefrei
- Jede Bytefolge unterliegt bestimmtem Muster, um die Dekodierung zu erleichtern

UTF-8 (contd.)

Bitmuster	Anzahl kodierter Zeichen	Bereich kodierter Zeichen
0xxxxxxx	128	0..7F
110xxxxx 10xxxxxx	2048	80..7FF
1110xxxx 10xxxxxx	65536	800..FFFF
11110xxx 10xxxxxx ...	2097152 (nur 1114112)	10000..10FFFF

Zeichenketten

- Kodierung von Zeichenfolgen: Verkettung der Kodierungen der einzelnen Zeichen
- Zeichenkette: engl. „character string“ oder einfach „string“
- Zwei Modi: Bytefolgen oder Zeichenfolgen
 - C, C++, (Python): Strings sind Folgen von Bytes; Interpretation der Bytes entsprechend dem Zeichensatz obliegt der Anwendung
 - Java, C#, (Python): Strings sind Folgen von Unicode-Zeichen; jedes Unicodezeichen ist intern durch mehrere Bytes repräsentiert
- Stringlänge: mehrere Repräsentationen
 - Datei: Stringende = Dateiende
 - C, C++: Stringende wird durch Null-Byte angezeigt
 - Pascal, Java, Python,...: Zusammen mit dem String wird die Länge gespeichert

Zusammenfassung

- Universalität binärer Daten
- Abtasttheorem
 - Erfassung stetiger Größen impliziert Ungenauigkeiten
- ganze Zahlen, Binary Coded Decimals
- 1er Komplement
- 2er Komplement
- Gleitkommaformate
 - Nur mit Festkomma-Formaten rechnet ein Computer „genau“
- ASCII, EBCDIC
- Unicode
 - Darstellung von „plain text“ gelöst;
 - Verarbeitung von Zeichenketten je nach Programmiersprache schwierig