Zeichenkodierung

Hans Jürgen Ohlbach

Keywords: ASCII, ISO-8859, Unicode, UTF, Glyphen, Fonts, Bitmap- und Vektordarstellung

Inhaltsverzeichnis

1 ASCII 2

2 ASCII-Erweiterungen 3

3 Unicode 4

3.1 UTF-16 Kodierung 5

3.2 UTF-8 Kodierung 7

4 Endiannes 8

5 HTML-Zeichensatz 9

6 Glyphen und Fonts 10

Das Wort „Computer“ heißt auf Deutsch „Rechner“. Die wenigsten Computer tun allerdings heutzutage hauptsächlich rechnen. Die meisten verarbeiten Texte in irgendeiner Form. Um Texte mit dem Computer zu verarbeiten, muss es möglich sein, die Buchstaben als Bitfolgen abzuspeichern. Im Gegensatz zu ganzen Zahlen, wo die Binärkodierung eine mathematische Vorschrift ergibt, um Zahlen binär zu speichern, ist die Binärkodierung von Buchstaben recht willkürlich. Daher hat man unterschiedliche Standards dafür entwickelt.

1 ASCII

Einer der ersten, und immer noch einer der wichtigsten, ist die 1963 von der *American Standards Association* entwickelte ASCII-Kodierung (*American Standard Code for Information Interchange*). ASCII weißt jedem Buchstaben eine Sequenz von 7 Bits zu. Das 8. Bit ist normalerweise 0, kann aber auch für andere Zwecke benutzt werden. Mit 7 Bits kann man $2^7 = 128$ Zeichen kodieren. Die ersten 32 Codes enthalten Steuerzeichen, dann kommen Sonderzeichen und Ziffern. Erst ab Nummer 65 kommen die Großbuchstaben, ab Nummer 97 kommen die Kleinbuchstaben.

ASCII-Tabelle

hex	dez	Zeichen	hex	dez	Zeichen	hex	dez	Zeichen	hex	dez	Zeichen
00	0	NUL	20	32	SP	40	64	@	60	96	‘
01	1	SOH	21	33	!	41	65	A	61	97	a
02	2	STX	22	34	”	42	66	B	62	98	b
03	3	ETX	23	35	#	43	67	C	63	99	c
04	4	EOT	24	36	$	44	68	D	64	100	d
05	5	ENQ	25	37	%	45	69	E	65	101	e
06	6	ACK	26	38	&	46	70	F	66	102	f
07	7	BEL	27	39	’	47	71	G	67	103	g
08	8	BS	28	40	(	48	72	H	68	104	h
09	9	TAB	29	41	)	49	73	I	69	105	i
0A	10	LF	2A	42	*	4A	74	J	6A	106	j
0B	11	VT	2B	42	+	4B	75	K	6B	107	k
0C	12	FF	2C	44	,	4C	76	L	6C	108	l
0D	13	CR	2D	45	-	4D	77	M	6D	109	m
0E	14	SO	2E	46	.	4E	78	N	6E	110	n
0F	15	SI	2F	47	/	4F	79	O	6F	111	o
10	16	DLE	30	48	0	50	80	P	70	112	p
11	17	DC1	31	49	1	51	81	Q	71	113	q
12	18	DC2	32	50	2	52	82	R	72	114	r
13	19	DC3	33	51	3	53	83	S	73	115	s
14	20	DC4	34	52	4	54	84	T	74	116	t
15	21	NAK	35	53	5	55	85	U	75	117	u
16	22	SYN	36	54	6	56	86	V	76	118	v
17	23	ETB	37	55	7	57	87	W	77	119	w
18	24	CAN	38	56	8	58	88	X	78	120	x
19	25	EM	39	57	9	59	89	Y	79	121	y
1A	26	SUB	3A	58	:	5A	90	Z	7A	122	z
1B	27	Esc	3B	59	;	5B	91	[	7B	123	{
1C	28	FS	3C	60	<	5C	92	\	7C	124	|
1D	29	GS	3D	61	=	5D	93	]	7D	125	}
1E	30	RS	3E	62	>	5E	94	ˆ	7E	126	˜
1F	31	US	3F	63	?	5F	95	_	7F	127	DEL

Einige Dinge fallen an der ASCII-Tabelle auf:

- Es gibt Sonderzeichen, die durch alte Hardware motiviert sind, und heute kaum noch brauchbar

sind, z.B. das Zeichen 7 (BEL) sollte eine Glocke ansteuern, die dann einen Ton von sich gibt.

- Es gibt Sonderzeichen, die ebenfalls durch alte Hardware motiviert sind, heute aber noch Ärger machen. In alten Schreibmaschinen und Fernschreibern bestand der Zeilenumbruch aus dem Wagenrücklauf (Zeichen 13 CR) und dem Zeilenvorschub (Zeichen 10 LF). Die waren im Prinzip unabhängig voneinander. Heute würde man das eher als *eine* Operation sehen, die mit *einem* Zeichen kodiert wird. Dafür würde sich CR anbieten. Leider ist es nicht gelungen, sich auf einen Standard zu einigen. Unix, Linux und Android benutzen LF, Windows benutzt CR LF, MAC OS benutzt CR. Wenn man also Dateien zwischen diesen Systemen austauscht, kann es zu merkwürdigen Zeilenumbrüchen kommen.

- Die Großbuchstaben fangen bei Nummer 65 an, und die Kleinbuchstaben fangen bei Nummer 97 an. Das kann man ausnutzen, um sehr effizient Klein- in Großbuchstaben umzuwandeln, und umgekehrt:
  Großbuchstabe $\mapsto$ Kleinbuchstabe: addiere 32.
  Kleinbuchstabe $\mapsto$ Großbuchstabe: subtrahiere 32.

- Ein Test, ob es sich bei einem Buchstaben um einen Großbuchstaben handelt, ist ebenfalls sehr effizient machbar: $65 \leq Buchstabe \leq 90$. Für Kleinbuchstaben testet man $97 \leq Buchstabe \leq 122$.

- Die alphabetische Ordnung spiegelt sich auch in der ASCCI-Kodierung wider. Daher kann man die alphabetische Reihenfolge von zwei Buchstaben einfach dadurch bestimmen, dass man ihre ASCII-Codes auf $<$ vergleicht.

Eine Alternative zu ASCII wurde von IBM entwickelt: Extended Binary Coded Decimal Interchange Code (EBSDIC) ist ein 8-Bit Code, der aber fast ausschließlich auf Großrechnern verwendung findet. Im Computeralltag spielt er daher kaum eine Rolle.

2 ASCII-Erweiterungen

Der 7-Bit ASCII Code enthält in seine 128 Zeichen keine Sonderzeichen, wie sie in vielen nicht-englischsprachigen Ländern verwendet werden, insbesondere nicht die deutschen Umlaute.

In ASCII ist das 8. Bit immer die 0. Das hat man sich zu Nutze gemacht, um weitere Zeichen hinzuzufügen. Wenn man das 8. Bit auf 1 setzt, hat man nochmal 128 Nummern zur Verfügung, auf die man Sonderzeichen verteilen kann. Sieht man sich die weltweit benutzten Zeichen an, reichen die 128 zusätzlichen Nummern leider bei weitem nicht, um alle Sonderzeichen unterzubringen.

Daher wurden unterschiedliche Erweiterungen von ASCII entwickelt, die alle mit ASCII übereinstimmen, falls das 8. Bit = 0 ist, aber die Sonderzeichen auf unterschiedliche Weise in die restlichen 128 Nummern verteilen wo das 8. Bit = 1 ist.

Die für Europa wichtigeste Familie von Erweiterungen ist die ISO-8859 Familie:

ISO-Nummer	Alternativer Name	Sprachfamilie
ISO-8859-1	Latin-1	Westeuropäisch
ISO-8859-2	Latin-2	Mitteleuropäisch
ISO-8859-3	Latin-3	Südeuropäisch
ISO-8859-4	Latin-4	Nordeuropäisch
ISO-8859-5		Kyrillisch
ISO-8859-6		Arabisch
ISO-8859-7		Griechisch
ISO-8859-8		Hebräisch
ISO-8859-9	Latin-5	Türkisch
ISO-8859-10	Latin-6	Nordisch
ISO-8859-11		Thai
ISO-8859-13	Latin-7	Baltisch
ISO-8859-14	Latin-8	Keltisch
ISO-8859-15	Latin-9	Westeuropäisch
ISO-8859-16	Latin-10	Südosteuropäisch

ISO-8859-15 ist dabei eine Erweiterung von ISO-8859-1, wo neben dem Eurosymbol noch einige französische Zeichen hinzugefügt wurden.

Die kompletten Zeichentabellen findet man im Internet.
Für deutsche Texte wichtig in ISO-8859-15 sind insbesondere:
ä = E4, Ä = C4, ö = F6, Ö = D6, ü = FC, Ü = DC, ß = DF und €= A4.

Die meisten dieser ISO-8859 Zeichenkodierungen lassen noch einige Nummern unbesetzt. Das haben sich wiederum andere Organisationen zu Nutze gemacht, um diese unbesetzten Nummern zu füllen. Z.B. wurde ISO-8859-1 von Microsoft für das Betriebssystem Windows erweitert zu Windows-1252. Überträgt man daher Texte von Windows auf andere Betriebssysteme, können die Texte sehr merkwürdig aussehen.

Die Website https://www.iana.org/assignments/character-sets/character-sets.xhtml gibt einen umfassenden Überblick über alle standardisierten Zeichenkodierungen.

3 Unicode

Jede der einzelnen 8-Bit Erweiterungen von ASCII kann nur maximal 256 Zeichen kodieren. Das reicht bei weitem nicht, um weltweit alle benutzten Zeichen unterzubringen. Allein für die chinesische Schrift sind über 87000 Zeichen bekannt, wobei allerdings nur 1500-2000 Schriftzeichen für die durchschnittliche Lese- und Schreibfähigkeit ausreichen sollten.

Um weltweit alle benutzten Schriftzeichen in einer Kodierung zu vereinen, wurde nach mehrjähriger Entwicklungszeit 1991 die erste Version der *Unicode Kodierung* veröffentlicht. In den folgenden Jahren wurden nach und nach immer mehr Schriftfamilien eingebaut.

Unicode definiert 17 *Ebenen* (engl. *planes*). Jede davon wird in 16 Bits kodiert. Sie kann also $2^{16} = 65536$ Zeichen unterbringen. Insgesamt ist also Platz für $17 \cdot 2^{16} = 1114112$ Zeichen. Nur wenige der

Ebenen sind derzeit tatsächlich bevölkert.

Plane 0 oder **Basic Multilingual Plane (BMP)** ist eine direkte Erweiterung von ASCII. D.h. die ersten 128 Nummern entsprechen der ASCII Kodierung. Die restlichen Nummern kodieren die am häufigsten im Gebrauch befindlichen Schriftsysteme. Die BMP ist weitgehend belegt, so dass weitere Schriftsysteme in den nächsten Ebenen Platz finden müssen.

Die Zeichen der BMP können in 2 Bytes (= 16 Bits) gepackt werden. Die Hexadezimaldarstellung dieser Zeichen geht dann von 0000_{16} bis FFFF_{16}.

Einige Programmiersysteme (nicht Java) beschränken sich bei der Zeichendarstellung auf 2 Bytes pro Zeichen. Diese können dann nur mit der BMP umgehen.

Plane 1 oder **Supplementary Multilingual Plane (SMP)** enthält vor allem historische Schriftsysteme und Zeichen, die selten in Gebrauch sind.

Die Hexadezimaldarstellung dieser Zeichen geht von 10000_{16} bis $1\mathrm{FFFF}_{16}$.

Plane 2 oder **Supplementary Ideographic Plane (SIP)** enthält ausschließlich chinesische, japanische und koreanische Schriften (CJK Schriften). Sobald dieser Bereich gefüllt ist, steht Plane 3 für Erweiterungen zur Verfügung.

Die Hexadezimaldarstellung dieser Zeichen geht von 20000_{16} bis $2\mathrm{FFFF}_{16}$.

Plane 14 oder **Supplementary Special-purpose Plane (SSP)** enthält einige wenige Kontrollzeichen zur Sprachmarkierung. Die Hexadezimaldarstellung dieser Zeichen geht von $\mathrm{E0000}_{16}$ bis EFFFF_{16}.

Die restlichen beiden Planes sind privat nutzbar.

3.1 UTF-16 Kodierung

Um alle Unicode Buchstaben in allen 17 Ebenen unterzubringen, braucht man Nummern zwischen 0 und $10\mathrm{FFFF}_{16}$. Dazu sind 21 Bits notwendig. 21 Bits würden problemlos in ein 32-Bit Integer Wort passen, wobei die führenden 11 Bits 0 wären. Unicode Zeichenketten könnte man daher als Integer-Arrays speichern. Dies entspräche der *UTF-32-Kodierung*.

Allerdings bestehen westliche Texte ganz überwiegend aus ASCII-kodierbaren Zeichen, mit ein paar wenigen Sonderzeichen dazwischen (Umlaute z.B.). D.h. die meisten Zeichen bräuchten von den 32 Bits tatsächlich nur 7 Bits. Die 25 weiteren Bits wären = 0. Das wäre eine enorme Platzverschwendung.

Die UTF-16 Kodierung trägt diesem Umstand teilweise Rechnung, indem sie anstelle von 32 Bit Integern, nur 16 Bit Integer für ein Zeichen bereit stellt. 16 Bit Integer sind in den meisten Programmiersystemen als *short int* verfügbar. In den 16 Bits lassen sich die 65536 Zeichen der Basic Multilingual Plane unverändert unterbringen.

Zeichen aus den höheren Ebenen brauchen mehr als 16 Bits. Deren Zeichen werden nach folgendem Schema auf 2 mal 16 Bits verteilt:

Von der Nummer des Zeichens wird zunächst die Zahl (10000_{16}) abgezogen (= Größe der BMP), wodurch eine 20-Bit-Zahl im Bereich von 00000hex bis FFFFFhex entsteht. Diese wird anschließend in zwei Blöcke zu je 10 Bit aufgeteilt, und dem ersten Block die Bitfolge 110110, dem zweiten Block dagegen die Bitfolge 110111 vorangestellt.

Das erste der beiden so entstandenen 16-Bit-Wörter bezeichnet man als *High-Surrogate*, das zweite als *Low-Surrogate*, und ihren Namen entsprechend enthält das High-Surrogate die 10 höherwertigen, das Low-Surrogate die 10 niederwertigen Bits des um 10000_{16} verringerten ursprünglichen Zeichencodes.

Beispiele:

Zeichen	Unicode	Unicode binär	Unicode-16 binär	UTF-16 hexadezimal
Buchstabe y	0079_{16}	00000000 011111001	00000000 011111001	00 79_{16}
Eurozeichen €	20 AC_{16}	00100000 10101100	00100000 10101100	20 AC_{16}
Violinschlüssel	$1D11E_{16}$	00000001 11010001 00011110	11011000 00110100 11011101 00011110	D8 34 DD $1E_{16}$
CJK-Ideogramm	$24F5C_{16}$	00000010 01001111 01011100	11011000 01010011 11011111 01011100	D8 53 DF $5CE_{16}$

Die beiden roten Bitmuster 110110 und 110111 in aufeinanderfolgenden 16-Bit Blöcken sagen dem jeweiligen Programmiersystem, dass es sich um einen Buchstaben außerhalb der BMP handelt. Die echte Nummer (meist *codepoint* genannt), kann man dann aus den hinteren Bits wieder zusammenbauen. Damit diese Bitmuster nicht mit Nummern aus der Ebene 0 verwechselt werden, besagt der Unicode Standard, dass Nummern mit diesen führenden Bitmustern in der BMP frei gehalten werden müssen.

UTF-16 hat mehrere Vorteile:

- die meisten westlichen Texte kommen mit der BMP aus, so dass sie als Folgen von 16 Bit Kodierungen gespeichert werden können.
- Ist eine Folge von UTF-16 kodierten Buchstaben an einer Stelle irgendwie beschädigt, dann kann man nach dieser Stelle wieder problemlos entscheiden, ob der Buchstabe aus der Ebene 0 oder aus höheren Ebenen stammt. Dadurch hat die Beschädigung keine Folgewirkung.

Zeichenkodierung in Java: Java benutzte ursprünglich eine einfach 16-Bit Kodierung von Unicode. Nachdem Unicode auf mehrere Ebenen erweitert wurde, wurde auch die Zeichenkodierung in Java entsprechend erweitert, und zwar folgendermaßen:

- der Typ `char` kann weiterhin nur 16-Bit Zeichen kodieren, und damit nur die BMP (Ebene 0).
- Folgen von Charactern, d.h. char Arrays, Strings usw. *werden UTF-16* kodiert.

  Bei Texten, die nur aus BMP-Zeichen bestehen, macht das keinen Unterschied zu früher. Für Texte mit Zeichen aus den höheren Ebenen gibt es Zugriffsmethoden auf die einzelnen 16-Bit Buchstaben, die dies unterscheiden können.

  Z.B. gibt es in der Klasse String die Methode
  `charAt(index)`
  die, egal welche Kodierung vorliegt, die 16-Bits an der Stelle `index` als Typ `char` aus dem String herausholt.

Darüber hinaus gibt es die Methode
`codePointAt(index)`
Diese überprüft, ob an den Stellen `index` und `index+1` das charakteristische UTF-16 Bitmuster 110110 und 110111 vorliegt. In diesem Fall extrahiert sie aus den beiden 16-Bit Folgen die Unicode Nummer des Zeichens aus Ebenen 1-17, und gibt sie als int-Wert zurück. (Diese Nummer nennt man auch den *code point*). Falls das charakteristische Muster nicht vorliegt, wird die einzelne 16-Bit Folge an der Stelle `index` als int-Wert zurückgegeben.

Für die meisten String-Operationen ist die UTF-16 Kodierung jedoch nicht sichtbar. Baut man z.B. einen String aus *zwei* Buchstaben der Ebenen 1-17 zusammen, dann braucht die UTF-16 Kodierung dafür 4 mal 16 Bits. Die Methode length(), die die Länge des Strings berechnet, liefert dann auch 4, anstelle von 2. Der Zugriff auf die einzelnen Komponenten des Strings geschieht mit Indizes, die in diesem Fall von 0-3 laufen. Ein Aufruf charAt(1) liefert dann die zweite 16-Bit Komponente, welche das Low-Surrogate des ersten Buchstabens ist.

Fazit: Obwohl Java mit Unicode arbeitet, kann man keine Programme, die entwickelt und getestet wurden mit Zeichen aus Ebene 0, unbesehen übernehmen, wenn auch Zeichen aus höheren Unicode Ebenen vorkommen können. Insbesondere kann man Programme, die für den westlichen Markt entwickelt wurden, auch multilingual, nicht ohne weiteres in Asien anwenden.

3.2 UTF-8 Kodierung

UTF-8 erlaubt eine noch stärkere Komprimierung von Unicode Zeichen als UTF-16. UTF-8 nutzt aus, dass Unicode einerseits eine Erweiterung von ASCII ist, und dass andererseits bei ASCII das 8. Bit immer 0 ist.

Ein Text, der ausschließlich aus ASCII-Buchstaben besteht, wird auch in UTF-8 als Folge von 8 Bit ASCII-Codes gespeichert. Damit ist der Text genauso kompakt wie in der Original ASCII-Kodierung, und jedes Textverarbeitungsprogramm, welches nur mit ASCII umgehen kann, hat kein Problem damit.

Buchstaben außerhalb des ASCII-Bereichs werden als Folgen von Bytes (8 Bits) gespeichert, wobei das erste Byte an der linkesten Position eine 1 hat. Diese 1 signalisiert, dass ein Nicht-ASCII Sonderfall vorliegt, und mehr als 8 Bits gebraucht werden. Wieviele Bytes gebraucht werden, bestimmen die Bitmuster an der linkesten Position:
110: 2 Bytes werden gebraucht
1110: 3 Bytes werden gebraucht
11110: 4 Bytes werden gebraucht.

Die Bits aus der Original Unicode-Kodierung werden nun auf diese 2,3 bzw. 4 Bytes verteilt, wobei jeweils die linkesten 2 Bits des 2., 3. und 4. Byte das Muster 10 enthält.

Die folgende Tabelle fasst das zusammen:

Unicode Bereich	UTF-8 Kodierung
0000 0000 - 0000 007F	0xxxxxxxx
0000 0080 - 0000 07FF	110xxxxxx 10xxxxxx
0000 0800 - 0000 FFFF	1110xxxxx 10xxxxxx 10xxxxxx
0001 0000 - 0010 FFFF	11110xxxx 10xxxxxx 10xxxxxx 10xxxxxx

Das führende 10-Muster im 2. 3. und 4. Byte hilft, wenn die Bitsequenzen irgendwie verfälscht wurden. Eine 10 signalisiert, dass das Byte noch zu einem 2er, 3er oder 4er Block gehört. Wenn kein 10-Muster mehr erscheint, kann der Text wieder normal weiterverarbeitet werden.

Die UTF-16 Kodierung wird vorwiegend in Programmiersprachen verwendet, wie z.B. in Java, während UTF-8 eher für die externe Datenspeicherung und den Datenaustausch gedacht ist.

4 Endiannes

In der Stellenwertnotation schreibt man Zahlen von links nach rechts auf. Z.B. ist bei der Ziffernfolge 345 die 3 als linkeste Ziffer die Hunderterstelle, und damit höchstwertige Ziffer. Die 5 ist die Einerstelle und damit die niedrigstwertige Ziffer. Wenn man die Ziffern durchnummeriert dann steht die 3 an Position 0, die 4 an Position 1 und die 5 an Position 3.

Im Binärsystem kann man das genauso machen: die höchstwertigen Bits oder Bytes haben die kleinste Position (auch Adresse genannt), die niedrigstwertigen Bits oder Bytes haben die höchste Position. In den meisten Prozessoren ist das auch so realisiert.

Die allermeisten arithmetische Operationen, z.B. Addition laufen allerdings von rechts nach links, von den niedrigwertigen Ziffern zu den höherwertigen Ziffern. Das würde nahelegen, die Ordnung umzudrehen: die niedrigstwertige Ziffer ist an Position 0 usw. Einige Computerhersteller hat das daher motiviert, diese Reihenfolgen umzudrehen.

Seither haben wir das Problem: es gibt die zwei Möglichkeiten für die Reihenfolge von Bytes und Wörtern:

Big-Endian (BE): das höchstwertige Byte steht an der kleinsten Adresse (so wie wir es gewohnt sind, Zahlen von links nach rechts zu lesen)

Little-Endian (LE): das niedrigstwertige Byte steht an der kleinsten Adresse.

Die wirkt sich nicht nur auf die Speicherung von Zahlen aus, sondern auch auf die Codierung von Buchstaben, wenn mehr als ein Byte gebraucht wird. Es betrifft daher nicht ASCII, auch nicht UTF-8, da dort die Reihenfolge normiert ist, wohl aber UTF-16 und UTF-32.

Die Hardwarehersteller konnten sich bisher leider nicht auf eine Version verständigen. Aber wenigstens wurde für die Datenübertragung in Netzwerken eine Byte-Reihenfolge festgelegt, die *Network Byte Order*. Stimmt die nicht mit der *Host Byte Order*, dann müssen die Bytes vor der Übertragung umgeordnet werden.

Ein Programm, welches Text einlesen oder Text ausgeben soll muss also nicht nur wissen, welche Kodierung gewählt wurde, sondern auch die Endianness. Innerhalb eines Rechners ist das meist kein Problem, da sich i.A. alle Programme an den gleichen Standard halten. Bei fremden Texten kann das aber zu einem großen Problem werden.

Byte Order Mark (BOM): Um Textverarbeitungsprogrammen eine Hilfestellung zu geben, mit denen sie erkennen können, welche Kodierung und Endianness vorliegt, wurde eine Kennzeichnung, die *Byte Order Mark* eingeführt. Dies ist eine charakteristische Bitfolge, mit der ein Text beginnen sollte.

Die wichtigsten BOMs sind:

UTF-8	EF BB BF$_{16}$
UTF-16 (BE)	FE FF$_{16}$
UTF-16 (LE)	FF FE$_{16}$
UTF-32 (BE)	00 00 FE FF$_{16}$
UTF-32 (LE)	FF FE 00 00$_{16}$

Eine Unterscheidung zwischen den UTF-Kodierung und den ISO-8859-Kodierungen ist dadurch leider noch nicht sicher möglich. Viele Programme versuchen daher intelligent zu raten. Das das nicht immer klappt, hat sicher schon jeder mal an E-Mails mit merkwürdigen Zeichen erlebt. Diese merkwürdigen Zeichen können dadurch entstanden sein, dass das Programm die falsche Kodierung geraten hat. Da hilft oft, den Menupunkt *encoding* zu suchen und zu ändern.

5 HTML-Zeichensatz

Die Webseitensprache HTML benutzt ebenfalls Unicode, und die Browser verstehen auch die UTF-8-Kodierung. Damit man HTML-Texte bequem editieren kann, wurden verschiedene Syntaxvarianten für nicht-ASCII Zeichen eingeführt. Die meisten Zeichen können entweder mit einem speziellen Namen, oder mit der *Numeric character reference (NCR)*, was nichts anderes als die Unicode Nummer ist, entweder hexadezimal oder dezimal angegen werden.

Die folgenden Beispiele verdeutlichen das.

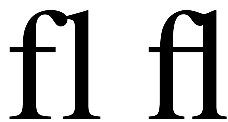
Zeichen	HTML	NCR hex	NCR dez
&	&	&	&
±	±	±	±

Vollständige Listen von diesen Zeichenkodierungen findet man im Internet.

6 Glyphen und Fonts

Die Bitkodierung eines Zeichens enthält noch keinerlei Information darüber, wie das Zeichen auf einem Bildschirm oder auf dem Papier dargestellt wird. Wie jeder Leser schon gesehen hat, gibt es dafür die unterschiedlichsten Möglichkeiten. Die graphische Darstellung eines Zeichens wird als *Glyphe* bezeichnet. Für jedes Zeichen gibt es dabei die unterschiedlichsten Varianten. Hier sind einige Varianten für den Buchstaben A:

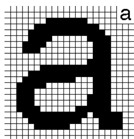
Manche der Buchstabenkombinationen sehen optisch besser aus, wenn man sie zu einer Glyphe vereinigt, sogenannte *Ligaturen*. Ein Beispiel ist die Folge aus f und l:



Die zweite Version ist eine Ligatur, wo aus f und l eine Glyphe gemacht wurde.

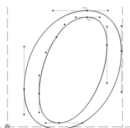
Eine Glyphe ist zunächst mal nur die Vorstellung eines Designers oder Künstlers. Um sie im Rechner zu benutzen, gibt es im Prinzip zwei Möglichkeiten.

Als Bitmap-Schrift: Hierbei wird für eine Glyphe ein Pixelraster festgelegt, und die Glyphe mit schwarzen und weißen Pixeln Zeile für Zeile ausgefüllt. Vergrößert kann das dann so aussehen:



Gerade für Bildschirmdarstellungen, wo es sowieso ein Pixelraster gibt, eignen sich Bitmap Schriften. Auf Papier haben sie den Nachteil, dass man sie nicht ohne weiteres Skalieren kann. Entweder man definiert für unterschiedliche Größen unterschiedliche Bitmaps, oder die Schrift sieht beim hochskalieren immer „verpixelter“ aus.

Als Vektorgraphik: Hierbei werden Umrisse durch mathematische Kurvenspezifikationen angeben, z.B. durch *Besierkurven* wie in folgendem Bild.



Der Vorteil ist, dass einerseits i.A. weniger Daten gebraucht werden als bei Bitmap Graphiken, und zum anderen die Form ohne Qualitätsverlust beliebig skalierbar ist.

Der Nachteil ist, dass die Umwandlung in darstellbare Pixels einiges an Rechenaufwand fordert. Daher ist es manchmal günstiger, aus der Vektorgraphik einmalig eine Serie von Bitmapgraphiken in passenden Größen zu erzeugen, die man dann immer wieder verwenden kann.

Zeichensätze/Fonts: Wenn man für jeden Buchstaben eines Alphabets, und alle Sonderzeichen, eine passende Glyphe definiert hat, hat man einen *Font*. Die Anzahl von Fonts, die im Umlauf sind, wächst und ändert sich fast jedes Jahr: auch hier scheint es Moden zu geben.

Mit den Problemen, die dabei entstehen, wird man konfrontiert, wenn z.B. der Drucker, den man benutzt, den Font nicht kennt, der in einem Dokument benutzt wird.

In diesem Text soll aber nicht weiter darauf eingegangen werden.

Stichwortverzeichnis

ASCII-Tabelle, 2

BE (Big Endian), 8
BMP (Basic Multilingual Plane), 5
BOM (Byte Order Mark), 9

code point, 6

EBSDIC, 3

Font, 11

Glyphen, 10

High-Surrogate, 6

ISO-8859 Familie, 3

LE (Little-Endian), 8
Ligatur, 10
Low-Surrogate, 6

NCR (Numeric character reference), 9

Schrift: Bitmap, 10
Schrift: Vektorgraphik , 10
SIP (Supplementary Ideographic Plane), 5
SMP (Supplementary Multilingual Plane), 5
SSP (Supplementary Special-purpose Plane), 5

Unicode, 4
UTF-16, 5
UTF-32, 5
UTF-8, 7