

Zeichendarstellung mit Unicode und UTF-8

Unicode

Um Probleme, die sich zum einen mit unterschiedlichen Zeichensätzen, zum anderen auch durch andere Sprachen, die mehr als 128 Zeichen umfassen, ergeben haben, wurde der Unicode-Standard entwickelt.

Unicode ist ein internationaler Standard, in dem langfristig für jedes sinnvolle Schriftzeichen oder Textelement aller bekannten Schriftkulturen und Zeichensysteme ein digitaler Code festgelegt wird. Ziel ist es, die Verwendung unterschiedlicher und inkompatibler Kodierungen in verschiedenen Ländern oder Kulturkreisen zu beseitigen. Unicode wird ständig um Zeichen weiterer Schriftsysteme durch das Unicode-Konsortium ergänzt. (Wikipedia, <https://de.wikipedia.org/wiki/Unicode>)

Im Unicode Standard hat jedes Zeichen einen eigenen "Unicode-Code", damit lassen sich derzeit 1.111.998 elementare Zeichen ("Codepunkte") abbilden. Darstellung: U+00DF (Mindestens 4x4Bit, bis zu U+10FFFF) Diese Codepunkte bilden den Unicode Zeichensatz.

Der Codepoint eines Unicode-Zeichens ist nur eine abstrakte Nummer. Die Schreibweise dieser Nummer im Unicode-Standard erfolgt in hexadezimalen Zahlen mit vorangestelltem "U+". Der Codepoint legt noch keine computerkompatible Darstellung fest, dies ist Aufgabe der eigentlichen Codierung. Da die Unicode-Codepoints von U+0000 bis U+10FFFF (hexadezimale Zahlendarstellung), mit einer beabsichtigten Lücke zwischen U+D7FF und U+E000, reichen, sind für eine vollständige Codierung des gesamten Codepoint-Bereichs als Binärzahl mindestens 3 Byte erforderlich.¹⁾

Frage: Wie kann man die "Zeichennummern", also die Codepunkte als Bitmuster darstellen? Dieser Vorgang ist die **Codierung** der Unicode Zeichen.

UTF-32	UTF-16	UTF-8
Jeder Codepoint wird durch 4 Bytes dargestellt. Das reicht für alle Codepoints verschwendet aber Ressourcen - einfach aber verschwenderisch.	Jeder Codepoint wird durch 2 Bytes dargestellt, halbiert also den Speicherverbrauch von UTF-32. 2 Bytes reichen nicht für alle Codepoints, deswegen muss man tricksen, indem man die oben erwähnte Lücke zwischen U+D7FF und U+E000 nutzt. Details finden sich hier.	UTF-8 nutzt 8 Bit lange Codeblöcke. UTF-8 verwendet je nach Bedarf zwischen 1 und 4 Byte, um einen Codepoint darzustellen. Ein codiertes Zeichen hat also keine feste Bitlänge.

UTF-8 hat im Vergleich zu den anderen UTF-Varianten mehrere praktische Vorteile:²⁾

- Die ersten 127 Zeichen und Bytes sind identisch mit ASCII, d. h. alle Texte, die in der Hauptsache Unicode-Zeichen mit Codepoints zwischen U+0000 und U+007F verwenden, bleiben problemlos lesbar.
- Auch Systeme, die UTF-8 nicht verstehen, können die Bytes trotzdem (eingeschränkt)

verarbeiten, weil es sich um – wenn auch eher abstruse – „normale“ Zeichen handelt.

- Selbst das byteorientierte Sortieren von UTF-8-Texten funktioniert und sortiert die Strings geordnet nach dem Zahlenwert der enthaltenen Codepoints.
- Bei Verwendung von westeuropäischen Sprachen wird im Vergleich zu UTF-16 viel Speicherplatz gespart, da die meisten Zeichen nur ein Byte benötigen.

Das führt zu folgender Darstellung aller Unicode Zeichen:

Unicode-Bereich	Binäre Zahldarstellung	1. Byte	2. Byte	3. Byte	4. Byte
U+0000-U+007F	00000000 0xxxxxxx	0xxxxxxx			
U+0080-U+07FF	00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
U+0800-U+D7FF, U+E000-U+FFFF	zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
U+10000-U+10FFFF	000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

UTF-8 Implementation des Unicode Zeichensatzes

Hier kann ein einzelnes Zeichen in der UTF-8-Codierung bis zu 4 Bytes umfassen, nach folgenden **Regeln**:

- Ist die Binärdarstellung des Unicode-Codes nicht länger als ein Byte und das das erste Bit eine 0, werden die restlichen 7 Bit gemäß des ASCII Codes verwendet, die 128 verbleibenden Möglichkeiten entsprechen also genau dem ASCII-Code.
- Ist die Binärdarstellung des Unicode-Codes länger als ein Byte oder der Code ist ein Byte lang und beginnt mit einer 1 geht man wie folgt vor:
 - Die binäre Darstellung der Zeichenummer³⁾ wird in 6 Bit lange Teile aufgeteilt.
 - Für jedes dieser 6 Bit Pakete wird ein Byte zur Darstellung verwendet, jedes Byte beginnt mit '10'.
 - Das erste Byte beginnt mit einer '1' für jedes Byte, das verwendet wird. Benötigt man also 3 Byte, um ein Zeichen in UTF-8 darzustellen, beginnt das erste Byte mit '111'. Bevor die Nutzdaten im ersten Byte beginnen, muss nach der Markierung der Anzahl der nötigen Bytes noch eine Null eingefügt werden⁴⁾

Beispiele:

(1)

$$y = 79_{16} = 0111\ 100_2$$

Beginnt mit einer Null und ist nicht länger als ein Byte → die letzten 7Bit werden verwendet, um zu codieren, also ein "ASCII k" in UTF-8

UTF-8: 0110 1011

(2)

$$\ddot{a} = E4_{16} = 1110\ 0100_2$$

Nur ein Byte lang, beginnt aber mit einer 1. Die 8 Bit müssen in 6 Bit Abschnitte geteilt und auf 2 Byte verteilt werden, beginnend von rechts, links wird stets mit 0en aufgefüllt:

000011 100100

- Das zweite Byte beginnt nach den Regeln mit 10, daran schließen die Nutzdaten an: 10 100100
- Das erste Byte beginnt mit 11 (weil man zwei Byte benötigt) dann wird mit 0en aufgefüllt, dann kommen die Nutzdaten: 11 000011

Die UTF-8 Codierung des Unicode-ä ist also 1100 0011 1010 0100. Die Nutzdaten, die den Code des Unicode Zeichens transportieren sind in jeden Byte nur die jeweils letzten 6 Bit.

(3)

乐

Chinese, Japanese, Korean (CJK) unified ideograph (U+4E50)

乐 → U+4E50 → 4E50₁₆ → 0100 1110 0101 0000₂

- 16 Bit Daten zu codieren, dafür braucht man 3 Byte (3 x 6 = 18)
- Der UTF-8 Code beginnt also mit der Startsequenz 1110
- Dann von rechts beginnend 6 Bit (01 000), das Byte beginnt mit 10 (Regel) also ist das dritte Byte 1001 1000
- Die nächsten 6 Bit analog: 1110 01 → 1011 1001
- Die fehlenden 4 Bit 0100 mit Padding + Startsequenz (111) ergeben das erste Byte 1110 0100

Die UTF-8 Codierung des Unicode-Zeichens 乔 ist also 3 Byte lang und sieht so aus: 1110 0100 1011 1001 1001 0000

2

(A1)

Wandle die nachfolgenden Zeichen des Unicode Zeichensatzes in die UTF-8-Codierung um. Der Hexadezimalcode des Unicode Zeichens ist jeweils angegeben.

Gehe jeweils wie in den Beispielen oben vor. Markiere die "Nutzdaten" die das eigentlich Unicode-Zeichen "transportieren".

1. I=49₁₆
2. Ö=D6₁₆
3. 弈=5F08₁₆
4. □=1F60A₁₆

Lösung 1

0**1001001**, ein Byte, erstes Bit 0.

Lösung 2

110**00011** 100**10110**

Lösung 3

1110**0101** 10**111100** 100**01000**

Lösung 4

1111**0000** 100**11111** 100**11000** 100**01010**



(A2)

Wie viele unterschiedliche Unicode-Zeichen lassen sich theoretisch mit 1 Byte, 2 Bytes, 3 Bytes und 4 Bytes unter Beachtung der UTF-8-Regeln darstellen?

Lösung

- 1 Byte: 7 nutzbare Bits $\rightarrow 2^7 = 128$ Zeichen
- 2 Bytes: 5+6 = 11 nutzbare Bits $\rightarrow 2^{11} = 2\,048$ Zeichen
- 3 Bytes: 4+6+6 = 16 nutzbare Bits $\rightarrow 2^{16} = 65\,536$ Zeichen
- 4 Bytes: 3+6+6+6 = 21 nutzbare Bits $\rightarrow 2^{21} = 2\,097\,152$ Zeichen

Modellvorstellung: Der UTF-8-Zug

Nr	Code	Browser	Appl	Goog	FB	Wind	Twtr	Joy	Sams	GMail	SB	DCM	KDDI	CLDR Short Name
1	U+1F600 0x1F600	😊	😊	😊	😊	😄	😊	😊	😊	😊	—	—	—	grinning face

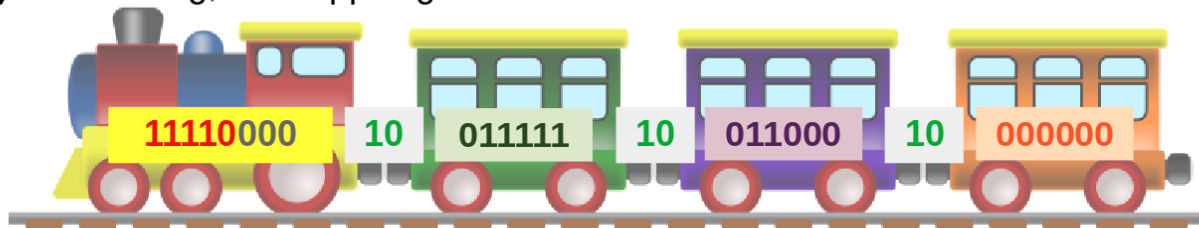
(1) Codepoint Hexadezimal → Codepoint binär

0x1F600 0 0 0 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0

(2) 6-Bit Ladungen – passt noch was in die Lock?

- 1 Byte - 7 Bit 7 Bit in der Lok
- 2 Byte - 11 Bit 5 Bit in der Lok + 6 Bit
- 3 Byte - 16 Bit 4 Bit in der Lok + 6 Bit + 6 Bit
- 4 Byte - 21 Bit 3 Bit in der Lok + 6 Bit + 6 Bit + 6 Bit

(3) UTF-8 Zug, die Kupplungen sind 10



(4) „UTF-8 Lok“, kennt die **Zahl der Zugteile + 0 als Trenner** + Padding + Ladung der Lok → Anhänger mit je 6 Bit

Material

a.png	19.9 KiB 28.11.2023 11:46
cs.png	25.6 KiB 28.11.2023 10:52
unicode.odp	1.2 MiB 29.11.2023 10:26
unicode.pdf	810.2 KiB 29.11.2023 10:26
zug.png	184.4 KiB 29.11.2023 10:25

CC-BY-SA Frank Schiebel, mit Material von Kimmig, ZPG Informatik BW

1) 2)

Teile dieser Wikiseite sind dem SelfHTML Wiki entnommen und stehen unter CC-BY-SA Lizenz

<https://wiki.selfhtml.org/wiki/Zeichencodierung/Unicode>

3)

des Codepoints

4)

Warum?

From:
<https://www.info-bw.de/> -

Permanent link:
<https://www.info-bw.de/faecher:informatik:oberstufe:codierung:zeichencodierung:unicode:start?rev=1701170980>

Last update: **28.11.2023 11:29**

