

FrOSCon 2025
2025-08-16

Marc Haber
Dipl.-Inform.

UTFwhat?!? Unicode für Anfänger

Agenda

- Historische Codierungen
- Unicode
- Transportcodierungen
- Herausforderungen und Fallen
- Sortieren, Vergleichen, Normalisieren
- Optionale Zugabe: Die Geschichte vor diesem Vortrag

Marc Haber, Dipl.-Inform.

- “Zugschluss”
- Wohnort St. Ilgen (bei Heidelberg)
- Jahrgang 1969, verheiratet, 4 Katzen
- Freier IT-Berater
- Arbeitet mit Linux und Netzwerken
- “alles was nichts mit Microsoft oder Apple zu tun hat”
- Seit 2001 Debian Developer
- Würde gerne mal wieder mit Debian arbeiten

Geschichte

- Computer verarbeiten Zahlen, Menschen Buchstaben.
- Codierung notwendig
- Telex: Baudot-Murray-Code (1901)
 - 5 bit pro Zeichen, aber 64 Zeichen
 - Zwei verschiedene Leerzeichen zur Umschaltung



Quelle: Wikipedia

Geschichte

- ASCII: American Standard Code for Information Interchange (1963)
 - 7 bit pro Zeichen, 128 Zeichen
 - Groß- und Kleinbuchstaben, Ziffern, Sonderzeichen
 - Aber kein Platz für internationale Umlaute

Mehr als Amerika

- ASCII 7 Bit mit nationalen Sonderzeichen
 - Ersatz von { | } [\] ~ durch Umlaute und ß
 - ISO 646 (1963!)
- Achtes Bit benutzen
 - ISO-8859 (1987)
 - Nationale Versionen
 - Versionen mit Strichgrafik
 - Kein Platz für unterschiedliche Sprachfamilien
 - ISO-8859-1 für westeuropäische Sprachen
 - ISO-8859-15 mit Euro-Zeichen (1999)

Honorable Mention

- EBCDIC
 - Extended Binary Coded Decimal Interchange Code
 - “IBMs Rache”
 - Außerhalb von Banken und Versicherungen ausgestorben.
- Windows-1252, “Codepages 850/437 etc”
- KOI8 für kyrillisch geschriebene Sprachen

1991: Unicode

- Unicode Consortium
 - Non-profit organization in den USA
- Möchte Text in allen digitalisierbaren Schriften unterstützen
- Unicode 16.0.0, 2024-09-10
 - 154998 Zeichen (“Characters”)
 - 168 Schriften (“Scripts”)
- ISO/IEC 10646 seit 1993, mit minimalen Abweichungen

Unicode

- Wird seit etwa der Jahrtausendwende in den wichtigsten Ökosystemen als “benutzbar” angesehen
- Also auch bei uns in der Free and Open Source Community.
- Hat ISO-8859, KOI8, Codepages, Windows-1252 abgelöst
 - “Altlasten”

Unicode-Zeichen

- Eigenschaften:
 - Uppercase
 - Lowercase
 - Decimal digit
 - Punctuation
- Operationen:
 - Case-änderungen
 - Vergleichen
 - Sortieren

Herausforderungen

- “Confusables”:
 - 1 (Ziffer), l (kleines L), l (großes l)
 - 0 (Ziffer), O (großes o)
 - RFC 8264 Section 12.5
- Homoglyphen
 - A in lateinisch, griechisch, kyrillisch
 - Römische Zahlen
 - Å (nordisch) und Å (Längeneinheit)

Herausforderungen

- (Pre-)Composed Characters
 - é ist eigener Codepoint
 - Kann aber auch aus e und ´ kombiniert werden
 - Reihenfolge egal!
 - In osteuropäischen Sprachen noch viel lustiger:
 - túz (ungarisch: Feuer), mit doppel-Akzent
 - Manche Modifikatoren sind typografisch wirksam, andere nicht.
 - Unterschiede in Sprachen
- Whitespace
- Richtungswechsel

Herausforderungen

- Unterschiedliche Ansichten über Zeichen
 - Buchstabenschriften
 - Silbenschriften
 - Wortschriften
- Unvollständige Fonts
 - Alltäglich!
 - Je exotischer, desto “extravaganter”, desto unvollständig.

Codierungen / Encodings

- Ein Zeichen (“Codepoint”) wird als U+hex geschrieben.
 - Aktuell: U+0000 - U+10FFFF
 - 0 to 1114111
- Codierung in Formate
 - Unicode Transformation Formats (“UTF”)

UTF-8

- Bei weitem am meisten verbreitetes Format
- Im Internet dominant für alle Sprachen und Länder
- In den meisten Standards verwendet
- Oft die einzige erlaubte Codierung
- Unterstützt von allen modernen Betriebssystemen und Programmiersprachen
- Auch IETF-standardisiert (RFC 3629)

UTF-8

- Rückwärtskompatibel zu 7-bit ASCII
 - Die ersten 128 Zeichen sind identisch
 - Codiert mit einem einzigen Byte pro Zeichen
 - Gleiche Binärrepräsentation
- Ein UTF-8 codierter Text der nur (englische) Buchstaben und Zahlen enthält ist identisch zur ASCII-Repräsentation dieses Textes
- Nicht 7-bit-ASCII-Zeichen werden mit 2-4 byte pro Zeichen codiert
- Je höher die Codepointnummer desto länger

UTF-8

Code point ↔ UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0yyyzzzz			
U+0080	U+07FF	110xxxyy	10yyzzzz		
U+0800	U+FFFF	1110wwww	10xxxxyy	10yyzzzz	
U+010000	U+10FFFF	11110uvv	10vwwwww	10xxxxyy	10yyzzzz

UTF-8 encoding example

Character	Binary code point	Binary representation	UTF-8 encoded bytes
W	U+0057 0000 0000 0101 0111	0101 0111	57
B	U+0392 0000 0011 1001 0010	1100 1110 1001 0010	CE 92
위	U+C704 1100 0111 0000 0100	1110 1100 1001 1100 1000 0100	EC 9C 84
Y	U+10345 0 0001 0000 0011 0100 0101	1111 0000 1001 0000 1000 1101 1000 0101	F0 90 8D 85

UTF-8

- Speicherplatzeffizient für westliche Sprachen:
 - Nur nicht-ASCII-Zeichen brauchen > 1 Byte
 - Sonderzeichen westlicher Sprachen kommen mit 2 Bytes aus
 - Blöd für asiatische und arabische Sprachen
- Um Zeichen X aus einem String zu gewinnen muss man immer von Anfang an lesen
- Byte Order Mark optional

Spaß mit Unicode

- Nicht alle Bytesequenzen sind valides UTF-8



- Kommt aber auch bei unvollständigen Fonts vor

Spaß mit UTF-8: Mojibake

Falsches Äœben von
Xylophonmusik quÃlt jeden
grÃÃeren Zwerg

Spaß mit UTF-8: Mojibake

- Code muss sich im Klaren sein wie Strings codiert sind
- Bereits UTF-8 codierte Strings lassen sich noch einmal UTF-8 codieren. Das ist aber falsch!
 - “ü” in Latin 1 0xFC, in UTF-8 0xC3 0xBC
 - 0xC3 in Latin 1 Ã, Ã in Unicode 0xC3 0x83
 - 0xBC in Latin 1 ¼, ¼ in Unicode 0xC2 0xBC
 - Mojibake: 0xC3 0x83 0xC2 0xBC
 - Dargestellt als Ã¼
- “Double-UTF-8”

Spaß mit UTF-8: Overlong Encoding

- Es gibt valide encodings von Zeichen, die länger sind als das kanonische Encoding dieses Zeichens
- Die meisten Libraries können diese Encodings trotzdem korrekt decodieren
- Aber: Schwachstellen in Software erlauben auf diese Weise das Aushebeln von Sicherheitsprüfungen.



Foto: Sandra Haber

UTF-32 (UCS-4)

- 4 bytes pro Zeichen. Immer
- Das einzige UTF mit fixer Länge pro Zeichen
- UTF-32 Repräsentation identisch mit der Codepointnummer
- Um Zeichen X aus einem String zu gewinnen muss man $4X$ errechnen
- Nicht speicherplatzeffizient
- Gibt es als Big und Little Endian
- Kennzeichnung mit Byte Order Mark ist optional 🙄 und selten verwendet 🤡

UTF-32 (UCS-4)

- Wird dort verwendet wenn es schnell sein muss und genug Speicherplatz vorhanden ist
- Neuere Python benutzt UTF-32 intern sobald mehr als US-ASCII in einem String ist

Honorable Mention: UTF-16

- War anfänglich eine Codierung mit fester Länge
- Variable Länge seitdem Codepoints $> U+FFFF$ vergeben wurden
- Vereinigt die Nachteile von UTF-8 mit den Nachteilen von UTF-32
- Hätte mit Vergabe von U+10000 deprecated gehört
- Selten benutzt
- **NICHT NEU VERWENDEN**

Honorable Mention: UTF-16

- Wird aber natürlich leider doch benutzt
 - Windows (“wchars”, z.b. in Filenamen)
 - Javascript
- Zur Vereinfachung wurde um 2000 WTF-8 (“Wobbly Transportation Format”) definiert
 - Nur zur internen Verwendung
 - “Any WTF-8 data must be converted to a Unicode encoding at the system’s boundary before being emitted. UTF-8 is recommended. **WTF-8 must not be used** to represent text in a file format or for transmission over the Internet.”

Byte Order Mark

- Unicode-Strings können mit U+FEFF “Zero Width No-Break Space” beginnen
- Daraus kann der Empfänger die Codierung und Endianness des Strings ablesen
- Nicht besonders weit verbreitet

Sortieren

- Bringt Elemente in eine bestimmte Ordnung
- Mathematisch: Halbordnung bezüglich \leq
- Schon innerhalb eines Scripts herausfordernd
- Deutsch: $\ddot{A}=A$, aber Schwedisch: $\text{\AA} > z$
- Wie sortiere ich unterschiedliche Sprachen? Ist
 - A (griechisch) $>$ A (lateinisch) oder ist
 - A (griechisch) $>$ A (kyrillisch) oder ist
 - A (kyrillisch) = A (lateinisch), aber ist
 - $\alpha = a$?
 - Und was ist mit Transliterationen?

Vergleichen

- Es gibt Zeichen, die gleich aussehen, aber unterschiedliche Bedeutungen haben:
 - Å (nordisch) und Å (Längeneinheit)
- Es gibt Zeichen, die gleich aussehen, aber aus unterschiedlichen Scripts sind
 - A (lateinisch) und А (kyrillisch)
 - Großbuchstaben mit Akzent im Französischen
- Diese Zeichen haben unterschiedliche Codepoints

Anforderungen: Vergleichen

- Suche: Angstrom → Ångström
 - Komfortfunktion
- Aber: Usernamen vergleichen?
 - Wenn es “Claus Ångström” (richtig geschrieben), soll ein Angreifer “Claus Ångström” (mit dem Einheitszeichen) nicht anlegen dürfen
- Vergleich ≠ Vergleichen
 - Muss man sich an jeder Stelle neu überlegen!

Unicode hat

- Verschiedene Positionen
- Gedrehte Zeichen
- Zeichen im Kreis und im Quadrat
- Unterschiedliche Breiten
- Indizes und Potenzen
- Brüche
- Ligaturen

Unicode Equivalence

- Wikipedia: Unicode Equivalence
- Unicode Annex #15¹ definiert Normalisierung

Subtype	Examples
Combining sequence	Ç ↔ C+◌̣
Ordering of combining marks	q+◌̇+◌̣ ↔ q+◌̣+◌̇
Hangul & conjoining jamo	가 ↔ ㄱ + ㅏ
Singleton equivalence	Ω ↔ Ω

Kanonische Äquivalenz: Quelle: Unicode Annex #15

Kompatible Äquivalenz

Subtype	Examples
Font variants	Œ → H
	H → H
Linebreaking differences	[NBSP] → [SPACE]
Circled variants	① → 1
Rotated variants	⌋ → {
	⌌ → }
Superscripts/subscripts	i ⁹ → i ₉
	ī ₉ → i ₉
Fractions	¼ → 1/4
Other	dž → dž

Äquivalenz

- Kanonische Äquivalenz
 - Gleiches Aussehen, gleiche Bedeutung
 - Soll beim Sortieren und Vergleichen gleich behandelt werden
 - Substituierbar
- Kompatible Äquivalenz
 - Unterschiedliches Aussehen, möglicherweise gleiche Bedeutung
 - Nicht unbedingt gleich zu behandeln
 - Nicht unbedingt substituierbar
- Kanonisch \subset Kompatibel

Äquivalenz

- \acute{a} (Zeichen) ist kanonisch äquivalent zu beliebiger Kombination aus a und $\acute{\prime}$
- Ω (Ohm) ist kanonisch äquivalent zum griechischen Omega
- Ein Leerzeichen ist kompatibel äquivalent zum geschützten Leerzeichen
- \mathbb{R} ist kompatibel äquivalent zu R
- i^2 ist kompatibel äquivalent zu $i2$.

Normalisierung laut Standard

- Normalization Form D (NFD):
 - Canonical Decomposition
- Normalization Form C (NFC):
 - Canonical Decomposition,
 - danach Canonical Composition
- Normalization Form KD (NFKD):
 - Compatibility Decomposition
- Normalization Form KC (NFKC):
 - Compatibility Decomposition
 - danach Canonical Composition

Beispiele für Normalisierung

Source

À
212B

:

NFD

A ◌
0041 030A

NFC

À
00C5

Source

À
00C5

:

NFD

A ◌
0041 030A

NFC

À
00C5

Ω
2126

:

Ω
03A9

Ω
03A9

Ô
00F4

:

O ◌
006F 0302

Ô
00F4

Source

š
1E69

:

NFD

s ◌ ◌
0073 0323 0307

NFC

š
1E69

đ
1E0B 0323

:

d ◌ ◌
0064 0323 0307

đ ◌
1E0D 0307

q̇
0071 0307 0323

:

q ◌ ◌
0071 0323 0307

q ◌ ◌
0071 0323 0307

Kompatibilitäts-Normalisierung

Source	NFD	NFC	NFKD	NFKC
fi FB01	: fi FB01	fi FB01	f i 0066 0069	f i 0066 0069
2⁵ 0032 2075	: 2 5 0032 2075	2 5 0032 2075	2 5 0032 0035	2 5 0032 0035
İ 1E9B 0323	: f ̇ ̇ 017F 0323 0307	İ ̇ 1E9B 0323	S ̇ ̇ 0073 0323 0307	Ş 1E69

Beispiel: Normalisierung Form C

- Kanonische Dekomposition
- gefolgt von kanonischer Komposition
- Löst Probleme “Akzent” und “Einheiten”
 - \acute{e} und e^+ werden beide auf \acute{e} abgebildet
 - Ohm und Omega werden beide zu Ω
- Aber nicht “Homograph”:
 - a (lateinisch) und a (kyrillisch) sind auch danach noch unterschiedliche Zeichen

Normalisierung verliert Information!

Normalisierung ist nicht notwendigerweise umkehrbar

- 10 Å
 - “eine physikalische Länge von 10 Ångström
- Nach Normalisierung nur noch
 - “die Zahl 10 gefolgt vom Buchstaben Å”
- A (Ampere) < Å (Ångström) < C (Celsius)?

Normalisierung verliert Information!

- Eine Applikation muss sich entscheiden, ob sie Strings normalisiert oder nicht normalisiert verarbeiten oder abspeichern möchte
- Normalisierung jeweils wiederholen
 - frisst Rechenzeit
- Normalisiert speichern
 - Verliert Information
- Eingabeform und normalisierte Form speichern
 - Frisst Speicher

Die “Lösung” im Standard

- Der Standard enthält Tabellen zur Normalisierung
- Aber: Volle Abdeckung ist nicht garantiert
 - Es gibt Fälle wo ähnlich oder gleich aussehende Dinge einander nicht zugeordnet sind
 - und deswegen unterschiedlich sortiert und verglichen werden
- Das ist sicherheitsrelevant

Wie es entstanden ist

- Ich bin Maintainer von adduser in Debian
- Adduser benutzt useradd als low-level tool
- Bugreport aus Osteuropa: “Regression: Ich kann neuerdings meinen Namen nicht mehr als Usernamen nehmen!”
- “Aber ich hab doch gar nichts geändert!”
- Geändert hatte sich useradd.
 - Nur noch ASCII als Username erlaubt

Wie es entstanden ist

- Ich so:
 - “Warum eigentlich nicht Unicode in Usernamen?”
 - “Wäre doch nett!”
 - “Hat doch jetzt die ganze Zeit funktioniert!”
- Diskussion auf Debian-Devel.
 - “Mach das nicht, lass das!”
- Eskalation bis nach Linux Weekly News
- Ich lese Standards und Dokumentation

Wie es entstanden ist

- Okay, das ist aufwendig
- Dafür gibt es Libraries
- Adduser ist
 - Top 5 in Popcon und im Installer
- Designziel: Wenige Dependencies
- Ausrede: RFC 8265 “Preparation, Enforcement, and Comparison of Internationalized Strings (PRECIS) - Representing Usernames and Passwords”
- Also haben wir es gelassen

Vielen Dank für Eure Geduld

Noch Fragen?

Vielen Dank für Eure Geduld



Noch Fragen?

Marc Haber
Fediverse: @Zugschluss@zug.network

mh+froscon25@inclusion.de
<https://www.inclusion.de/>

Foto: Sandra Haber