Encoding

Inhalt

1) Was heißt Encoding?
2) ASCII
3) ISO
4) Unicode
5) UTF-8
6) UTF-16
7) Encoding unter Linux

Was heißt Encoding?

- Wörter, bzw. Strings bestehen aus Zeichen
  → Zeichen sind: „a, á, ह, 請, щ, etc.

- Zeichen können in maschinellen Speichern nicht direkt abgelegt werden
  → jedem Zeichen wird eine Nummer zugeordnet
  → **Zeichencode** (codepoint)
  → durch ein oder mehrere Byte repräsentiert

- Zeichen werden zu **Zeichensätzen** (character set) zusammengefasst

- **Zeichencodierung** (encoding) ist der „Schlüssel“, der Zeichencodes umsetzt
  → Ausgabe als lesbare Zeichen möglich


100001
A


100001
A

! Es gibt viele verschiedene Zeichensätze !

- viele unterschiedliche Zuordnungen zwischen den Zeichencodes und Zeichen

- Falsches Encoding kann zu fehlerhafter Darstellung der Zeichen führen
  → über wird zu ber

  Man muss nicht alles bis ins letzte Detail wissen, aber man sollte Zeichencodierung erkennen und anwenden können

ASCII

- American Standard Code for Information Interchange
- Eine der ersten Zeichencodierungen
- Umfasst das lateinische Alphabet in Groß- und Kleinschreibung, 10 arabische Zahlen, Interpunktions-, Sonder- und Steuerzeichen
  → entspricht der Tastatur der englischen Sprache
- Insgesamt 128 Zeichen (33 nicht druckbar)

Ein ASCII codiertes Zeichen ist 7 Bit groß - was heißt das?

Jedes Zeichen ist mit einem Bitmuster aus 7 Bit codiert

A =

	1	0	0	0	0	0	1

ein Byte sind immer 8 Bit, ein Bit bleibt also ungenutzt

0	1	0	0	0	0	0	1

wird mit 0 „aufgefüllt“, ändert den Zahlenwert nicht

daraus ergeben sich 2^7 mögliche Zeichencodes, also 128 verfügbare Zeichen

- ASCII enthält keine Zeichen, die innerhalb der englischen Sprache nicht gebraucht werden
  - → Akzente, wie: à
  - → Umlaute
  - → Alphabete wie Kyrillisch, Griechisch, etc.
  - → logographische Schriftsysteme wie Chinesisch
  - → Emojis
  - → etc.

Um diese Zeichen darstellen zu können, wurden andere Zeichensätze benötigt

ISO 8859

- Normenfamilie von der Internationalen Organisation für Normung (ISO) entwickelt
- Enthält 15 8-Bit-Zeichensätze
- Die ersten 128 Positionen entsprechen bei allen Teilstandards der ASCII-Norm
- Unterscheiden sich jeweils in den regionalen Sonderzeichen
  → z.B. Griechisch benötigt andere Buchstaben als Hebräisch

ISO 8859

-1, Latin - 1	Westeuropäisch
-2, Latin - 2	Mitteleuropäisch
-3, Latin - 3	Südeuropäisch
-4, Latin - 4	Nordeuropäisch
-5,	Kyrillisch
-6	Arabisch
-7	Griechisch
-8	Hebräisch
-9, Latin - 5	Türkisch
-10, Latin - 6	Nordisch
-11	Thai
-12	Existiert nicht
-13, Latin - 7	Baltisch
-14, Latin - 8	Keltisch
-15, Latin -9	Westeuropäisch
-16, Latin - 10	Südosteuropäisch

Jedes Zeichen ist mit einem Bitmuster aus 8 Bit codiert

- Jedes Zeichen der ISO 8859 Normen ist in 8 Bit/ 1 Byte codiert.

1	1	0	0	0	0	0	0

codiert z.B. in Latin – 9 das Zeichen À (nicht in allen Normen einheitlich)

Latin – 9 ist die neuere Version des Westeuropäischen Zeichensatzes; enthält z.B. neu das € Zeichen

daraus ergeben sich 2^8 mögliche Zeichencodes, also 256 verfügbare Zeichen

Unicode

- Universal Character Set (UCS)/ ISO 10646
- ISO 8859 deckt auch nicht alle Zeichensysteme ab und ist nicht einheitlich
  → Chinesisch, Japanisch, Arabisch, etc.
- UCS enthält alle bekannten Schriftzeichen eindeutig codiert
- Erweiterung der Codierung von 8 auf 32 Bit, also von 1 auf 4 Byte
  → multibyte-Kodierung

Jedes Zeichen ist mit einem Bitmuster aus 32 Bit codiert

A =

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1

jedes Zeichen besteht aus 4 Byte; bei ASCII konformen Zeichen

macht das 3 Byte ohne Information mit gleichem Speicheraufwand

daraus ergeben sich 2^{32} mögliche Zeichencodes, also 4.294.967.296 verfügbare Zeichen

- Mit Unicode ist meistens UCS -2 gemeint, die sog. „Basic Multilingual Plane“
  → erste Ebene des Unicode

- Codierung erfolgt in 16 statt in 32 Bit

- Auf den anderen Ebenen sind Hiroglyphen, selten genutzte Chinesische Zeichen etc. codiert
  → hierfür 16 Bit pro Zeichen nicht mehr ausreichend

- Volles Unicode Set: UCS-4 bzw. UTF-32

UTF - 8

- UTF = Unicode Transformation Format

- Erlaubt flexible Breite für verschiedene Zeichen
  → Bytes ohne Information werden nicht codiert
  → 1-6 Bytes

- Umfasst alle Zeichen des vollen Unicode Standards

- Heute am häufigsten genutzter Zeichensatz für Websites (ca. 88%)

Wie funktioniert die Codierung?

Codierung in UTF - 8

Da die Breite unterschiedlich ist, kann nicht wie bei den anderen Zeichensätzen in „gleich große Stücke“ unterteilt werden

→ die Maschine muss wissen welche Zeichen aus mehr als einem Byte bestehen

Lösung: Jedes Zeichen gibt an, ob es ein Multibyte Character ist und aus wie vielen Bytes es besteht

Größe = 1 Byte: 0 ? ? ? ? ? ? ?

Beginnt ein Symbol in UTF-8 mit einer 0, folgt kein weiteres Byte
→ ASCII konform

Größe = 2 Byte: 1 1 0 ? ? ? ? ? 1 0 ? ? ? ? ? ?

Beginnt ein Symbol hingegen mit 110, folgt genau ein weiteres Bytes
→ das Folgebyte beginnt mit 10

Größe = 3 Byte: 1 1 1 0 ? ? ? ? 1 0 ? ? ? ? ? ?

1 0 ? ? ? ? ? ?

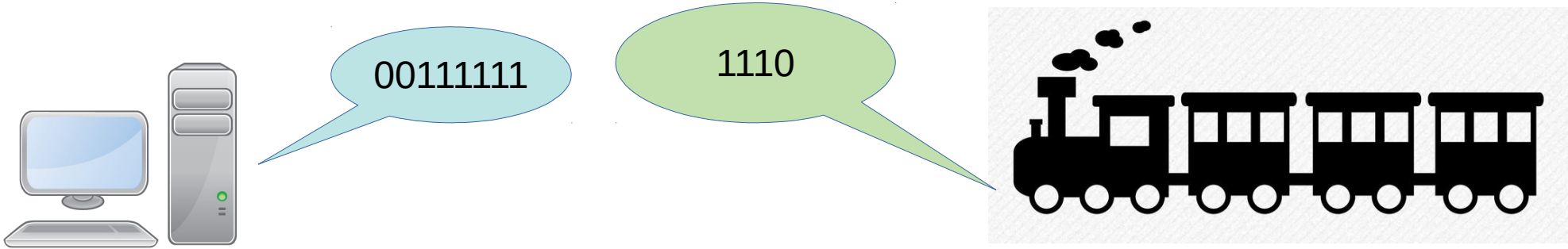
Beginnt ein Symbol mit 1110, folgen genau 2 weitere Bytes
→ jedes Folgebyte beginnt mit 10

Größe = 4 Byte:

1	1	1	1	0	?	?	?	1	0	?	?	?	?	?	?
1	0	?	?	?	?	?	?	1	0	?	?	?	?	?	?

Beginnt ein Symbol mit 11110, folgen 3 weitere Bytes
→ alle weiteren Bytes beginnen mit 10

<u>Bildlich gesprochen:</u>
Jedes UTF-8 Zeichen ist eine Lokomotive, die Waggons hinter sich herziehen kann. Bevor sie in den Bahnhof einfahren kann, muss sie aber erst angeben, wie lang ihr Zug insgesamt ist.



UTF - 16

- Ähnlich UTF-8, aber:
  → Komplex von 1 oder 2 16-Bit Einheiten
  → kann auch alle Unicode Zeichen abbilden

- Im BMP Bereich sehr effektiv, braucht aber bei ASCII Zeichen viel Speicherplatz

  Problem: es gibt 2 verschiedene Varianten
  → UTF-16BE (big-endian)
  → UTF-16LE (little-endian)

  Was ist der Unterschied?

Big-Endian und Little-Endian

Little- und Big-Endian geben die Ordnung an, in der die Bytes im Speicher abgelegt werden:

BE: Das „große Ende“ wird zuerst abgelegt (also an der kleineren Speicheradresse)

LE: Das „kleine Ende“ wird zuerst abgelegt

Big-Endian Beispiel:

Wir speichern die Zahl 4F52

→ erstes Byte (4F) liegt im Speicher auf Adresse 1000

→ zweites Byte (52) liegt im Speicher auf Adresse 1001

Little-Endian Beispiel:

Wir speichern die Zahl 4F52

→ erstes Byte (4F) liegt im Speicher auf Adresse 1001

→ zweites Byte (52) liegt im Speicher auf Adresse 1000

Das heißt:
Wird nicht angegeben um welche Speicherreihenfolge es sich handelt, könnte die Zahl sowohl als 4F52, als auch als 524F interpretiert werden.

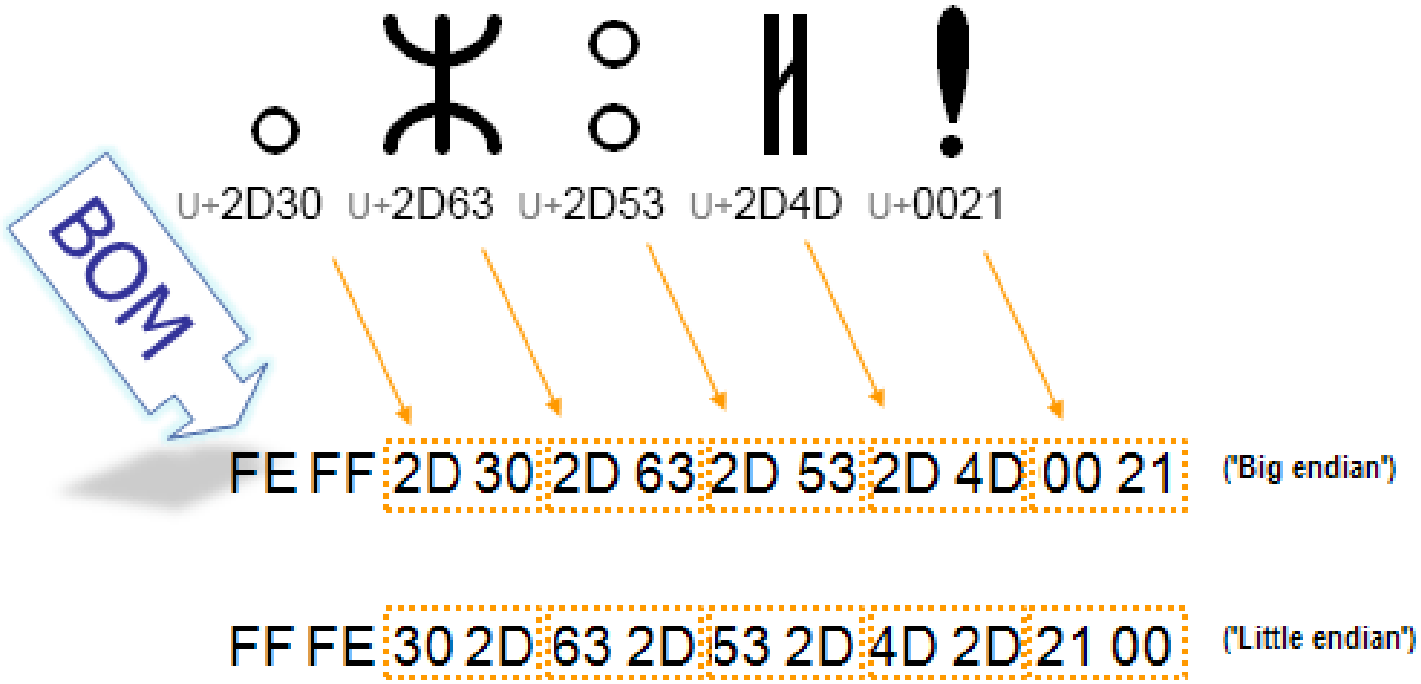
Lösung: Byte Order Marks (BOM)

Bytes am Anfang des Datenstroms, die kein anderes Zeichen im Unicode codieren:
→ Big-Endian: U+FE FF
→ Little-Endian: U+FF FE
(taucht in UTF-8 als EF BB BF auf)

Da das BOM kein eigentliches Zeichen codiert, ist es bei korrekter Verwendung normalerweise nicht sichtbar

Ein Programm, das kein BOM „erwartet“ kann es nicht auslesen!

Beispiel:



Quelle: https://www.w3.org/International/questions/qa-byte-order-mark.de

Encoding unter Linux

Encoding erkennen

Hex dump:

Liest aus Dateien und gibt die einzelnen Bytes hexadezimal, dezimal, oktal oder in ASCII auf die Standardausgabe

(hexadezimal per Default)

Befehl: hexdump/ hd

Octal dump:

Liest aus Dateien und gibt die einzelnen Bytes hexadezimal, dezimal, oktal oder in ASCII auf die Standardausgabe

(octal per Default)

Befehl: od

Aber:

Es existiert immer „Abwärtskompatibilität“ zu ASCII. Das heißt:

1) ASCII ist auch immer ISO-Latin (ein in ASCII codierter Text kann mit einem ISO-Latin Encoding fehlerfrei ausgelesen werden)
2) ASCII ist auch immer UTF-8 (s.o.)

Kann das auch in die „andere Richtung“ gelten?

Nein, denn:

Abwärtskompatibilität gibt es aus dem Grund, dass die ASCII Zeichen sowohl in ISO-Latin, als auch in UTF-8 ihren ASCII-Code beibehalten.

Enthält ein Text Sonderzeichen, die zwar in ISO Standards oder im Unicode vorkommen, existiert dafür kein entsprechender Code in ASCII.

Daraus folgt:

ISO-Latin und UTF-8 sind nicht immer auch ASCII, *können* aber ASCII konform sein!

Beispiel hexdump

1. dump:

```
00000000  56 65 72 73 70 c3 a4 74  75 6e 67 0a 0a           |Versp..tung..|
0000000d
```

2. dump:

```
00000000  56 65 72 73 70 e4 74 75  6e 67 0a 0a           |Versp.tung..|
0000000c
```

Das Wort ist in beiden Dateien „Verspätung“. Welche der Dateien ist Iso-Latin und welche UTF-8?

Beispiel hexdump

1. dump:

```
00000000  56 65 72 73 70 c3 a4 74  75 6e 67 0a 0a           |Versp..tung..|
0000000d
```

2. dump:

```
00000000  56 65 72 73 70 e4 74 75  6e 67 0a 0a           |Versp.tung..|
0000000c
```

Der Dump unterscheidet sich an einem Buchstaben, dem „ä“. Das ist in Iso-Latin ein Byte, in UTF-8 2 Byte groß.

Encoding ändern

iconv:

Konvertiert von einem Encoding in ein anderes:

Befehl:

iconv -f (from) -t (to) -o (output)

z.B.:

iconv -f ISO-8859-15 -t UTF-8 -o utf8.txt iso.txt (Eingabedatei)

uniconv:

Konvertiert von einem Encoding in ein anderes:

Befehl:

uniconv -out (output)-decode -encode

z.B:

uniconv -out utf8.txt -decode iso-8859-15 -encode utf-8 iso.txt (Eingabedatei)

Gut zu wissen:

- Newline Character \n ist folgendermaßen codiert:
  - → dezimal: 10
  - → hexadezimal: 0a
  - → octal: 12

- Es gibt Unterschiede zwischen den Systemen:
  - → Unix: \n (s.o.)
  - → Windows: LF/CR
  - → Mac: CR