Kodierung

by

Dr. Günter Kolousek

Überblick

- Kodierung (auch Code): Abbildung, die jedem Zeichen eines Quellalphabets (Menge!) eindeutig ein Zeichen eines Zielalphabets zuordnet.
  - kodieren vs. dekodieren
  - Kodierung mit
    - fixer Länge (z.B. ASCII)
    - variable Länge (z.B. UTF-8)
- Zweck
  - Speicherung
  - Informationsaustausch
  - Verarbeitung

Anwendungen

- Zeichenkodierung
  - $\rightarrow$ ”`character_encoding`”
- Zahlenkodierungen
  - ganze Zahlen vs. Gleitkommazahlen
  - Zahlen mit einer variablen Länge zur Datenübertragung
- Leitungskodierung
  - z.B.: Manchesterkodierung, Morsecode
- Fehlererkennende und fehlerkorrigierende Codes
  - z.B.: CRC
- Komprimierung
  - z.B.: Huffman-Kodierung

Zahlen mit variabler Länge

- Zweck: Übertragen und Speichern einer beliebig großen Zahl
- LEB128
  - Little Endian Base 128
    - Unsigned LEB128
    - Signed LEB128

Unsigned LEB128

1. Zahl binär darstellen
2. 0en bis auf Vielfaches von 7 links auffüllen
3. in 7er Gruppen teilen
4. auf 8 Bits bringen: MSB setzen in jeder Gruppe außer der höchstwertigsten
5. Daten beginnend mit dem niederwertigsten Byte übertragen

Unsigned LEB128 – 2

1. $123456_{10} = 11110001001000000_2$
2. 000011110001001000000
3. 0000111 1000100 1000000
4. 00000111 11000100 11000000
5. Übertragen: 11000000 11000100 00000111

Signed LEB128

1. Zahl binär darstellen
   - negativ $\rightarrow$ positiv, 0 Bit hinzu, 2er-Komplement
2. VZ bis auf Vielfaches von 7 links auffüllen
3. in 7er Gruppen teilen
4. auf 8 Bits bringen: MSB setzen in jeder Gruppe außer der höchstwertigsten
5. Daten beginnend mit dem niederwertigsten Byte übertragen

Achtung: Empfänger muss wissen, ob *Signed LEB128* oder *Unsigned LEB128*!

Signed LEB – 2

1. $-123456_{10} = -1 \cdot 1111000100100000_2 = -1 \cdot 011110001001000000_2 = 10000111011100000_2$
2. 11110000111011000000
3. 1111000 0111011 1000000
4. 01111000 10111011 11000000
5. Übertragen: 11000000 10111011 01111000

Konfiguration & Programmierung

- Betriebsystem konfigurieren!
- Terminal konfigurieren!
- Editor konfigurieren!
  - aber auch: Fonts installieren…
- HTML

```
<meta charset="UTF-8">
<meta http-equiv="Content-type"
      content="text/html; charset=UTF-8">
```

- Webbrowser
  - → wenn im HTML nicht spezifiziert…

Konfiguration & Programmierung – 2

- HTTP
  - Header für Inhalt

    ```
    Content-Type: text/plain; charset="UTF-8"
    ```
- Datenbank
  - Datenbank, Tabelle, Spalte
- XML
  - jeder konforme XML-Prozessor muss UTF-8 unterstützen

  ```
  <?xml version="1.0" encoding="UTF-16"?>
  ```
- Programmierung
  - z.B. Java

    ```
    String name;
    // ...
    byte[] bytes = name.getBytes("utf-8");
    // ...
    name = new String(bytes, "utf-16")
    ```

Anwendungen – 2

- Verschlüsselung
  - aber nicht jeder Code ist eine Verschlüsselung!
  - z.B. DES, AES, RSA,…
- Identifizierung von Gegenständen,…
  - z.B.: ISBN-10 bzw. ISBN-13 (International Standard Book Number), ISSN (für Zeitschriften), GTIN (Global Trade Item Number), QR-Code
- Geekcode

```
GCS s a++ C UL++ L+++ E++ !tv b++ e++++ h----
```

  - if you are really curious… → http://www.joereiss.net/geek/ungeek.html

ISBN

- Beispiel: 3765457280 bzw. 978-3765457289
- Aufbau
  - Präfix: 978 oder 979 (keiner bei ISBN-10)
  - Gruppennummer (national, sprachlich): 3 (deutsch)
  - Verlagsnummer (variabel, abhängig von Gruppennummer): 7654
  - Titelnummer: 5728
  - Prüfziffer: 9
- Prüfziffer für ISBN-13
  - Berechnung
    - $z_{13} = (10 - ((z_1 + z_3 + z_5 + z_7 + z_9 + z_{11} + 3(z_2 + z_4 + z_6 + z_8 + z_{10} + z_{12})) \bmod 10)) \bmod 10$
  - Überprüfung
    - $(z_1 + z_3 + z_5 + z_7 + z_9 + z_{11} + z_{13} + 3(z_2 + z_4 + z_6 + z_8 + z_{10} + z_{12})) \bmod 10 = 0$

Kodierung für die Datenübertragung

- Quellenkodierung
- Kanalkodierung
- Leitungskodierung

Quellenkodierung

- Aufgabe: Signale der Quelle einer Redundanzreduktion zu unterwerfen
- ursprüngliches Signal enthält oft redundante Anteile, die nicht benötigt werden (z.B. Audio, Video) oder Datenkompression
- verlustlos vs. verlustbehaftet

Quellenkodierung

- Aufgabe: Signale der Quelle einer Redundanzreduktion zu unterwerfen
- ursprüngliches Signal enthält oft redundante Anteile, die nicht benötigt werden (z.B. Audio, Video) oder Datenkompression
- verlustlos vs. verlustbehaftet
  - verlustlos
    - Lauflängenkodierung
    - Kodierung mit variabler Länge, z.B. Huffman-Kodierung
  - verlustbehaftet
    - z.B. JPEG
    - z.B. Audio: MPEG-1 Level III (mp3)
    - z.B. Audio & Video: MPEG-4 (mp4)

Lauflängenkodierung

- lange Folgen sich wiederholender Zeichen $\rightarrow$ # der Wiederholungen und Zeichen
- Bsp.: AAAAAAXXXXTTTTQUUU $\rightarrow$ 6A4X4T1Q3U
- Binäre Daten $\rightarrow$ Angabe des Zeichens nicht notwendig
  - 0000011110000001010 $\rightarrow$ 5461111

Huffman-Kodierung

- wird für Texte oder binäre Daten (z.B. PNG) verwendet
- variable Länge
  - häufige Zeichen weniger Bits als seltene Zeichen
  - redundanzfrei
  - $\rightarrow$ optimale Kodierung!
- präfixfrei
  - kein Codewort ist der Beginn eines anderen Codewortes
  - $\rightarrow$ keine Trennzeichen zwischen Codewörtern nötig!
- basierend auf Baum
  - Blätter stehen für die Codewörter
- Quellalphabet T
- Codealphabet C, $n = |C|$

Huffman-Kodierung – 2

- Kodieren
  1. n ermitteln
  2. je Symbol $t \in T$: relative Häufigkeit p_t ermitteln
     - Anzahl ermitteln ÷ Gesamtanzahl aller Symbole
  3. je Symbol: einen Knoten mit relativer Häufigkeit erstellen
     - d.h. je ein Baum mit genau einem Knoten
  4. Wiederholen bis nur mehr ein Baum:
     - 4.1 Alle n Bäume mit geringster Häufigkeit in Wurzel auswählen
     - 4.2 Ausgewählte Bäume zu neuem Baum zusammenfassen
     - 4.3 Summe der Häufigkeiten der direkten Kinder addieren und in neuer Wurzel notieren
  5. Codewörterbuch erstellen
  6. Je Symbol im Codewörterbuch nachschlagen

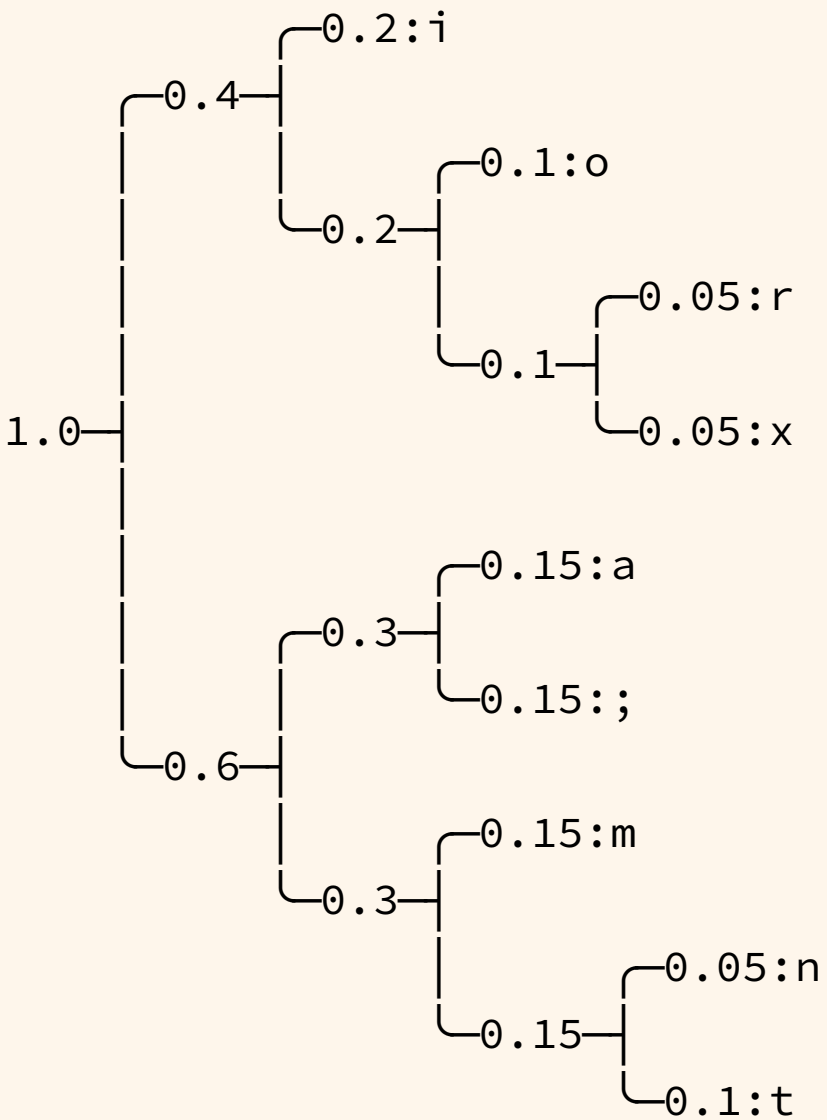
Huffman-Kodierung – Beispiel

- Text: `maxi;mini;otto;maria`
- Quellalphabet: $T = \{m, a, x, i, ;, n, o, t, r\}$
- Codealphabet: $C = \{1, 0\}$

1. $n = 2$
2. relative Häufigkeiten ermitteln:
   $p_m = 0.15, p_a = 0.15, p_x = 0.05, p_i = 0.2, p_; = 0.15, p_n = 0.05, p_o = 0.1, p_t = 0.1, p_r = 0.05$
3. ...

Huffman-Kodierung – Beispiel – 2

4. Baum



Huffman-Kodierung – Beispiel – 3

5. Codewörterbuch

i	00
o	010
a	100
;	101
m	110
r	0110
x	0111
t	1111
n	1110

Huffman-Kodierung – 3

- Wortlänge
  - naive Kodierung: $\log_2(9) = 3.17 \rightarrow$ 4 Bits je Zeichen $\rightarrow$ 80 Bits
  - Huffman $\rightarrow$ 61 Bits $\rightarrow$ 3.05 Bits je Zeichen
    - mittlere Wortlänge konkret: $61 \div 20 = 3.05$
    - mittlere Wortlänge mittels Auftrittswahrscheinlichkeiten: $(2 \cdot 0.2 + 3 \cdot 0.1 + 4 \cdot 0.05 + 4 \cdot 0.05 + 3 \cdot 0.15 + 3 \cdot 0.15 + 3 \cdot 0.15 + 4 \cdot 0.05 + 4 \cdot 0.1) \div 9 = 3.05$
- Dekodieren
  1. aus Codewörterbuch Baum erstellen
  2. Je Symbol im Baum bis Blatt navigieren $\rightarrow$ Symbol gefunden

Kanalkodierung

- Fehlerarten:
  - Rauschen (Störgröße mit breitem Frequenzspektrum)
  - Kurzzeitstörungen (magnetische Felder)
  - Signalverformung (z.B. physikalische Eigenschaften Kabel)
  - Nebensprechen (durch kapazitive Kopplung)
- Aufgabe: Erkennung und Korrektur von Fehlern
  - Erkennung: $\rightarrow$ Neuübertragung
  - Korrektur: Netzwerke mit hoher
    - Fehlerwahrscheinlichkeit (z.B. GSM)
    - Latenz ($\rightarrow$ Neuübertragung dauert lange)
- Idee: zusätzliche Prüfbits (redundante Information)
- Methoden
  - Berechnung und Übertragung eines Codewertes
  - Hinzufügen nicht gültiger Codewörter zum Code

Hamming-Distanz

- Hamming-Distanz Δ zweier Codewörter: Anzahl der unterschiedlichen Bitpositionen
- Hamming-Distanz d eines Codes: Minimaler Werte aller Distanzen
- Beispiel: $C = \{1001, 1111, 0100\}$
  - $\Delta(x, y) = |x \oplus y|_1$
  - $\Delta(1001, 1111) = 2$
  - $\Delta(1001, 0100) = 3$
  - $\Delta(1111, 0100) = 3$
  - $\rightarrow d(C) = 2$
- Fehlererkennung: $d - 1$
- Fehlerkorrektur: $\lfloor \frac{d-1}{2} \rfloor$

Fehlererkennung – Paritätsbits

- Parität einer Zahl: Eigenschaft, ob diese gerade oder ungerade ist.
- $\rightarrow$ Hinzufügen von Paritätsbits, sodass die Anzahl der Einsen gerade ist
- Arten
  - eindimensionale Parität
  - zweidimensionale Parität

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

$\rightarrow$ erkennt alle 1-, 2- und 3-Bitfehler sowie die meisten 4-Bitfehler!

Fehlererkennung – Prüfsummen

- Prinzip:
  - Sender: mathematische Operationen $\rightarrow$ Prüfsumme $\rightarrow$ mitübertragen
  - Empfänger: mathematische Operationen $\rightarrow$ Prüfsumme $\rightarrow$ vergleichen
- Beispiel: IPv4 Prüfsummen
  - Sender
    - Daten als 16-Bitwörter
    - aussummieren mittels Einerkomplementarithmetik (normale binäre Addition, jedoch wird eine 1 am Ende addiert, wenn Übertrag)
    - von Ergebnis Einerkomplement bilden $\rightarrow$ mitübertragen
  - Empfänger
    - Berechnung wie Sender
    - zum Ergebnis Prüfsumme addieren
    - Ergebnis ungleich 0xFFFF $\rightarrow$ Fehler

Beispiel: IPv4 Prüfsummen

- mit 8-Bitworten:

1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	1
3	1	0	1	0	0	1	0	1
4	0	0	1	0	0	1	0	0

- Sender
  - Rechengang
    1. Zeile 1 + Zeile 2: 10000001
    2. Ergebnis von Schritt 1 + Zeile 3: 100100110
    3. Übertrag addieren: 00100111
    4. Zeile Ergebnis von Schritt 3 + Zeile 4: 01001011
    5. Einerkomplement bilden: 10110100 (= Prüfsumme)
- Empfänger

```
01001011  wie Sender Schritte 1-4
10110100  Prüfsumme
--------
11111111
```

Fehlererkennung – CRC

- ebenfalls Prüfsumme
  - ausgelegt, dass Fehler durch Rauschen mit hoher Wahrscheinlichkeit erkannt wird.
  - kann einfach in HW implementiert werden.
- Nachricht der Länge m wird als Polynom mit dem Grad $m-1$ aufgefasst.
- Polynom wird durch ein gewähltes Polynom mit dem Grad k (Generatorpolynom) dividiert.
- Der entstehende Rest wird zur Bildung der Prüfziffern herangezogen.
- Bei ”gutem” Generatorpolynom, dann
  - alle Fehler mit ungerader Anzahl an Bitfehlern
  - alle Bündelfehler der Länge $\leq$ k
- $\rightarrow$ Polynomarithmetik und Binärarithmetik

Fehlererkennung – CRC – 2

Nachricht	Nachricht multipliziert mit Generator	Codewort
000	$0 \odot (x \oplus 1) = 0$	0000
001	$1 \odot (x \oplus 1) = x \oplus 1$	0011
010	$x \odot (x \oplus 1) = x^2 \oplus x$	0110
011	$(x \oplus 1) \odot (x \oplus 1) = x^2 \oplus 1$	0101
100	$x^2 \odot (x \oplus 1) = x^3 \oplus x^2$	1100
101	$(x^2 \oplus 1) \odot (x \oplus 1) = x^3 \oplus x^2 \oplus x \oplus 1$	1111
110	$(x^2 \oplus x) \odot (x \oplus 1) = x^3 \oplus x$	1010
111	$(x^2 \oplus x \oplus 1) \odot (x \oplus 1) = x^3 \oplus 1$	1001

- erzeugte Codewörter bilden 3-Bit-Binärcode mit angehängtem Paritätsbit.
- einfaches Generatorpolynom $\rightarrow$ kein Vorteil gegenüber Paritätsbits…
- Prinzip: Alle Wörter, die nicht durch Generatorpolynom teilbar $\rightarrow$ Fehler!

Fehlererkennung – CRC – 3

1. Multipliziere Nachricht p mit x^k. D.h. es werden k Nullbits am rechten Ende der Nachricht angehängt. Leicht durch Verschieben realisierbar.

   - p = 10001001, d.h. als Polynom: $x^7 \oplus x^3 \oplus 1$
   - als Generatorpolynom wählen wir CRC-4, d.h.: $x^4 \oplus x \oplus 1$.
   - p mit x^k multiplizieren: $(x^7 \oplus x^3 \oplus 1) \odot x^4 = x^{11} \oplus x^7 \oplus x^4$. Als Bitmuster: 100010010000!

Fehlererkennung – CRC – 4

2. Teile erhaltenes Ergebnis (Modulo-2 Arithmetik) durch das Generatorpolynom: $\rightarrow$ Restpolynom

```
100010010000
10011
-----
00010001
   10011
   -----
   00010000
      10011
      -----
      000110
```

- Division durch sukzessives Abziehen des Generatorpolynoms
- Differenzoperator: $\ominus$ herangezogen (leicht durch XOR)

Fehlererkennung – CRC – 5

3. Restpolynom zum Ergebnis von Punkt 1 addieren: $(x^{11} \oplus x^7 \oplus x^4) \oplus (x^2 \oplus x)$. D.h. es ergibt sich der Bitstring 100010010110. Ebenfalls leicht durch XOR realisierbar, da letzte Stellen alle 0 (siehe Punkt 1) und Anzahl der Stellen des Restes…
4. Übertragung
5. Empfänger dividiert empfangenes Polynom durch Generatorpolynom (wie Punkt 2). Entsteht ein Rest ungleich Null, dann ist ein Fehler aufgetreten.

Leitungskodierung

- Aufgabe: Umwandlung digitaler Signale zur Übertragung über den (physischen) Übertragungskanal
- hauptsächlich im Basisband
- binäre Signale meist durch 2 Pegeln
  - positives Potential U+ (z.B. 5V) $\equiv$ 1, Nullpotential $\equiv$ 0
  - positives Potential U+ $\equiv$ 1, negatives Potential U- $\equiv$ 0
- 3 Kriterien
  - Gute Ausnützung der Bandbreite
  - Gute Regenerierung des Sendetaktes
  - Möglichst geringer Gleichspannungsanteil

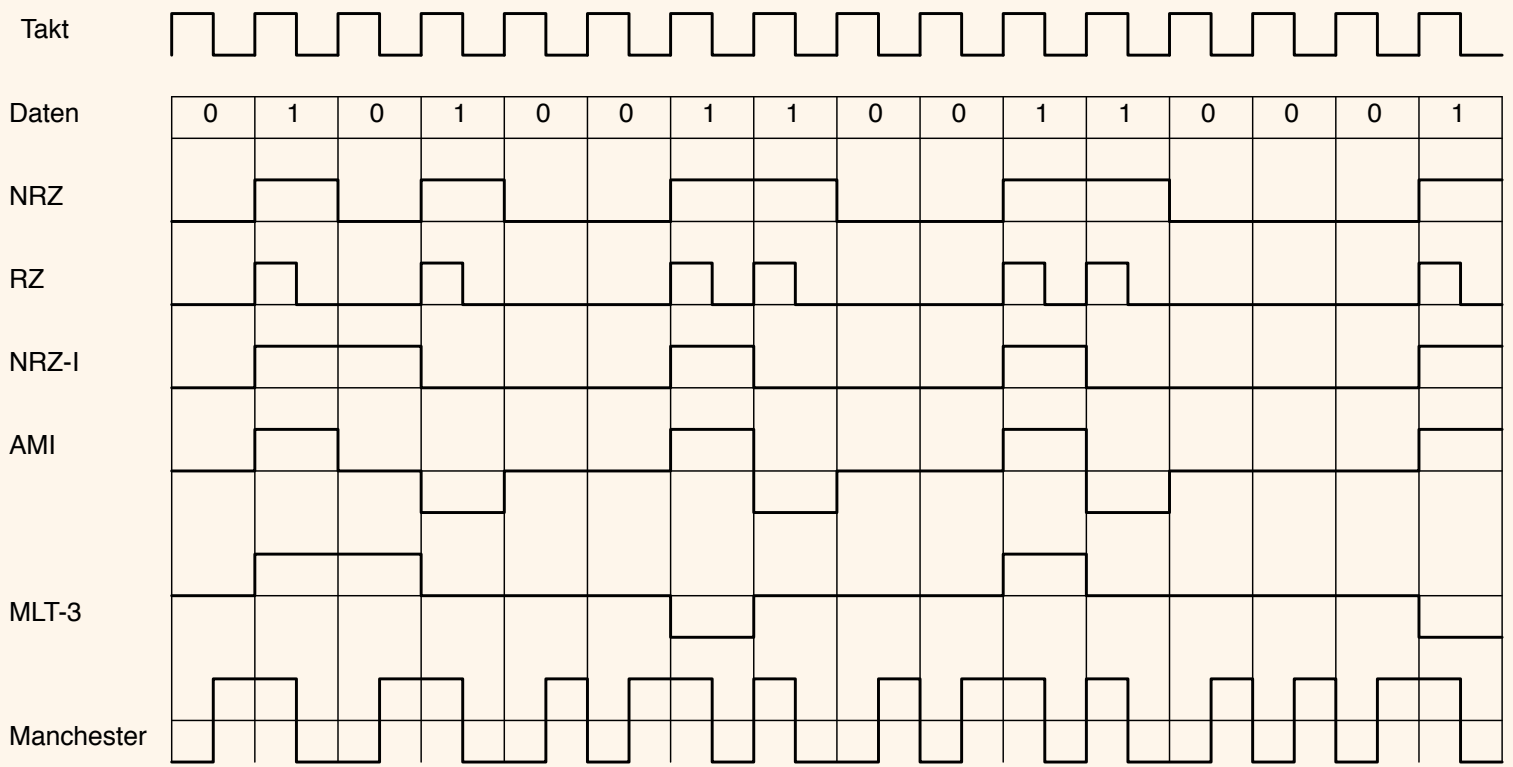
Leitungskodierung – 2

- NRZ (Non-Return to Zero): eigentlich keine richtige Kodierung…
- RZ (Return to Zero)
  - Vermeidung langer Perioden von U+ bei bei langen Folgen von Einsen
    - sonst: höherer Gleichspannungsanteil, schlecht Regenerierung
    - allerdings: höhere Frequenz!
- NRZ-I (Non-Return to Zero Inverse)
  - Bei jeder 1 $\rightarrow$ Pegelwechsel, 0 $\rightarrow$ keine Änderung
    - löst Problem aufeinanderfolgender Einsen
    - …aber nicht aufeinanderfolgende Nullen
- AMI (Alternate Mark Inversion)
  - 1 abwechselnd U+ und U-, 0 $\equiv$ Nullpotential
  - kein Gleichspannungsanteil, lange Perioden von 0er!

Leitungskodierung – 3

- MLT-3 (Multilevel Transmission Encoding)
  - 1 abwechselnd …,0,U+,0,U-,0,U+,…
  - 0 keine Änderung
  - ähnlich AMI
- Manchester Code
  - 1 $\equiv$ Übergang von U+ zu U- (negative Flanke)
  - 0 $\equiv$ positive Flanke
  - de facto kein Gleichspannungsanteil, gute Taktrückgewinnung
  - Möglichkeit Codeverletzungen zu erkennen
    - oder absichtlich einbauen, um z.B. Anfang/Ende eines Frame zu erkennen
  - Verdopplung des Frequenzbandes!

Leitungskodierung – 4



Blockkodierung

- Ziele
  - Vermeidung langer Folgen von 0en und 1en
  - Zusatzinformationen mitübertragen wie beim Manchester-Code
- Notation: mBnx
  - m … Anzahl der Bits
  - B … ”Bits”
  - n … Länge des Blocks
  - x … Anzahl der verschiedenen Symbole
    - B … binär
    - T … ternär
    - Q … quarternär

Blockkodierung – 2

- 4B5B
  - 4 Bits werden zu 5 Bits umkodiert
  - 16 Bitkombinationen auf 32 Codewörter
  - Hälfte der Codewörter können zusätzlich verwendet werden
    - z.B. Fehlererkennung
  - Nie mehr als 3 Nullen aufeinanderfolgend
    - kann optimal mit NRZ-I kombiniert werden
  - nur 25% höhere Bandbreite
- 4B3T
  - 4 Bits auf 3 ternäre Signalparameter
  - 16 Bitkombinationen auf 27 (3^3) Codewörter
  - redundante Signalgruppen werden benutzt, um Gleichspannungsanteil auszugleichen
    - dazu bisherigen Gleichspannungsanteile summieren und entsprechend einen von zwei möglichen ternären Codes wählen