**Zusammenfassung INT**

René Bernhardsgrütter – 10.05.2013

**Wsk** = Wahrscheinlichkeit, **ZV** = Zufallsvariable,   
**IG** = Informationsgehalt, **WB** = Wörterbuch, **CW** = Codewort,

**Alg** = Algorithmus

# Wsk als Zahlenwert

X = **ZV** (Grossbuchstabe)

xm = mögliches Ergebnis (Kleinbuchstabe)

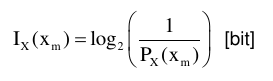
**Wsk** von zwei diskreten, unabhängigen **ZV** X und Y:

# Informationsgehalt I in bit

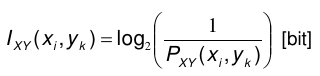
X,Y = **ZV**

xm, yk = mögliches Ergebnis

Für 1 **ZV** X:



Für 2 oder mehrere **ZV** X, Y:



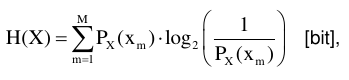
# Entropie H in bit

= mittlerer **IG** aller möglichen Ergebnisse xm einer **ZV** X.

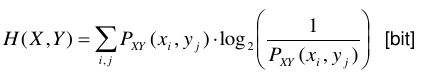
X,Y = **ZV**

xm, yk = mögliches Ergebnis

Für 1 **ZV** X:

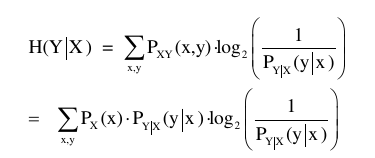


Für 2 oder mehrere **ZV** X, Y:



# Entropierate H in bit/Symbol

Entropierate H in bit/Symbol **einer Quelle mit Gedächtnis**:



Ergibt:



*Wenn es sich* ***um eine Quelle ohne Gedächtnis*** *handelt, die normale Entropie-Formel verwenden!*

# Redundanz R als Zahlenwert

R = Redundanz

X = **ZV**

M = Anzahl Werte der **ZV** (MMünze = 2)

Falls R = 0, dann ist nichts mehr (verlustlos) komprimierbar.

*Achtung: Redundanz R ≠ Rate R ≠ Kompressionsrate R!*

# Mittelwert einer diskreten ZV

X = Diskrete ZV (z. B. Würfel)

M = Anzahl verschiedene Werte von X

P(xm) = WSK, dass der Wert xm vorkommt

xm = Der effektive Zahlenwert!

Mittelwert der diskreten ZV X (also der Zahlen-Wert, der im Schnitt vorkommt:

# Quellen

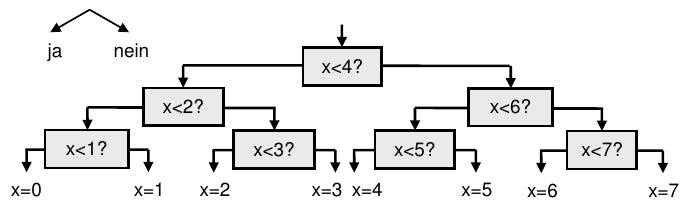
**DMS**: Discrete Memoryless Source, unabhängig und gleichwahrscheinlich.

**BMS**: Binary Memoryless Source, unabhängig und gleichwahrscheinlich und binär.

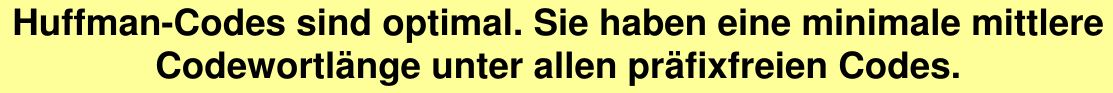
**BSS**: Binary Symetric Source, unabhängig und gleichwahrscheinlich und binär und pinJedemFall = 0.5.

# Binärer Fragebaum

Man fragt wie in einem Binärbaum. Man wiegt dabei immer die zwei Gruppen gegeneinander ab, also beim ersten Schritt Nummer 0..3 vs 4..7 als zwei Gruppen auffassen und diese ‚auf die Waage‘ legen:



# Huffman-Codierung



**Alg**:

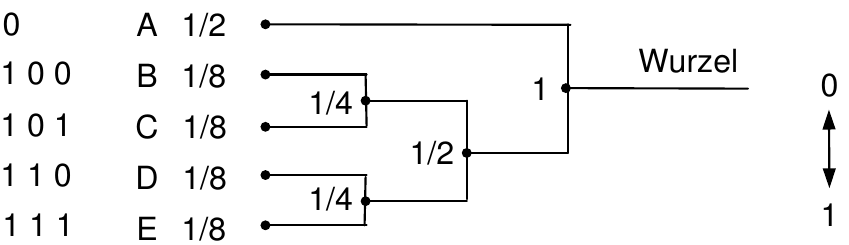
① Ordnen der Symbole nach **Wsk** (grösste zuerst) und so aufschreiben.

② Zwei Symbole mit der kleinsten **Wsk** zu einem neuen Symbol verbinden. Summierte **Wsk** hinschreiben.

③ ② wiederholen, bis nur noch genau 1 Ast übrig ist.

④ Richtung definieren: z. B. ↑ = 0, ↓ = 1 und das hinschreiben.

⑤ Von der Wurzel aus **CW** aufbauen.



**Blocklänge**: Anzahl der Eingangszeichen auf der linken Seite bzw. am Ende des Baums. Normalerweise 1 (A, B, C, …), kann aber auch mehr sein.

Nachteile:

* **CW** müssen synchronisiert werden.
* Präfixfreie **CW** nötig.
* Gute Quellenstatistik erforderlich.
* Komplex (z. B. beim Wechsel auf Doppelsymbole).

## Mittlere CWlänge E[L] resp. Rate R resp. Coderate R

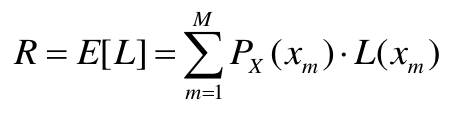
R = Rate in bit/Symbol

E[L] = Mittlere **CW**länge in bit/Symbol

X = Code (Quelle)

xm = Wert des aktuellen Symbols

L(xm) = **CW**länge von xm



Achtung: Unbedingt mit der Summenformel ausrechnen!

Achtung: Rate R ≠ Redundanz R ≠ Kompressionsrate R!

## Entropie H in bit

Normale Formel für eine **ZV** X verwenden.

Wenn **H(X) < E[L]** ist, gehen **Infos verloren**.

Wenn **H(X) = E[L]** ist, ist der Huffman-Code **optimal**.

Wenn **H(X) ≥ E[L]** ist, müssen keine Infos verloren gehen.

# LZ-Verfahren generell

*Ergänzung von Handnotizen.*

Strings variabler Länge werden in **CW** fixer länge codiert.

Bei sehr grossen Files ist Kompression mit **WB** optimal bzgl. Entropie.

Vorteile:

* **Universell anwendbar**. Quellenstatistikunabhängig.
* **Asymptotisch optimal**. Bei langen Bitfolgen geht die mittlere **CW**länge R = E[L] gegen die Entropie H(X).

Nachteile:

* Wörterbuchgrösse beschränkt.
* Bei kleinen Eingangsfolgen kaum Kompression möglich.

## Kompressionsrate R

**Merkhilfe**: „Kompressionsrate“ ≃ „Kompressionsfaktor“

**Je kleiner der Kompressionsfaktor, desto besser die Kompression.**

Achtung: Kompressionsrate R ≠ Rate R ≠ Redundanz R!

# LZ77

*Ergänzung von Handnotizen.*

* Decoder einfacher als Encoder, er hat alle Token und muss diese zusammensetzen. Encoder muss ständig Werte mit Sliding Window vergleichn.
* Nahe beieinanderliegende Muster komprimieren gut, weit entfernte schlecht.

## Tokenlänge in bit

S = Offset in bit = Anzahl möglicher Zeichen, die im Such-Buffer zurückgefahren werden darf

L = Vorschaubufferlänge in bit

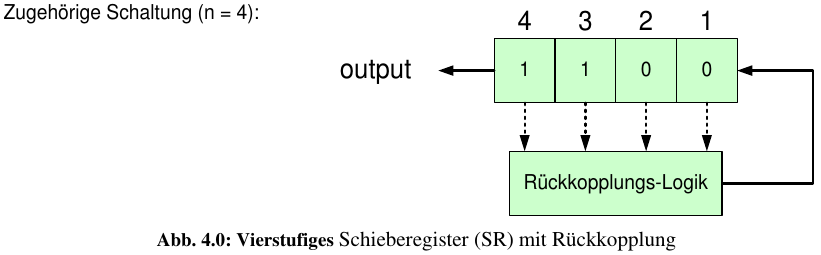
# PN-Sequenzen

Erzeugen statistisch ziemlich gute Zufälligkeit (wie Münzwurf, Wahrscheinlichkeit P beinahe 0.5).

## Linear Feedback Shift Register (LFSR)

Beispiel: Ein vierstufiges LFSR mit den Wertigkeiten

23, 22, 21 und 20 und den momentan gültigen binären Signalen   
1 – 1 – 0 – 0 weist den dezimalen Zählerstand 12 auf:



**n = Anzahl Gatter**, oben ist n = 4.

Im Allgemeinen haben n-stufige LFSR-Sequenzen die Periode

„-1“, weil 0 (binär und dezimal) nie dargestellt wird. Denn dies würde nie ‚verlassen‘ werden, sondern gäbe immer wieder 0.

# m-/Maximallängen-Sequenzen

= PN-Sequenzen, die die Maximallänge erreichen. Mit *primitiven Polynomen* kann eine solche (gewollte) m-Sequenz erzeugt werden.

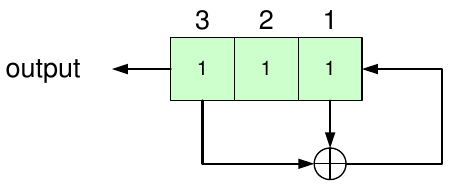
**Bsp:**

n = 3 => Periode P = 2³ - 1 = 7

Primitives Polynom = (3, 1, 0)

Feedbackpolynom = x3 + x1 + x0

Schaltung (hier mit Seed 1 1 1):



Das primitive Polynom sagt, bei welchen Gattern eine XOR-Verknüpfung gemacht wird. Oben: 3 + 1 + 0:

Also Gatter 3 ⊕ 1, dann das Ergebnis bei 0 wieder rein (= Anfang).

Das läuft nun sieben Schritte durch, dann ist man wieder bei 1 1 1, dem Seed (=> Periode).

Wenn man ein „schlechtes“ Polynom nimmt, ist es kein Maximallängenpolynom, hat also nicht 2n - 1 verschiedene Zustände!

## Zufallseigenschaften von m-Sequenzen

* m-Sequenzen sind fast ausgeglichen in Bezug auf die Anzahl Einsen und Nullen.
* Die m-Sequenz der Länge P und die zyklisch verschobene Kopie haben fast 50 % übereinstimmende Bits und 50 % verschiedene Bits.

Jede m-Sequenz ist eine PN-Sequenz, jedoch nicht immer umgekehrt!

## Anzahl 0en bzw. 1en berechnen

Wenn z. B. n = 4 (also 4 Gatter), dann ist bei optimalem primitivem Polynom 2n verschiedene Wörter möglich (wobei 0000 nie verwendet wird, das ist bei Anzahl 0en relevant).

## Wahrscheinlichkeiten berechnen

Klar: und

Bei m-Sequenzen ist die Anzahl 1en immer 1 grösser als die Anzahl 0en. In Abhängigkeit der Periode P:

## Relative Häufigkeit der Runs

n = Anzahl Gatter

P = Periode der LFSR-Sequenz (= 2n-1)

k = Länge der Runs, 1…n

p = Wahrscheinlichkeit bei der entsprechenden Länge

r = Anzahl Runs, muss gezählt werden (von den Outputbits, aneinandergereiht)!

Ein „**Run**“ ist das Aufeinanderfolgen mehrere Nullen oder Einsen. So weist zum Beispiel die Bitfolge ... 00110101000111001101... folgende Runs auf: Run 1 kommt 6x vor, Run 2 kommt 4x vor und Run 3 kommt 2x vor.

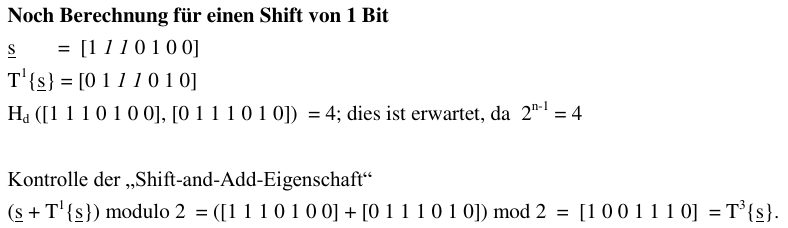
## Seed bei m-Sequenz verschieben

**Die m-Sequenz der Länge P und die zyklisch verschobene Kopie haben fast 50 % übereinstimmende Bits und 50 % verschiedene Bits.**

Shift = nach rechts schieben, links mit 0en auffüllen.

Es gelten die folgenden Eigenschaften, n = Anzahl Reigster:

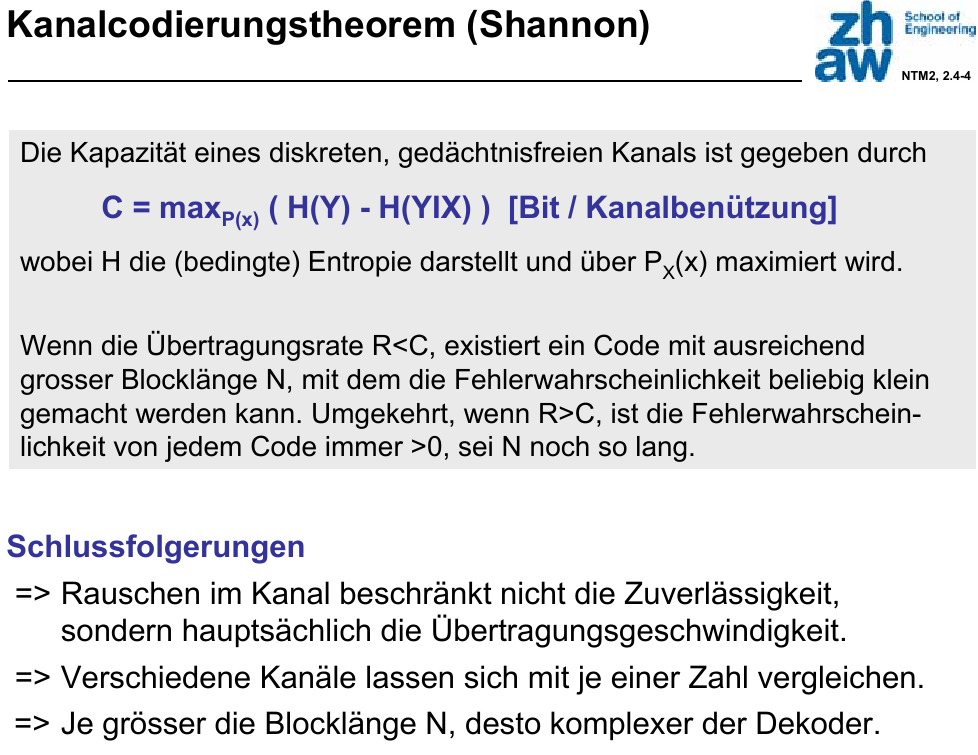
* Die Hammingdistanz zwischen der m-Sequenz s und der zyklisch um einen Rechts-Shift verschobenen Sequenz beträgt 2n-1.
* Es gilt eine „Shift-and-Add-Eigenschaft“ die folgendes besagt:   
  Die Modulo-2 Addition der Sequenz s mit der verschobenen Sequenz Tu {s} führt zu einer (ebenfalls verschobenen) Sequenz Tv{s}. Aber es ist wiederum die Sequenz s, eben nur verschoben.



# Primitive Polynome bis 50

|  |  |  |  |
| --- | --- | --- | --- |
| (1, 0) | (26, 6, 2, 1, 0) | (14, 5, 3, 1, 0) | (39, 4, 0) |
| (2, 1, 0) | (27, 5, 2, 1, 0) | (15, 1, 0) | (40, 5, 4, 3, 0) |
| (3, 1, 0) | (28, 3, 0) | (16, 5, 3, 2, 0) | (41, 3, 0) |
| (4, 1, 0) | (29, 2, 0) | (17, 3, 0) | (42, 5, 4, 3, 2, 1, 0) |
| (5, 2, 0) | (30, 6, 4, 1, 0) | (18, 5, 2, 1, 0) | (43, 6, 4, 3, 0) |
| (6, 1, 0) | (31, 3, 0) | (19, 5, 2, 1, 0) | (44, 6, 5, 2, 0) |
| (7, 1, 0) | (32, 7, 5, 3, 2, 1, 0) | (20, 3, 0) | (45, 4, 3, 1, 0) |
| (8, 4, 3, 2, 0) | (33, 6, 4, 1, 0) | (21, 2, 0) | (46, 8, 5, 3, 2, 1, 0) |
| (9, 4, 0) | (34, 7, 6, 5, 2, 1, 0) | (22, 1, 0) | (47, 5, 0) |
| (10, 3, 0) | (35, 2, 0) | (23, 5, 0) | (48, 7, 5, 4, 2, 1, 0) |
| (11, 2, 0) | (36, 6, 5, 4, 2, 1, 0) | (24, 4, 3, 1, 0) | (49, 6, 5, 4, 0) |
| (12, 6, 4, 1, 0) | (37, 5, 4, 3, 2, 1, 0) | (25, 3, 0) | (50, 4, 3, 2, 0) |
| (13, 4, 3, 1, 0) | (38, 6, 5, 1, 0) |  |  |

# Kanalcodierung



|  |
| --- |
| Wenn in einem zeitdiskreten Kanal die Werte, welche Eingangs- und Ausgangsvariablen einnehmen können, endlich oder zählbar unendlich sind, bezeichnet man den Kanal als diskreten Kanal. |

C = Kanalkapazität in Bit/Kanalbenutzung (hier nicht Bit/s, weil hier der Kanal wichtig ist)

R = Übertragungsrate /= Coderate bei AWGN-Kanal!

N = Blocklänge



**Es gilt**: Falls die Übertragungsrate beträgt, existiert ein Code mit ausreichend grosser Blocklänge , mit dem die Fehlerwahrscheinlichkeit beliebig klein gemacht werden kann.

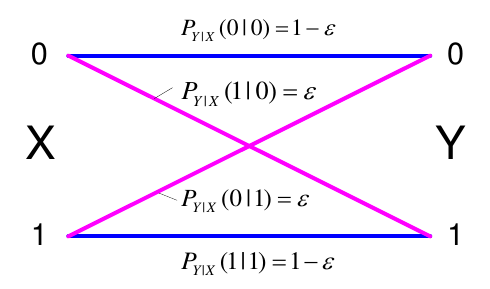
Umgekehrt gilt: Falls beträgt, so ist die Fehlerwahrscheinlichkeit von jedem Code immer , sei noch so lang. Man hat dann also keine Informationen ungestört übertragen! Alles wird verfälscht! Keine Kommunikation möglich!

## Binary Symmetric Channel (BSC)

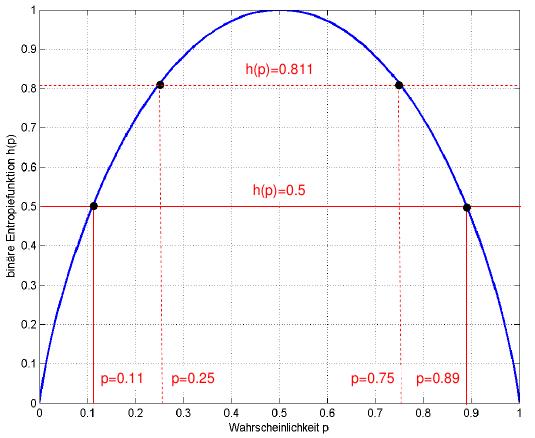
BSC ist gedächtnisfrei.

ε = Übertragungswahrscheinlichkeit, dass am Ausgang eine 1 erscheint, wenn am Eingang eine 0 gesendet wurde   
(vice versa)

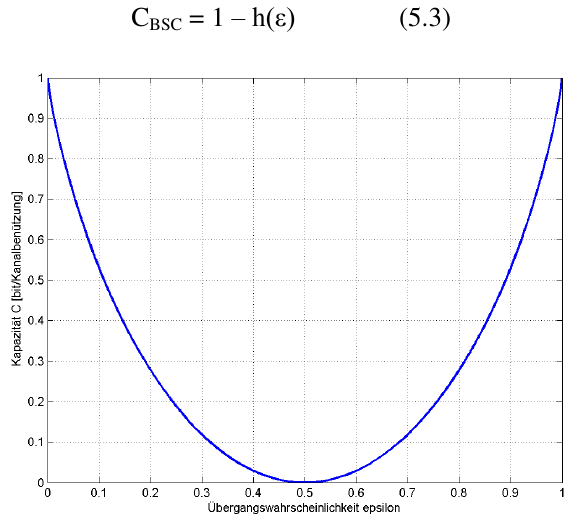
BER = Bitfehlerrate (wenn kein Forward Error Correction angewendet wurde, ist BER = ε)

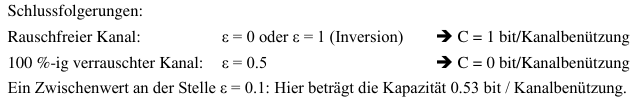


### Entropie H



### Kanalkapazität





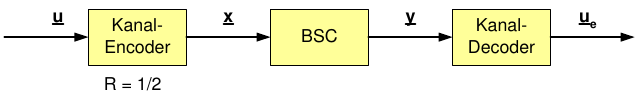
## AWGN-Kanal (Additive White Gaussian Noise)

= ein Kanal, der sehr selten sehr kleine und sehr grosse Rauschaplituden hat, dafür oft mittlere. (gem. Gauss’scher Normalverteilung der Rauschleistung).

Coderate R ≠ Übertragungsrate RBSC!

Bsp: für Coderate R < 0.53 sind mehr als 100 Codebits nötig, um 53 Infobits zu übertragen.

Bsp: R = 0.5



|  |  |  |
| --- | --- | --- |
| u = Infowort | x = **CW** | N = **CW** Blocklänge |

Jedes |x| = N/2, wenn |u| = N. Die Redundanz ermöglicht eine zuverlässige Datenübertragung.



N0 = Rauschleistungsdichte in

B = Bandbreite in

S = Signalleistung in

## Binäre (N, K)Block-Codes

K = Anzahl Infobits bei einem Code-Block (es gibt maximal 2K verschiedene Infoworte)

P = Anzahl Prüfbits

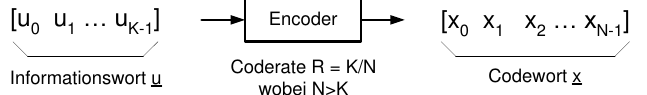
N = Anzahl Bits eines encodeten Blockes

R = Coderate

u = Infowort (ein Block)

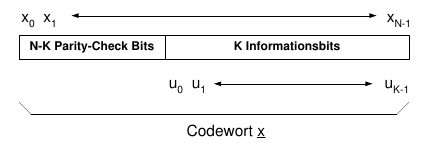
x = **CW**

N > K. Daher werden Block-Codes als **(N, K) Block-Codes** bezeichnet.



### „Systematische (N, K) Block-Code C“

Das Infowort u wird 1:1 in dem **CW** x mitübertragen. Die restlichen Bits sind Paritätsbits (z. B. vorne oder hinten…): Dann ist es ein systematischer (N, K) Block-Code.



### „Linearer (N, K) Block-Code C“

Falls die modulo-2 Summe zweier Codewörter wieder ein **CW** ergibt, dann ist der Block Code linear:

**CW**1 = 000, **CW**2=110 => **CW**1 + **CW**2 (mod) = 110 => wieder ein **CW**.

Trifft auf alle Kombinationen der **CW** zu => linearer Code

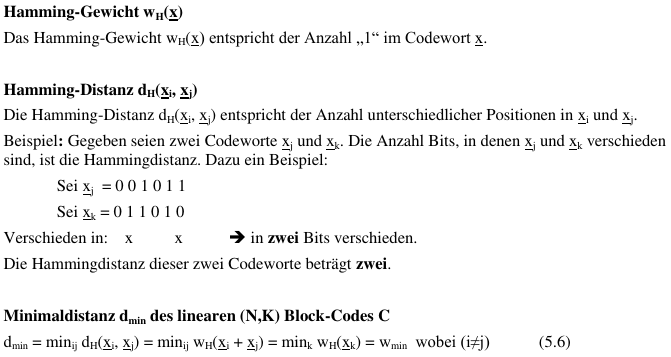
### „Linearer, zyklischer (N, K) Block-Code C“

Falls die zyklische Verschiebung eines **CW** wieder ein **CW** ergibt, ist der Code ausserdem zyklisch.

### Fehlerkorrekturbegriffe

BER = Bit Error Rate = Bitfehlerrate

FEC = Forward Error Rate = Vorwärtskorrektur



### Fehlerdetektion/-erkennung

e = Fehlervektor (also ein fehlerhaft übertragenes Wort)

Bsp: Wenn 000 gesendet und 011 empfangen wurde, ist 001 der Fehlervektor e.

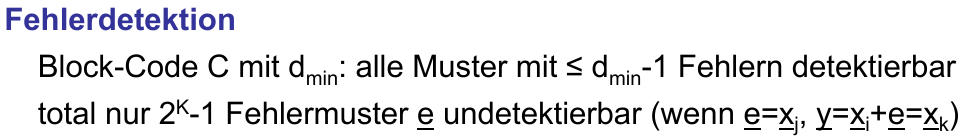
#### Welche Fehler-Muster bei einem Code erkennbar sind

F = Anzahl Bit, die in dem empfangenen **CW** falsch sein können wobei das **CW** dennoch erkennbar bleibt

C = Verwendeter Block-Code

Man versteht dies im Sinne von „Alle **CW** mit F bit falsch können erkannt werden“.

Siehe auch:



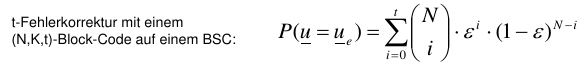
### Fehlerkorrektur



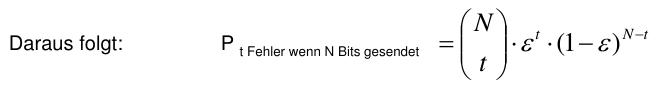
**t-Fehlerkorrektur** = Anzahl t oder weniger Fehler korrigierbar.

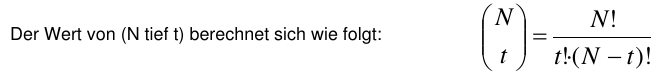
Die **Wahrscheinlichkeit**, dass auf einem BSC mit einem

(N,K,t) Block-Code ein **CW** **korrekt** übertragen wird, ist:

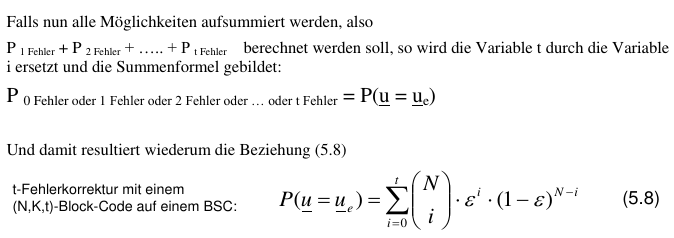




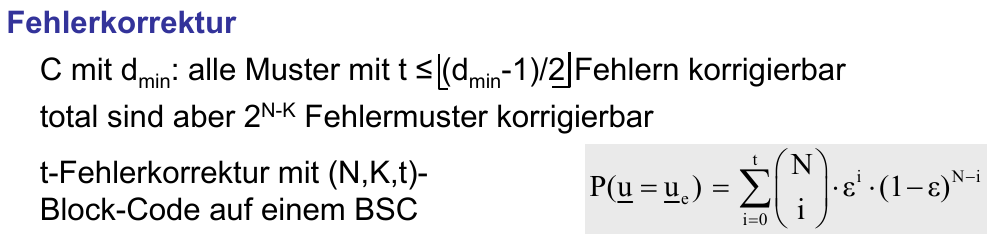




Mit , N = **CW**länge



Siehe auch:



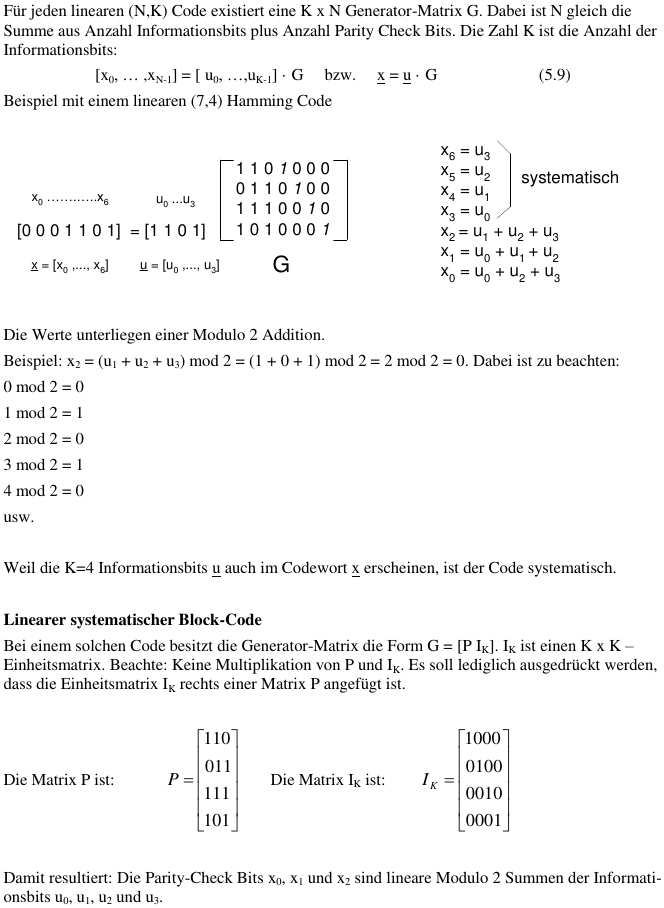
#### Wahrscheinlichkeiten-Überblick

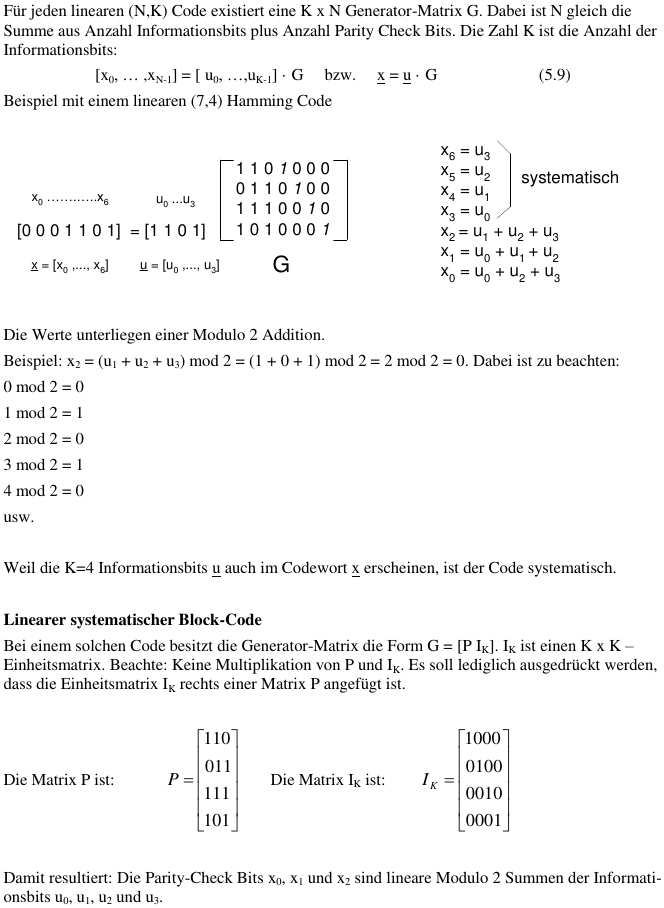
### Zuverlässigkeit eines Block-Codes

Wenn die Coderate R < 0.53 ist (bei einem Epsilon von 0.1), so ist der Code zuverlässig.

## Generator-Matrix

|  |  |
| --- | --- |
| x = **CW** | u = Infowort |
| G = K \* N-Matrix (Zeilen \* Spalten) | |





Wenn es heisst: „Betrachten Sie den **linearen (6,3) Block-Code**

C = { [000000],

[**011*100***],

[**101*010***],

[**110*001***],

[110110],

[101101],

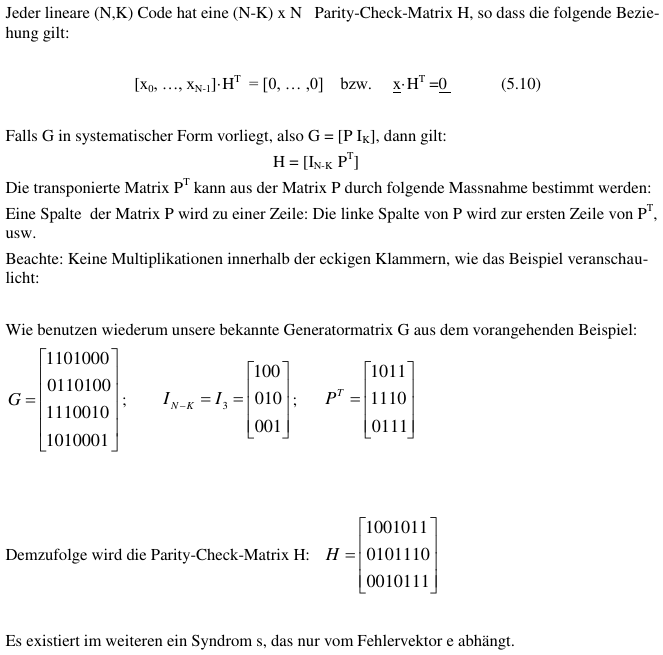
[011011],

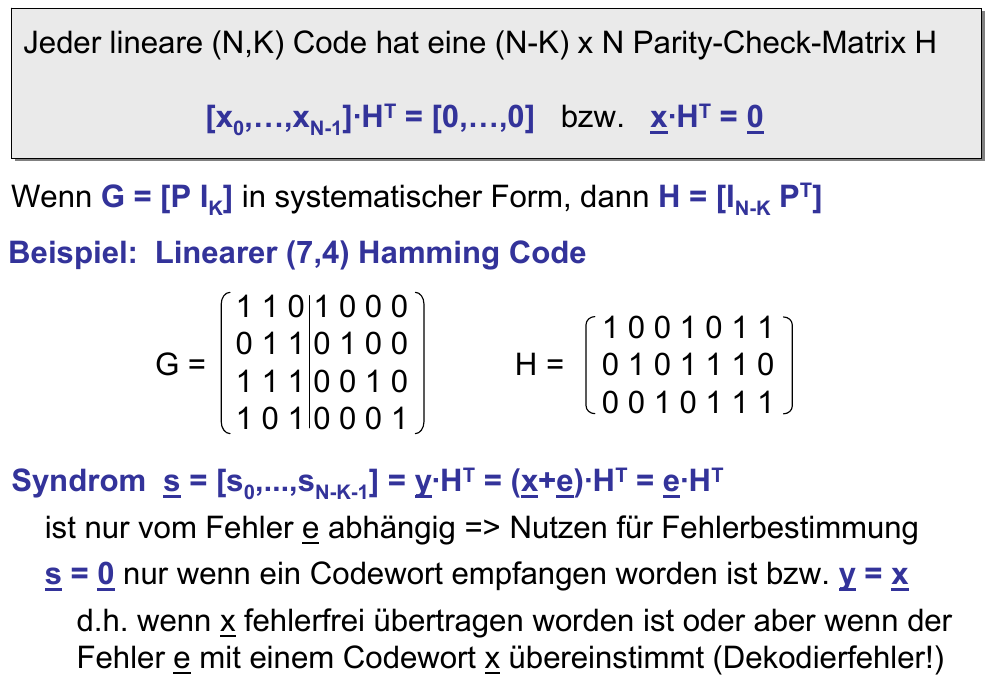
[000111]}, Bestimmen Sie die Generator-Matrix G in systematischer Form.“

Es ergibt sich G:

indem man diejenigen Codewörter mit der Einheitsmatrix am Schluss als G verwendet.

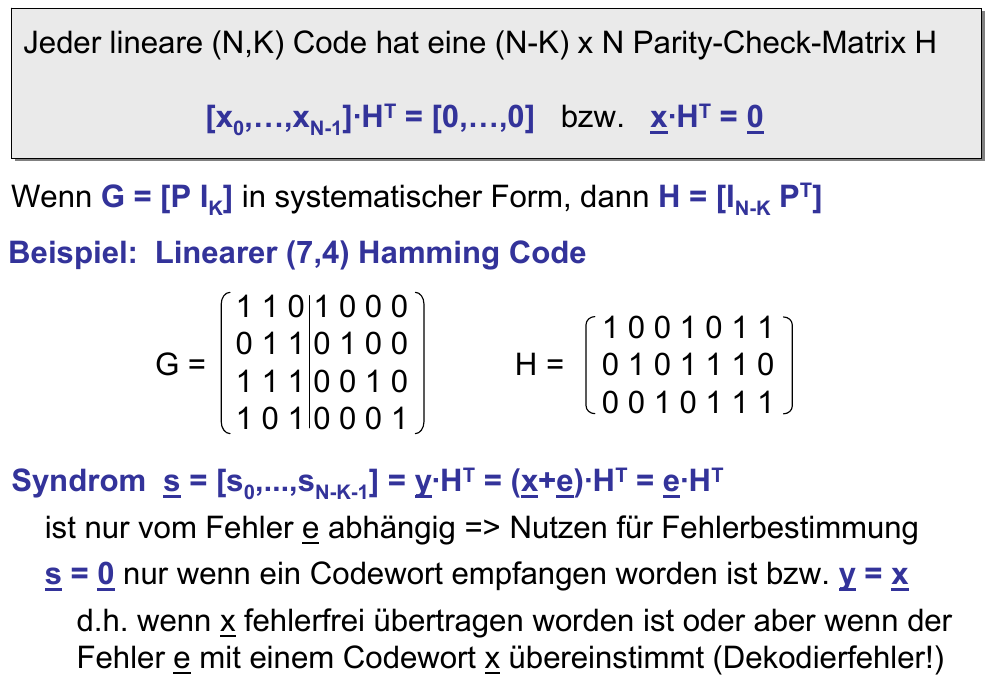
## Parity-Check-Matrix



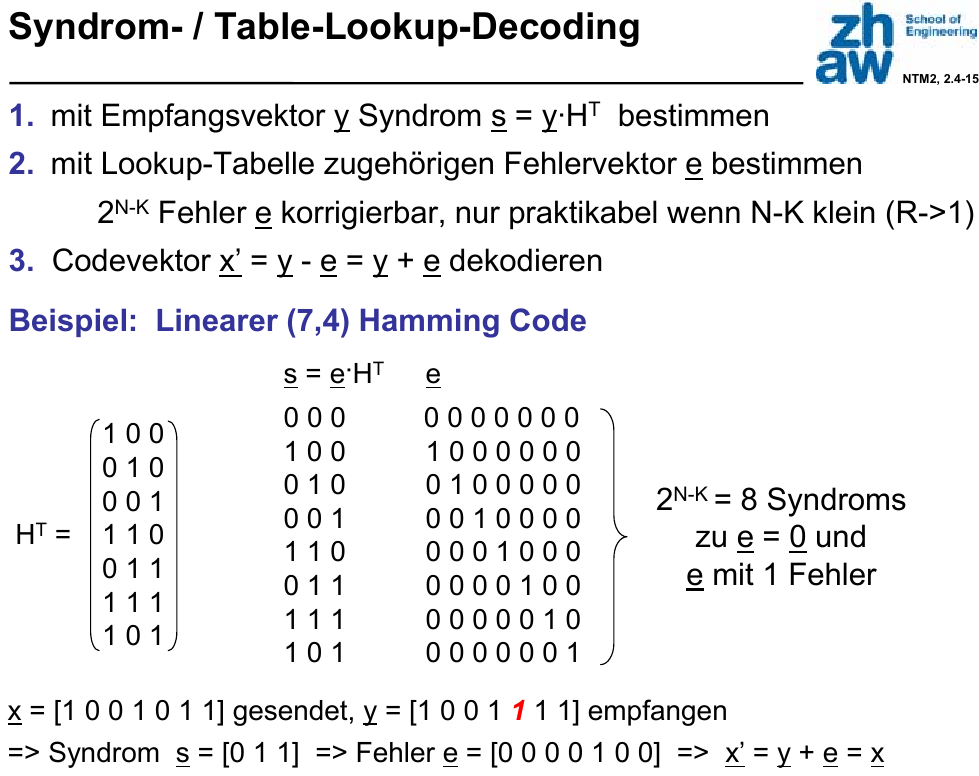


P Nutzbits I PT

I = Einheitsmatrix



s = y \* HT, y = Empfangsvektor, also die Codewörter



# CRC

RC = Codierungsrate

K = Anzahl Nutzbits

PAnzahl = Anzahl Prüfbits

N = Länge des übertragenen Datenworts = K + P Bits

c = CW

u = Nutzdatenwort (bsp: =

U = Nutzdatenplynom von u (bsp: )

s = Syndromvektor

PBits = Prüfbits

[G] = Generatormatrix

[H] = Kontrollmatrix

p = Grad des Generators (bis z³ => p = 3)

E(z) = Rest der Polynomdivision

**Code mit dmin kann ekor Fehler korrigieren:**



**Code mit dmin kann eerk Fehler erkennen:**



**Burstfehler**: Nebeneinanderliegende Bits, die falsch übertragen wurden. Bei P Prüfbits können P nebeneinanderliegende Bitfehler entdeckt werden.

**Codieren gem. Script:**

Wobei zp das jeweilige Polynom von U ist.

**Decodieren gem. Script:**

Wenn das ohne Rest aufgeht, also Result = 0, dann wurde das c fehlerfrei übertragen.

**Nullproblem**: Produziert ein Datenstrom zufällig einen CRC gleich null, so ist der CRC auch dann null, wenn dem Datenstrom zusätzliche Nullen angehängt werden, oder – falls der Datenstrom mit einer oder mehreren Nullen endet – einige dieser letzten Nullen entfernt werden.

Ist dem Ende des Datenstroms der CRC angehängt (so wie es ein Sender eben verschickt), und bei der Übertragung werden (nach dem gesendeten CRC) noch zusätzliche Nullen angefügt, so können diese zusätzlichen Nullen am Ende nicht erkannt werden.

**AntiProblem**: Man invertiert die CRC-Ergebnisse vor schicken und nach empfangen.

## Wiki-Bsp

Generatorpolynom: 110101 (zuvor festgelegt)

Rahmen: 11011 (Nutzdaten)

Rahmen p-1 Nullen: 1101100000 (hier ist p = 5)

### Encodierung

Der Rahmen + Anhang wird von links her durch das Generatorpolynom dividiert. Es wird immer XOR verwendet:

1101100000

110101

------

0000110000

110101 // Solange rechts, bis in oberer Zeile eine 1

------

101 (Rest)

Der Rest wird nun an den Rahmen ohne Anhang angehängt:

Übertragener Rahmen: 1101100101

### Decodierung

Mittels **Division durch das Generatorpolynom** kann jetzt die **fehlerfreie** Übertragung erkannt werden:

1101100101

110101

------

110101

110101

------

000000 // Wenn == 0, dann fehlerfrei!

Oder wenn fehlerhaft übertragen (1**0**01100101=:

1001100101

110101

------

100110

110101

------

100111

110101

------

100100

110101

------

100011

110101

------

10110 // Rest != 0, also wahrsch. fehlerhaft!

#### Mögliche Resultate

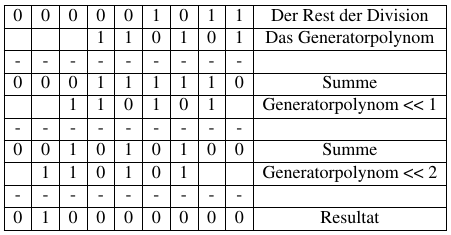
1. Der Rest der Division ist **null** und die Nachricht ist **richtig**
2. Der Rest der Division ist **null** und die Nachricht ist **fehlerhaft** (dieser Fall ist **unwahrscheinlich**, kann aber vorkommen, wenn das Fehlerpolynom ein Vielfaches des Generatorpolynoms ist oder wenn der Fehler im Datenteil und im CRC-Wert ist)
3. Der Rest der Division ist **ungleich null** und die Nachricht ist **fehlerhaft**
4. Der Rest der Division ist **ungleich null** und die Nachricht ist **richtig** (dieser Fall tritt ein, wenn lediglich der angehängte Rest fehlerhaft übertragen wird; dies ist jedoch ebenfalls unwahrscheinlich, da der übertragene Rest im Vergleich zur Gesamtlänge des Pakets kurz ist)

### Fehlererkennung

Es werden alle Burstfehler ≤ Länge vom Generatorpolynom erkannt, ausser es hat das gleiche Bitmuster wie das Generatorpolynom. Ein Generatorpolynom mit gerader Länge erkennt jede ungerade Anzahl Bitfehler.

### Fehlerkorrektur

Wir korrigieren höchstens 1bit-Fehler. Man kann die Stelle jener wie folgt finden:



# Faltungscodes

Bieten Vorwärtskorrektur (=Redundanz, um Fehler eher korrigieren zu können) bei Kanaltechnik. Verwendet Gedächtnis. Können nicht systematisch konstruiert werden, sondern werden ausprobiert.

Sind linear!

K = Anzahl Infobits (<n)

N = **CW**länge

LC = Gedächtnislängenordnung (Anz. Takte, die ein Bit braucht, bis es ganze Schaltung durchlaufen hat)

R = Coderate

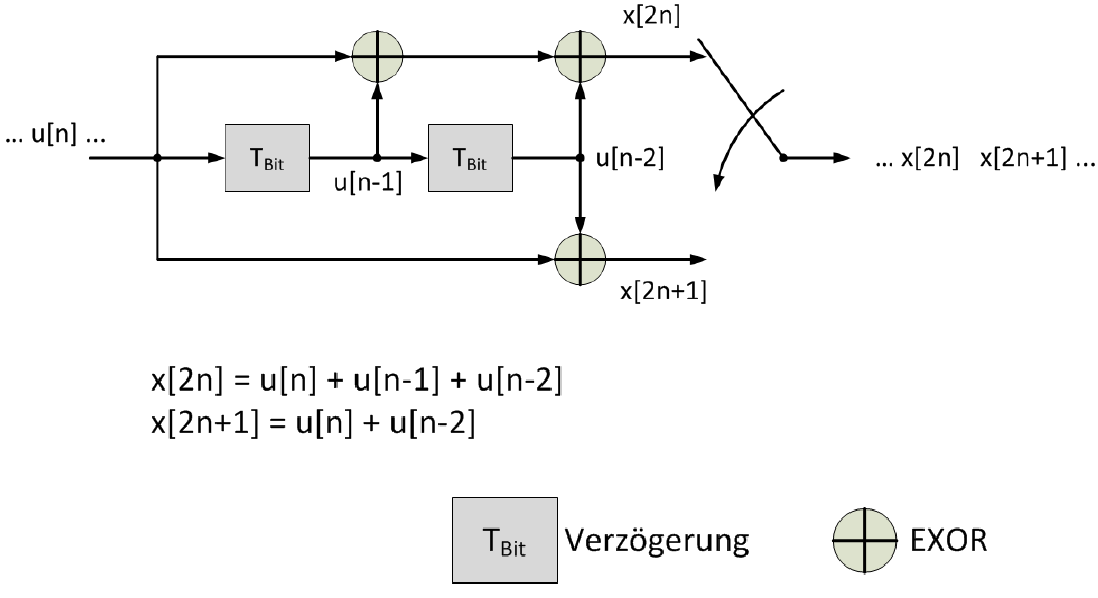
**Wenn R klein**: Relativ viele Parity-Check-Bits.

**Wenn R gross**: Relativ wenige Parity-Check-Bits.

## Encoder

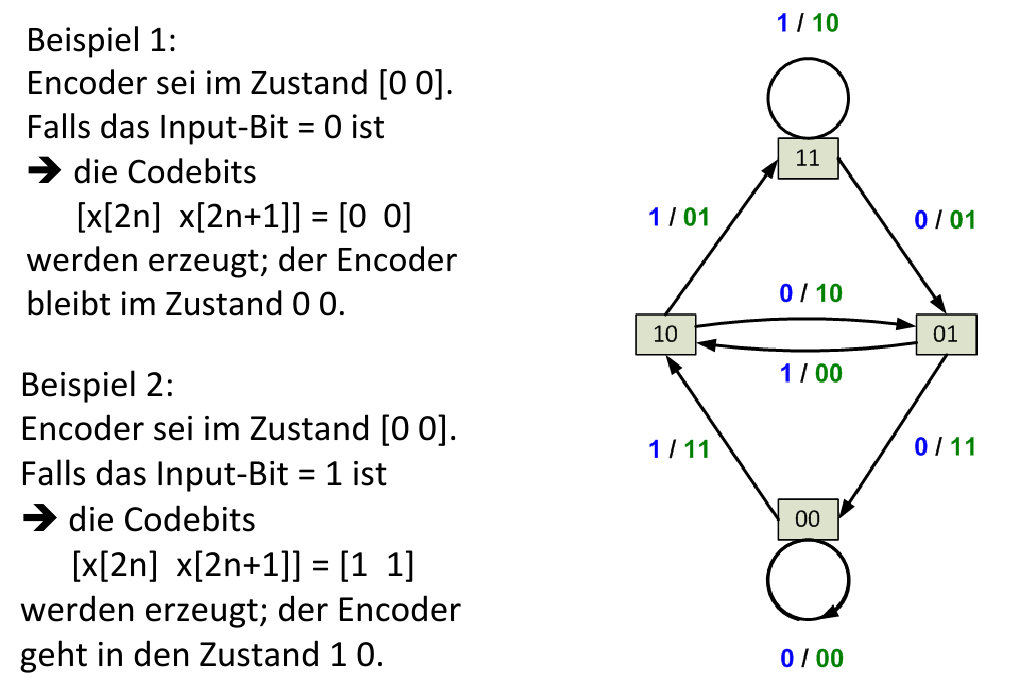
M = Anzahl Gatter/Gedächtnisbreite

R = Rate



Dieser Encoder hat: und das Gedächtnis .

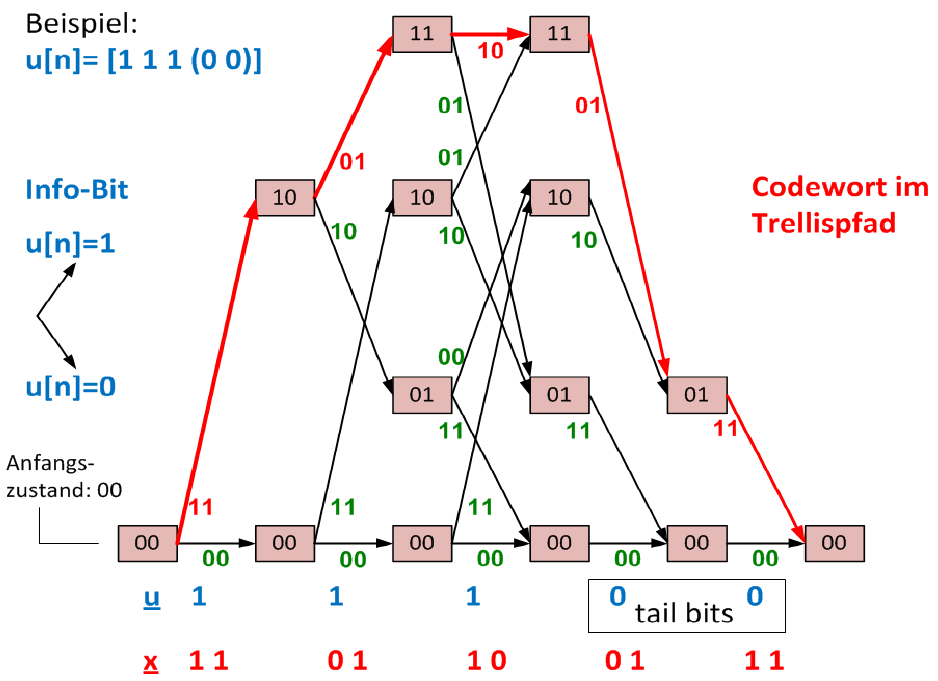
**Zustandsdiagramm des Encoders:**



XX : Aktueller Zustand. **X** / **YY**: **X** = Input-Bit, **YY** = Erzeugte Codebits.

**Trellisdiagramm**: Stellt die Zustandsübergänge in Abhängigkeit der Zeit dar. In der Vertikalen sind die Zustände aufgetragen. In der Horizontalen ist der zeitliche Verlauf aufgetragen.

Die erlaubten Zustandsübergänge sind mit Pfeilen gezeichnet. Zu diesen Pfeilen sind die K Info-Bits, die die Übergänge erzeugen, notiert.

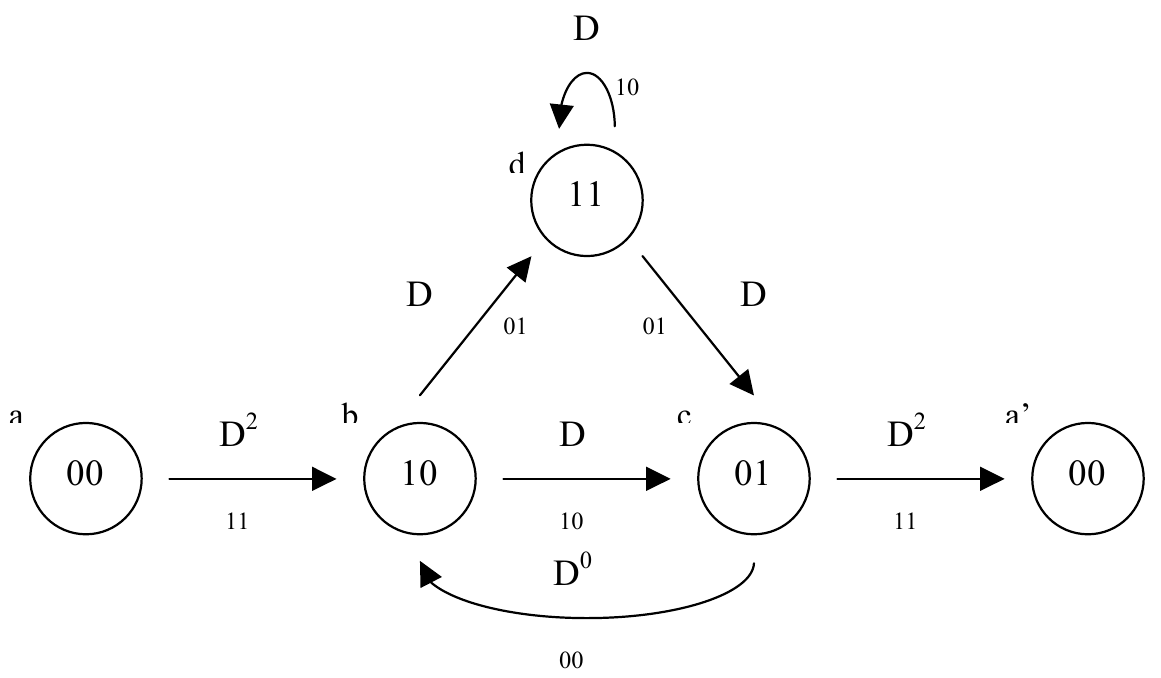


u1 u2 u3 trail1 trail2

## Distanz von Codefolgen

Lineare Codes => Alle Nicht-Null-**CW** mit dem Null-**CW** x[n]=[000..00] vergleichen. Gesucht sind also: Umwege (im Zustandsdiagramm) zum ‘Nullpfad’, der mit der kleinsten Anzahl Einsen.

Das Zustandsdiagramm wird im Zustand 00 aufgetrennt und an den Pfaden die Distanz zur Nullfolge (Di) notiert. **Also wenn beim Encoder beim aktuellen Schritt ‚10‘ raus kommt, ist das Gewicht dort 1 (dmin(10, 00) ) = 1).** Für einen Beispiel-Encoder sieht das wie folgt aus:



Die Distanz der gesamten Codefolge zur Nullfolge ergibt sich aus der Multiplikation aller Einzeldistanzen.

In unserem Fall ist der kürzeste Weg: a → b → c → a’.

Die Distanz zur Nullfolge ist: D2 \* D \* D2 = D5 => **dmin = dfree = 5**