Fehlerkorrektur

Digitale AV Technik, MIB 5

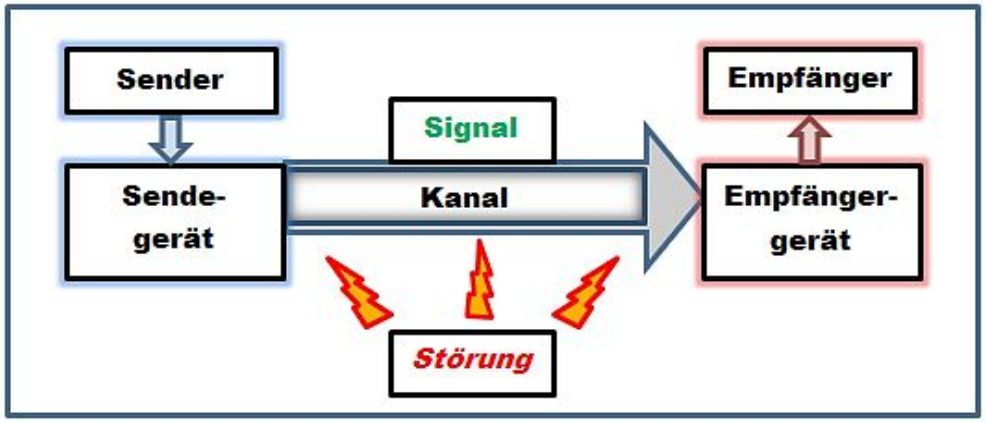
Aus Sicht der Informationstheorie

Problem	Kompression	**Fehlerkorrektur**
Ziel	Effizienz	Verlässlichkeit
Anwendung	Quellencodierung	**Kanalcodierung**

Algorithmische Perspektive

Problem	**Fehlerkorrektur**
Algorithmen	**Hamming code**
	Reed-Solomon Code
	Turbo-Code

Fehlererkennung



Störungen sind unvermeidlich. Zunächst ist es schon mal hilfreich zu erkennen, ob es einen Fehler bei der Datenübertragung gab oder nicht.

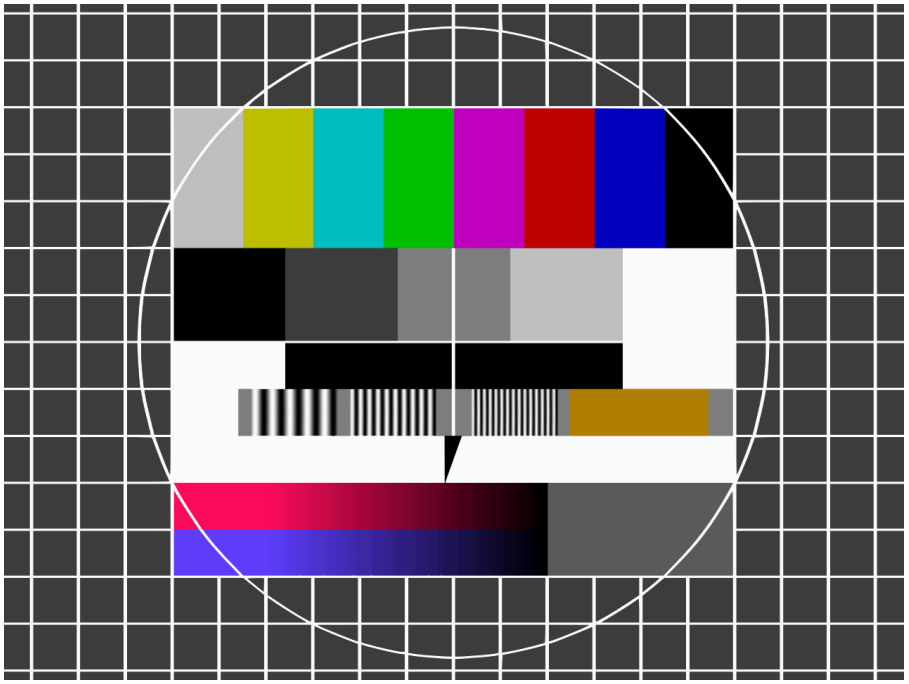
😉 Der erste Bitfehler der Geschichte

Aigeus, König von Athen, wartete besorgt auf die Rückkehr seines Sohnes Theseus, der nach Kreta gereist war, um gegen den Minotaurus zu kämpfen. Vor der Abreise hatten sie vereinbart, dass Theseus bei einer erfolgreichen Rückkehr weiße Segel setzen würde. Doch auf der Heimfahrt vergaß die Besatzung vor lauter Freude, die schwarzen Segel gegen weiße auszutauschen. Als Aigeus die schwarzen Segel sah, nahm er an, sein Sohn sei tot, und stürzte sich aus Verzweiflung ins Meer.

Fehler erkennen

- Wann sind Fehler überhaupt kritisch?
- Wie kann überprüft werden, ob eine Nachricht fehlerfrei übertragen wurde?

Testsignale



Bildquelle: Rotkaeppchen68 (in de.wikipedia), CC BY-SA 3.0, via Wikimedia Commons

Man sendet ein vereinbartes Signal, dass der Empfänger bereits kennt. Dadurch lassen sich systematische (also immer wieder gleich auftretende) Fehler erkennen.

Blockcodes

Ein Code, also eine Bitkette wird um eine Anzahl Bits erweitert, die keinen zusätzlichen Inhalt beitragen, sondern zum Schutz der Daten hinzugefügt werden.

Paritätsbits

Man prüft die Parität der Daten eines Blocks, also ob sie eine gerade (oder ungerade) Anzahl an **1en** enthalten.
Man verwendet im einfachen Fall nur ein Schutzbit und setzt dieses so, dass immer eine gerade Anzahl an **1en** entsteht.
Der Empfänger prüft einfach die Parität und weiß, dass ein Fehler dabei war, wenn keine gerade Anzahl an **1en** vorliegt.

Beispiel Parität

Beispiele:

100111 -> Parität 0

1101 -> Parität 1

10101 -> Parität 1

Berechnung im Rechner mit **XOR**:

```
def compute_parity(binary_string):
    parity = 0
    for bit in binary_string:
        parity ^= int(bit)
    return parity
```

Prüfsummen (checksums)

Bei vielen Anwendungen wird eine Prüfsumme berechnet und mit übertragen. Der Empfänger berechnet die selbe Summe und vergleicht das Ergebnis.

Beispiel: EAN

Zyklische Redundanzüberprüfung (CRC)

CRC sind weit verbreitet bei der Datenübertragung auch innerhalb einer Festplatte.

Die Idee ist es den Binärcode als Polynom zu interpretieren. Dieses Polynom wird dann zyklisch durch ein allen bekanntes Prüfpolynom dividiert und der Rest wird als Schutzbits angehängt.

(die folgenden Folien sind von Ulrich Berger und Oualid Benabdallah hier geklaut)

Erinnerung: Rechnen mit Modulo 2

- Addition: „normal Addieren dann mod. 2 rechnen“

  $0 + 0 = 0$

  $0 + 1 = 1$

  $1 + 0 = 1$

  $1 + 1 = 0$

  entspricht **XOR**

- Multiplikation: „normal Multiplizieren, dann mod. 2 rechnen“

  $0 * 0 = 0$

  $0 * 1 = 0$

  $1 * 0 = 0$

  $1 * 1 = 1$

  entspricht **AND**

Polynomdivision

Polynomdivision mit Binärzahlen kann effizient mit XOR umgesetzt werden:

- Schritt 1: Stelle Polynome als Binärzahlen dar
- Schritt 2: Verschiebe den Divisor soweit nach links bis die führenden Stellen übereinstimmen
- Schritt 3: Berechne ein XOR zwischen Dividend und dem verschobenen Divisor
- Schritt 4: Falls das Ergebnis des XOR einen geringeren Grad als der Divisor hat ist dies der Rest der Division, andernfalls weiter mit Schritt 2 wobei das Ergebnis des XOR den neuen Dividend bildet.

Beispielrechnung (1)

Schritt 1:

$$f : x^7 + x^3 + x + 1 =$$

$$1 * x^7 + 0 * x^6 + 0 * x^5 + 0 * x^4 + 1 * x^3 + 0 * x^2 + 1 * x^1 + 1 * x^0$$

$$\Longrightarrow \ 10001011$$

Analog: $g : x^5 + x^3 + x^1 + 1 \ \Longrightarrow \ 101011$

Beispielrechnung (2)

Schritt 2-4: Teile f durch g schriftlich (an der Tafel)

$$f : g \rightarrow 10001011 : 101011 = 101$$
$$\Longrightarrow \text{ Faktor: } x^2 + 1$$
$$\Longrightarrow \text{ Rest: } x^3 + x^2$$
$$f = g * Faktor + Rest$$
$$x^7 + x^3 + x + 1 = (x^5 + x^3 + x^1 + 1) * (x^2 + 1) + x^3 + x^2$$

CRC Algorithmus

1. Datenblock erweitern: An die Nachricht werden N Nullen angehängt, wobei N die Länge des Prüfpolynoms minus 1 ist.
2. Division: Der erweiterte Datenblock wird durch das Prüfpolynom modulo 2 dividiert.
3. Restwert anhängen: Der Rest der Division wird als Prüfsumme an die ursprüngliche Nachricht angehängt.
4. Empfang: Der Empfänger führt dieselbe Berechnung durch. Ein Rest von 0 bedeutet fehlerfreie Übertragung.

Fazit Fehlererkennung

Die genannten Methoden erkennen, ob es einen Fehler bei der Übertragung gab, aber nicht wo. Daher lässt sich der Fehler auch nicht korrigieren.

Fehlerkorrektur

Wie könnte man mit Hilfe der Prüfzeichen oder Prüfbits erkennen, wo genau der Fehler liegt?

Mehrdimensionale Paritätsbits

Grundidee: Verwende ein Paritätsbit für Zeilen (also für die einzelnen Codeworte) und eins für die Spalten (also für die einzelnen Bits in den Codeworten).

Beispiel:

```
1 0 0 1 0 1 1
0 1 1 0 1 1 0
1 0 1 0 1 1 0
1 0 0 0 0 0 1
1 0 1 1 1 1 1
1 1 1 0 1 0 0
1 0 0 0 0 0 1
```

- **Overhead:** 2n + 1 zusätzliche Bits für n^2 Datenbits
- 3-fehlererkennend
- 1-fehlerkorrigierend

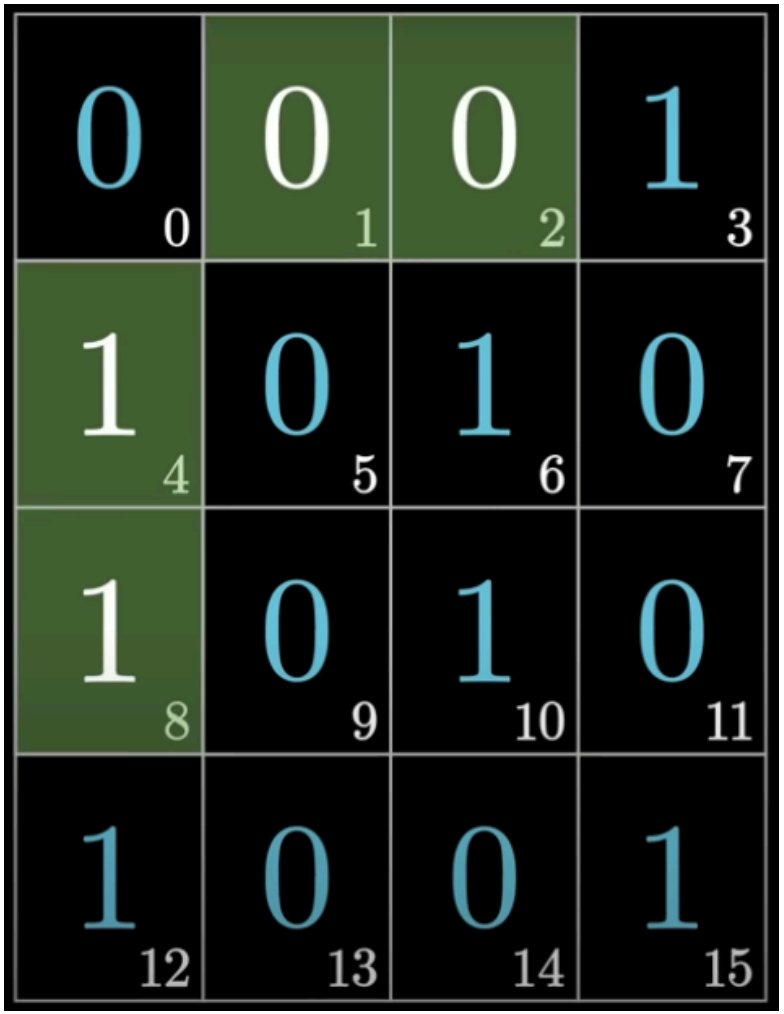
Was sind Hamming-Codes?

- Hamming-Codes sind eine Form von **fehlerkorrigierenden Codes**, die in digitalen Kommunikationssystemen verwendet werden.
- Sie können **Einzelfehler korrigieren** und **mehrere Fehler erkennen**.

Im folgenden sind einige Screenshots aus dem sehr zu empfehlenden Video von Grant Sanderson: 3Blue1Brown - Hamming Codes

Wie funktionieren Hamming-Codes? (1)

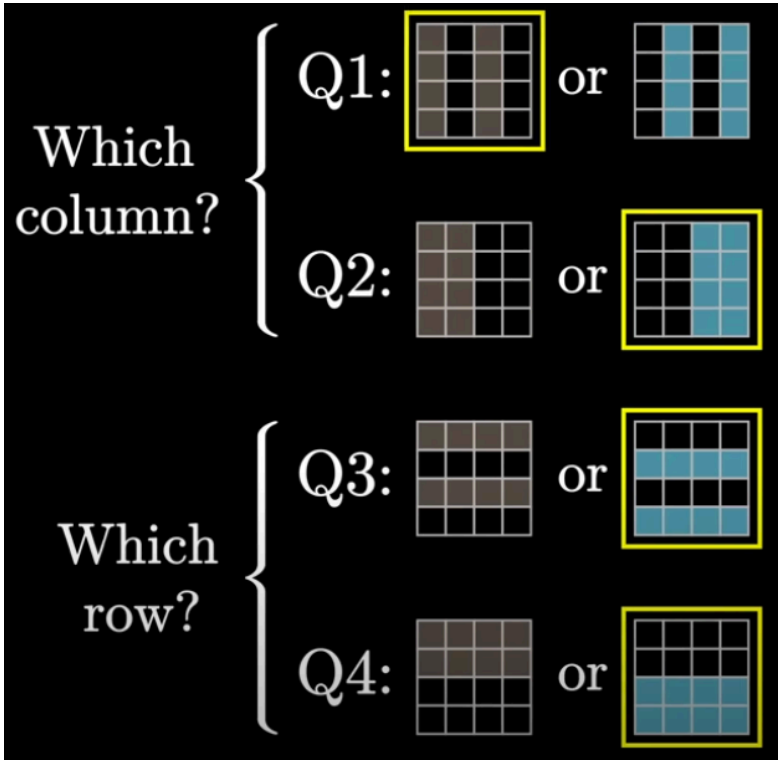
1. **Datenbits und Paritätsbits:**
- Datenbits werden durch **zusätzliche Paritätsbits** ergänzt (grün hinterlegt im Bild rechts).
- Die Positionen der Paritätsbits folgen einer **2er-Potenz-Reihe**: 1, 2, 4, 8, usw.



Wie funktionieren Hamming-Codes? (2)

2. **Berechnung der Paritätsbits:**

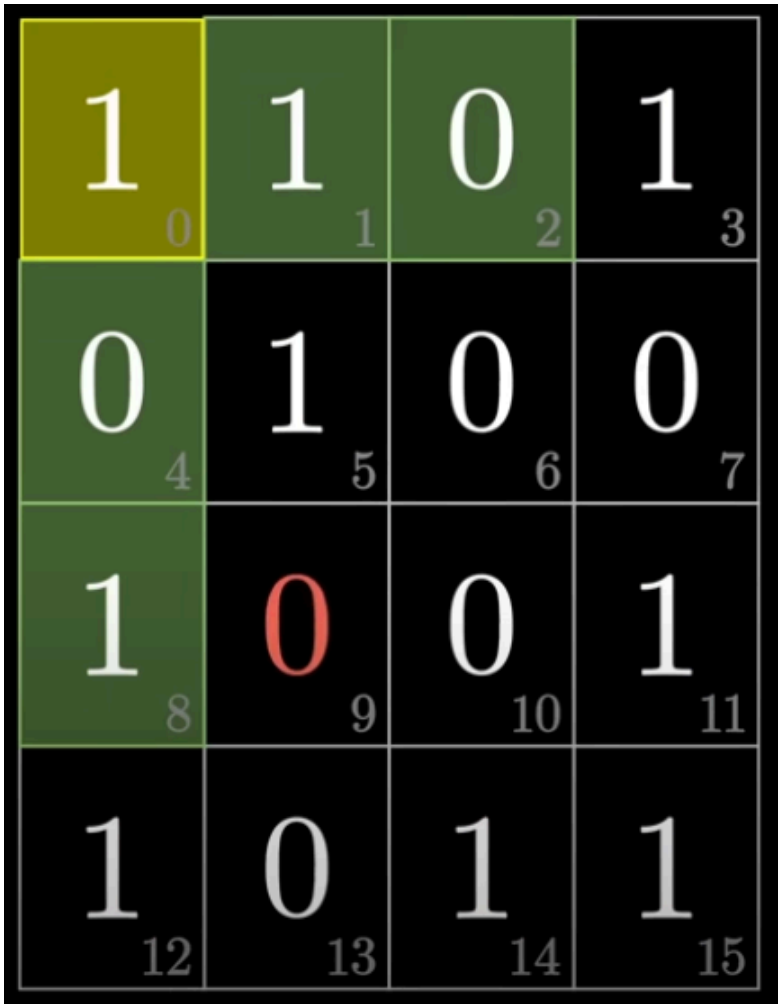
- Jedes Paritätsbit prüft eine bestimmte Menge von Datenbits.
- Das Ziel: Sicherstellen, dass die Anzahl der **1en** in einer bestimmten Gruppe von Bits **gerade** (oder ungerade) ist.



Wie funktionieren Hamming-Codes? (3)

3. **Fehlerkorrektur**

- Wenn ein Fehler auftritt, kann der **Ort des Fehlers** durch die **kombinierte Ausgabe der Paritätsbits** genau bestimmt werden.
- Das fehlerhafte Bit wird dann einfach **umgekippt** um den Fehler zu korrigieren.



🔑 Wichtige Konzepte des Verfahrens:

- **Redundanz:** Zusätzliche Bits werden hinzugefügt, um Fehler erkennen zu können.
- **Hamming-Distanz:** Die minimale Anzahl von **Bitflips**, die erforderlich ist, um einen gültigen Code in einen anderen zu verwandeln. Hamming-Codes haben eine Distanz von **3**, was bedeutet, dass sie **einen Fehler korrigieren** und **zwei Fehler erkennen** können.
- **Syndromberechnung:** Die Ausgabe der Paritätsbits ergibt ein sogenanntes **Syndrom**, das direkt den fehlerhaften Bitindex angibt.

📽️ Warum ist das Video so gut?

- **3Blue1Brown** verwendet **anschauliche Animationen**, um das komplexe Thema leicht verständlich zu machen.
- Er erklärt die **logischen Grundlagen** auf eine intuitive Weise, die sowohl für Anfänger als auch für Fortgeschrittene interessant ist.

Hier nochmal der Link zum Video:

🌐 3Blue1Brown - Hamming Codes

Fazit und Ausblick

Reed-Solomon Codes

Turbocodes