INSTITUT FÜR INFORMATIONSVERARBEITUNG

LEIBNIZ UNIVERSITÄT HANNOVER

# MEDIENVERARBEITUNGSLABOR

## Kanalcodierung

| NAME | MAT.–NR. | GRUPPE | |
|---|---|---|---|
| VERSUCHSLEITER | VERSUCHSTAG | | ENDTESTAT |

# Inhaltsverzeichnis

# Glossar

**ARQ** Automatic-Repeat-Request.

**BCH-Code** Bose-Chaudhuri-Hocquenghem Code.

**Codespreizung** Permutation oder Verschränkung der erzeugten Codewörter zur Umwandlung von Fehlerbündeln in Einzelbitfehler.

**Codewort** Alle Ausgangssequenzen, welche durch die Codierung erzeugt werden können.

**FEC** Forward-Error-Correction.

**Informationsstelle** Die Binärstellen eines systematischen Codes welche die Eingangsdatensequenz enthalten.

**LUT** Lookup-Tabelle.

**Maske** Konstante Bitsequenz welche mit einer Eingangsdatensequenz XOR-verknüpft wird.

**Modul** Basiselement eines QR-Codes zur Speicherung von einem Bit.

**Prüfstelle** Die Binärstellen eines systematischen Codes welche die redundanten Korrekturdaten enthalten.

**QR-Code** Quick-Response Code.

**RS-Code** Reed-Solomon Code.

**Segment** Eine Bitsequenz welche genau eine TLV-codierte Nachricht enthält.

**Symbol** Konkrete Instanz eines QR-Codes.

**TLV** Type-Length-Value.

**Wort** Für RS-Codes: Aggregation von acht Bits.

# 1. Einleitung

Wenn in der Informationsverarbeitung Nachrichten übertragen werden, so muss dies immer über ein Medium geschehen welches wir als Kanal bezeichnen. Selbst auf einem zuverlässigen Kanal müssen wir in der Praxis mit geringen Störeinflüssen rechnen. Diese Störeinflüsse können aus sehr unterschiedlichen Gründen auftreten, sei es durch Interferenz auf einer Funkstrecke oder durch unsauber verarbeitetes Material in einem Glasfaserkabel. Sie können aber dazu führen, dass komprimierte Daten anschließend im Quellendecoder nicht mehr korrekt decodiert werden können. Durch eine geschickte Kanalcodierung sollen fehlerhaft übertragene Nachrichten erkannt und repariert werden. Moderne Verfahren erlauben sogar eine sinnvolle Funkverbindung über weite Entfernungen bei geringer Sendeleistung, wie bei den Voyager Missionen der NASA. Alltäglichere Anwendungsgebiete sind z.B. die Datenspeicherung auf optischen Medien (CD-Audio, DVD, Blu-Ray), Mobilfunk und DVB-T.

In diesem Laborversuch sollen die Grundlagen der Kanalcodierung am Beispiel von Quick-Response Codes (QR-Codes) aufgezeigt werden. QR-Codes wurden ursprünglich von der japanischen Firma Denso-Wave zur Kennzeichnung von Bauteilen in der Fahrzeugfertigung entwickelt und sind mittlerweile in der Norm ISO/IEC 18004 standardisiert. Sie stellen eine Weiterentwicklung der eindimensionalen Strichcodes dar, welche aber nur eine geringe Datenmenge speichern konnten. QR-Codes haben sich durch eine breite Unterstützung durch Smartphones auch am Massenmarkt durchsetzen können und werden vor allem zur Verbreitung von URLs und Kontaktinformationen eingesetzt.

Der erste Teil des Laborumdrucks beschäftigt sich mit den Grundlagen der Kanalcodierung und mit binären linearen Blockcodes. Im zweiten Teil wird der Aufbau von QR-Codes sowie deren Codierung und Decodierung erläutert. Abschließend folgen eine Liste der benötigten MATLAB-Funktionen sowie die zu bearbeitenden Aufgaben.

# 2. Grundlagen

## 2.1. Kanalmodell

Abbildung 2.1 zeigt das Blockschaltbild eines digitalen Nachrichtenübertragungssystems. Die für diesen Laborversuch wesentlichen Komponenten sind sowohl der Kanalcoder/-decoder als auch der Übertragungskanal. Um einen optimalen Kanalcoder zu spezifizieren muss zuerst der Kanal modelliert werden, insbesondere der darauf üblichen Störeinflüsse. Häufig kommen hierfür statistische Methoden zur Anwendung. Im Rahmen dieses Versuchs soll das Hauptaugenmerk erst einmal auf die Unterscheidung zwischen unterschiedlichen Fehlerstrukturen gelegt werden:

**Einzelbitfehler:** Die einzelnen Bits wurden unabhängig voneinander gestört

**Bündelfehler:** Mehrere aufeinanderfolgende Bits wurden gestört

Im Gegensatz zu Einzelbitfehlern sind die Fehler in einem Fehlerbündel statistisch voneinander abhängig. Der Kanal lässt sich dann häufig vereinfacht durch zwei Zustände beschreiben. Der Basiszustand hat eine geringe Fehlerwahrscheinlichkeit. Tritt ein Fehler auf, so wechselt der Kanal in einen Zustand mit erhöhter Fehlerwahrscheinlichkeit.

**Beispiel Compact-Discs:** Durch Kratzer auf der Oberfläche werden immer größere räumlich benachbarte Datenbereiche gestört.

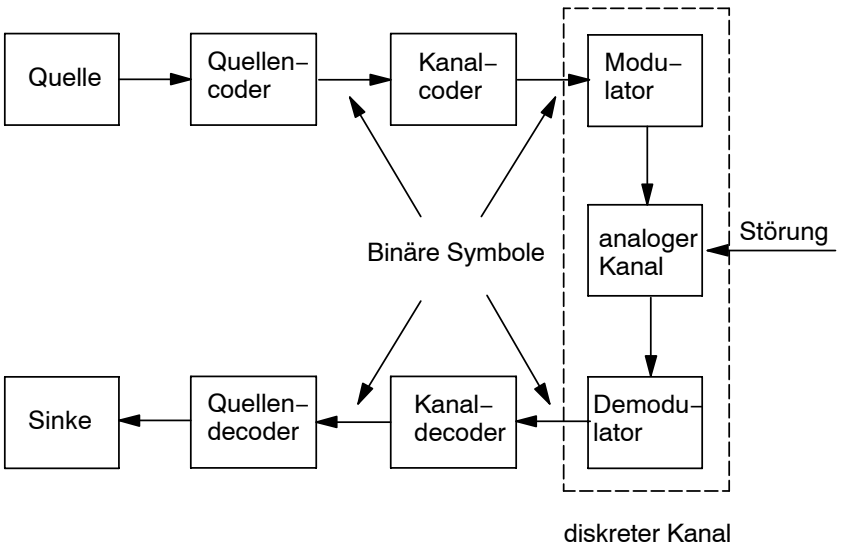



Abbildung 2.1.: Blockschaltbild eines digitalen Nachrichtenübertragungssystems

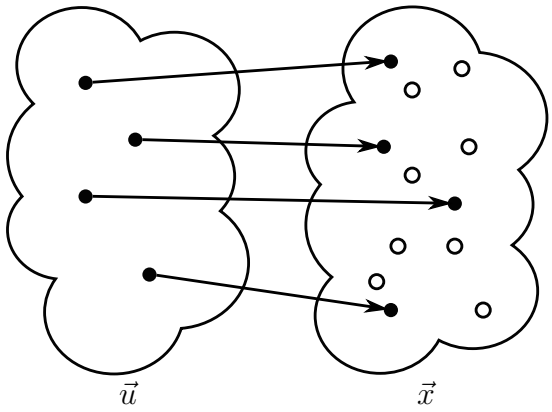



Abbildung 2.2.: Kanalcodierung als injektive Abbildung

Die Kanalcodierung muss also immer an die Eigenschaften des vorliegenden Kanals angepasst werden. Die folgenden Abschnitte werden zuerst einige grundlegende Konzepte und Werkzeuge der Kanalcodierung einführen. Anschließend zeigt Kapitel 3, wie diese Konzepte im ISO/IEC 18004 Standard für QR-Codes umgesetzt wurden.

## 2.2. Kanalcodierung

Ziel der Kanalcodierung ist es, Nutzinformationen so über den Kanal zu übertragen, dass sie möglichst fehlerfrei beim Empfänger ankommen. Die zwei typischen Strategien sind Automatic-Repeat-Request (ARQ) und Forward-Error-Correction (FEC). In ARQ-Protokollen wird nach der Detektion potentieller Übertragungsfehler eine Neuübertragung der Daten angefordert. FEC hingegen codiert die Nutzinformationen in einer Form, in der typische Übertragungsfehler erkannt und korrigiert werden können.

In diesem Labor werden binäre Blockcodes zum Einsatz kommen. Die Nutzinformationen liegen hier in binärer Form vor. Jeweils $K$ aufeinanderfolgende Quellensymbole werden zu einem Block $\vec{u} = (u_0, \ldots, u_{K-1})$ zusammengefasst. Eine *Codierung* ist eine injektive Abbildung (siehe Abbildung 2.2):

$$f : \{0,1\}^K \rightarrow \{0,1\}^N, \vec{u} \mapsto \vec{x}, N \geq K. \tag{2.1}$$

$\vec{x} = (x_0, \ldots, x_{N-1})$ wird als Codewort bezeichnet. Die Menge aller durch $f$ erzeugbaren Codewörter wird als *Code* bezeichnet. Da $\vec{x}$ mehr Binärstellen hat als $\vec{u}$, ohne dass zusätzliche Information hinzugefügt wird, enthält der erzeugte Code eine höhere Redundanz als die Eingangsdaten.

**Definition 1 (*Relative Redundanz*)**

*Sei $K$ die Anzahl der Binärstellen in den Eingangsdaten und $N$ die Anzahl der Binärstellen eines Codewortes. Die relative Redundanz eines Code berechnet sich dann gemäß*

$$r = \frac{N - K}{N}. \tag{2.2}$$

**Definition 2 (*Coderate*)**

*Der relative Informationsgehalt eines Codewortes*

$$R = 1 - r = \frac{K}{N} \tag{2.3}$$

Eine *Decodierung* ist das Gegenstück zur Codierung:

$$f^{-1} : \{0,1\}^N \to \{0,1\}^K, \vec{y} \mapsto \vec{u}. \tag{2.4}$$

Aufgrund der Injektivität der Codierung ist nicht jedes Element der Zielmenge ein gültiges Codewort, deshalb handelt es sich bei $f^{-1}$ um eine Links-Inverse. Durch Übertragungsfehler kann es deshalb vorkommen, dass der empfangene Block $\vec{y}$ nicht eindeutig auf ein $\vec{u}$ abgebildet werden kann. Wir werden daher im Folgenden aus geometrischen Überlegungen herleiten, wie eine Decodierung in solchen Fällen verfahren sollte.

**Definition 3 (*Hamming-Distanz*)**

*Die Anzahl der Codeelemente entsprechender Stellen, in denen sich zwei gleich lange Codewörter eines Binärcode unterscheiden. Die Hamming-Distanz zweier Codewörter $\vec{x}_i$ und $\vec{x}_j$ lässt sich berechnen gemäß*

$$D(\vec{x}_i, \vec{x}_j) = w(\vec{x}_i \oplus \vec{x}_j) = w(\vec{x}_R), \tag{2.5}$$

*wobei $w(\vec{x}_R)$ das Gewicht von $\vec{x}_R$ d.h. die Anzahl der mit 1 belegten Stellen von $\vec{x}_R$ ist. Die Modulo-2-Summe $\vec{x}_R = \vec{x}_i \oplus \vec{x}_j$ enthält dabei genau an den Stellen eine 1, an denen sich $\vec{x}_i$ und $\vec{x}_j$ unterscheiden.*

**Definition 4 (*Codedistanz*)**

*Die kleinste aller Hamming-Distanzen der Codewörter eines Code*

$$d = min\ D(\vec{x}_i, \vec{x}_j), \forall\ i \neq j \tag{2.6}$$

Abbildung 2.3 zeigt beispielhaft einen fehlerkorrigierenden Code. Die schwarzen Punkte markieren zum Code gehörende Codewörter. Durch Übertragungsfehler können einzelne Bits gestört werden, wodurch ein benachbartes Codewort empfangen wird. Falls das empfangene Codewort nicht zum definierten Code gehört, so kann dieser Fehler eindeutig erkannt werden. Man spricht in diesem Fall von *Fehlererkennung*. Weiterhin ist es möglich ein gestörtes Codewort dem, nach Definition der Hamming-Distanz, nächstgelegensten gültigen Codewort zuzuordnen. Hier spricht man von *Fehlerkorrektur*. Hieraus ergibt sich auch gleich ein mögliches Decodierungsverfahren. Bei der *Brute-Force*-Decodierung wird die empfangene Nachricht mit allen $2^K$ Codewörtern verglichen.

Aus geometrischen Überlegungen können nun Beziehungen zwischen der Codedistanz und der Anzahl der erkennbaren, bzw. korrigierbaren Fehler abgeleitet werden. Um mindestens $e$ Fehler erkennen zu können, muss die Codedistanz

$$d \geq e + 1 \tag{2.7}$$

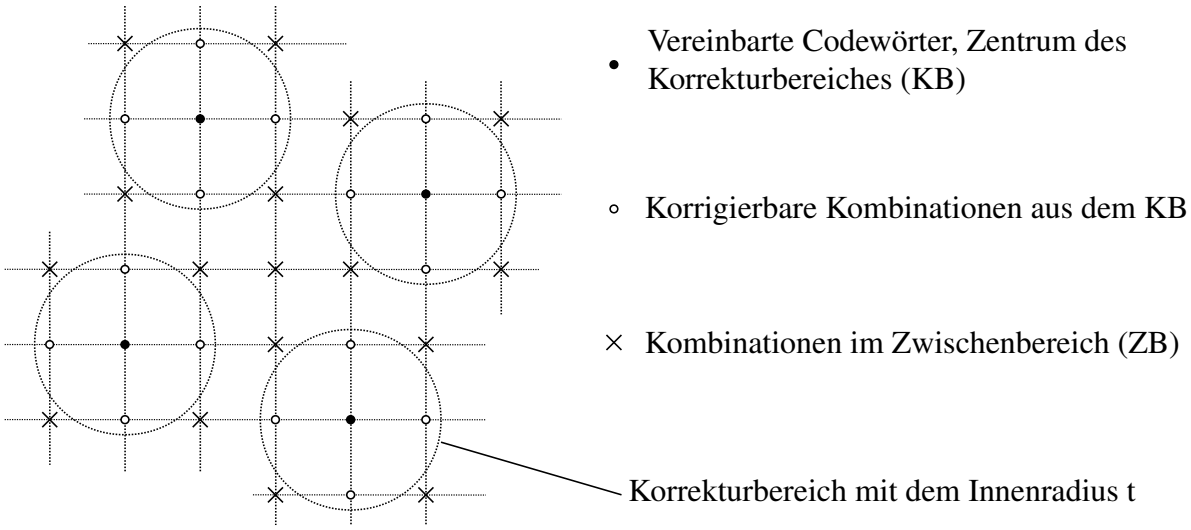



Abbildung 2.3.: Geometrische Darstellung eines fehlerkorrigierenden Code

sein. Die Codedistanz eines $t$-fehlerkorrigierenden Code muss die Bedingung

$$d \geq 2t + 1 \tag{2.8}$$

erfüllen.

**Definition 5 (*Systematische Codes*)**

*Systematische Codes haben die Eigenschaft, dass die zu codierenden Informationsbits in unveränderter Form auch im erzeugten Codewort enthalten sind. Für Blockcodes ergibt sich folgender Zusammenhang:*

| ← K → | | ← K → | ← N - K → |
|---|---|---|---|
| $u_0, \ldots, u_{K-1}$ | $\longleftrightarrow$ | $u_0, \ldots, u_{K-1}$ | $x_K, \ldots, x_{N-1}$ |

*Die Stellen $u_i, i \in [0, K-1]$ werden auch als Informationsstellen bezeichnet und $x_j, j \in [K, N-1]$ als Prüfstellen.*

In diesem Labor werden lineare Blockcodes und Reed-Solomon Codes (RS-Codes) zum Einsatz kommen. Es handelt sich bei RS-Codes um einen Spezialfall von Bose-Chaudhuri-Hocquenghem Codes (BCH-Codes), welche wiederrum einen Spezialfall der linearen Blockcodes darstellen. Abbildung 2.4 zeigt unterschiedliche Klassen von Blockcodes. In Abschnitt 2.3 sollen zunächst die Grundlagen linearer Blockcodes erarbeitet werden.

## 2.3. Lineare Blockcodes

Unter linearen Blockcodes werden alle Codes zusammengefasst, welche sich durch folgende Codiervorschrift beschreiben lassen:

$$\vec{x} = \vec{u} \cdot G. \tag{2.9}$$

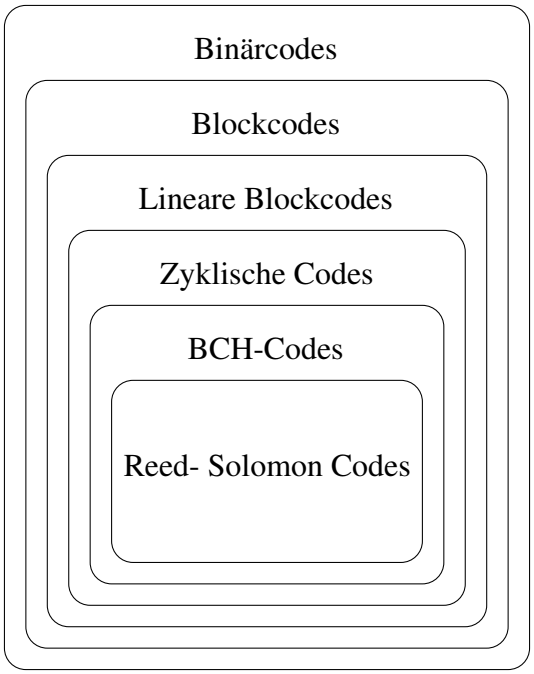



Abbildung 2.4.: Codeklassenhierarchie als Mengendiagramm

Die Vektoren $\vec{x} = (x_0, \ldots, x_{N-1})$ und $\vec{u} = (u_0, \ldots, u_{K-1})$ beschreiben jeweils die erzeugten Codewörter und die zu codierenden Informationsstellen. Da es sich um einen binären Code handelt muss jedes Element $x_i$ und $u_j$ die Werte $0$ oder $1$ annehmen. Bei allen Berechnungen wird die *Modulo-2*-Arithmetik verwendet, d.h Additionen werden ohne Übertrag berechnet und entsprechen somit einer stellenweisen *XOR*-Verknüpfung. Multiplikationen hingegen entsprechen einer stellenweisen *UND*-Verknüpfung. Die Operationen lassen sich durch folgende Tafeln beschreiben:

| $+$ | $0$ | $1$ |
|---|---|---|
| $0$ | $0$ | $1$ |
| $1$ | $1$ | $0$ |

| $\cdot$ | $0$ | $1$ |
|---|---|---|
| $0$ | $0$ | $0$ |
| $1$ | $0$ | $1$ |

Der Code wird im Wesentlichen durch die Matrix $G \in \{0,1\}^{K \times N}$ definiert, $G$ wird daher auch als *Generatormatrix* bezeichnet. Da fehlerkorrigierende Codes per Definition *injektiv* sind, lässt sich die Decodierung nicht über die Inverse von $G$ erreichen. Wir führen deshalb die *Parity-Check-Matrix* $H \in \{0,1\}^{(N-K) \times N}$ ein. Sie ist so gewählt dass gilt:

$$\vec{x} \cdot H^T = 0 \tag{2.10}$$

Sei $\vec{y}$ ein empfangenes Codewort. Gleichung (2.10) gilt gdw. $\vec{y}$ während der Übertragung nicht gestört wurde. Aufgrund der Linearität des Code gilt

$$S = \vec{y} \cdot H^T = (\vec{x} + \vec{e})H^T = \vec{e} \cdot H^T, \tag{2.11}$$

d.h. $S = (s_0, \ldots, s_{N-K-1})$ ist nur von dem Fehlervektor $\vec{e} = (e_0, \ldots, e_{N-1})$ abhängig. Ist $\vec{e}$ bekannt, so kann das ungestörte Codewort durch

$$\vec{x} = \vec{y} \oplus \vec{e} \tag{2.12}$$

Tabelle 2.1.: Eine Beispiel Syndromtabelle für einen ein-Fehler korrigierenden ($N = 7$, $K = 4$) Blockcode

| $S$ | | $\vec{e}$ |
|---|---|---|
| 000 | $\rightarrow$ | 0000000 |
| 001 | $\rightarrow$ | 0000001 |
| 010 | $\rightarrow$ | 0000010 |
| 011 | $\rightarrow$ | 0000100 |
| 100 | $\rightarrow$ | 0001000 |
| 101 | $\rightarrow$ | 0010000 |
| 110 | $\rightarrow$ | 0100000 |
| 111 | $\rightarrow$ | 1000000 |

rekonstruiert werden. $S$ wird auch als *Syndrom* bezeichnet. Durch das Syndrom können $2^{N-K}$ Fehlervektoren unterschieden werden. Die Abbildung von dem berechneten Syndrom zu einem Fehlervektor geschieht über eine Lookup-Tabelle (LUT) (wie z.B. in Tabelle 2.1). Da der Code injektiv ist kann diese Zuordnung nicht eindeutig sein. In der Praxis wird die Syndromtabelle so aufgebaut, dass jedem Syndrom der, laut Kanalmodell, am häufigsten auftretende Fehlervektor zugewiesen wird. Im direkten Vergleich zu einer *Brute-Force*-Decodierung können wir somit die Rechenzeitkomplexität von ca. $2^K$ zu einem einfachen Speicherzugriff reduzieren, was wir uns aber durch Speicherung der vollständigen Syndromtabelle im Arbeitsspeicher erkaufen.

## 2.4. Reed-Solomon Codes

Bei RS-Codes handelt es sich um eine Teilmenge der linearen Blockcodes. Somit könnten sie auch durch eine Generatormatrix beschrieben werden. Durch Ausnutzung von Eigenschaften endlicher algebraischer Körper können RS-Codes in einer Polynomalgebra allerdings deutlich effizienter codiert und decodiert werden. Der Fehlervektor kann somit ohne den Umweg einer Syndromtabelle direkt berechnet werden. Auf eine tiefergehende Erklärung soll an dieser Stelle verzichtet werden, kann aber im Skript zur Vorlesung *Kanalcodierung* nachgeschlagen werden. Die wesentlichen Eigenschaften von RS-Codes seien aber dennoch zu erwähnen:

- Sie haben eine geringe Laufzeit- und Speicherkomplexität
- Sie werden ausschließlich als systematische Codes verwendet.
- RS-Codes arbeiten nicht auf binären Symbolen sondern auf Wortgrößen von jeweils 8 Bit. Werden mehrere Bits innerhalb einer Wortgrenze gestört, so wird dies nur als ein Einzelfehler gewertet. Somit können RS-Codes, auf Bitebene betrachtet, besonders gut mit Bündelfehlern umgehen.
- Für jedes Fehlerkorrekturwort (à 8 Bit) das der Nachricht angehängt wird kann mindestens ein beliebiges gestörtes Wort beim Empfänger erkannt werden.
- Für jeweils zwei Fehlerkorrekturwörter die der Nachricht angehängt werden kann mindestens ein beliebiges gestörtes Wort beim Empfänger korrigiert werden.

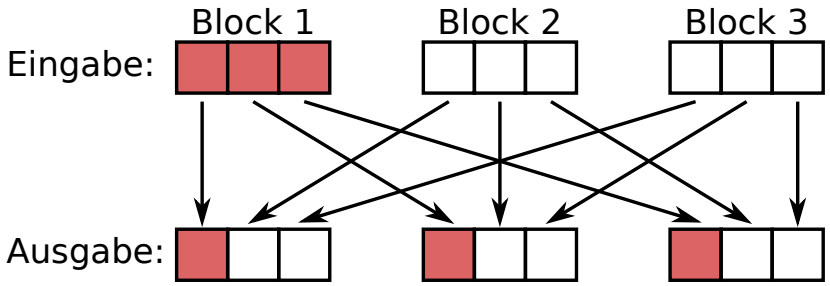



Abbildung 2.5.: Beispiel einer Codespreizung mit drei Eingabeblöcken. Rote Kästchen symbolisieren während der Übertragung gestörte Bits.

## 2.5. Codespreizung

Bei Verwendung von Blockcodes ist es häufig sinnvoll die Blocklänge klein zu halten. Einerseits hängt die Komplexität der (De-)Codierung direkt von der Blocklänge ab, wodurch einer Anwendung auf Hardware mit eingeschränkter Rechenleistung Grenzen gesetzt sind. Andererseits erhöhen große Blocklängen die Latenz der Übertragung, da jeweils nur vollständig übertragene Blöcke am Decoder verarbeitet werden können. Wurde der Blockcode auf eine bekannte Fehlerrate optimiert, so kann dies zu Problemen führen. So können Bündelfehler die Korrekturkapazität eines Blocks überschreiten, ohne jedoch im Mittel die vorher festgelegte Fehlerrate zu verletzen. Durch eine Codespreizung werden Bündelfehler eines Blocks in Einzelbitfehler mehrerer Blöcke umgewandelt. Eine mögliche Implementation wäre die Codeverschränkung, so wie sie in Abbildung 2.5 gezeigt wird.

# 3. QR-Codes

## 3.1. Kanalmodell

Bevor die einzelnen Komponenten eines QR-Code erläutert werden muss man sich zuerst Gedanken über den Anwendungskontext und den Übertragungskanal machen. Abbildung 3.1 zeigt den typischen Übertragungsweg einer Nachricht. Im Vergleich zu anderen üblichen Anwendungen in der Nachrichtentechnik, z.B. das Versenden von Netzwerkpaketen über ein Kupferkabel, spielen hier zusätzliche Faktoren eine wesentliche Rolle:

- Optische Parameter:
  - Beleuchtung
  - Kameraorientierung
  - Skalierung, Perpektive
- Fehlerursachen:
  - Schmutz, Kratzer
  - Papierverzug
  - Verdeckungen

Diese Faktoren sind eine Folge des verwendeten Übertragungskanals und müssen daher in der Kanalcodierung berücksichtigt werden.

Die folgenden Abschnitte beschreiben den Aufbau eines QR-Code, weitere Informationen und Tabellen sind im Appendix A zu finden.

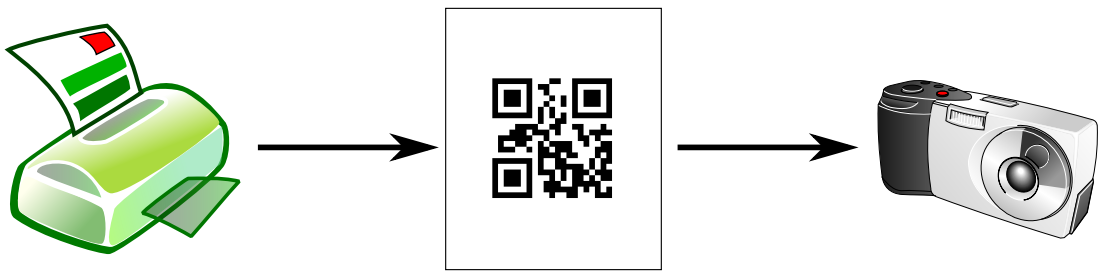
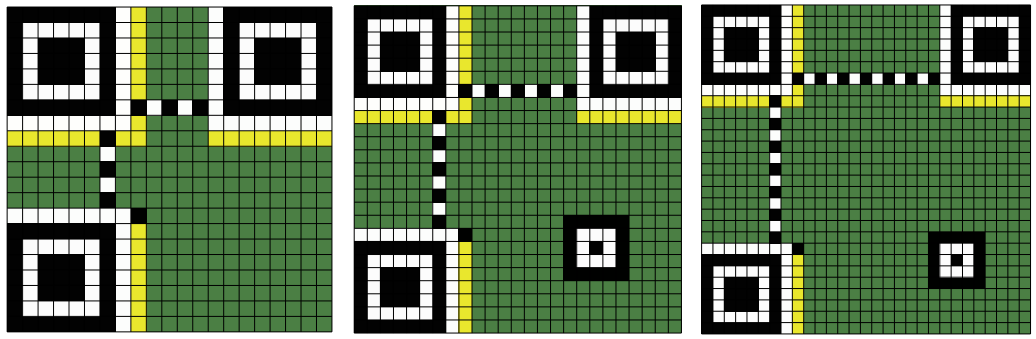
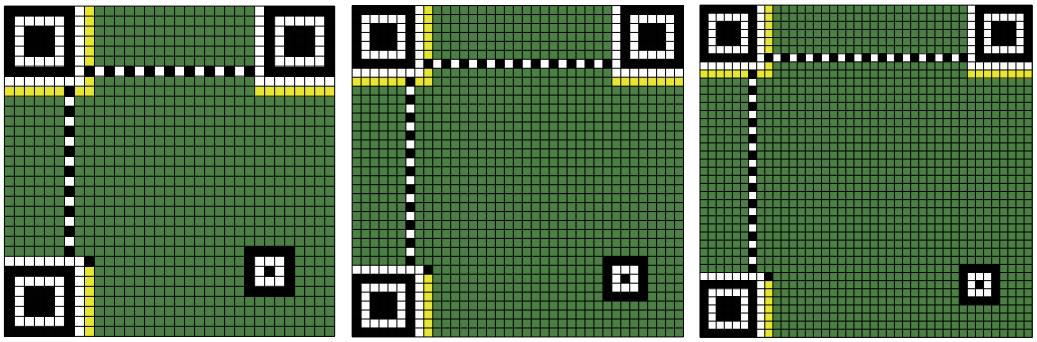


Abbildung 3.1.: Typischer Übertragungskanal für QR-Codes



(a) Version 1 (b) Version 2 (c) Version 3



(d) Version 4 (e) Version 5 (f) Version 6

Abbildung 3.2.: Symbole mit unterschiedlichen Versionen

## 3.2. Aufbau

QR-Codes können mit unterschiedlichen Daten- und Fehlerkorrekturkapazitäten erzeugt werden. Die beiden Stellschrauben zur Anpassung dieser Parameter sind:

1. Die Größe des Symbols
2. Die Coderate

Ein Modul ist die Basiseinheit eines Symbols und kann entweder hell ($= 0$) oder dunkel ($= 1$) gefärbt sein. Es hat somit eine Datenkapazität von einem Bit. Die Größe des Symbols wird über seine *Version* spezifiziert. Das kleinste gültige Symbol hat die Version $1$ und besteht aus $21 \times 21$ Modulen. Jede weitere Version erweitert seinen Vorgänger in beide Richtungen um jeweils vier Module. Abbildung 3.2 zeigt den Aufbau der ersten sechs Versionen.

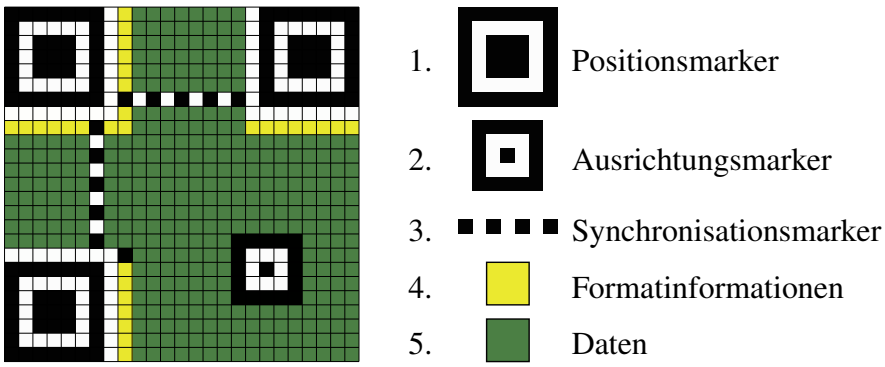



Abbildung 3.3.: Strukturelemente eines QR-Codes

Wie in Abbildung 3.3 gezeigt, besteht ein QR-Code aus unterschiedlichen Strukturelementen:

1. Positionsmarker
2. Ausrichtungsmarker
3. Synchronisationsmarker
4. Formatinformationen
5. Daten

Anhand der Positionsmarker können sowohl Position als auch die Orientierung des Symbols bestimmt werden. Zusätzlich sollte um jedes Symbol eine freie Zone von mindestens vier Modulen vorhanden sein. Dies erleichtert es dem Decoder relevante Module von umliegenden Stördaten zu unterscheiden. Die Formatinformationen und Daten werden gesondert in den folgenden Abschnitten behandelt.

## 3.3. Formatinformation

Jedes Symbol beinhaltet einen Header in dem wichtige Formatinformationen abgespeichert sind. Er enthält den verwendeten Fehlerkorrektur-Level und die zur Datencodierung verwendete Maske. Da diese Daten zum Decodieren wesentlich sind, müssen sie auch vor Beschädigung geschützt werden. Zur Anwendung kommt hier ein linearer ($N = 15$, $K = 5$) Blockcode, welcher bis zu drei Bitfehler korrigieren kann.

### 3.3.1. Aufbau

Die Formatinformationen haben insgesamt eine Länge von $15$ Bits. Aus Gründen der Redundanz ist diese Bitfolge zweifach in jedem Symbol gespeichert, es reicht jedoch in der Regel auch nur eine Instanz zu lesen. Abbildung 3.4 zeigt Position und Aufbau der Formatinformationen in einem Symbol.

Von den $15$ Bits sind $5$ Bits Nutzdaten. Die restlichen $10$ Bits werden zur Fehlerkorrektur

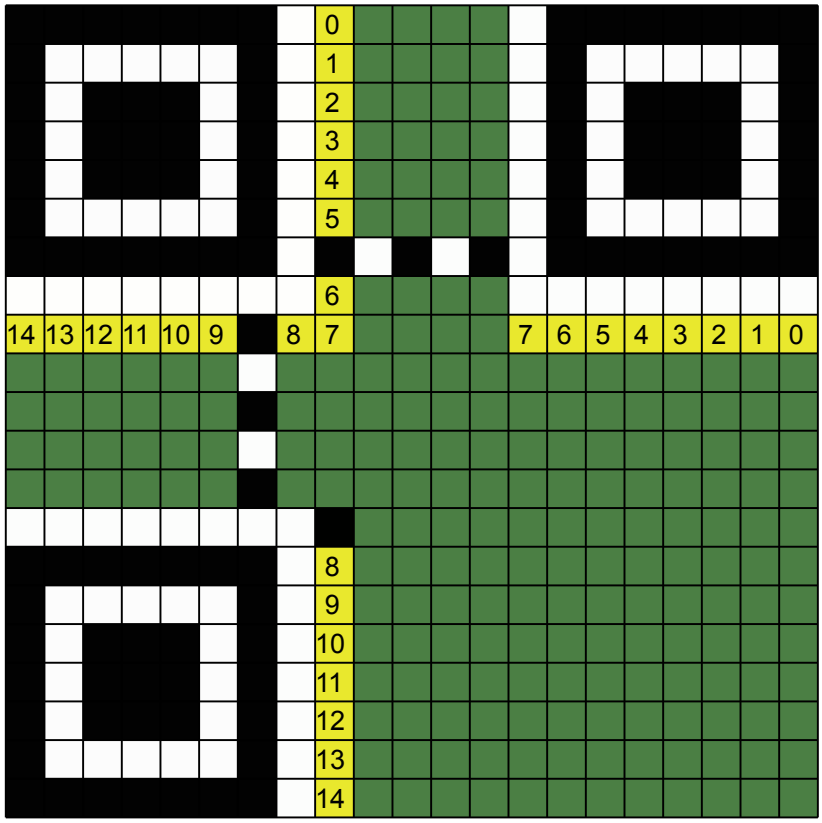



Abbildung 3.4.: Formatinformationen

verwendet:

| 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ECL | | MASK | | | ECC | | | | | | | | | |

Die obersten zwei Bits geben den Fehlerkorrektur-Level (ECL) an mit dem die Daten codiert sind. Tabelle A.1 zeigt alle vier Optionen und die jeweils erreichte Korrekturkapazität an. Die nachfolgenden drei Bits beschreiben die auf die Daten angewandte Maske (MASK). Näheres dazu steht im Abschnitt 3.4.5. Die untersten zehn Bits beinhalten die Prüfstellen (ECC).

### 3.3.2. Fehlerkorrektur

Zur Erzeugung der Prüfstellen wird ein systematischer BCH-Code verwendet. In diesem Fall kann die (De-)Codierung auch sinnvoll mit einer LUT erledigt werden. Da zur Speicherung der Informationsstellen lediglich fünf Bits verwendet werden umfasst die Tabelle $2^5 = 32$ Einträge. Die vollständige LUT ist unter Tabelle A.2 gegeben. Die Codierung erfolgt durch ein einfaches nachschlagen des Codewortes zu den gegebenen Informationsstellen. Die Informationsstellen werden hierzu als Ganzzahl interpretiert und bilden so die gesuchte Zeilennummer in der Tabelle. Die Decodierung kann durch einen Brute-Force

Ansatz gelöst werden. Ein empfangenes, eventuell gestörtes, Codewort wird mit jeden Tabelleneintrag verglichen. Es wird der Eintrag gewählt, der die geringste Hamming-Distanz zum Codewort hat.

### 3.3.3. Maske

Die codierte Formatinformation wird zusätzlich mit der konstanten Bitsequenz $\vec{M}_{Format} = 101010000010010$ maskiert:

$$\vec{F}_{masked} = \vec{F} \oplus \vec{M}_{Format}.$$

Der Operator $\oplus$ ist dabei ein bitweises XOR. Dies soll verhindern, dass lange Folgen von Nullen oder Einsen in der gespeicherten Bitsequenz vorkommen. Beim decodieren wird dieselbe Operation zum Entfernen der Maske verwendet:

$$\vec{F} = \vec{F}_{masked} \oplus \vec{M}_{Format}.$$

## 3.4. Daten

Der Datenbereich beinhaltet sowohl die Informationsstellen als auch die Prüfstellen der zu übertragenen Nachricht. Es gibt im QR-Code Standard neben der Möglichkeit Rohdaten zu speichern unter anderem auch Modi zur Codierung von numerischen oder alphanumerischen Daten. Der Vorteil dieser Formate liegt in der kompakteren Darstellung der Informationen, und somit in einer höheren Datenkapazität.

### 3.4.1. Datensegmente

Die Daten werden im sogenannten Type-Length-Value (TLV) Format abgespeichert. Dazu werden die Daten in Segmente aufgeteilt, wobei jedes Segment einem festgelegten Muster entsprechend aufgebaut ist. Zuerst wird ein $4$-Bit Indikator erzeugt, welcher den Typ der folgenden Nutzdaten in diesem Segment beschreibt. Tabelle A.3 zeigt alle gebräuchlichen Formate, wobei Textdaten häufig im Byte Format (z.B. Latin-1 codiert) abgespeichert werden. Anschließend folgt die Anzahl der in diesem Segment enthaltenen Zeichen. Die Anzahl der Bits, die hierfür benötigt werden, ist in Tabelle A.4 angegeben. Das Segment wird mit den Nutzdaten abgeschlossen. Im Anschluss kann ein neues Segment angelegt werden. Sollte dies nicht der Fall sein, so wird hinter dem letzten Segment der Vorgang mit dem Terminator (Indikator 0000) abgeschlossen.

| Segment 1 | | | | Segment N | | | Terminator |
|---|---|---|---|---|---|---|---|
| Type | Length | Value | $\cdots$ | Type | Length | Value | 0000 |

Ungenutzte Bits werden bis zur nächsten Byte-Grenze mit Nullen aufgefüllt, also bis die Anzahl der geschriebenen Bits durch acht teilbar ist. Sollten nun weiterhin ungenutzte Bytes im Datenbereich verbleiben, abhängig von Version und Fehlerkorrektur-Level, so werden diese abwechselnd mit den Platzhaltern $EC_{16} = 11101100_2$ und $11_{16} = 00010001_2$ aufgefüllt.

**Beispiel:**
Die empfangene Bitsequenz laute:

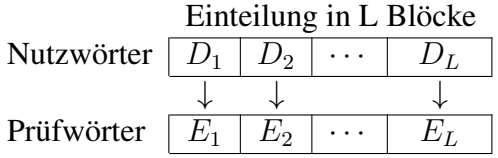
$$\underbrace{0100}_{T}\underbrace{00000010}_{L}\underbrace{0100111101001011}_{V}\underbrace{0000}_{T}1110110000010001111011 00\ldots$$

Sie lässt sich folgendermaßen in ihre Einzelkomponenten aufteilen:

- Typindikator (hier Byte-Modus)
- Anzahl der folgenden Zeichen (hier 2)
- Daten (hier “OK” in Latin-1 Codierung)
- Typindikator (hier Terminator)
- Platzhalter

## 3.4.2. Fehlerkorrektur

Entsprechend der Tabelle A.6 werden die Eingangsdaten aus dem vorherigen Abschnitt in L Blöcke aufgeteilt. Diese Blöcke $D_1, \ldots, D_L$ enthalten die Informationsstellen. Zu jedem Block $D_i$ wird anschließen durch Codierung mit dem passenden RS-Code ein Block $E_i, i \in [1, L]$ mit den dazugehörigen Prüfstellen erstellt.




**Beispiel:**
Als Eingabe dient ein Datenstrom der nach Abschnitt 3.4.1 erzeugt wurde. Der Datenstrom hat eine Länge von $40$ Wörtern und wir wollen den Fehlerkorrektur-Level *H* verwenden. Aus Tabelle A.6 ist zu ersehen, dass QR-Codes der Version $4$ in diesem Fehlerkorrektur-Level bis zu $36$ Informationswörter speichern können, also etwas zu wenig. Für Version $5$ beträgt dieser Wert $46$ Wörter und ist somit ausreichend. Aus derselben Tabelle ist zu ersehen, dass der Datenstrom in vier Blöcke aufgeteilt werden muss. Die ersten beiden Blöcke haben eine Länge von je $11$ Informationswörtern und werden mit einem (33, 11)-RS-Code codiert. Die anderen beiden Blöcke haben eine Länge von je $12$ Informationswörtern. Sie werden mit einem (34, 12)-RS-Code codiert.

## 3.4.3. Codespreizung

Die Codespreizung geschieht bei QR-Codes durch eine Verschränkung der erzeugten Blöcke. Die Verschränkung erfolgt dabei auf Wortebene. Zuerst werden jeweils die ersten Wörter aller Informationsblöcke herausgeschrieben. Anschließend folgen die zweiten Wörter, usw., bis schließlich alle Informationsblöcke vollständig geschrieben wurden. Anschließend wird dasselbe Vorgehen mit den Prüfblöcken wiederholt.

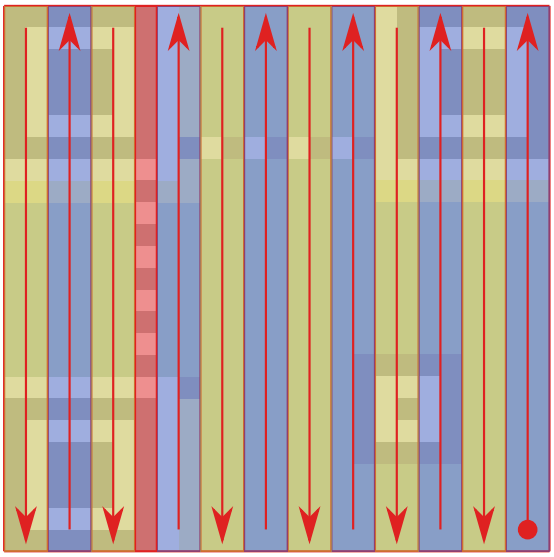


Abbildung 3.5.: Abbildung der Bitsequenz: Mäanderförmiger Verlauf

**Beispiel:**
Aufbauend auf dem Beispiel des vorangegangenen Abschnitts wird wieder ein QR-Code der Version 5 mit Fehlerkorrektur-Level *H* verwendet. Die Verschränkung der jeweils vier Informationsblöcke $D_i$ und Prüfblöcke $E_i$, $1 \leq i \leq 4$, erfolgt auf folgende Weise:

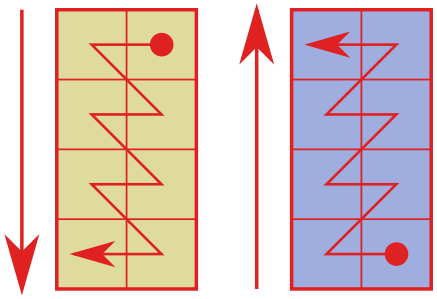
| Informationsblöcke | | | | | | | | Prüfblöcke | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $D_1(1)$ | ↓ | $D_1(2)$ | ↓ | $\cdots$ | $D_1(11)$ | ↓ | | $E_1(1)$ | ↓ | $E_1(2)$ | ↓ | $\cdots$ | $E_1(22)$ | ↓ |
| $D_2(1)$ | | $D_2(2)$ | | $\cdots$ | $D_2(11)$ | | | $E_2(1)$ | | $E_2(2)$ | | $\cdots$ | $E_2(22)$ | |
| $D_3(1)$ | ↓ | $D_3(2)$ | ↓ | $\cdots$ | $D_3(11)$ | ↓ | $D_3(12)$ | $E_3(1)$ | ↓ | $E_3(2)$ | ↓ | $\cdots$ | $E_3(22)$ | |
| $D_4(1)$ | ↗ | $D_4(2)$ | ↗ | $\cdots$ | $D_4(11)$ | ↗ | $D_4(12)$ | $E_4(1)$ | ↗ | $E_4(2)$ | ↗ | $\cdots$ | $E_4(22)$ | |

wobei $D_i(j)$ das $j$-te Wort im Informationsblock $i$ bezeichnet.

### 3.4.4. 2D-Mapping

Die nun vorliegende Bitsequenz muss auf den zweidimensionalen Datenbereich des Symbols abgebildet werden. Diese Abbildung ist durch eine Kurve definiert. Die Reihenfolge der Datenmodule, durch welche die Kurve wandert, entspricht der Sequenz der zu codierenden Ausgangsdaten. Zur Beschreibung des Kurvenverlaufs werden zunächst jeweils zwei benachbarte Spalten des Symbols zusammengefasst. Der vertikale Synchronisationsmarker sollte dabei übersprungen werden. Beginnend bei der unteren rechten Ecke wandert die Kurve, wie in Abbildung 3.5 zu sehen ist, mäanderförmig entlang der Doppelspalten über das gesamte Symbol.

Innerhalb einer Doppelspalte wandert die Kurve entlang einer Sägezahnform. Die genaue Form hängt von der aktuellen vertikalen Laufrichtung ab und ist in Abbildung 3.6 gegeben.



(a) Abwärts (b) Aufwärts

Abbildung 3.6.: Abbildung der Bitsequenz: Sägezahnförmiger Verlauf

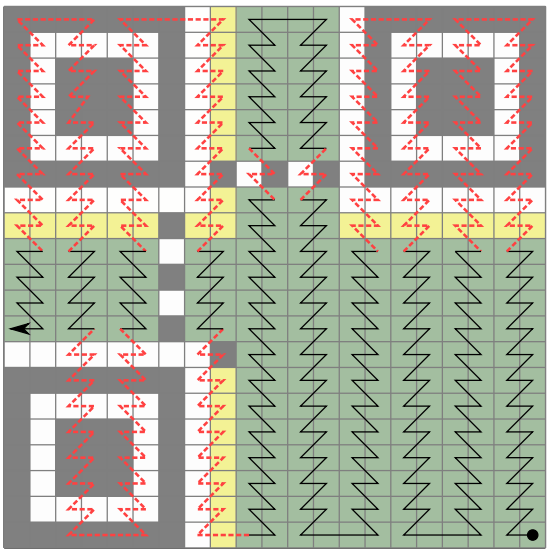
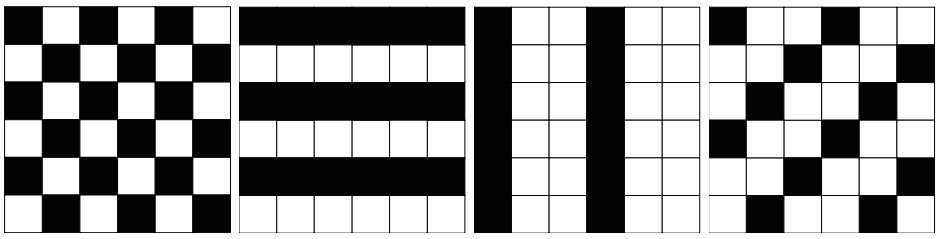


Abbildung 3.7.: Abbildung der Bitsequenz: Vollständiges Beispiel

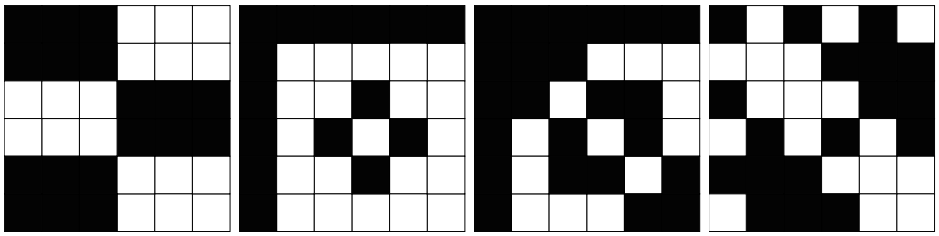
Wandert die Kurve entlang ihres Verlaufs auf Nicht-Datenmodule, so sollten diese einfach ignoriert werden. Abbildung 3.7 zeigt ein vollständiges Beispiel anhand eines QR-Codes der Version 1.

### 3.4.5. Maske

Die codierten Daten werden zusätzlich mit einer zweidimensionalen Maske maskiert. Es stehen dabei acht unterschiedliche Maskentypen zur Auswahl, wobei die Selektion über den Maskenindikator in den Formatinformationen geschieht. Tabelle A.5 und die dazugehörige Abbildung 3.8 zeigen die Berechnungsvorschriften aller verwendeten Maskentypen. Das Koordinatenpaar $(x, y)$ beschreibt die Position eines Moduls innerhalb des Symbols. Für Version 1 z.B. startet die Zählung bei $(0, 0)$ in der linken oberen Ecke und endet



(a) 000 (b) 001 (c) 010 (d) 011



(e) 100 (f) 101 (g) 110 (h) 111

Abbildung 3.8.: $6 \times 6$ Ausschnitte der Datenmasken. Dunkle Module symbolisieren eine $1$, helle eine $0$.

bei $(20, 20)$ rechts-unten. Sowohl bei der Codierung als auch der Decodierung gilt:

**foreach** *(x, y)* $\in$ *DataModules* **do**
| $Symbol(x, y) = Symbol(x, y) \oplus M_{Data}(x, y)$
**end**

# 4. Matlab

## 4.1. Laborspezifische Funktionen

**Funktion 1 (`read_symbol`)**

```
symbol = read_symbol(filename)
```

*Liest ein QR-Code Symbol aus einer Bilddatei. Eine eventuell vorhandene freie Randzone wird dabei entfernt. Die Rückgabematrix enthält boolsche Werte und wird so herunterskaliert, dass ein Eintrag der Matrix genau einem Modul entspricht.*
***Beispiel:*** `symbol = read_symbol('qr.png');`

**Funktion 2 (`number_to_bitstream`)**

```
bitstream = number_to_bitstream(value, bits)
```

*Wandelt einen positiven Integralwert in einen binären Datenvektor um. Der Parameter* `bits` *gibt die Anzahl der Bits an, die zur Codierung der Zahl verwendet werden. Ungenutzte Bits werden mit* 0 *aufgefüllt.*
***Beispiel:*** `teststream = number_to_bitstream(10, 4);`
*Schreibt den Dezimalwert* 10 *als* 4*-Bit Binärzahl auf den Stream.*

**Funktion 3 (`bitstream_to_number`)**

```
value = bitstream_to_number(bitstream, bits)
```

*Wandelt einen binären Datenvektor in einen positiven Integralwert um. Der Parameter* `bits` *gibt die Anzahl der Bits an, die zur Codierung der Zahl verwendet werden.*
***Beispiel:*** `val = bitstream_to_number(teststream, 4);`
*Liest eine* 4*-Bit Binärzahl aus dem Stream und speichert sie in die Variable* `val`*.*

**Funktion 4 (`text_to_bitstream`)**

```
bitstream = text_to_bitstream(text)
```

*Wandelt einen String in einen binären Datenvektor um.*

**Funktion 5 (`bitstream_to_text`)**

```
text = bitstream_to_text(bitstream)
```

*Wandelt einen binären Datenvektor in einen String um.*

**Funktion 6 (`simulate_noise`)**

```
noisy_bitstream = simulate_noise(bitstream, errors[, burst])
```

*Erzeugt eine festgelegte Anzahl an Bitfehlern an zufälligen Positionen im Bitstream. Die Anzahl der Bitfehler wird über den Parameter* `errors` *gesteuert. Ist der Parameter* `burst` *auf* `true` *gesetzt, so wird ein Bündelfehler mit der festgelegten Länge an einer zufälligen Position im Bitstream erzeugt.*

**Funktion 7 (`poly_div`)**

```
[result rest] = poly_div(dividend, divisor)
```

*Berechnet den Quotienten und den Rest einer Polynomdivision mit binären Koeffizienten.*

## 4.2. Wichtige Matlab-interne Funktionen

**Funktion 8 (`global`)**

```
global varname
```

*Stellt die Variable* `varname` *aus dem globalen* Workspace *innerhalb einer Funktion zur Verfügung. Muss am Anfang jeder Funktion aufgerufen werden in der die globale Variable verwendet wird.*

**Funktion 9 (`xor`)**

```
C = xor(A, B)
```

*Berechnet ein elementweises exklusiv-oder auf den Eingangsmatrizen.*

**Funktion 10 (`sum`)**

```
result = sum(A)
```

*Summiert über alle Elemente des Eingabevektors* `A`.

**Funktion 11 (`mod`)**

```
result = mod(A, B)
```

*Berechnet elementweise den Rest von* `A/B`. *Es darf sich bei* `B` *um einen Skalar handeln.*

# 5. Aufgaben

## 5.1. Vorbereitungsaufgaben

1. In welchen Anwendungsfällen ist eine ARQ-Strategie einer FEC-Strategie vorzuziehen?
2. Wann ist eine Decodierung durch eine Syndromtabelle gegenüber einer *Brute-Force*-Decodierung vorzuziehen?
3. Geben Sie eine untere Schätzung der Codedistanz des BCH-Codes an, welcher zur Codierung der Formatinformationen genutzt wird.
   **Hinweis:** Siehe dazu auch Abschnitt 3.3.
4. Geben Sie die Menge der Nutzdaten an, welche mit QR-Codes der Version $2$ und $5$ in allen vier Fehlerkorrekturleveln höchstens codiert werden können.
5. Wieviele Fehler können mit diesen QR-Codes erkannt und korrigiert werden?

## 5.2. Versuchsdurchführung

Alle Aufgaben wurden zur Bearbeitung in MATLAB konzipiert. Nachdem Sie die MATLAB-Umgebung gestartet haben sollten Sie zuerst in das Arbeitsverzeichnis *„./project/nva/kanalcodierung“* wechseln. Führen Sie anschließend den Befehl `startup()` aus um die Arbeitsumgebung zu initialisieren.

### 5.2.1. Fehlererkennung

In ARQ-Protokollen kommen fehlererkennende Codes zum Einsatz. Im ersten Versuch sollen zur Einführung zwei Varianten solcher Codes untersucht werden. Die erste dieser Varianten stellt das *Parity-Bit* dar. Wird die zu übertragene Nachricht $\vec{x} = x_{N-1}, ..., x_0 \quad , x_i \in \{0, 1\}$ in binärer Darstellung beschrieben, so lässt sich das *Parity-Bit* durch die Formel

$$p = x_0 \oplus x_1 \oplus ... \oplus x_{N-2} \oplus x_{N-1}$$

berechnen. Anschließend wird die Nachricht $\vec{y} = x_{N-1}, ..., x_0, p$ über den Kanal übertragen. Auf Empfängerseite kann zur Fehlererkennung das *Parity-Bit* der Nachricht $\vec{y}$ berechnet werden. Ist es $0$, so wurde die Nachricht korrekt übertragen.

1. Schreiben Sie eine Funktion, welche das *Parity-Bit* für beliebige binäre Vektoren berechnet.

2. Verwenden Sie die Funktion `simulate_noise` um unterschiedliche Fehlermuster auf so codierten Testdaten zu simulieren. Welche Arten von Fehlermustern können durch eine *Parity-Bit*-Codierung erkannt werden?

Um mehr Fehlermuster sicher erkennen zu können, kommen in diversen Dateiformaten (PNG, ZIP, ...) und Übertragungsprotokollen (Ethernet, SATA, ...) *Cyclic-Redundancy-Checks* (CRC) zum Einsatz. Statt einem *Parity-Bit* werden, z.B. beim CRC32, 32 *Parity-Bits* berechnet und der Nachricht angehängt. Die Berechnungsvorschrift ist hierbei nur unwesentlich komplizierter. Zuerst werden die Bits der zu übertragenen Nachricht als Koeffizienten einer Polynomdarstellung interpretiert. Daraus folgt:

$$x(D) = x_{N-1}D^{N-1} + ... + x_1D^1 + x_0.$$

Die *Parity-Bits* ergeben sich als Rest einer Polynomdivision mit einem vorher festgelegten CRC-Polynom:

$$p(D) = (x(D) \cdot D^{32}) \; mod \; CRC_{32}(D)$$

In binärer Schreibweise bedeutet dies, dass an die Nachricht zuerst 32 Nullen angehängt werden. Die anschließende Polynomdivision kann mit der Funktion `poly_div` ausgeführt werden. Die zu übertragende Nachricht lautet dann $\vec{y} = x_{N-1}, ..., x_0, p_{31}, ..., p_0$. Zur Überprüfung einer korrekten Übertragung muss auf Empfängerseite die Bedingung

$$y(D) \; mod \; CRC_{32}(D) = 0$$

ausgewertet werden.

3. Schreiben Sie eine Funktion, welche die *Parity-Bits* für beliebige binäre Vektoren bezüglich einer CRC32-Codierung berechnet.

4. Verwenden Sie die Funktion `simulate_noise` um unterschiedliche Fehlermuster auf so codierten Testdaten zu simulieren. Welche Arten von Fehlermustern können durch eine CRC32-Codierung erkannt werden?

**Hinweis:**
Zur Erzeugung von Testdaten können Sie beliebige Strings mit der Funktion `text_to_bitstream` in binäre Vektoren umwandeln. Das CRC-Polynom ist in der Variable `CRC32` hinterlegt.

## 5.2.2. QR-Codes: Formatinformation

Es wurden die Symbole in den Dateien *„testdaten/Aufgabe_2_1.png“*, *„testdaten/Aufgabe_2_2.png“* und *„testdaten/Aufgabe_2_3.png“* während der Übertragung gestört.

1. Decodieren Sie für jedes Symbol manuell die darin enthaltenen Formatinformationen. Halten Sie insbesondere auch die folgenden Zwischenergebnisse fest:

   - $\vec{F}_{masked}$ in Binärschreibweise (direkt aus dem Symbol)
   - $\vec{F}$ in Binärschreibweise (nach dem Entfernen der Maske)
   - $\vec{F}_{decoded}$ in Binärschreibweise (nach der Fehlerkorrektur)

- ECL $\in \{'L', 'M', 'Q', 'H'\}$
- MASK $\in [0, 7]$

Verwenden Sie hierfür die Tabellen A.1 und A.2 aus dem Anhang.

2. Bestimmen Sie für jedes Symbol die grundlegenden Codeeigenschaften. Halten Sie insbesondere die folgenden Parameter fest:
   - Version
   - Coderate
   - Erkennbare Fehler
   - Korrigierbare Fehler

   Verwenden Sie hierfür die Tabelle A.6 aus dem Anhang.

### 5.2.3. Lineare Blockcodes

In dieser Aufgabe soll die Klasse der *Hamming-Codes* untersucht werden. Bei den *Hamming-Codes* handelt es sich um eine Untermenge der linearen Blockcodes, welche die Nebenbedingungen $N = 2^m - 1,\ K = 2^m - m - 1,\ N - K = m$ erfüllen. Sie haben immer die Codedistanz 3. Die Konstruktion der Generatormatrix $G$ und der Parity-Check-Matrix $H$ erweist sich für diese Codeklasse als besonders einfach. $H$ kann durch die Auflistung aller vom Nullvektor verschiedenen binären Spaltenvektoren der Länge $m$ erzeugt werden. Da wir im Allgemeinen systematische Codes verwenden sollten die Spalten so umsortiert werden, dass die $m$ rechtsseitigsten Spalten die Einheitsmatrix bilden. Die Reihenfolge der anderen $K$ Spalten kann beliebig gewählt werden. Diese beiden Schritte werden in Abbildung 5.1 beispielhaft für $m = 3$ aufgezeigt. Die Generatormatrix $G$ kann anschließend direkt aus $H$ abgeleitet werden:

$$H = \begin{pmatrix} A_{m\times K} & I_{m\times m} \end{pmatrix}_{m\times N} \quad \Rightarrow \quad G = \begin{pmatrix} I_{K\times K} & A^T_{K\times m} \end{pmatrix}_{K\times N},$$

wobei es sich bei $I_{m\times m}$ um die Einheitsmatrix mit der Seitenlänge $m$ und bei $A_{m\times K}$ um eine binäre Matrix mit $m$ Zeilen und $K$ Spalten handelt.

1. Schreiben Sie eine Funktion, welche für ein beliebiges $m$ die dazu passenden Matrizen $G$ und $H$ bestimmt.
2. Codieren Sie einige Testnachrichten mit dem so erzeugten Code. Achten Sie darauf, dass die Codierung in *Modulo-2*-Arithmetik erfolgen muss.
3. Verwenden Sie die Funktion `simulate_noise` um unterschiedliche Fehlermuster zu simulieren. Berechnen Sie anschließend das Syndrom um aufgetretene Fehler zu erkennen. Welche Fehlermuster können erkannt werden?
4. Erzeugen Sie nun eine sinnvolle Syndromtabelle. Die Syndromtabelle ordnet jedem der $2^m$ unterschiedlichen Syndrome ein Fehlermuster zu. Die konkrete Zuordnung ist hierbei im Prinzip beliebig.
   **Hinweis:** Welche Fehlermuster können mit diesem Code sicher korrigiert werden?

$$H' = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \Rightarrow H = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Abbildung 5.1.: Erstellung der Parity-Check-Matrix am Beispiel eines ($N = 7$, $K = 4$) *Hamming-Codes*. Nach Auflistung aller möglicher Spaltenvektoren in $H'$ müssen die Spalten $1$, $2$ und $4$ verschoben werden um einen systematischen Code zu garantieren.

5. Verwenden Sie die Funktion `simulate_noise` um unterschiedliche Fehlermuster zu simulieren. Welche Fehlermuster können erkannt, bzw. korrigiert, werden?

### 5.2.4. QR-Codes: Daten

Es wurden die Symbole in den Dateien *„testdaten/Aufgabe_4_1.png“*, *„testdaten/Aufgabe_4_2.png“* und *„testdaten/Aufgabe_4_3.png“* empfangen. Verwenden Sie das Fehlersimulationstool (Abbildung 5.2) um die Fehlerkorrektureigenschaften von QR-Codes zu analysieren. Beantworten Sie insbesondere folgende Fragen:

1. Welchen Einfluss hat eine Codespreizung auf Einzelbitfehler?
2. Welchen Einfluss hat eine Codespreizung auf Bündelfehler?
3. Welchen Einfluss hat das gewählte 2D-Mapping auf Einzelbitfehler?
4. Welchen Einfluss hat das gewählte 2D-Mapping auf Bündelfehler?
5. Welche Annahmen wurden bei der Entwicklung von QR-Codes über die typischen Fehlermuster gemacht?
6. Erläutern Sie die Funktionsweise des komplexeren Sägezahn 2D-Mappings.

Sie können das Tool in der MATLAB-Umgebung mit dem Befehl `fehlersimulator(read_symbol('filename'))` starten.


Abbildung 5.2.: Das Fehlersimulationstool. Fehlerhafte Module werden durch rote Kästchen symbolisiert.

# A. QR-Code Tabellen

Tabelle A.1.: Die vier unterstützten Fehlerkorrektur-Level. Der Indikator gibt eine zwei-Bitsequenz an, welche in den Formatinformationen verwendet wird.

| Indikator | Fehlerkorrektur-Level | Korrekturkapazität |
|---|---|---|
| 01 | L | 7% |
| 00 | M | 15% |
| 11 | Q | 25% |
| 10 | H | 30% |

Tabelle A.2.: BCH Lookup-Tabelle zur Brute-Force Decodierung der Formatinformationen.

| Index | Informationsstellen | Prüfstellen | Index | Informationsstellen | Prüfstellen |
|---|---|---|---|---|---|
| 1 | 00000 | 0000000000 | 17 | 10000 | 1010011011 |
| 2 | 00001 | 0100110111 | 18 | 10001 | 1110101100 |
| 3 | 00010 | 1001101110 | 19 | 10010 | 0011110101 |
| 4 | 00011 | 1101011001 | 20 | 10011 | 0111000010 |
| 5 | 00100 | 0111101011 | 21 | 10100 | 1101110000 |
| 6 | 00101 | 0011011100 | 22 | 10101 | 1001000111 |
| 7 | 00110 | 1110000101 | 23 | 10110 | 0100011110 |
| 8 | 00111 | 1010110010 | 24 | 10111 | 0000101001 |
| 9 | 01000 | 1111010110 | 25 | 11000 | 0101001101 |
| 10 | 01001 | 1011100001 | 26 | 11001 | 0001111010 |
| 11 | 01010 | 0110111000 | 27 | 11010 | 1100100011 |
| 12 | 01011 | 0010001111 | 28 | 11011 | 1000010100 |
| 13 | 01100 | 1000111101 | 29 | 11100 | 0010100110 |
| 14 | 01101 | 1100001010 | 30 | 11101 | 0110010001 |
| 15 | 01110 | 0001010011 | 31 | 11110 | 1011001000 |
| 16 | 01111 | 0101100100 | 32 | 11111 | 1111111111 |

Tabelle A.3.: Die gültigen Typ-Indikatoren für Datensegmente.

| Indikator | Name | Beschreibung |
|---|---|---|
| 0001 | Numerisch | Ziffern |
| 0010 | Alphanumerisch | Buchstaben und Ziffern |
| 0100 | Byte | Rohdaten |
| 1000 | Kanji | Japanischer Schriftsatz |
| Spezialsymbole | | |
| 0000 | Terminator | Signalisiert das Ende der gesamten Nachricht |
| 0111 | Extended Channel Interpretation | Signalisiert eine speziellere Codierung der nachfolgenden Daten |
| 0011 | Structured Append | Verkettung mehrerer Symbole zu einer langen Nachricht |
| 0101<br>1001 | FNC1 | Datencodierung gemäß GS1 Spezifikation |

Tabelle A.4.: Anzahl der benötigten Bits für die Datensegmentlänge.

| Version | Numerisch | Alphanumerisch | Byte | Kanji |
|---|---|---|---|---|
| 1 - 9 | 10 | 9 | 8 | 8 |
| 10 - 26 | 12 | 11 | 16 | 10 |
| 27 - 40 | 14 | 13 | 16 | 12 |

Tabelle A.5.: Berechnungsvorschriften der Masken. Ist die Bedingung erfüllt, so steht in der Maske an der entsprechenden Position eine $1$, ansonsten eine $0$.

| Indikator | Vorschrift $M_{Data}(x, y)$ |
|---|---|
| 000 | $(x + y)\ mod\ 2 = 0$ |
| 001 | $y\ mod\ 2 = 0$ |
| 010 | $x\ mod\ 3 = 0$ |
| 011 | $(x + y)\ mod\ 3 = 0$ |
| 100 | $(x + y)\ mod\ 2 = 0$ |
| 101 | $(x \cdot y)\ mod\ 2 + (x \cdot y)\ mod\ 3 = 0$ |
| 110 | $((x \cdot y)\ mod\ 2 + (x \cdot y)\ mod\ 3)\ mod\ 2 = 0$ |
| 111 | $((x + y)\ mod\ 2 + (x \cdot y)\ mod\ 3)\ mod\ 2 = 0$ |

Tabelle A.6.: Versionsinformationen und Blockaufteilung.

| Version | Gesamtgröße in Codewörtern | Fehlerkorrektur-Level | Blöcke | | |
|---|---|---|---|---|---|
| | | | Informationswörter | Prüfwörter | Gesamt |
| 1 | 26 | L | 19 | 7 | 26 |
| | | M | 16 | 10 | 26 |
| | | Q | 13 | 13 | 26 |
| | | H | 9 | 17 | 26 |
| 2 | 44 | L | 34 | 10 | 44 |
| | | M | 28 | 16 | 44 |
| | | Q | 22 | 22 | 44 |
| | | H | 16 | 28 | 44 |
| 3 | 70 | L | 55 | 15 | 70 |
| | | M | 44 | 26 | 70 |
| | | Q | 17 | 18 | 35 |
| | | | 17 | 18 | 35 |
| | | H | 13 | 22 | 35 |
| | | | 13 | 22 | 35 |
| 4 | 100 | L | 80 | 20 | 100 |
| | | M | 32 | 18 | 50 |
| | | | 32 | 18 | 50 |
| | | Q | 24 | 26 | 50 |
| | | | 24 | 26 | 50 |
| | | H | 9 | 16 | 25 |
| | | | 9 | 16 | 25 |
| | | | 9 | 16 | 25 |
| | | | 9 | 16 | 25 |
| 5 | 134 | L | 108 | 26 | 134 |
| | | M | 43 | 24 | 67 |
| | | | 43 | 24 | 67 |
| | | Q | 15 | 18 | 33 |
| | | | 15 | 18 | 33 |
| | | | 16 | 18 | 34 |
| | | | 16 | 18 | 34 |
| | | H | 11 | 22 | 33 |
| | | | 11 | 22 | 33 |
| | | | 12 | 22 | 34 |
| | | | 12 | 22 | 34 |
| 6 | 172 | L | 68 | 18 | 86 |
| | | | 68 | 18 | 86 |
| | | M | 27 | 16 | 43 |
| | | | 27 | 16 | 43 |
| | | | 27 | 16 | 43 |
| | | | 27 | 16 | 43 |
| | | Q | 19 | 24 | 43 |
| | | | 19 | 24 | 43 |
| | | | 19 | 24 | 43 |
| | | | 19 | 24 | 43 |
| | | H | 15 | 28 | 43 |
| | | | 15 | 28 | 43 |
| | | | 15 | 28 | 43 |
| | | | 15 | 28 | 43 |