

Zahlendarstellung und Kodierung

- Zahlensysteme
- Darstellung negativer Zahlen
- Darstellung Fest- und Fließkommazahlen
- Kodierungen zur Zahlen- und Zeichendarstellung
- Fehlerkorrigierende Codes

Zeichenkodierung

Sollen nicht nur Zahlen, sondern auch Buchstaben, Sonderzeichen, etc. z.B. zur Textverarbeitung kodiert werden, so ist eine andere Kodierung erforderlich

26 Buchstaben

➔ **mindestens 5 Bit sind zur Kodierung erforderlich**

Heute gebräuchliche Kodierungsbreiten zur Textdarstellung: **7 - 16 Bit**

ASCII – Kodierung

Anmerkungen:

- Die höherwertigen Bits der Kodierung eines Zeichens sind in der Kopfzeile abzulesen, die niederwertigen Bits in der ersten Spalte

Beispiel: A → 100 0001₂

- 128 mögliche Zeichen → 2 * 26 Buchstaben, 10 Ziffern, 32 Kommunikationssteuer und 32 Interpunktionszeichen

Unicode (<http://www.unicode.org>)

- ❑ Firmenspezifische und nationale, nicht miteinander kompatible 7-Bit- und 8-Bit-Zeichenkodierungen bis weit in die 80-er Jahre
- ❑ Chinesische, japanische, koreanische, arabische, thailändische, bengalische, hebräische und viele andere Schriftzeichen zum großen Teil überhaupt nicht oder nur über **komplizierte Kodierungssequenzen variabler Länge** kodiert
- ➔ **Weltweit genormte Kodierung aller Zeichen ist notwendig, um den einfachen Datenaustausch möglich zu machen ➔ Unicode**

- ❑ BCD-Kodierung (engl. Binary coded decimal):
 - Kodierung der zehn Dezimalziffern durch ihr 4-Bit Dualäquivalent (Tetrade)
 - 6 der 16 möglichen Kodierungen stellen keine gültige Dezimalziffer dar, sie heißen daher **Pseudotetraden**
 - Nachteil: Suboptimale Speicherplatzausnutzung und Probleme bei der Ausführung arithmetischer Operationen
- ❑ Gray-Kodierung:
 - **Einschrittige** Kodierung
 - Vorteile bei der A/D-Wandlung und für mechanische Abtaster
 - Keine feste Stellenwertigkeit → Die Ausführung arithmetischer Operationen ist recht schwierig

ASCII – Kodierung

ASCII (American Standard Code for Information Interchange) ist die bekannteste Kodierung zur Textdarstellung. Es handelt sich um eine **7-Bit-Kodierung**

	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	'	p
0001	SOH	DC 1	!	1	A	Q	a	q
0010	STX	DC 2	"	2	B	R	b	r
0011	ETX	DC 3	#	3	C	S	c	s
0100	EOT	DC 4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	^	7	G	W	g	w
1000	BS	CAN	(8	H	X	h	x
1001	HT	EM)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[k	{
1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M]	m	~
1110	SO	RS	.	>	N	^	n	_
1111	SI	US	/	?	O	~	o	DEL

Platzbeschränkungen

Für deutsche Umlaute oder andere nationale Besonderheiten ist kein Platz:

2 Möglichkeiten:

- Nationale Varianten der ASCII-7-Bit Kodierung: z. B. ISO-7-Bit oder DIN 66003 (deutsche Variante mit Umlauten)
- Erweiterung auf eine 8 Bit Kodierung (Byte-Orientierung der Rechner): Nutzung der zusätzlichen 128 Zeichen für nationale Besonderheiten und weitere Sonderzeichen.

Weitere Möglichkeit des zusätzlichen Bits beim Übergang von 7 auf 8 Bit ist die **Fehlererkennung** (wird später behandelt)

Geschichte des Unicodes

- ❑ Notwendigkeit für eine genormte Kodierung aller Zeichen, die in der Welt geschrieben werden oder wurden
- ❑ **Unicode-Konsortium** begann Ende der 80-er Jahre mit der Entwicklung einer solchen Zeichenkodierung
- ❑ Im Konsortium sind viele Soft- und Hardwarefirmen wie Apple, IBM und Microsoft, sowie nichtamerikanische Firmen vertreten:
 - ➔ die bisherigen nationalen, internationalen und firmenspezifischen Zeichenkodierungen werden durch die Unicode-Norm abgelöst
 - Übergang wird nur sehr langsam erfolgen (Kompatibilität zu bisherigen Kodierungen)

Unicode: Entwurfsprinzipien (1)

- Unicode-Zeichenkodierungen haben sämtlich eine Länge von 16 Bits
- Der gesamte Kodeumfang von 16 Bits steht für die Zeichenkodierung zur Verfügung
- Die Unicode-Norm kodiert Zeichen, nicht Zeichendarstellungen oder Erscheinungsbild
- Zeichen haben eine wohl definierte Semantik
- Die Unicode-Norm kodiert einfachen (nicht formatierten) Text
- Unicode-Zeichen werden in logischer Reihenfolge gespeichert

2.5 Fehlerkorrigierende Codes

- Einfaches Schema zur Fehlererkennung
 - Quersumme
- Hammingcode zur Fehlerkorrektur
 - Aufbau des Hammingcodes
 - Erkennung und Korrektur von 1-Bit-Fehlern
 - Erkennung von 2-Bit-Fehlern

Beispiel: Datenwort: $x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0$
 $1\ 0\ 0\ 1\ 1\ 0\ 1\ 1$
 $\hookrightarrow x_1 \oplus x_2 \oplus \dots \oplus x_7 = 1 = x_0$
 Codewort: $x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0$
 $1\ 0\ 0\ 1\ 1\ 0\ 1\ 1$
 $\hookrightarrow x_0 \oplus x_1 \oplus \dots \oplus x_7 = 0$
 gerade Parität!
ergänzt auf gerade Anzahl von "1"

Beispiel: Codewort $x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0$
 $1\ 0\ 0\ 0\ 1\ 0\ 1\ 1$
 $\hookrightarrow x_0 \oplus x_1 \oplus \dots \oplus x_7 = 1$
 \Rightarrow Fehler
qscc: Keine Positionsinformation!

Unicode: Entwurfsprinzipien (2)

- Gleiche Zeichen werden gleichgesetzt
- Die Unicode-Norm erlaubt die dynamische Erstellung von zusammengesetzten Zeichen
- Für zusammengesetzte Zeichen mit eigener Kodierung stellt die Unicode-Norm eine Übertragung in die äquivalente Sequenz aus Grundzeichen und modifizierenden Zeichen bereit
- 1-zu-1-Umsetzung zwischen Unicode und anderen weit verbreiteten Normen ist garantiert

Fehlererkennung durch Quersummenbits

Jedem Datenwort, z. B. 8-Bit-Wort x_7, \dots, x_0 , wird ein neuntes Bit x_8 , das Prüfbit, angehängt, das sich durch Addition mod 2 oder Quersumme berechnet:

$$x_8 = x_7 \oplus \dots \oplus x_0$$

Damit wird jedes Kodewort auf gerade Parität (Quersumme) ergänzt.

Der zusätzliche Speicheraufwand ermöglicht damit eine einfache Fehlerprüfung:

$$x_8 \oplus (x_7 \oplus \dots \oplus x_0) = 0$$

Einfache Bit-Fehlererkennung

$$x_8 \oplus (x_7 \oplus \dots \oplus x_0) = 0$$

Die Gleichung wird falsch, wenn ein **einzelnes** Bit oder eine **ungerade** Zahl von Bits fehlerhaft geworden ist.

➔ Einfache Bitfehlererkennung

Der Code kann jedoch **keine** Fehler erkennen, bei denen eine **gerade** Zahl von Bits fehlerhaft ist, und ermöglicht auch keine Fehlerkorrektur.

Einfache Bit-Fehlererkennung

- Ein Bit verfälscht \rightarrow Anzahl der Einsen nicht mehr gerade

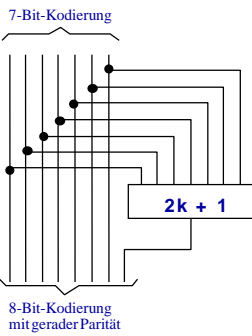
Verfälschung einer geraden Zahl von Binärstellen ist nicht erkennbar

- **Alternative:** Man kann auch ein Paritätsbit anfügen, das zu einer ungeraden Anzahl von Einsen führt (*odd parity*) und die ungerade Anzahl überprüfen

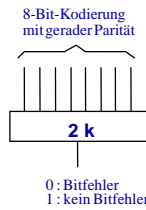
Schema einer Fehlererkennung

Beispiel: Fehlererkennung bei der Übertragung von ASCII-Zeichen

Redundante Kodierung:



Fehlererkennung:



Fehlerkorrektur durch Hammingcodes

- Fehlerkorrekturcodes können ein fehlerfreies Codewort von einem fehlerhaften Codewort unterscheiden.
- Bei fehlerhaften Codewörtern sind die entsprechenden Bits im Codewort innerhalb bestimmter Grenzen korrigierbar.
 - Mehr als ein Prüfbit pro Datenwort notwendig
- Um die Fehlerkorrektur vornehmen zu können, wird in den Korrekturbits eindeutig kodiert, ob und ggf. welche Bits im gesamten Codewort (Daten- und Korrekturbits) fehlerhaft sind

Ein-Bit-Fehlerkorrektur

Notwendiger Aufwand für die sog. **Hammingcodes**:

$$2^k \geq m + k + 1$$

- k : Zahl der zusätzlichen Korrekturbits
- m : Zahl der Datenbits

Aufbauprinzipien

Unterschiedliche Aufbauprinzipien sind möglich.

- Hier:** Einfaches Realisierungsschema
- Vorteil:** Beliebige Erweiterbarkeit auf größere Datenwortlängen

Der Ansatz verwendet k verschiedene Prüfgleichungen für die Quersummen QS_0 bis QS_{k-1}

Funktionsweise der Fehlererkennung

- 7-Bit-Kodierung wird auf eine redundante 8-Bit-Kodierung erweitert
- 8-te Bit wird durch die Quersumme (Exklusiv-Oder-Schaltung) erzeugt
- Diese erzeugt genau dann eine 1, wenn an den Eingängen eine ungerade Anzahl von Einsen anliegt (Paritätsbit)
- es wird eine 8-Bit-Kodierung erzeugt, die immer eine gerade Anzahl von Einsen aufweist (*even parity*)

Ein-Bit-Fehlerkorrektur

- Anzahl der Bits im zu übertragenden Codewort: n
- Einer der Bits kann falsch sein $\rightarrow n$ Fehlerfälle + falscher Fall
- $n+1$ Fälle
- Kodierung dieser $n+1$ Fälle mit k Korrekturbits
- $2^k \geq n+1$
- k Korrekturbits sind Teil des Codeworts
- $n = m + k$
- m : Anzahl der Datenbits

Mindestaufwand: Ein-Bit Fehlerkorrekturcodes

Zahl der Datenbits m	Zahl der Prüfbits k	Bits insgesamt: $m + k$
1	2	$m + 2$
2 bis 4	3	$m + 3$
5 bis 11	4	$m + 4$
12 bis 26	5	$m + 5$
27 bis 57	6	$m + 6$
...

• Beispiel: 7-bit Codewort $\overset{7}{x} \overset{6}{x} \overset{5}{x} \overset{4}{x} \overset{3}{x} \overset{2}{x} \overset{1}{x}$

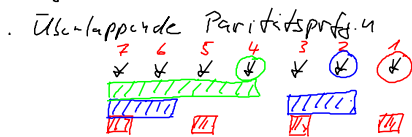
• Zahl der Datenbits 4, Zahl der Prüfbits 3

• So generiertes Syndrom aus Prüfbits $k_3 k_2 k_1$ soll Position des fehlerhaften Bits im Codewort kodieren

Fehlerposition	Syndrom
1	$k_3 k_2 k_1$
2	$0 0 1$
3	$0 1 1$
4	$1 0 0$
5	$1 0 1$
6	$1 1 0$
7	$1 1 1$

Ansatz (1)

- Erste Paritätsprüf. soll erstes Bit k_1 des Syndroms liefern
 \hookrightarrow Paritätsprüf. über Positionen 1, 3, 5, 7
- k_2 : Positionen 2, 3, 6, 7
- k_3 : " 4, 5, 6, 7



- Paritätsprüf. sollen unabhängig sein \rightarrow Prüfbits sollen sich nicht gegenseitig prüfen
 \hookrightarrow Positionen 2^i für Prüfbits k_{i+1} mit $i=0,1,2,\dots$

Die i -te Quersumme (QS_i) ergänzt alle Positionen im Kodewort, die die Potenz 2^i in ihrer Ziffernwertigkeit enthalten ($i = 0, 1, \dots, k-1$) auf gerade Parität

- $k_1 = QS_0$ über alle ungeraden Stellen 1, 3, 5, 7, ...
- $k_2 = QS_1$ über die Stellen 2, 3, 6, 7, 10, 11, ...
- $k_3 = QS_2$ über die Stellen 4, 5, 6, 7, 12, 13, 14, 15, ..
- $k_4 = QS_3$ über die Stellen 8, 9, 10, 11, 12, 13, 14, 15, ..

0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
...

Ansatz (2)

- Es werden gerade Quersummen verwendet
- Datenwörter werden zum Einfügen der Prüfbits gespreizt, indem die Kodewortposition 2^i für die i -te Quersummenprüfstelle k_{i+1} freigehalten wird (m_i bezeichnen die Datenbits)

Damit erhält man folgendes Schema:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	Position
	4								3				2		1	0	i
m_{12}	k_3	m_{11}	m_{10}	m_9	m_8	m_7	m_6	m_5	k_4	m_4	m_3	m_2	k_3	m_1	k_2	k_1	Stellen

Ansatz (3)

- Beim Lesen eines evtl. fehlerhaften Kodeworts kann durch **erneutes Bilden der Quersummen** ermittelt werden, ob und an welcher Position ein Fehler aufgetreten ist
- Die Quersummen geben dabei, als Dualzahl interpretiert, die fehlerhafte Position im Kodewort an
- Liegt kein Fehler vor, dann sind alle Prüfgleichungen erfüllt

Beispiel (1)

Hammingcode mit 4 Datenbits und 3 Prüfbits, also 7-Bit-Kodewörtern

Datenwort : 1 1 0 1
 \rightarrow $\begin{matrix} 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 1 & 0 & - & 1 & - & - \\ m_4 & m_3 & m_2 & k_3 & m_1 & k_2 & k_1 \end{matrix}$

Bestimmung der Prüfbits:

$$k_1 = m_1 \oplus m_2 \oplus m_4 = 1 \oplus 0 \oplus 1 = 0$$

$$k_2 = m_1 \oplus m_3 \oplus m_4 = 1 \oplus 1 \oplus 1 = 1$$

$$k_3 = m_2 \oplus m_3 \oplus m_4 = 0 \oplus 1 \oplus 1 = 0$$

\rightarrow **Kodewort :** 1 1 0 0 1 1 0

Beispiel (2)

Bildet man erneut die Prüfbits bei fehlerfreiem Kodewort, ergibt sich als Syndrom $k_3 k_2 k_1$ der Wert 000.

Kodewort: 1 1 0 0 1 1 0

$$k_1 = k_1 \oplus m_1 \oplus m_2 \oplus m_4 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$

$$k_2 = k_2 \oplus m_1 \oplus m_3 \oplus m_4 = 1 \oplus 1 \oplus 1 \oplus 1 = 0$$

$$k_3 = k_3 \oplus m_2 \oplus m_3 \oplus m_4 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$$

Kodewort: 1 1 0 0 0 1 0

$$k_1 = k_1 \oplus m_1 \oplus m_2 \oplus m_4 = 0 \oplus 0 \oplus 0 \oplus 1 = 1$$

$$k_2 = k_2 \oplus m_1 \oplus m_3 \oplus m_4 = 1 \oplus 0 \oplus 1 \oplus 1 = 1$$

$$k_3 = k_3 \oplus m_2 \oplus m_3 \oplus m_4 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$$

$k_3 k_2 k_1 = 011_2 = 3_{10}$ Fehler an 3. Position

Beispiel (3)

Ist ein Prüfbit verfälscht, z. B. k_3 , wird nur die betroffene Prüfgleichung beeinflusst, hier QS_2

Kodewort: $\begin{matrix} m_4 & m_3 & m_2 & k_3 & m_1 & k_2 & k_1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{matrix}$

$$k_1 = k_1 \oplus m_1 \oplus m_2 \oplus m_4 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$

$$k_2 = k_2 \oplus m_1 \oplus m_3 \oplus m_4 = 1 \oplus 1 \oplus 1 \oplus 1 = 0$$

$$k_3 = k_3 \oplus m_2 \oplus m_3 \oplus m_4 = 1 \oplus 0 \oplus 1 \oplus 1 = 1$$

$k_3 k_2 k_1 = 100_2 = 4_{10}$ Fehler an 4. Position

Hammingcode: 2-Bitfehler (1)

- Die Betrachtungen gelten genau dann, wenn garantiert werden kann, dass tatsächlich **nur 1-Bitfehler** auftreten
- Der Hammingcode kann sehr wohl mehrfache Bitfehler erkennen, aber nur 1-Bitfehler korrigieren

Voraussetzung: Nur 1- oder 2-Bitfehler treten auf !

- Im Fall von 2-Bitfehlern werden diese vom Hammingcode erkannt, können aber nicht von 1-Bitfehlern unterschieden werden:

Bei dem Syndrom $k_3 k_2 k_1 = 0101$ kann sowohl einen 1-Bitfehler an der 5. Position als auch einen 2-Bitfehler an zwei unbekannt Positionen vorliegen.

Hammingkode : 2-Bitfehler (2)

- Um 1-Bitfehler von 2-Bitfehlern unterscheiden zu können, fügt man dem Hammingkode ein weiteres Paritätsbit hinzu

1-Bitfehler:

- Erkennbar durch das Paritätsbit
- Erkennbar und korrigierbar durch den Hammingkode

2-Bitfehler:

- Nicht erkennbar durch das Paritätsbit
- Erkennbar durch den Hammingkode, **aber** nicht korrigierbar, da die fehlerhaften Positionen nicht ermittelt werden können

n-Bitfehler (n>2): andere Verfahren



Hammingkode: Zusammenfassung

- An Standard-Codewörter lässt sich ein weiteres Quersummenbit anfügen, so dass Zweibitfehler im Codewort erkennbar werden.
- 32-Bit Wortlänge werden durch 7 Prüfbits zur Einzelfehlerkorrektur und Doppelfehlererkennung ergänzt.
- 64 Bit lange Datenwörter erfordern 8 Korrekturbits.
- Die Algebra endlicher Körper bietet über Generator- und Nullmatrix Verfahren, die zu jedem Datenwort ein Codewort erzeugen.



Hammingkode: Beispiel 2-Bitfehler

- Fall 1: Kein Fehler

$$\begin{array}{cccccccc} m_5 & k_4 & m_4 & m_3 & m_2 & k_3 & m_1 & k_2 & k_1 & PB \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{array} \quad k_4 k_3 k_2 k_1 = 0000$$

- Fall 2: 1-Bit-Fehler

$$0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \quad k_4 k_3 k_2 k_1 = 0110$$

Fehler an 6. Position

- Fall 3: 2-Bitfehler

$$0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \quad k_4 k_3 k_2 k_1 = 0101$$

2-Bit-Fehler

