Tutoriumsblatt 3 mit Musterlösung

Aufgabe 3.1: Abtasttheorem

Mit der Digitalisierung des Telefonnetzes (durch ISDN) wurde festgelegt, dass zur digitalen Telefonie ein Kanal mit 64 kBit/s benötigt wird.

a) Auf welchen Annahmen basiert die Bitrate von 64 kBit/s?

b) Hängt diese Bitrate auch mit dem verwendeten Medium (Twisted Pair) zusammen?

Lösung 3.1

1.a) Siehe Folien: menschliche Sprache benötigt nur eine Bandbreite von 3.1 kHz, von 300 Hz bis zu 3.4 kHz. Da man nach dem Abtasttheorem mindestens mit der doppelten der höchsten benötigten Frequenz abtasten muss, ergäbe sich die Notwendigkeit, 6800 Abtastungen des Sprachsignals pro Sekunde vorzunehmen. Dies ist allerdings nur pure Theorie... da unsere Hardware nicht perfekt ist, muss man etwas häufiger abtasten (genauer Wert von der Redaktion vergessen, so um die 2.2), so dass man aufgerundet auf 8000 Abtastungen pro Sekunde kommt. Die Aufrundung auf exakt 8000 hat man wohl einfach nur deshalb gemacht, weil es so eine schöne Runde Zahl ist. Also mehr oder weniger reine Willkür.

Die Festlegung von 256 Quantisierungsintervallen für Sprache ist auch mehr oder weniger willkürlich. Es wird angenommen, dass diese Anzahl zur vernünftigen Rekonstruktion des Signals ausreicht. Vielleicht würde es sogar ein Bit weniger tun, aber der Mensch rechnet gerne in kompletten Bytes. ;-)

Wir tasten also 8000 mal pro Sekunde 8 Bit ab, was 64 kBit/s ergibt.

1.b) Man kann hier sagen, in gewisser Weise ja - zwar nicht in der Berechnung selbst (dort werden nur die eigentlich wichtigen Faktoren der menschlichen Sprache und Wahrnehmung betrachtet), aber durch das Shannon-Theorem vorgegeben. Die Telefonleitung hin zur Zentrale kann Kilometer lang sein, und es treten Dämpfung und Störeinflüsse auf. Je höher die verwendeten Frequenzen, desto stärker ist die Dämpfung, so dass man auf einem Kupferkabel auch einen umgekehrt proportionalen Zusammenhang zwischen nutzbarer Bandbreite und Entfernung hat! Da die einfache TP-Telefonverkabelung auch sehr störanfällig ist, wird zudem auch in den niedrigen Frequenzbereichen das Signal stark verzerrt. Daher ist – je nach Entfernung zwischen der eigenen Wohnung und der Zentrale – keine höhere Datenrate möglich!

Dies ist auch die Beschränkung für DSL. DSL erreicht höhere Datenraten einerseits durch Verwendung höherwertiger QAM-Verfahren, andererseits durch Nutzung massiv größerer Bandbreiten. Ersteres erhöht die Fehleranfälligkeit gegenüber Störungen, letzteres wird von der Entfernung begrenzt... was der Grund ist, warum die erreichbare Datenrate bei DSL immer vom Wohnort abhängt, bzw. vom Ausbaustatus des Glasfaserkabelnetzes, durch das die Länge der alten Kupferleitungen deutlich reduziert werden kann.

Aufgabe 3.2: Code Division Multiplexing (CDM)

In modernen Mobilfunknetzen wird teilweise Code Division Multiplex (CDM) als Mehrfachzugriffsverfahren eingesetzt. Bei CDM wird jedes Datenbit $x \in \{0, 1\}$ zur Übertragung mittels eines Spreizcodes $a = a_0 a_1 \cdots a_{m-1}$ mit $a_i \in \{-1, +1\}$ in eine Signalfolge übersetzt, wobei gilt:

x	gesendete Bitfolge
0	$a = a_0 a_1 \cdots a_{m-1}$
1	$-a = -a_0 - a_1 \cdots - a_{m-1}$

Es gibt also pro Spreizcode genau zwei Codeworte.

Damit mehrere Sender gleichzeitig senden können, ohne die Übertragung der anderen Sender zu stören, ist es notwendig, dass die verwendeten Codes orthogonal sind. Zwei Codes a mit $a_i \in \{-1, +1\}$ und b mit $b_i \in \{-1, +1\}$ sind orthogonal, wenn ihr Skalarprodukt 0 ergibt, d.h. wenn gilt:

$$\sum_{i=0}^{m-1} a_i * b_i = 0$$

Betrachten Sie in dieser Aufgabe Spreizcodes der Länge $m = 4$.

a) Wie viele verschiedene Spreizcodes sind bei dieser Länge möglich?
Sind alle Spreizcodes orthogonal zueinander?

b) Geben Sie alle zu $a = 1, 1, 1, 1$ orthogonalen Spreizcodes an.

Lösung 3.2

a) Bei einer Länge von $m = 4$ gibt es $2^m = 16$ verschiedene Spreizcodes.
Es sind aber nicht alle Spreizcodes orthogonal, z.b.: $a = 1, 1, 1, 1$ und $b = -1, -1, -1, -1$, hier gilt: $a * b = -4 \neq 0$.

b) Es gibt 6 Codes die orthogonal zu a sind:
(1, 1, -1, -1)
(1, -1, 1, -1)
(1, -1, -1, 1)
(-1, 1, 1, -1)
(-1, 1, -1, 1)
(-1, -1, 1, 1)

Aufgabe 3.3: Bit-Stuffing

Führen Sie anhand des folgenden Beispiels Bit-Stuffing durch:

- Die zu übertragende Bitfolge sei `1011 0001 0011 0110 0110 0101 1101 01`.
- Das Flag zur Kennzeichnung von Start und Ende der Übertragung sei `1001101`.
- Bit Stuffing soll erfolgen, indem eine `0` eingefügt wird.

Lösung 3.3

Bit Stuffing: Verhindere durch das Einfügen eines weiteren Bits in die Datenfolge, dass das Flag innerhalb der Bitfolge vorkommt. Im vorliegenden Fall: Füge in jede Sequenz `1001101`, die in der Datenfolge vorkommt, eine `0` ein.

Und wo fügen wir diese `0` nun ein?

Idealerweise sollte man die `0` so weit hinten wie möglich einfügen, um den Overhead zu reduzieren: je länger die erlaubten Folgen sind, desto geringer wird die Wahrscheinlichkeit, dass sie vorkommen, und desto seltener muss man Bitstuffing vornehmen. Als Beispiel kann man einfach mal nur die ersten drei Bit hernehmen und sagen: „verhindern wir doch mal, dass `100` auftaucht." Dann müsste man immer nach `10` eine `1` einfügen, um zu verhindern, dass das Muster auftauchen kann. Also seeehr oft. Dann doch lieber weiter hinten etwas einfügen, also mehr Bits nehmen.

Also ist es am besten, das Bit erst so weit hinten wie möglich einzufügen: `100110`$\boxed{\texttt{0}}$.

Damit ergibt sich die Folge: `1011 0001 0011 0`$\boxed{\texttt{0}}$`11 0011 0`$\boxed{\texttt{0}}$` 01 0111 0101`. Man ersetzt also nur sehr selten.

Bei der zweiten Ersetzung könnte man fragen, warum überhaupt eine `0` einfügen muss, dort war doch danach schon eine, so dass das Flag gar nicht vorkam. Nun ja, das weiß der Empfänger aber nicht – nach Codierungsregeln wird der Empfänger nach `100110` eine `0` entfernen, daher muss man auch eine einfügen.

Aufgabe 3.4: CRC

a) Gegeben ist die Bitsequenz `1011 1001`, die mittels CRC gesichert übertragen werden soll. Berechnen Sie die Prüfsumme unter Verwendung des Generatorpolynoms $G(x) = x^4 + x + 1$.

b) Warum hat sich CRC zur Fehlerprüfung in der Datenkommunikation durchgesetzt? Welche Vorteile bietet sie verglichen mit z.B. der Verwendung eines Paritätsbits?

c) Kann man mittels einer CRC auch Fehler korrigieren?

d) Warum wird die berechnete Prüfsumme nicht wie alle anderen Kontrollinformationen mit im Header übertragen, sondern an die Nutzdaten angehängt?

Lösung 3.4

4.a) Bitsequenz `1011 1001`; Generatorpolynom $G(x) = x^4 + x + 1$, d.h. 10011

```
1 0 1 1 1 0 0 1 0 0 0 0
1 0 0 1 1
- - - - -
      1 0 0 0 0
      1 0 0 1 1
      - - - - -
            1 1 1 0 0
            1 0 0 1 1
            - - - - -
              1 1 1 1 0
              1 0 0 1 1
              - - - - -
                1 1 0 1 0
                1 0 0 1 1
                - - - - -
                  1 0 0 1
```

Damit ist der CRC 1001.

Zu übertragende Bitfolge: `1011 1001'1001`

Das Ergebnis der Division braucht man gar nicht mitzuhalten - nur der Rest ist relevant.

4.b) CRC ist sehr leistungsfähig. Sie ist in der Lage, die meisten Fehlerbursts zu erkennen. Hat das Prüfpolynom den Grad n, lassen sich zuverlässig alle Fehlerbursts bis zur Länge n erkennen. Auch längere Fehlerbursts können noch zu einem großen Prozentsatz erkannt werden. Lasst hier eure Teilnehmer diskutieren, was denn nicht erkannt wird – da eine Übertragung als korrekt erkannt wird, wenn die erhaltene Bitfolge ein Vielfaches des Generatorpolynoms ist, lassen sich also nur die Fehlermuster nicht erkennen, die einem Vielfachen des Generatorpolynoms entsprechen. Das Parity Bit erkennt nur ungerade Anzahlen an falschen Bits. Daduch gibt es sehr viel mehr Fehlermuster als bei CRC, die nicht erkannt werden können.

4.c) Als Antwort sagt man sicher erst einmal 'nein'. Wir haben ja nur den Rest, aber nicht wie bei Kreuzparität noch eine zweite Prüfsumme, mit der wir den Fehler eingrenzen können...

Aber trotzdem kann man unter bestimmten Bedingungen Fehler korrigieren.

Im Allgemeinen wird CRC nur zur Fehlererkennung eingesetzt, aber in ATM gab es eine CRC zur Fehlerkorrektur: vier Byte Header wurden mit einem Byte CRC geschützt. Das Generatorpolynom wurde so geschickt gewählt, dass jeder der 40 möglichen Einzelbitfehler in Header + CRC zu einem

anderen Rest führte. Der Wert des Rests identifiziert hier also die Fehlerposition. Das funktioniert allerdings nur bei Einzelfehlern und der Overhead ist hier sehr hoch (1 Byte CRC für 4 Byte Header). Daher lohnt es sich im Allgemeinen nicht, ein Hamming-Code wäre effizienter.

4.d) Der Grund ist Effizienz – wir können unser Generatorpolynom einfach mittels Schieberegistern in Hardware implementieren, unsere Bitsequenz durchjagen und gleichzeitig aufs Kabel setzen; die berechnete Prüfsumme können wir direkt im Anschluss hinterherschicken. Ebenso kann der Empfänger on-the-fly einen empfangenen Bitstrom durch seine Schieberegister jagen und die Korrektheit der Bitfolge direkt beurteilen.

Aufgabe 3.5: Fehlerkorrigierende Codes

Sie wollen Nachrichten übertragen, welche nur aus den Zeichen „A“, „B“, „C“ und „D“ bestehen können. Gegeben sei die folgende Codierungsvorschrift, die regelt, welche Bitfolgen zur Repräsentation der jeweiligen Zeichen übertragen werden:

- „A“: `000000`
- „B“: `111000`
- „C“: `000111`
- „D“: `111111`

a) Sie empfangen nun die im Folgenden dargestellten Bitfolgen. Welche Zeichen werden jeweils dekodiert? Lässt sich dies in jedem Fall eindeutig entscheiden?

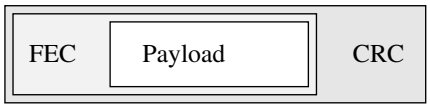
   – `100000`
   – `001111`
   – `101111`
   – `101010`

b) Wie viele Bitfehler lassen sich mit der gegebenen Codierung erkennen?

c) Wie viele Bitfehler lassen sich mit der gegebenen Codierung korrigieren?

d) Sie wollen nun fehlererkennende (CRC) und fehlerkorrigierende (FEC) Codes gemeinsam verwenden, um Fehler auf jeden Fall zu erkennen und soweit möglich auch korrigieren zu können. Dazu können Sie zwischen drei Szenarien wählen:

Szenario 1



Szenario 2

Szenario 3

Im ersten Szenario wird zunächst die CRC-Prüfsumme zu den Nutzdaten (Payload) berechnet, anschließend werden Prüfbits zur Fehlerkorrektur über Payload und CRC-Prüfsumme berechnet.

Im zweiten Szenario geht man genau umgekehrt vor: erst werden Korrekturbits berechnet, danach erfolgt über Payload und Korrekturbits eine Berechnung der CRC-Prüfsumme.

Im dritten Szenario werden sowohl CRC- als auch FEC-Prüfbits nur über den Payload berechnet.

Welches dieser Szenarien halten Sie für das Sinnvollste?

Lösung 3.5

5.a)

- `100000`: „A“ – nur ein Bit Abweichung vom Code.
- `001111`: „C“ – nur ein Bit Abweichung vom Code.
- `101111`: „D“ – nur ein Bit Abweichung vom Code.
- `101010`: „B“ – zwei Bit Abweichung vom Code.

In jedem der Fälle hat die empfangene Bitfolge zu einem der Codewörter einen geringeren Hamming-Abstand als zu allen anderen, also korrigieren wir dementsprechend. Ob es korrekt ist, ist eine andere Frage, man nimmt einfach an, dass Bitfehler nicht besonders häufig sind.

5.b) Der Hamming-Abstand ist $D = 3$. (Es müssen paarweise alle Zeichen miteinander verglichen werden; der minimale Hamming-Abstand ist der Hamming-Abstand des gesamten Codes.) Daher können zwei Fehler erkannt werden.

5.c) Ebenso: der Hamming-Abstand sagt uns, dass nur 1-Bit-Fehler korrigiert werden können. Darstellen lässt sich das Ganze anhand von „Sphären“ – zeichne das korrekte Wort in die Mitte und all die Bitfolgen, die sich in einer Position davon unterscheiden, in die gleiche Sphäre.

Nimm jetzt eine Bitfolge her, die sich in zwei Positionen von einem erlaubten Wort unterscheidet – z.B. „110000“ als Verfälschung von „A“... schade, das haben wir bereits bei „B“ in die Sphäre gepackt. Der 2-Bit-Fehler wird nicht erkannt bzw. als 1-Bit-Fehler aufgefasst und falsch korrigiert.

Teil a) verwirrt hier natürlich: ja, wir können eventuell auch 2-Bit-Fehler korrigieren. Aber auch nur, falls das entstandene Wort nicht zufällig nur 1 Bit von einem korrekten Wort entfernt ist – dass in einem Fall auch ein 2-Bit-Fehler korrigiert werden kann, liegt nur daran, dass der Coderaum nicht vollständig ausgenutzt ist und es Bitmuster gibt, die nicht in der Sphäre eines erlaubten Wortes liegen. Also nicht durch Sonderfälle beeindrucken lassen!

5.d) Das zweite Szenario hätte in der Praxis den großen Nachteil, dass bei erkanntem CRC-Fehler FEC und Payload einfach verworfen werden. Es wird nicht mehr versucht, irgendwas zu korrigieren. Das ist reine Implementierungsache und ließe sich natürlich auch anders implementieren – dann könnte man sagen, dass man erst erkennen und dann korrigieren sollte. Aber: wenn man keinen Fehler erkennt, muss man trotzdem die Korrektur anwenden, da CRC nicht alle Fehler erkennen kann. Wenn man einen Fehler mit CRC erkennt, weiß man nicht, ob die FEC korrekt funktionieren wird oder ob zu viele Fehler aufgetreten sind. Man muss sie einfach nur anwenden. Damit kann man die CRC auch ganz weglassen, da man keine Information mehr aus ihr zieht – FEC muss auf jeden Fall angewendet werden, unabhängig vom Ergebnis der CRC.

Also wären Szenario 1 oder 3 sinnvoller. Hierbei bietet Szenario 1 den großen Vorteil, dass auch Fehler in der CRC-Prüfsumme korrigiert werden können, bevor der Payload auf Fehler überprüft wird. In Szenario 3 könnten nur Fehler im Payload korrigiert werden, aber Fehler in der CRC sind genauso wahrscheinlich und können zum Verwerfen eines korrekten Pakets führen. (Man könnte aber auch argumentieren, dass Szenario 3 besser ist, da bei gleicher Anzahl an FEC-Bits weniger Bits geschützt werden und damit mehr Fehler im Nutzdatenteil korrigiert werden können. Oder dass man mit kleinerer FEC-Prüfsumme auskommt und daher weniger Overhead erzeugt.)

Letztlich ist Szenario 1 meines Wissens die gebräuchliche Variante, z.B. bei WLAN: die CRC wird auf Schicht 2 hinzugefügt, FEC erfolgt hier zusammen mit der Codierung auf Layer 1.