Die Sicherungsschicht ("Data Link Layer")

Die Sicherungsschicht ist die Schicht 2 (Layer 2) im OSI-Referenzmodell. ~~sowie im TCP/IP-Modell.~~

OSI-Referenzmodell

Anwendungsschicht	(Layer 7)
Darstellungsschicht	(Layer 6)
Sitzungsschicht	(Layer 5)
Transportschicht	(Layer 4)
Vermittlungsschicht	(Layer 3)
Sicherungsschicht	(Layer 2)
Bitübertragungsschicht	(Layer 1)

TCP/IP-Modell

Anwendung
Transport
Internet#
Host-zu-Netz

Beim TCP/IP-Modell kommt die Sicherung nicht gesondert vor, sie ist Teil der "Host-zu-Netz"-Schicht.

Beide Protokolle haben ihre Vor- und Nachteile. Da man besonders auf Vorteile nicht verzichten möchte, wird meist eine "Hybridform" (also eine Mischform) aus beiden Protokollen benutzt. Diese sieht dann so aus:

(5)	Anwendungsschicht
(4)	Transportschicht
(3)	Vermittlungsschicht
(2)	Sicherungsschicht
(1)	Bitübertragungsschicht

Hauptaufgabe der Sicherungsschicht

Die Aufgabe der Sicherungsschicht ist die gesicherte Übertragung von Daten.
Vom Sender werden hierzu die Daten in Rahmen ("frames") aufgeteilt und sequentiell (also nacheinander) an den Empfänger gesendet.
Wenn der Empfänger ~~die~~ einen gesendeten Rahmen erhalten hat, schickt er an den Sender eine Bestätigung (einen "Bestätigungsrahmen").

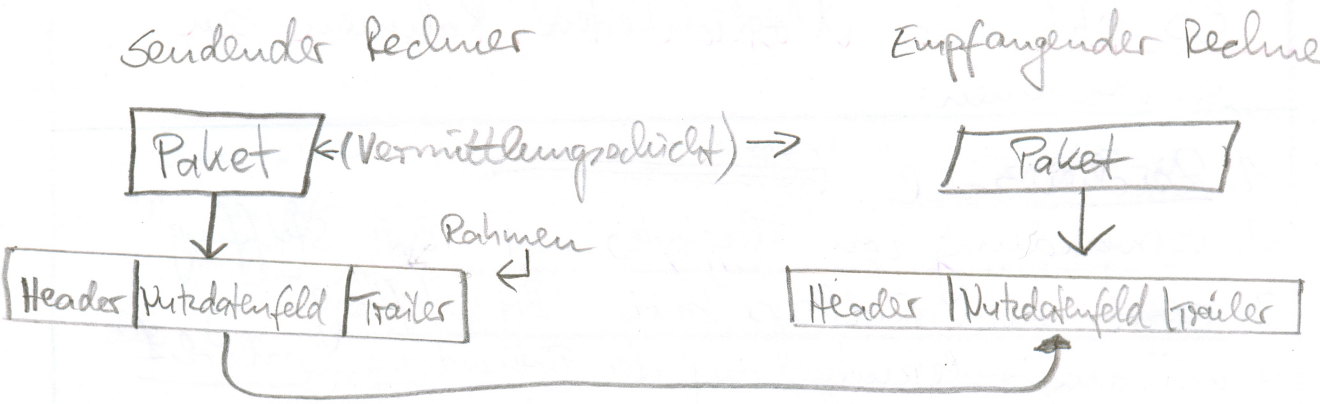
(Buch S. 55)

Die Sicherungsschicht befindet sich über der Bitübertragungsschicht (Layer 1) und unter der Vermittlungsschicht (Layer 3).
Jede Schicht bietet der Schicht über ihr ihren Dienst an, während sie Dienste der unteren Schicht in Anspruch nimmt.

Die Sicherungsschicht bietet also der Vermittlungsschicht ihre Dienste an und nimmt die Dienste der Bitübertragungsschicht in Anspruch.

Sie erhält von der Vermittlungsschicht Daten in Form von Paketen.

Aus diesen Paketen formt sie nun Rahmen, in denen zusätzliche Informationen stehen.



Doch was für zusätzliche Infos sollen denn jetzt in diesen Rahmen ("frames") stehen?

Zum Beispiel:
Es ist ja die Aufgabe der Sicherungsschicht, eine fehlerfreie Übertragung ~~zu gest~~ von 1 nach ~~2~~ 3 zu gestalten.
Deswegen wird das zu übertragene Nutzdatenfeld mit sogenannten "Redundanzbits" aufgefüllt, die die Prüfsumme bilden.
Wenn diese Prüfsumme, die beim Empfänger neu berechnet wird, mit der vom Sender übereinstimmt, dann war die Übertragung fehlerfrei.
Andernfalls schickt der Empfänger eine negative Bestätigung an den Sender, damit der weiß, dass er diesen Rahmen nochmal schicken muss...
Aber eins nach dem anderen...

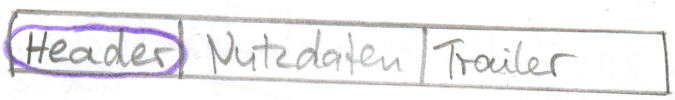
Das erste Problem, das sich ergibt, ist natürlich die Frage, wie überhaupt ein Rahmen entsteht. Die Bitübertragungsschicht kann ja lediglich einen Bitstrom übertragen (also nur Nullen und Einsen).

Es gibt vier Möglichkeiten, Rahmen zu kennzeichnen:

1. Zeichenzahl (~~Character oriented~~)
2. Verwendung von "Flagbytes" mit "Byte Stuffing"
3. Start- und Endflags mit "Bit-Stuffing"
4. Kodierungsverletzungen auf der Bitübertragungsschicht

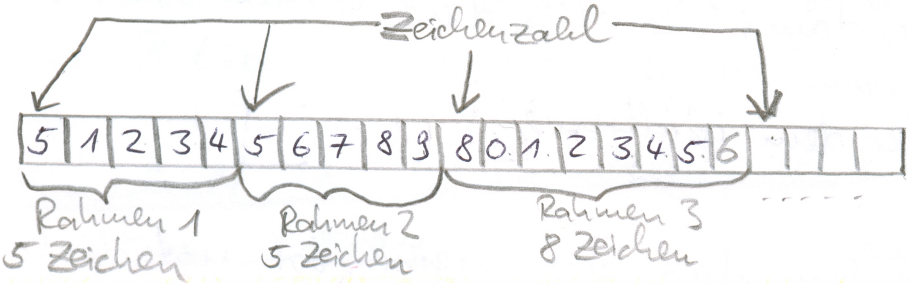
(Buch S. 215)

So sah nochmal der Rahmen aus:



In einem Rahmen ist also ein sogenannter "Header" mit Informationen. Dieser ist zum Beispiel bei der ersten Variante der Rahmenbildung nützlich.

Im Header steht in einem Feld, wie groß der ~~erste~~ Rahmen ist. Also wie viele Felder dieser Rahmen hat. Der Empfänger weiß bei einer 5, dass der Rahmen 5 Felder hat und danach logischer Weise der nächste Rahmen beginnt (und auch dort steht dann, wie groß dieser Rahmen ist usw.).



Das ist natürlich echt eine gute Idee.

ABER: Es kann bei der Übertragung ja vorkommen, dass, sagen wir mal in Rahmen 2 zu Beginn aus der 5 durch einen Übertragungsfehler eine 7 wird. Somit hätten wir dann eine falsche Rahmenlänge - nicht gut...

Verwendung von "Flagbytes" mit "Bytestuffing"

Kommen wir zur zweiten Variante:

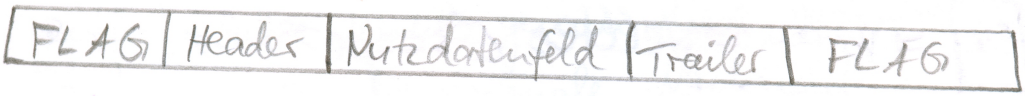
Das Problem eben (bei der Zeichenzahlvariante) war ja, dass der Empfänger bei einem Übertragungsfehler das Ende bzw. den Anfang des Rahmens nicht finden kann, also die Grenzen nicht finden kann und somit ~~dem~~ den Sender nicht darüber informieren kann, ~~wie~~ wie viele Zeichen ausgelassen werden müssen, ~~die~~ um ~~die~~ den Anfang der Wiederholungsübertragung zu finden.

Dieses Problem wird ~~es~~ behoben, ~~dass~~ in dem an den Anfang und an das Ende ~~des~~ eines Rahmens sozusagen eine "Fahne", also eine "Flag", gesetzt wird. Die erste Flag bedeutet "Der Rahmen beginnt hier" und die zweite Flag bedeutet "Der Rahmen endet hier".

Die "Flag" hat die Bitfolge 01111110

Bei dem Erhalt dieser Bitfolge weiß der Empfänger also: "Jetzt geht der Rahmen los". Sobald er wieder darauf stößt, weiß er: "Das war's".

Die Übertragung eines Rahmens sieht also so aus:

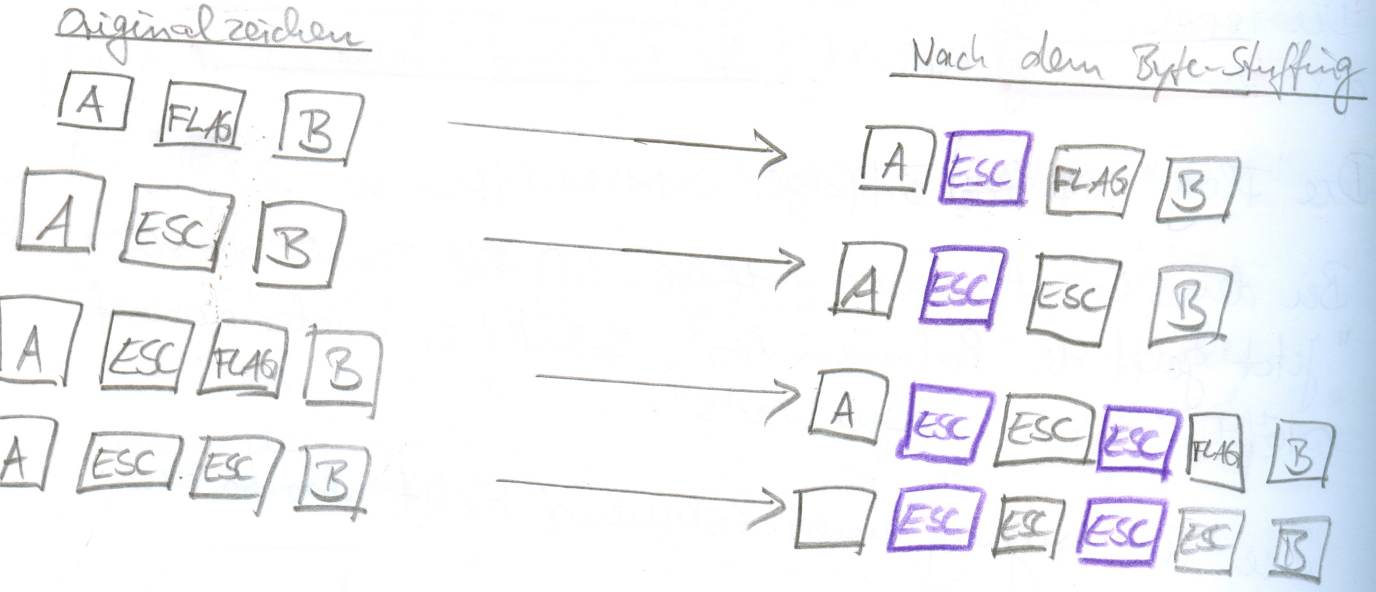


Eine berechtigte ~~Frage~~ ist natürlich:

Was ist, wenn in den Nutzdaten die Kombination 01111110 auftaucht? Dann denkt doch der Empfänger, dass der Rahmen zu Ende ist, oder?

Antwort: Genau so ist es. Da muss also eine Lösung her.

Die wurde gefunden, in dem Folgendes gemacht wird:
Die Lösung ist, dass die Sicherungsschicht des Absenders ein spezielles ~~Espa~~ Escape-Byte (ESC) vor jedem "zufälligen" Flagbyte in den Daten einfügt.
Die Sicherungsschicht auf der Empfangsseite nimmt das Escape-Zeichen wieder raus, bevor sie die Daten an die Vermittlungsschicht (Layer 3) weiterschickt.
Auch ein "zufällig" auftretendes Escapezeichen in den Nutzdaten wird mit vorangestelltem Escapezeichen versehen.
Hier ein paar Beispiele:



Dieses "Byte-Stuffing" (also Byte-stopfen) hat einen kleinen Nachteil: Es ist auf Rahmen einer Bytegröße reduziert. Es gibt aber auch Codes, die mehr als nur ein Byte (also 8 Bit) lang sind. Daher suchte man nach einer anderen Methode, um Zeichen in beliebiger Länge zu unterstützen.
Das "Bit-Stuffing" war geboren...

Start- und Endflag mit "Bitstuffing"

Hier gibt es nur eine kleine Änderung.

Der Rahmen wird, genau wie beim "Byte-Stuffing", von den Flag eingegrenzt.
Das Flag hat natürlich auch hier die Zeichenfolge 01111110.

Wenn jetzt genau diese Folge in den Nutzdaten vorkommt, hätten wir das gleiche Problem wie beim Byte-Stuffing. Auch hier wird jetzt gestopft.
Wir sehen eine Folge von sechs Einsen, eingegrenzt von zwei Nullen.

Die Sicherungsschicht macht Folgendes:
Immer, wenn die Gefahr besteht, das wir sechs Einsen in Folge haben (was ja hieße, dass der Rahmen als beendet interpretiert werden würde), wird zwischen dem 5. und dem 6. Bit eine 0 "gestopft".
Egal, ob nacher wirklich eine 1 kommt oder nicht.

Im Klartext:
Nach 5 aufeinander folgenden Einsen wird eine 0 gestopft, dann geht es weiter.
Die Sicherungsschicht beim Empfänger weiß, dass nach 5 aufeinander folgenden Einsen eine 0 gestopft wurde, entfernt also jeweils genau diese gestopfte 0 und hat somit wieder die originale Nachricht.

Beispiel:

Originale Nutzdaten:	0110111111111111111110010
Gestopfte Daten	01101111101111101111010010
Daten nach Entfernen der Füllbits	0110111111111111111110010

Kodierungsverletzungen auf der Bitübertragungsschicht

Hierzu heißt es im Buch (S. 218):

> Die letzte Methode der Rahmenbildung kann nur ~~[illegible]~~ in Netzen angewendet werden, in denen die Kodierung im physikalischen Medium mit Redundanz verbunden ist. Einige LANs kodieren beispielsweise ein Datenbit unter Verwendung von zwei physikalischen Bits.

Was soll das jetzt heißen?
Redundanz ließe sich ja übersetzen mit "Weitschweifigkeit".
Man überträgt also mehr als man übertragen muss.
Hier heißt es, das ein Datenbit mit zwei physikalischen Bits kodiert wird.
Hier ist die Regel (meistens ist es wie folgt definiert):
Eine 1 wird durch die zwei physikalischen Bits als 1-0-Folge kodiert.

Eine 0 als 0-1-Folge.
(Auch "Hoch/Niedrig"- bzw. "Niedrig/Hoch"-Paar genannt).

Nochmal tabellarisch:

Bitzeichen	Codierung
1	10
0	01
Rahmengrenze	00
Rahmengrenze	11

} Die nicht definierten Paare, also 00 bzw. 11 sind eine "Kodierungsverletzung" und werden in einigen Protokollen als Rahmenbegrenzung benutzt.

Dieses Verfahren nennt man auch "Manchesterkodierung"

Fehlererkennung und Fehlerkorrektur

Wie gesagt (besser geschrieben ☺): Bei der Übertragung der Rahmen können Fehler auftreten.
Diese Fehler müssen natürlich irgendwie erkannt werden.
Zum Erkennen von Fehlern gibt es sogenannte "Fehlererkennungscodes".

Zum korrigieren dieser Fehler gibt es sogenannte "Fehlerkorrekturcodes".

Wann wird was eingesetzt?
Erst einmal sollte klar sein, dass es größeren Umstand macht, Fehler zu korrigieren als sie lediglich zu erkennen. Wenn wir uns das verinnerlicht haben, dann dürfte auch klar sein, dass für die Fehlerbehebung mehr "Redundanzbits" mitgesendet werden müssen als zur Fehlererkennung.
Wenn ein Fehler lediglich erkannt wird, dann muss der fehlerhafte Rahmen neu gesendet werden (was nicht notwendig ist, wenn der Fehler sofort korrigiert wird).

Auf die Frage, wann welcher Code sinnvoll wäre, gibt das Buch sehr verständlich eine Antwort:

> Einige Kanäle wie Glasfaser sind extrem zuverlässig; daher ist es günstiger, einen Fehlererkennungscode zu verwenden, und die gelegentlich fehlerhaften Blöcke erneut zu übertragen.

> Bei Kanälen wie Funkverbindungen treten viele Fehler auf; daher ist es besser, jeden Block mit ausreichend Redundanz auszustatten, damit der Empfänger ermitteln kann, wie der ursprüngliche Block ausgesehen hat, anstatt sich nur auf eine ~~ebenso~~ eventuell ebenso fehlerhafte Neuübertragung zu verlassen.

Buch, S. 221

Hamming-Distanz

Zum Verständnis der Fehlerbehandlung brauchen wir jetzt erstmal ein bisschen Theorie:

Die Nachricht:

Eine Nachricht besteht aus m Datenbits und aus r Redundanz- bzw. Prüfbits (m steht für die "message" selbst und r natürlich für "redundance"). Die insgesamte Bitlänge ist n = m+r.

Eine Einheit mit n Daten- und Prüfbits wird auch n-Bit-Codewort genannt.

Der "Code" ist die Gesamtheit aller einzelnen Codewörter. Nehmen wir an, ein Code hat ~~besteht~~ zwei Codewörter (also zwei verschiedene Zeichen). Sagen wir Zeichen A und B, wobei Zeichen A als 10001001-Folge und B als 10110001-Folge dargestellt wird.

Zeichen	Codewort
A	10001001
B	10110001

Jetzt schauen wir mal, an wie vielen Stellen wir unterschiedliche Zahlen haben. Dazu schreiben wir die beiden Codewörter untereinander. Stellen, die sich unterscheiden, hebe ich ~~gelb~~ rot vor, Stellen, die gleich sind, grün!

A	1	0	0	0	1	0	0	1
B	1	0	1	1	0	0	0	1

Die Anzahl der unterschiedlichen Stellen ist der Hammingabstand dieses Codes.

Bei mehreren Codewörtern schaut man sich den geringsten Unterschied an. Das ist dann der Hamming-Abstand!

Wir haben in unserem Beispiel also einen Hammingabstand von 3.
Was sagt uns das jetzt?
Folgendes: Es müssen 3 Einzelbitfehler vorkommen, um aus dem einen Codewort das andere zu machen (logisch, wenn A, also die Folge 10001001 an drei Stellen, nämlich an der 3., 4. und 5. Stelle, verfälscht wird, haben wir 10110001, und das ist das Zeichen für B).

Wenn in unserem Code nur 2 Fehler vorkommen, dann könnte man jederzeit feststellen, dass ein Fehler passiert ist (denn dann würde aus A nie ein B werden sondern irgendein Codewort, das nicht zum Code gehört).

Regel 1:
Um d Fehler erkennen zu können, brauchen wir mindestens einen Hammingabstand von d+1.

Regel 2:
Um d Fehler beheben zu können, müssen die Codewörter einen möglichst großen Abstand (also Hammingabstand) haben, mindestens einen Abstand von 2d+1. Dann sind die Codewörter nämlich so weit voneinander entfernt, dass bei d Fehlern das Originalcodewort immer noch am nächsten ist als alle anderen und somit kann der Fehler schließlich behoben werden.

Auf den nächsten Seiten schauen wir uns mal den sogenannten Cyclic-Redundancy-Check (kurz CRC) an. Dieser ist ein Algorithmus zur Fehlererkennung!
Doch zuvor gibt es noch ein kleines Beispiel zur ~~Fehlererkennung~~ Fehlerbehebung mittels Hamming-Distanz.

Fehlerbehebung durch Hamming-Distanz

Es hieß ja, dass man bei einer Anzahl von d Einzelbitfehlern einen Hammingabstand von $2d+1$ braucht, um diese zu beheben.

Im Umkehrschluss heißt das für ~~$2d+1=f$~~ einen Hammingabstand von f (so nennen wir jetzt mal den Abstand, wobei $f = 2d+1$ ist) Folgendes:

Wenn wir einen Abstand von f haben, wie viele Einzelbitfehler sind dann höchstens möglich, um diese noch beheben zu können?

$$2d+1 = f \quad |-1$$
$$2d = f-1 \quad |:2$$
$$d = (f-1)/2$$

Bei einem Hamming-Abstand von f können also $(f-1)/2$ Fehler behoben werden.

Und jetzt mal konkret:

Nehmen wir mal folgende vier Codewörter:

A	0000000000
B	0000011111
C	1111100000
D	1111111111

Die Hamming-Distanz ist in diesem Fall 5.

Also können $(5-1)/2 = 4/2 = 2$ ~~Fehler~~ Einzelbitfehler behoben werden. Egal, wo 2 Bits verändert werden, es gibt immer nur ein Codewort, das diesem veränderten Codewort am nächsten ist (und das ist das ursprüngliche, also das original gesendete).

Kleiner Test:
Welches Codewort wurde ursprünglich gesendet?

Zeichen: 00110 11111

Lösung: Vergleichen wir die jeweiligen Abstände einfach mal in einer Tabelle:

	00110 11111
A 00000 00000	7
B 00000 11111	2
C 11111 00000	8
D 11111 11111	3

} Abstand

Der klare Gewinner ist B, denn kein anderes Zeichen hat einen gleichwertigen oder sogar kleineren Abstand...

Jetzt aber zum CRC-Modell!

Los geht's auf der nächsten Seite...

Cyclic-Redundancy-Check

CRC ist, wie bereits angedeutet, ein Algorithmus zur Fehlererkennung!

Es gibt die Nutzdaten, die es zu übertragen gilt. Diese Nutzdaten, also der zu übertragende Block, nennen wir B(x).
Wir hatten ja vorhin bereits festgestellt (obwohl, wir haben es einfach nur hingenommen), dass wir, um Fehler erkennen zu können, Redundanz brauchen (also zusätzliche Bits neben den zu übertragenden Bits).
Mit wie vielen und vor allem mit welchen Bits wir den Block auffüllen müssen, das bekommen wir durch eine Polynomdivision raus.
Hier können wir gerne die "Schema-F-Schablone" benutzen :).

Was wir jetzt machen:
Den zu übertragenden Block, also B(x), müssen wir durch ein vorher festgelegtes Polynom, das sogenannte "Generatorpolynom" G(x), teilen.

Nehmen wir mal als Generatorpolynom
$G(x) = x^4 + x + 1$ an (Polynom 4-ten Grades).
Wie können wir daraus eine Bitfolge mit Einsen und Nullen machen?
Ganz einfach: Wir gehen die Hochzahlen durch und da, wo wir im Generatorpolynom etwas stehen haben, kommt eine 1, sonst eine 0.
Wird beim Beispiel deutlich:

x^4	x^3	x^2	x^1	x^0 (=1)
↓	↓	↓	↓	↓
1	0	0	1	1

$= x^4 + x + 1$

Also ist unser Generatorpolynom $G(x)$ in Binärdarstellung

$G(x) = 10011$

Wir haben ein Polynom 4-ten Grades (4 ist der höchste Exponent), deswegen füllen wir den Datenblock mit 4 Redundanzbits auf und das teilen wir dann durch das Generatorpolynom: Nehmen wir als $B(x)$ mal die Folge 11010110 11 an.

Also:

Rahmenbits: 1101011011 | Redundanzbits: 0000 : 10011 (= G(x)) = Q(x) + R(x)

Das, was bei dieser Rechnung rauskommt ist erstmal ein weiteres Polynom, das wir $Q(x)$ nennen und ein Restpolynom, das wir $R(x)$ nennen.

Wie wird das nun berechnet?

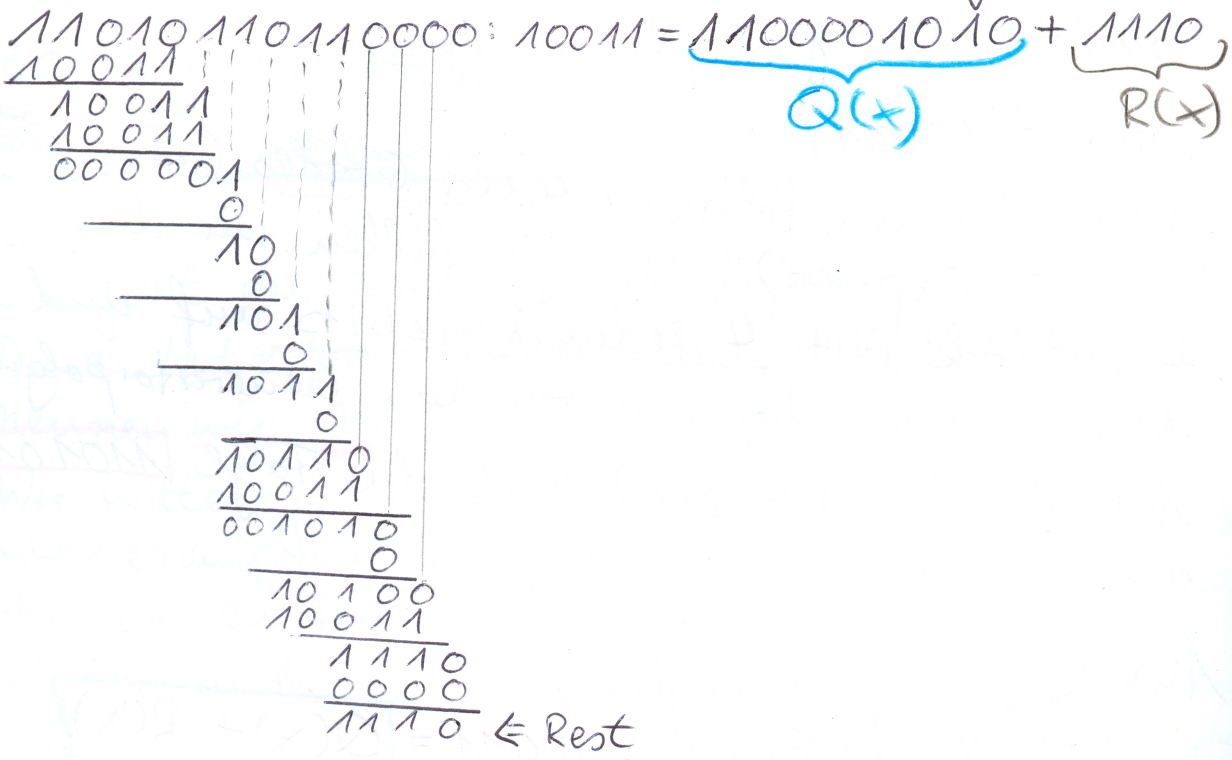
Ganz einfach:
Wir teilen immer durch 10011 und gucken, ob das in die obere Eins-Null-Folge reinpasst. Das ist dann der Fall, wenn die gleiche Bitanzahl vorhanden ist.
Probe auf Exempel:

Erste Zeile:

11010110110000 : 1011 =
1011

Ja, passt rein, also schreiben wir hier eine 1 (sonst 0).
Unter dem Bruchstrich kommen jetzt überall Einsen hin, wo sich die obere und die untere Zeile unterscheiden.

Hier jetzt die komplette Rechnung:

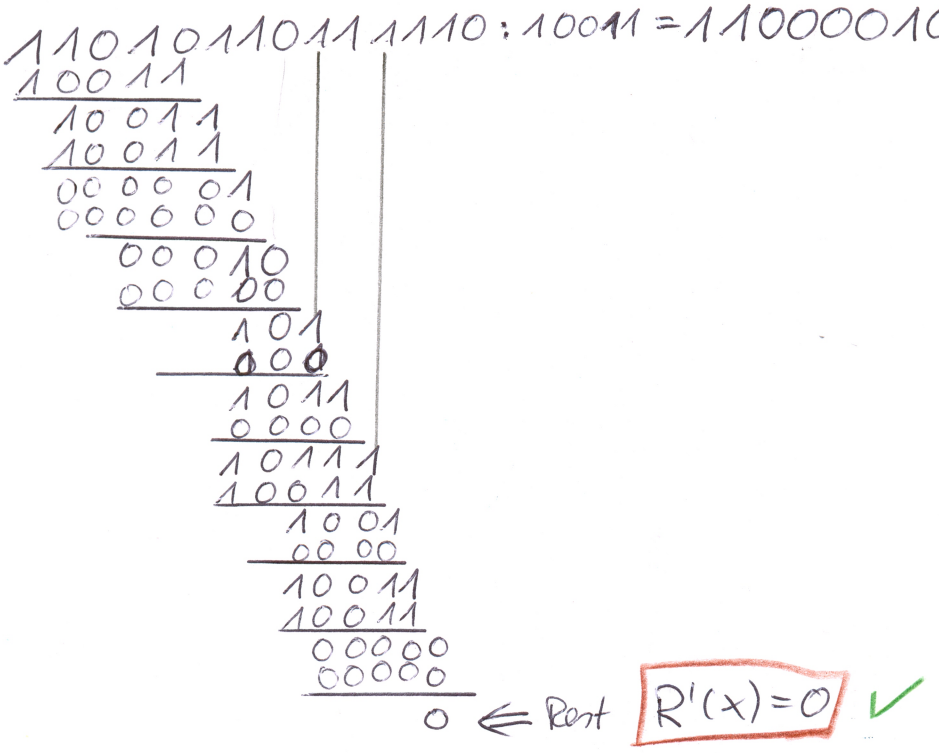


Interessant ist für uns das Restpolynom 1110. Mit genau diesen Bits müssen wir jetzt die Redundanzbits auffüllen, also statt der vier Nullen jetzt 1110. Somit sieht der übertragene Rahmen wie folgt aus:

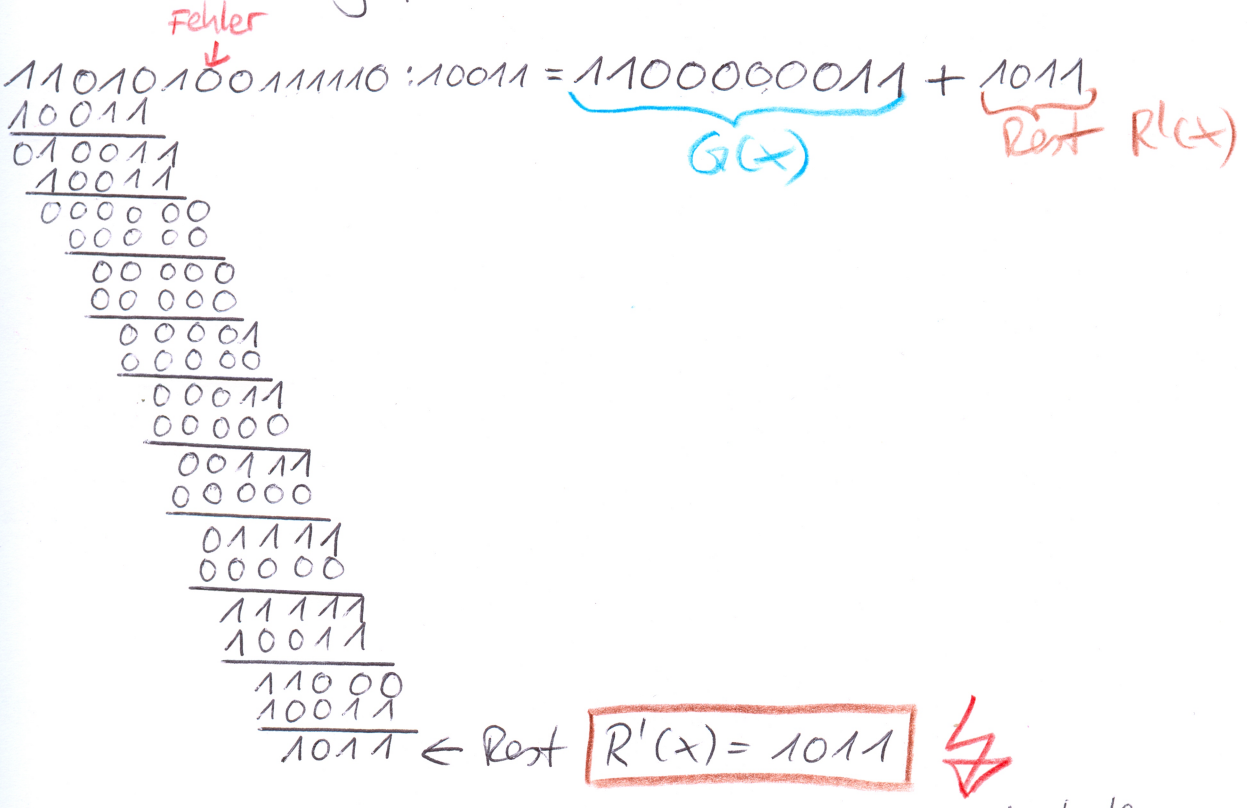
11010110111110
Nutzdatenbits | Redundanzbits.

Wenn der übertragene Rahmen genauso ankommt, dann ergibt die Division durch das Generatorpolynom eine Rechnung ohne Rest ($R'(x)=0$).
Sollte irgendwo auch nur ein Bit anders sein, kommt ein Restpolynom raus und ~~der~~ der Empfänger (der ja jetzt genau diese Prüfung macht) weiß, dass ein Fehler bei der Übertragung passiert sein muss. Also schickt er dem Sender die Botschaft, dass er die Nachricht bitte nochmal senden soll (negative Acknowlege ⇒ negative Bestätigung).

Berechnung für den Fall, das alles korrekt ist:



Berechnung für Fehler an Stelle 7:



Es gibt einen Rest, also ist ein Fehler aufgetreten...