Kapitel 4
Zyklischer Redundanzcode (CRC) mit Fehlererkennung

4.1 Einleitung

Es sei X eine Nachrichten**quelle**, die Zeichen aus ihrem Zeichenvorrat $X = \{x_1, x_2, \cdots\cdots, x_n\}$ über einen Binärkanal an eine Nachrichten**senke** Y schickt. Durch Störungen auf dem Übertragungskanal werden Teile der von X ausgesendeten Zeichen beschädigt und kommen in Y falsch an. Siehe Bild 83.

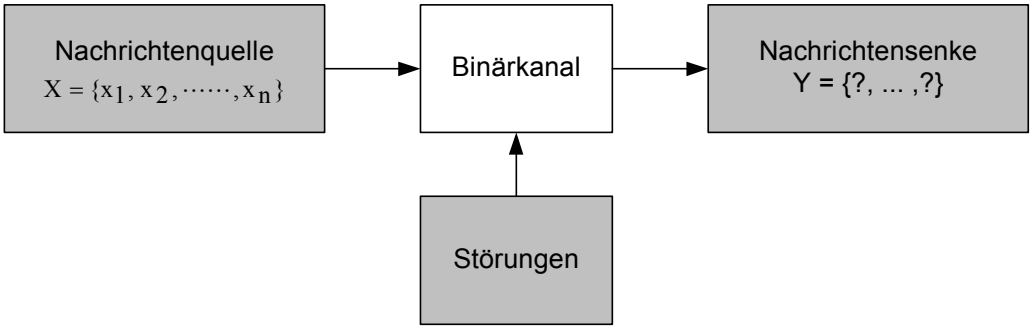


Bild 83: Gestörter Binärkanal

So ist die Entropie H(Y) der Nachrichtensenke **kleiner** als die Entropie der Nachrichtenquelle H(X). Mit anderen Worten, Information geht verloren. Wir erinnern uns (vgl. Seite 188 u. 191).

1, Wenn infolge von Störungen aus verschiedenen Zeichen einer Nachrichtenquelle X mit dem Zeichenvorrat $X = \{x_1, x_2, ..., x_n\}$ in der Nachrichtensenke Y das gleiche Zeichen y_j entsteht, dann resultiert hieraus eine Rückschlussunsicherheit, die sogenannte **Äquivokation**. Sie ist derjenige Teil der Quellenentropie H(X), der während der Übertragung verloren geht. Die Veranschaulichung der Äquivokation von Seite 188 ist hier nochmals dargestellt. Siehe Bild 84.

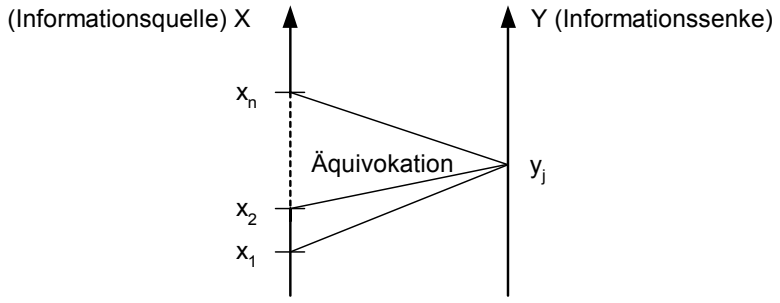


Bild 84: Veranschaulichung der Äquivokation

2, Wenn infolge von Störungen auf dem Übertragungskanal sich ein Teil der empfangenen Zeichen y_j von den gesendeten Zeichen x_i unterscheidet, dann entsteht eine Vorhersageunsicherheit, die sogenannte **Irrelevanz**. Sie erzeugt je nach Störung eines Zeichens x_i verschiedene Zeichen y_j. Die auf der Seite 191 gezeigte Veranschaulichung der Irrelevanz ist hier nochmals abgebildet. Siehe Bild 85.

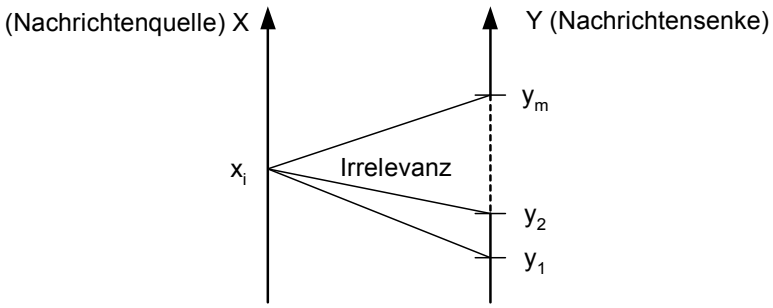
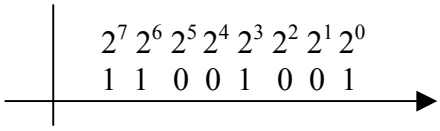


Bild 85: Veranschaulichung der Irrelevanz

So ist die Irrelevanz eine **Fehlinformation**, die in der Nachrichtensenke Y auftritt, die aber nicht von der Nachrichtenquelle X stammt.

4.2 Cyclic Redundancy Code (CRC)

Um Übertragungsfehler aufzuspüren, wird in der Praxis bevorzugt der **Polynomcode** eingesetzt. Er ist auch als zyklischer Redundanzcode oder crc (cyclic redundancy code) bekannt und stammt von William Wesley Peterson. Er war ein US-amerikanischer Mathematiker und Elektroingenieur (1924 – 2009). Der Polynomcode basiert auf der Idee, einen Bitstrom als Polynom mit der Basis x=2 und den Koeffizienten 0 und 1 zu behandeln. Zum Beispiel kann der Bitstrom



als Polynom in folgender Weise dargestellt werden:

$$\mathbf{f(x) = 1x^7 + 1x^6 + 0x^5 + 0x^4 + 1x^3 + 0x^2 + 0x^1 + 1x^0}$$

oder kürzer

$$\mathbf{f(x) = x^7 + x^6 + x^3 + 1}$$

Wenn eine Nachrichtenquelle einen Bitstrom zur Senke schickt, dann kann dieser Bitstrom als Polynom T(x) aufgefasst werden, mit dem einige Berechnungen durchführbar sind. Insbesonders interessiert die Frage, ob das Polynom T(x) durch ein anderes Polynom G(x) teilbar ist. Führt man eine solche Division durch, lautet das Ergebnis allgemein

$$\mathbf{T(x) : G(x) = I(x) + R(x)}$$

Man erhält das Polynom I(x) und das Restpolynom R(x). Ist R(x) = 0, dann ist der Dividend durch den Divisor ohne Rest teilbar. Ist $R(x) \neq 0$, dann ist der Dividend durch den Divisor nicht restlos teilbar. Was kann man mit dieser Erkenntnis anfangen? Sie ist der Schlüssel zu einem Verfahren, mit dem sich Übertragungsfehler aufspüren lassen. Eine Idee vom Prinzip dieses Verfahrens vermittelt das folgende Bild 86 am Beispiel des Zehner-Zahlensystem.

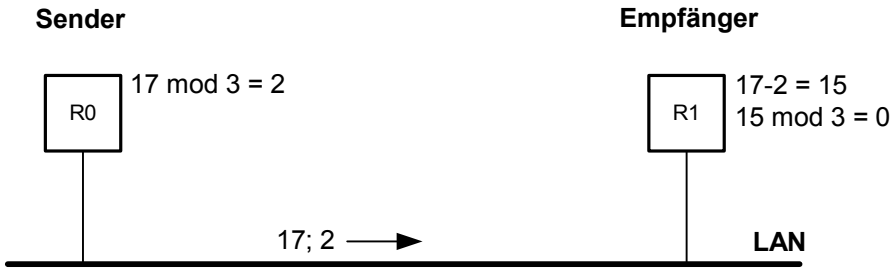


Bild 86: Methode zur Erkennung von Übertragungsfehlern

Der Sender R0 hat die Zahl 17, die beim Empfänger R1 fehlerfrei ankommen soll. Sowohl der Sender als auch der Empfänger führen eine modulo Division durch, und einigen sich zu diesem Zweck auf einen gemeinsamen Divisor, z.B. 3.

Der Sender berechnet 17:3 = 5 Rest 2, und schickt den Dividenden 17 und den Rest 2 durch das LAN zum Empfänger. Der Empfänger subtrahiert vom empfangenen Dividenden den Rest, also 17-2 = 15, und führt anschließend mit dem Ergebnis die Division 15:3 = 5 Rest 0 durch. Der **Rest 0** ist ein Indiz dafür, dass kein Übertragungsfehler aufgetreten ist.
Um diese Methode für die Praxis nutzen zu können, müssen wir sie auf Polynomberechnungen im binären Zahlensystem anwenden. Dazu benötigen wir die Regeln der modulo 2 Arithmetik. Eine allgemeine Betrachtung der modulo m Rechnung ist daher von Vorteil.

4.3 Die modulo m Rechnung

Es sei M = {-14,-4,6,16,26} eine Menge von Zahlen im Zehnersystem. Zu jeder Zahl x suchen wir die „Restzahl“ r, und führen zu diesem Zweck die modulo m Rechnung durch.

Das Verfahren lautet: Subtrahiere **x-m** oder addiere **x+m** solange, bis eine Restzahl r entsteht, die in der sogenannten Restklasse **0...m-1** enthalten ist.

Ist beispielsweise m = 10, dann liefert die **modulo 10 Rechnung** folgende Ergebnisse:

x modulo 10	= r
-14 modulo 10 = -14+10 = -4; -4+10	= 6
-4 modulo 10 = -4+10	= 6
6 modulo 10	= 6
16 modulo 10 = 16-10	= 6
26 modulo 10 = 26-10 = 16; 16-10	= 6

Was sagen uns die Ergebnisse? In der modulo m Rechnung wird nicht zwischen der Zahl x und der Restzahl r unterschieden. So gilt für das angegebene Beispiel

$$\mathbf{6 = \{...,-14,-4,6,16,26,...\}}$$

dass alle rechts stehenden Zahlen durch die Zahl 6 repräsentiert werden. Dies kann man allgemein folgendermaßen ausdrücken:

$$\mathbf{r = r+k \bullet m; \qquad k =...,-2,-1,0,+1,+2,...}$$

Verifizierung:

$6 = 6-2\bullet10 = -14$ (k = -2)
$6 = 6-1\bullet10 = -4$ (k = -1)
$6 = 6+0\bullet10 = 6$ (k = 0)
$6 = 6+1\bullet10 = 16$ (k =+1)
$6 = 6+2\bullet10 = 26$ (k =+2)

Die Schlussfolgerung lautet: In der modulo m Rechnung gibt es genau m unterscheidbare Zahlen. Es sind dies die Elemente 0.....m-1 eines Körpers, den man auch als Galois-Feld GF(m) bezeichnet. Im GF(10) lauten die Zahlen **0,1,.....9**. Im GF(2) sind es die Zahlen **0,1**.

4.3.1 Die Arithmetik im Galois-Feld GF(2)

Wir führen eine modulo2 Addition und eine modulo2 Subtraktion durch und überprüfen bei beiden Rechnungsarten, ob die Ergebnisse im GF(2) liegen.

Addition:

0+0 = 0; 0 modulo2 = 0
0+1 = 1; 1 modulo2 = 1
1+0 = 1; 1 modulo2 = 1
1+1 = 2; 2 modulo2 = 0

Subtraktion:

0-0 = 0; 0 modulo2 = 0
0-1 = -1; -1 modulo2 = 1
1-0 = 1; 1 modulo2 = 1
1-1 = 0; 0 modulo2 = 0

Bei der modulo2 Addition werden die Ergebnisse der Operationen 0+0 und 1+1 durch 0 repräsentiert. Die Ergebnisse der Operation 0+1 und 1+0 durch 1.
Bei der modulo2 Subtraktion werden die Ergebnisse der Operationen 0-0 und 1-1 durch 0 repräsentiert. Die Ergebnisse der Operationen 0-1 und 1-0 durch 1.
Also liegt bei beiden Rechnungsarten ein Galois-Feld GF(2) mit den Elementen M={0,1} vor. Desweiteren zeigen die Ergebnisse, dass es zwischen der modulo2 Addition und der modulo2 Subtraktion keinen Unterschied gibt. Die Operationen + und - lassen sich somit durch EXOR-Verknüpfungen realisieren.

Beispiele:

modulo2 Addition	modulo2 Subtraktion
1011 0100	0101 0000
+1101 0000 (EXOR)	-1000 0001 (EXOR)
0110 0100	1101 0001

Kehren wir zurück zur Polynomberechnung. Wir benutzen die Regeln der modulo2 Arithmetik und führen damit die Division T(x) : G(x) = I(x) + R(x) durch.

Beispiel 1:

Es sei **$T(x) = 1x^4+1x^3+0x^2+1x^1+1x^0$** und **$G(x) = 1x^2+1x^1+1x^0$**. Gesucht wird das Restpolynom **$R(x)$**.
Division T(x) durch G(x):

$$T(x) \quad : \quad G(x) \quad = \quad I(x) \quad + \quad R(x)$$

$$\begin{array}{l}
1x^4+1x^3+0x^2+1x^1+1x^0 : 1x^2+1x^1+1x^0 = 1x^2+0x^1+1x^0 \quad + \quad 0 \\
\underline{1x^4+1x^3+1x^2} \ \downarrow \ \downarrow \\
\qquad\qquad \mathbf{1x^2}+1x^1+1x^0 \\
\qquad\qquad \underline{1x^2+1x^1+1x^0} \\
\qquad\qquad \text{Rest: } 0
\end{array}$$

Im Beispiel 1 ist wegen R(x)=0 der Dividend T(x) durch den Divisor G(x) ohne Rest teilbar. Außerdem gibt es im Sinne der modulo m Rechnung zwischen dem „Restpolynom 0“ und dem Polynom $T(x)=1x^4+1x^3+0x^2+1x^1+1x^0$ keinen Unterschied.

Beispiel 2:

Es sei **$T(x) = 1x^5+0x^4+1x^3+0x^2+0x^1+1x^0$** und **$G(x) = 1x^2+1x^1+1x^0$**. Gesucht wird auch hier das Restpolynom **$R(x)$**.

Division T(x) durch G(x):

$$T(x) \quad : \quad G(x) \quad = \quad I(x) \quad + \quad R(x)$$

$$\begin{array}{l}
1x^5+0x^4+1x^3+0x^2+0x^1+1x^0 : 1x^2+1x^1+1x^0 = 1x^3+1x^2+1x^1 \quad + \quad x+1 \\
\underline{1x^5+1x^4+1x^3} \ \downarrow \ \downarrow \ \downarrow \\
\quad \mathbf{1x^4}+\mathbf{0x^3}+0x^2+0x^1+1x^0 \\
\quad \underline{1x^4+1x^3+1x^2} \ \downarrow \ \downarrow \\
\qquad \mathbf{1x^3}+\mathbf{1x^2}+0x^1+1x^0 \\
\qquad \underline{1x^3+1x^2+1x^1} \ \downarrow \\
\qquad \text{Rest: } \ \mathbf{1x^1}+1x^0 \text{ bzw. } x+1
\end{array}$$

Im Beispiel 2 ist wegen R(x)=x+1 der Dividend T(x) durch den Divisor G(x) **nicht** restlos teilbar. Auch hier gilt natürlich, dass es zwischen T(x) und R(x) keine Unterscheidung gibt.

4.4 Fehlererkennung mit der Polynomcodemethode

Die Polynomcodemethode benutzt die Polynomdivision, wie wir sie anhand von zwei Beispielen kennen gelernt haben. In Anlehnung an das Verfahren nach Bild 86, einigen sich der Sender und der Empfänger auf einen gemeinsamen Divisor, dem sogenannten **Generatorpolynom** G(x). Dabei müssen in G(x) das höchstwertigste und das niederwertigste Bit gleich 1 sein. Die Grundidee ist, auf der Senderseite an den originalen Bitstrom, der dem Polynom T(x) entspricht, eine Prüfsumme so anzuhängen, dass sich auf der Empfängerseite das resultierende Polynom durch G(x) restlos teilen lässt:

Der detaillierte Algorithmus lautet wie folgt:

1, Sender:

- Wenn k der Grad von G(x) ist, dann füge k Nullbits an T(x) an, so dass das Polynom $T(x)x^k$ entsteht. Ist n die Anzahl der Bits in T(x), dann enthält das resultierende Polynom n+k Bits.

  Beispiel: Sei $T(x) = 1x^3+0x^2+1x^1+0x^0 = x^3+x$ und
  $G(x) = 1x^3+0x^2+1x^1+1x^0 = x^3+x+1$

  Der Grad k von G(x) ist 3. Somit wird

  $$T(x)x^k = (1x^3+0x^2+1x^1+0x^0)x^3 = \underbrace{1x^6+0x^5+1x^4+0x^3}_{n=4} + \underbrace{0x^2+0x^1+0x^0}_{k=3}$$

- Führe die modulo2 Division $T(x)x^k : G(x)$ aus.

  Beispiel Fortsetzung:

  $$T(x)x^k : G(x) = I(x) + R(x)$$

  $$1x^6+0x^5+1x^4+0x^3+0x^2+0x^1+0x^0 : 1x^3+0x^2+1x^1+1x^0 = 1x^3+0x^2+0x^1+1x^0 + x+1$$

  $$\underline{1x^6+0x^5+1x^4+1x^3} \quad \downarrow \quad \downarrow \quad \downarrow$$
  $$\mathbf{1x^3}+0x^2+0x^1+0x^0$$
  $$\underline{1x^3+0x^2+1x^1+1x^0}$$
  Rest: $1x^1+1x^0$ bzw. **x+1**

  Das entstandene Restpolynom x+1 ist die gesuchte Prüfsumme, die in der Praxis oft auch als **frame check sequence** (fcs) bezeichnet wird.

- Ziehe das Restpolynom, also die fcs, durch modulo2 Subtraktion von $T(x)x^k$ ab. Das Ergebnis-Polynom, wir nennen es C(x), ist ein Polynomcode, der als **cyclic redundancy code** (crc) bezeichnet wird. Warum crc ein zyklischer Code ist, werden wir etwas später sehen. **Hinweis**: Weil in der modulo2 Arithmetik zwischen Subtraktion und Addition nicht unterschieden wird, erscheint in der folgenden Rechnung +R(x).

  Beispiel Fortsetzung:

  $$T(x)x^k + R(x) = C(x)$$

  $$1x^6+0x^5+1x^4+0x^3+0x^2+0x^1+0x^0 + \underbrace{1x^1+1x^0}_{fcs} = \underbrace{1x^6+0x^5+1x^4+0x^3+0x^2+\mathbf{1x^1}+\mathbf{1x^0}}_{crc}$$

Das vom Sender berechnete Polynom C(x) schickt nun sein serieller Controller (z.B. Ethernet Controller oder UART; Universal Asyncronous Receiver Transmitter) als Bitstrom durch das LAN zum Empfänger.

2, Empfänger:

- Führe die modulo2 Division C(x) : G(x) aus.
  Beispiel Fortsetzung:

$$\begin{array}{rcl} C(x) & : & G(x) = I(x) + R(x) \\ 1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0 & : & 1x^3+0x^2+1x^1+1x^0 = 1x^3+0x^2+0x^1+1x^0 + 0 \end{array}$$

$$\begin{array}{l} \underline{1x^6+0x^5+1x^4+1x^3} \quad \downarrow \quad \downarrow \quad \downarrow \\ \qquad\qquad \mathbf{1x^3}+0x^2+1x^1+1x^0 \\ \qquad\qquad \underline{1x^3+0x^2+1x^1+1x^0} \\ \qquad\qquad\quad \text{Rest: } 0 \end{array}$$

Solange C(x) ein zulässiges Codewortpolynom ist, ist eine restfreie Division durch G(x) möglich, dann ist R(x)=0. Ein fehlerhaftes Codewortpolynom erkennt man daran, dass C(x) modulo G(x) ≠ 0 ist.
In unserem Beispiel ist R(x)=0 bzw. die fcs=0, und damit ist der Bitstrom unverstümmelt beim Empfänger angekommen. Ist das wirklich so? Kann beim Empfänger auch ein fehlerbehafteter Polynomcode ankommen, der trotzdem zu R(x)=0 führt?
Wenn C(x) auf dem Weg zum Empfänger beschädigt wird, dann addiert sich zu C(x) das Fehlerpolynom F(x). Mit anderen Worten, beim Empfänger kommt C(x)+F(x) an. Ist zufälligerweise das Fehlerpolynom mit dem Generatorpolynom identisch, dann gilt F(x)=G(x) und der Empfänger erhält C(x)+G(x). Demzufolge berechnet der Empfänger (modulo2)

$$(C(x)+F(x)) : G(x) = \frac{C(x)}{G(x)} + \frac{F(x)}{G(x)} = \frac{C(x)}{G(x)} + \frac{G(x)}{G(x)} = \mathbf{0} + \mathbf{0}$$

Die modulo2 Division $\frac{C(x)}{G(x)}$ liefert immer 0, und die modulo2 Division $\frac{F(x)}{G(x)} = \frac{G(x)}{G(x)}$ auch.

Das heißt, Fehlerpolynome die dem Generatorpolynom G(x) entsprechen, können nicht entdeckt werden, alle anderen werden dagegen erkannt. Daher ist es gut, zusätzliche Fehlerüberwachungssysteme einzusetzen (z.B. die Paritätsüberprüfung).

4.5 Zyklische Eigenschaften

Wir rotieren das originale Codewortpolynom C(x) und 1 Bitstelle links- oder rechtsherum, und bekommen das Polynom $C^{(1)}(x)$.
Beispiel: 1te Linksrotation

$$\begin{array}{rcl} C(x) & \rightarrow & C^{(1)}(x) \\ 1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0 & \rightarrow & \mathbf{0x^6+1x^5+0x^4+0x^3+1x^2+1x^1+1x^0} \end{array}$$

Wir führen die modulo2 Division $C^{(1)}(x) : G(x) = R(x)$ aus und bekommen
Beispiel: Fortsetzung

$$\begin{array}{rcl} C^{(1)}(x) & : & G(x) = I(x) + R(x) \\ 1x^5+0x^4+0x^3+1x^2+1x^1+1x^0 & : & 1x^3+0x^2+1x^1+1x^0 = 1x^2+0x^1+1x^0 + 0 \end{array}$$

$$\begin{array}{l} \underline{1x^5+0x^4+1x^3+1x^2} \quad \downarrow \quad \downarrow \\ \qquad\quad \mathbf{1x^3+0x^2}+1x^1+1x^0 \\ \qquad\quad \underline{1x^3+0x^2+1x^1+1x^0} \\ \qquad\quad \text{Rest: } 0 \end{array}$$

Augenscheinlich ist $C^{(1)}(x)$ ein gültiges Codewortpolynom, denn $C^{(1)}(x)$ modulo G(x) = 0. Ist k der Grad von C(x), dann entstehen weitere gültige Codewortpolynome durch zyklische Rotationen von C(x) um 2,3,..k Stellen. Nach der (k+1)ten Rotation entsteht dann wieder das Originalpolynom C(x). Es gibt also genau k+1 redundante Codewortpolynome.

Dies ist auch der Grund für den Begriff **c**yclic **r**edundancy **c**ode. In unserem Beispiel ist der Grad k von C(x) = 6. Daher gibt es k+1=7 gültige Codewortpolynome. Hier eine Zusammenfassung:

1		$\mathbf{C(x) = 1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0}$
2	1te Linksrotation:	$C^{(1)}(x) = 0x^6+1x^5+0x^4+0x^3+1x^2+1x^1+1x^0$
3	2te Linksrotation:	$C^{(2)}(x) = 1x^6+0x^5+0x^4+1x^3+1x^2+1x^1+0x^0$
4	3te Linksrotation:	$C^{(3)}(x) = 0x^6+0x^5+1x^4+1x^3+1x^2+0x^1+1x^0$
5	4te Linksrotation:	$C^{(4)}(x) = 0x^6+1x^5+1x^4+1x^3+0x^2+1x^1+0x^0$
6	5te Linksrotation:	$C^{(5)}(x) = 1x^6+1x^5+1x^4+0x^3+1x^2+0x^1+0x^0$
7	6te Linksrotation:	$C^{(6)}(x) = 1x^6+1x^5+0x^4+1x^3+0x^2+0x^1+1x^0$

7te Linksrotation: $C^{(7)}(x) = \mathbf{C(x) = 1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0}$

4.6 Modulo2 Division und Hardware

Wir wollen jetzt als nächsten Schritt die uns bereits bekannte, vom Empfänger durchgeführte modulo2 Division, durch eine Hardware-Logik realisieren. Zur Erinnerung:

$$\begin{array}{rcl} C(x) & : G(x) = & I(x) + R(x) \\ 1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0 & : 1x^3+0x^2+1x^1+1x^0 = & 1x^3+0x^2+0x^1+1x^0 + 0 \\ \underline{1x^6+0x^5+1x^4+1x^3} \downarrow \quad \downarrow \quad \downarrow & & \\ \mathbf{1x^3}+0x^2+1x^1+1x^0 & & \\ \underline{1x^3+0x^2+1x^1+1x^0} & & \\ \text{Rest: } 0 & & \end{array}$$

Im Bild 87 ist diese Logik zu sehen.

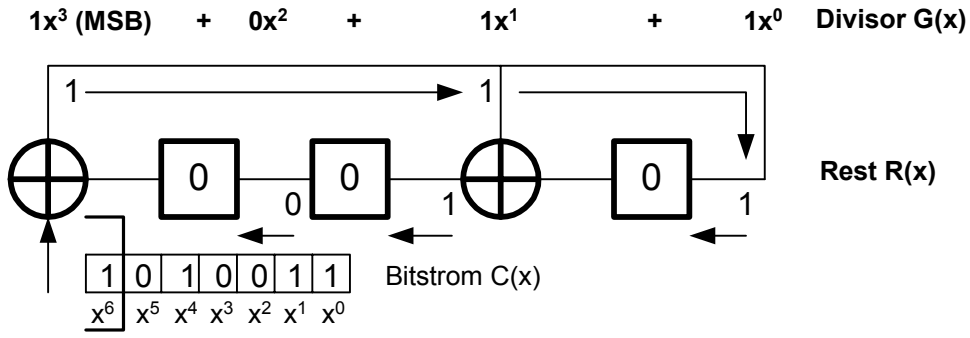


Bild 87: Modulo2 Divisions-Hardware

Sie besteht aus einer Anzahl von FlipFlops (□) und XORs (⊕). Die FlipFlops sind zu einem Schieberegister konfiguriert, das die temporären Restwerte während des Divisonsvorgangs aufnimmt, und am Ende der Division die fcs enthält. Anfänglich sind die FlipFlops gelöscht, das Schieberegister also leer, und damit die fcs=0. Die Anzahl der FlipFlops ist abhängig von der Anzahl der + Zeichen im Divisor G(x). An jeder Position eines +Zeichens erscheint im Schieberegister ein FlipFlop.

Die Anzahl der XORs ist abhängig von den Koeffizienten der Polynomglieder im Divisor G(x). Ist der **Koeffizient = 1**, fügt sich an der betreffenden Stelle in die FlipFlop-Kette ein XOR ein. Dies ist an den Stellen $1x^3$, $1x^1$ und $1x^0$ der Fall. Weil an der Stelle $1x^0$ keine Verknüpfung stattfindet, ist es dort weggelassen.
Das XOR an der Stelle $1x^3$ hat zwei Eingänge. Der eine Eingang bekommt das höchstwertige Bit des Schieberegisters, und der andere Eingang die seriell einlaufenden Bits des Dividenden C(x).

1. Schritt: Die modulo2 Divisions-Hardware beginnt ihre Arbeit an der Divisorstelle $1x^3$ mit der XOR-Verknüpfung des höchstwertigen Bits in C(x) und dem höchstwertigen Bit des Schieberegisters: **1+0=1**, bzw. MSB=1. Über die Verbindungsleitung gelangt das MSB zu den Divisorstellen $1x^1$ und $1x^0$. Damit steht es an dem einen Eingang des zweiten XORs und am Eingang des niederwertigsten FlipFlops an. Das zweite XOR verknüpft das MSB mit dem Ausgang des niederwertigsten FlipFlops und liefert 1+0=1. Danach stehen an den Eingängen der drei FlipFlops neue Informationen an, nämlich 011. Sie entsprechen dem temporären Rest $R(x) = 0x^2+1x^1+1x^0$. Siehe Bild 87. Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 1te Bearbeitungsschritt ist beendet. Siehe Bild 87a.

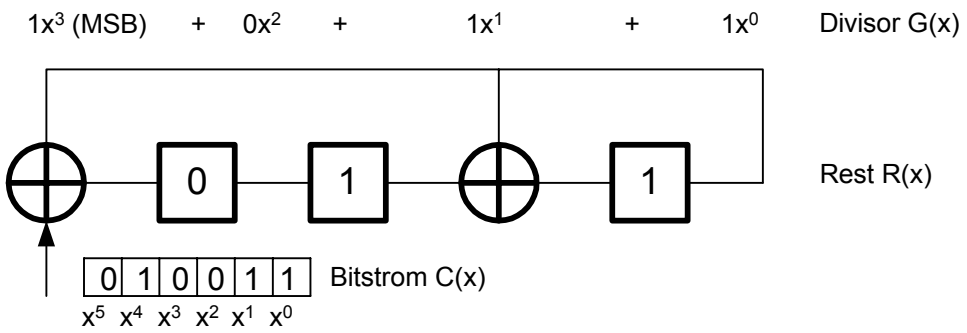


Bild 87a: Situation **nach** der Bearbeitung von $1x^6$ des Bitstroms C(x)

2. Schritt: Es folgt die Bearbeitung von Bit $0x^5$ des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **0+0=0**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **0+1=1**,
- der temporäre Rest lautet $R(x)=\mathbf{1x^2+1x^1+0x^0}$. Siehe Bild 87b.

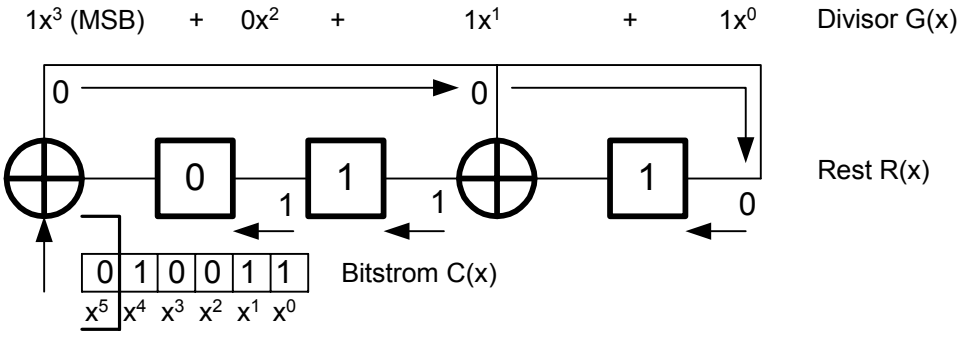


Bild 87b: Bearbeitung von Bit $0x^5$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 2te Bearbeitungsschritt ist beendet. Siehe Bild 87c.

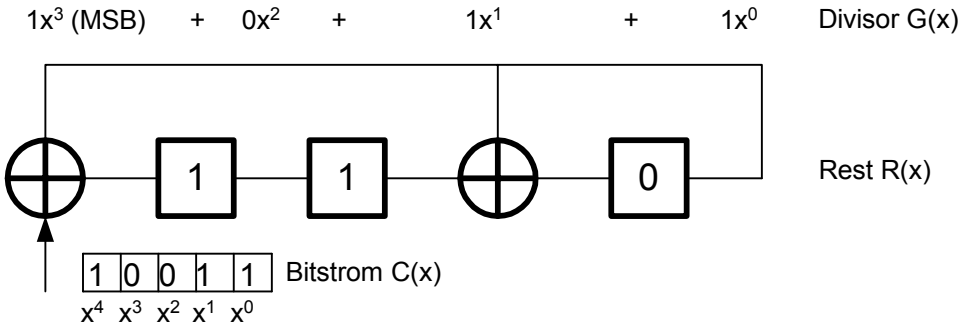


Bild 87c: Situation **nach** der Bearbeitung von $0x^5$ des Bitstroms C(x)

3. Schritt: Es folgt die Bearbeitung von Bit $1x^4$ des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **1+1=0**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **0+0=0**,
- der temporäre Rest lautet R(x)=$\mathbf{1x^2+0x^1+0x^0}$. Siehe Bild 87d.

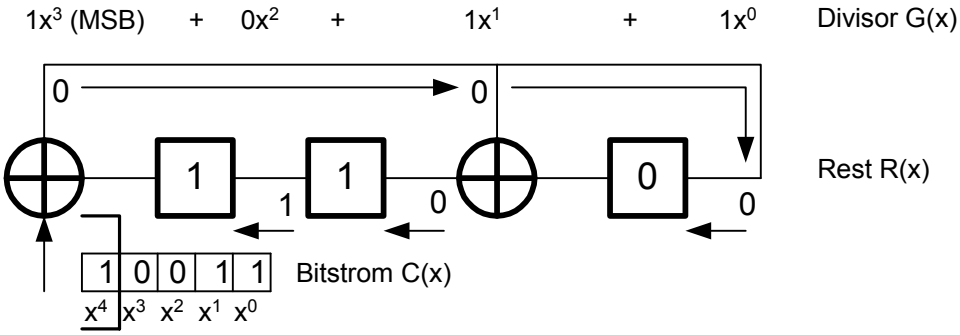


Bild 87d: Bearbeitung von Bit $1x^4$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 3te Bearbeitungsschritt ist beendet. Siehe Bild 87e.

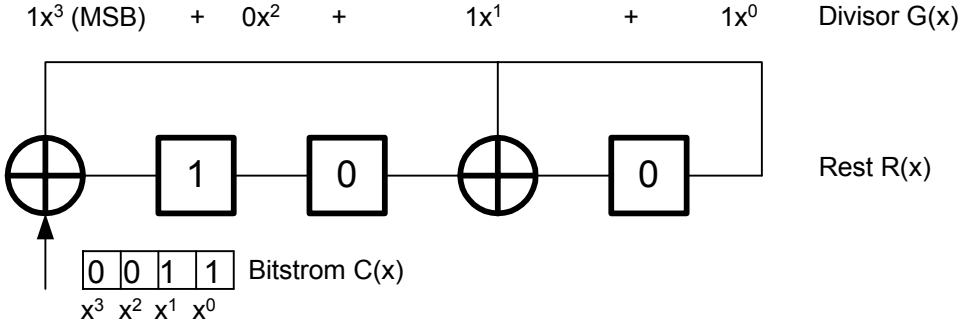


Bild 87e: Situation **nach** der Bearbeitung von $1x^4$ des Bitstroms C(x)

4. Schritt: Es folgt die Bearbeitung von Bit $0x^3$ des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **0+1=1**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **1+0=1**,
- der temporäre Rest lautet R(x)=$\mathbf{0x^2+1x^1+1x^0}$. Siehe Bild 87f.

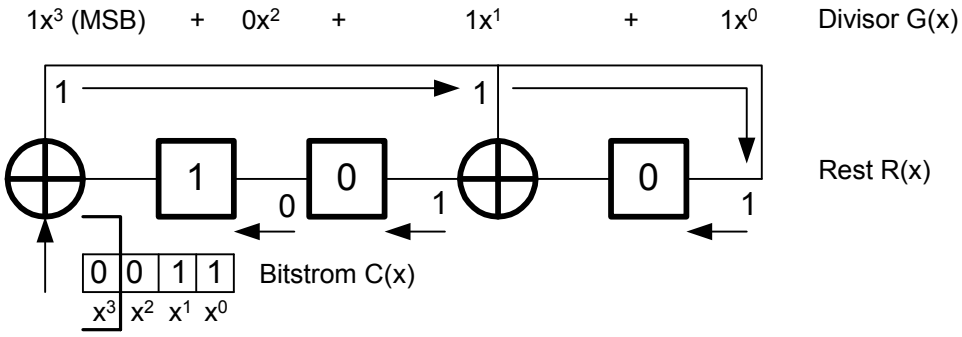


Bild 87f: Bearbeitung von Bit $0x^3$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 3te Bearbeitungsschritt ist beendet. Siehe Bild 87g.

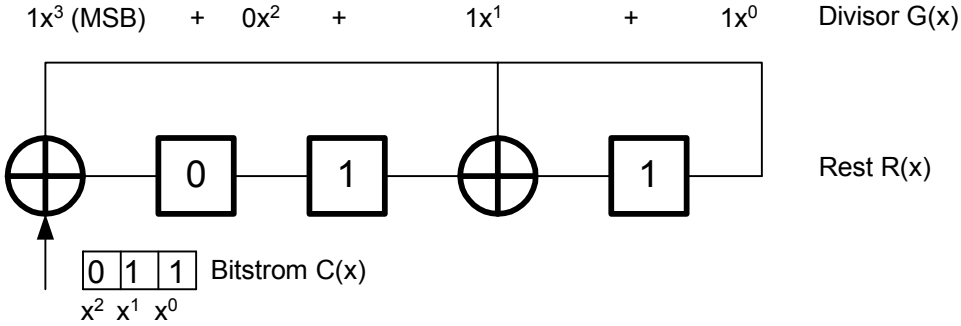


Bild 87g: Situation **nach** der Bearbeitung von $0x^3$ des Bitstroms C(x)

5. Schritt: Es folgt die Bearbeitung von Bit 0x2 des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **0+0=0**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **0+1=1**,
- der temporäre Rest lautet R(x)=$\mathbf{1x^2+1x^1+0x^0}$. Siehe Bild 87h.

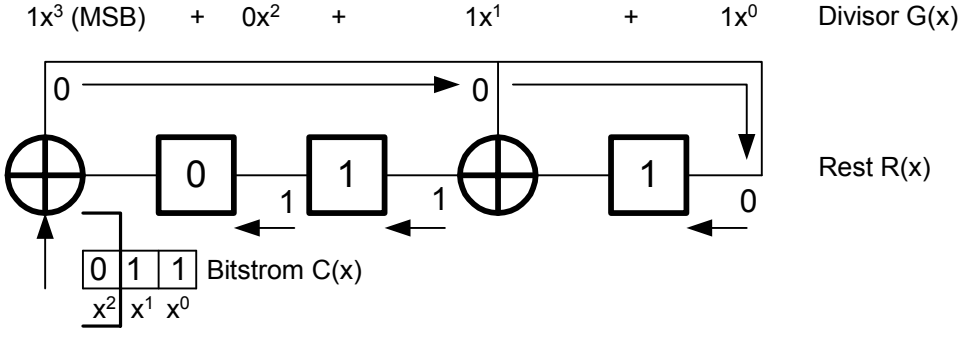


Bild 87h: Bearbeitung von Bit $0x^2$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 5te Bearbeitungsschritt ist beendet. Siehe Bild 87i.

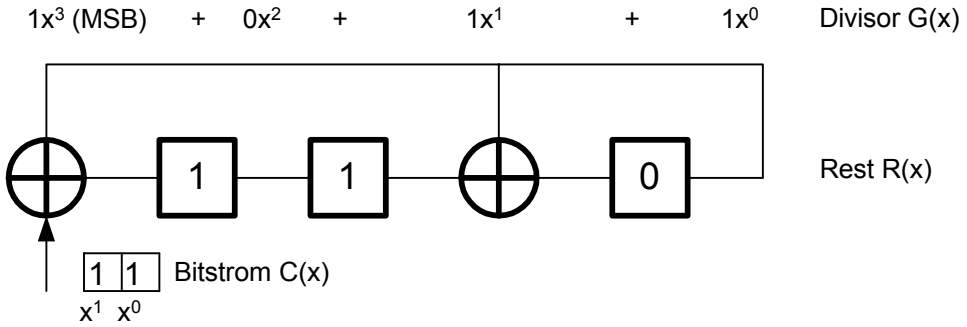


Bild 87i: Situation **nach** der Bearbeitung von $0x^2$ des Bitstroms C(x)

6. Schritt: Es folgt die Bearbeitung von Bit $1x^1$ des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **1+1=0**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **0+0=0**,
- der temporäre Rest lautet R(x)=$\mathbf{1x^2+0x^1+0x^0}$. Siehe Bild 87j.

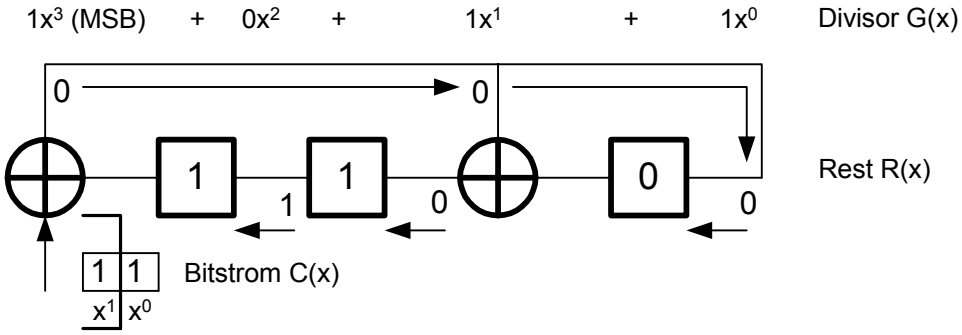


Bild 87j: Bearbeitung von Bit $1x^1$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 6te Bearbeitungsschritt ist beendet. Siehe Bild 87k.

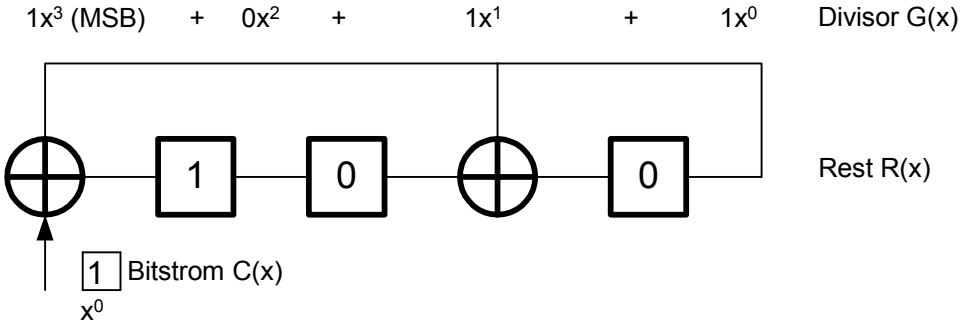


Bild 87k: Situation nach der Bearbeitung von $1x^1$ des Bitstroms C(x)

7. Schritt: Es folgt die Bearbeitung von Bit $1x^0$ des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **1+1=0**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **0+0=0**,
- der temporäre Rest lautet R(x)=**$0x^2+0x^1+0x^0$**. Siehe Bild 87l.

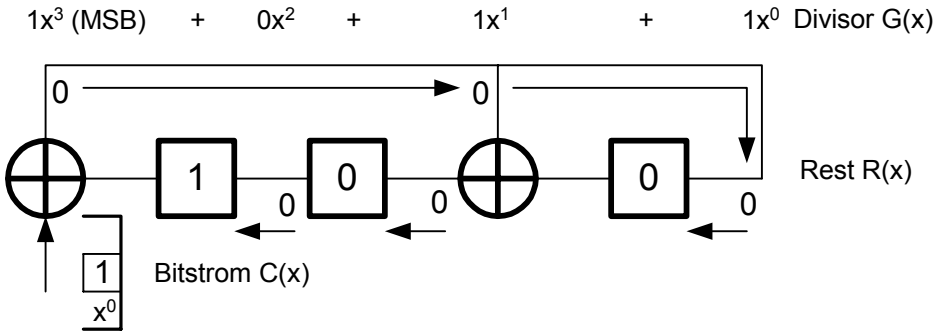


Bild 87l: Bearbeitung von Bit $1x^0$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 7te Bearbeitungsschritt ist beendet. Gleichzeitig ist auch das Bitreservoir im Codewortpolynom C(x) ausgeschöpft.
Die Division C(x) modulo G(x) = 0 ist damit abgeschlossen. Die Hardware hat das erwartete Ergebnis R(x)=0 bzw. fcs=0 geliefert. Siehe Bild 87m.

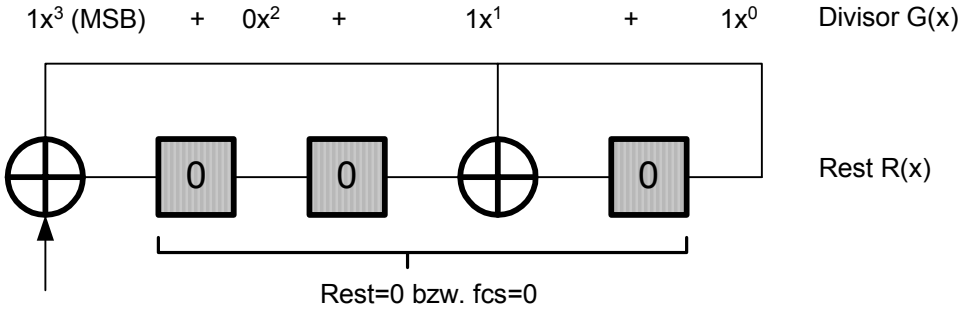


Bild 87m: Das Ergebnis der Hardware-Division C(x) modulo G(x)=0

4.7 Modulo2 Division und Software

Wir wollen ein C-Programm entwickeln, das die Mechanismen der modulo2 Divisions-Hardware simuliert.

1, Vorbereitung:
Wir formen aus dem Divisor $G(x)=1x^3+0x^2+1x^1+1x^0$ die Maske $m(x)=\mathbf{0}x^3+0x^2+1x^1+1x^0$. Mit anderen Worten, wir ersetzen den Koeffizienten 1 im höchstwertigen Polynomglied des Divisors G(x) durch 0.

2, Algorithmus:
Er ermittelt fortlaufend die MSBs (Most Significant Bits) durch XOR-Verknüpfung des augenblicklich höchstwertigen Bits im Bitstrom C(x) und dem höchstwertigen Bit des Schieberegisters solange bis alle Bits in C(x) bearbeitet sind.

a, Stellt der Algorithmus **MSB=1** fest, berechnet er:
- Augenblicklichen Inhalt des Schieberegisters R(x) linksschieben: $R(x)<< \rightarrow R'(x)$
- addiere (modulo2) zu $R'(x)$ die Maske m(x): $R'(x) + m(x)$
- das Ergebnis ist das neue R(x).

Beispiel: Vergleiche mit **Bild 87**; dort ist MSB=1. Deshalb folgt

$R(x) = 0x^2+0x^1+0x^0$;augenblickliches R(x)
$R'(x) = 0x^2+0x^1+0x^0$;R(x)<<

$$\begin{array}{rl} R'(x) + m(x) = & 0x^2+0x^1+0x^0 \\ + & \underline{0x^2+1x^1+1x^0} \\ & \mathbf{0x^2+1x^2+1x^2} \end{array}$$;neues R(x)

Das neue $R(x)= 0x^2+1x^2+1x^2$ ist mit dem Ergebnis der modulo2 Hardware-Division identisch. Vergleiche mit **Bild 87a**.

b, Stellt der Algorithmus **MSB=0** fest, schiebt er:
- lediglich den augenblicklichen Inhalt des Schieberegisters um 1 Bitposition nach links.

Beispiel: Vergleiche mit **Bild 87b**; dort ist MSB=0. Deshalb folgt

$R(x) = 0x^2+1x^1+1x^0$;augenblickliches R(x)
$R(x)<< \rightarrow 1x^2+1x^1+0x^0$;neues R(x)

Das neue $R(x)= 1x^2+1x^2+0x^2$ ist mit dem Ergebnis der modulo2 Hardware-Division identisch. Vergleiche mit **Bild 87c**.

4.7.1 frame check sequence-Berechnung für 1 Oktet

Das folgende C-Programm (siehe Bild 88) wird sowohl von der Informationsquelle als auch von der Informationssenke benutzt. Es realisiert den oben beschriebenen Algorithmus mit dem realen Generator-Polynom CRC-16, und liefert eine 16-Bit **f**rame **c**heck **s**equence (fcs) für einen Bitstrom, der aus 8 Bits (1 Oktet) besteht.
Das CRC-16 Generator-Polynom hat den Grad k=16, besteht also aus 17 Bits und hat folgende Form:

CRC-16:
$\mathbf{1x^{16}+0x^{15}+0x^{14}+0x^{13}+1x^{12}+0x^{11}+0x^{10}+0x^{9}+0x^{8}+0x^{7}+0x^{6}+1x^{5}+0x^{4}+0x^{3}+0x^{2}+0x^{1}+1x^{0}}$

Daraus ergibt sich die Maske (Zur Erinnerung: Den Koeffizienten an der höchstwertigen Stelle x^{16} gleich 0 setzen):

mask:
$\mathbf{0x^{16}+0x^{15}+0x^{14}+0x^{13}+1x^{12}+0x^{11}+0x^{10}+0x^{9}+0x^{8}+0x^{7}+0x^{6}+1x^{5}+0x^{4}+0x^{3}+0x^{2}+0x^{1}+1x^{0}}$

Hinweis: Die IEEE 802.4 empfiehlt ein Generator-Polynom mit dem Grad k=32.

Das C-Programm enthält die Funktion **crc**, die mit drei Parametern aufgerufen wird:

- unsigned short int fcs ;den temporären Rest, der den anfänglichen Wert 0 hat und ;in einem 16 Bit-Datentyp enthalten ist,
- unsigned short int c ;den Bitstrom (C(x) = 1 Oktet), der sich im Low-Byte eines ;16 Bit-Datentyps aufhält, und
- unsigned short int mask ;der Maske, die ebenfalls in einem 16 Bit-Datentyp ;aufbewahrt ist.

```
unsigned short int crc(unsigned short int fcs,unsigned short int c,unsigned short int mask)
{
 unsigned char i;

 c<<=8;
 for(i=0;i<8;i++)
     {
      if((fcs ^ c) & 0x8000) fcs = (fcs<<1)^mask;
      else fcs<<=1;
      c<<=1;
     }
 return fcs;
}
```

Bild 88: fcs-Berechnung für 1 Oktet

Die Funktion beginnt ihre Arbeit, indem sie als erstes die Variable c um 8 Bitpositionen nach links schiebt, um damit den Bitstrom (1 Oktet), der sich ja im Low-Byte der Variablen c aufhält, an die höchstwertige Stelle x^{16} des Generator-Polynoms zu bringen: **c<<=8**;

Bei einer anfänglichen fcs=0 entsteht dann folgende Situation; siehe Bilder 89 und 89a:

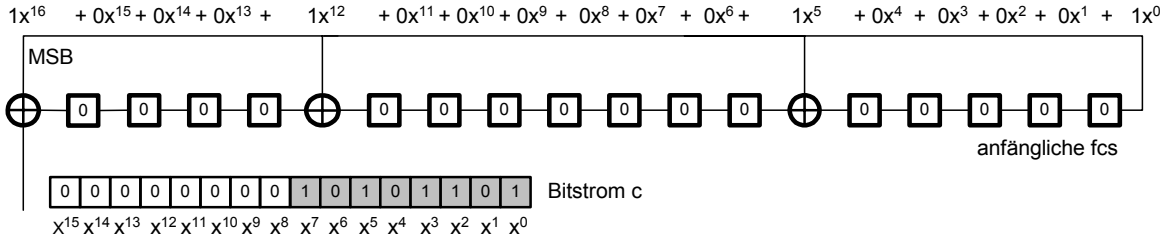


Bild 89: Bitstrom c vor dem Linksschieben

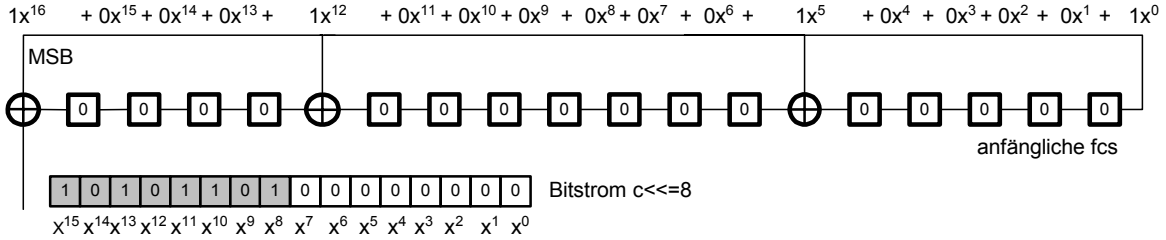


Bild 89a: Bitstrom c nach dem Linksschieben

Die Funktion crc setzt ihre Arbeit fort, und bearbeitet in der for-Schleife **for(i=0;i<8;i++)** Bit für Bit die Variable c. Für jedes Bit ermittelt sie zunächst das MSB: **(fcs^c) & 0x8000**

Beispiel gemäß Situation nach Bild 89a:

1,

fcs = 0000 0000 0000 0000
^c = **1010 1101** 0000 0000
fcs^c = 1010 1101 0000 0000

2,

fcs^c = 1010 1101 0000 0000
& 8000 = 1000 0000 0000 0000
(fcs^c) & 8000 = **1**000 0000 0000 0000

Ergebnis: MSB=**1**

Die crc-Simulationssoftware liefert auf ihre Weise das gleiche Ergebnis wie die modulo2-Divisionshardware (vgl. mit Bild 89b).

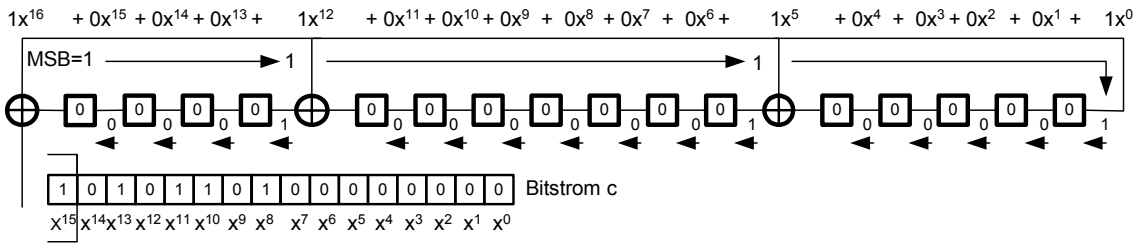


Bild 89b: Die modulo2 Divisions-Hardware liefert MSB=1

Die Divisions-Hardware schließt den ersten Schritt ab, und liefert durch Linksschieben der neuen Informationen das erste temporäre fcs: Siehe Bild 89c.

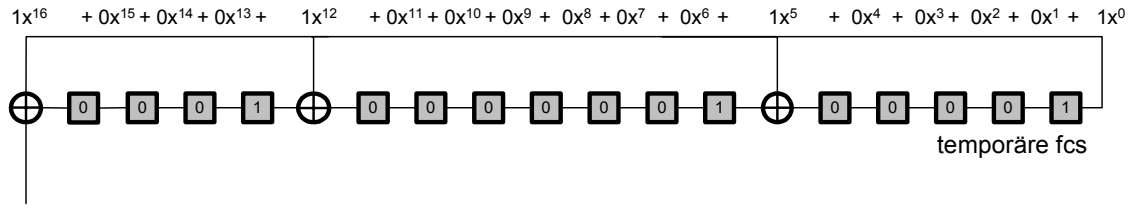


Bild 89c: Temporäre fcs nach dem ersten Schritt

Zum gleichen Ergebnis kommt die crc-Simulationssoftware. Wegen MSB=1 folgt:
fcs = (fcs<<1)^mask; (vgl. mit Bild 88)

3, fcs = 0000 0000 0000 0000

fcs<<1 = 0000 0000 0000 0000
^mask = 0001 0000 0010 0001
fcs = 0001 0000 0010 0001

Die Funktion crc setzt ihre Arbeit fort, und bearbeitet nach **c<<=1** das nächste Bit im Bitstrom. Vergleiche mit Bild 88.

Die modulo2 Divisions-Hardware bearbeitet ebenfalls das nächste Bit im Bitstrom, nämlich $0x^{14}$, und liefert MSB=0. Siehe Bild 89d.

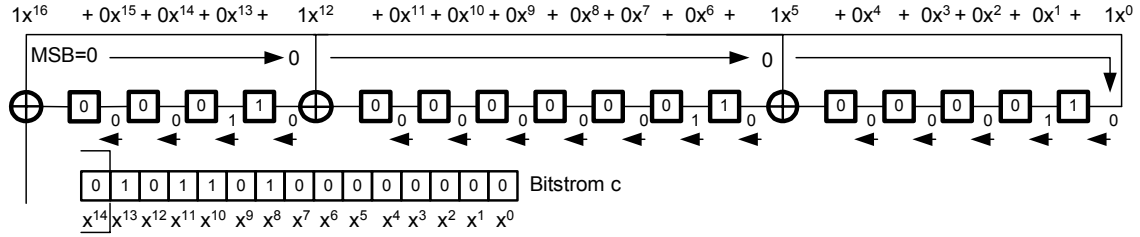


Bild 89d: Die modulo2 Divisions-Hardware liefert MSB=0

Durch das übliche Linksschieben um eine Bitposition kommt sie zur nächsten temporären fcs. Siehe Bild 89e.

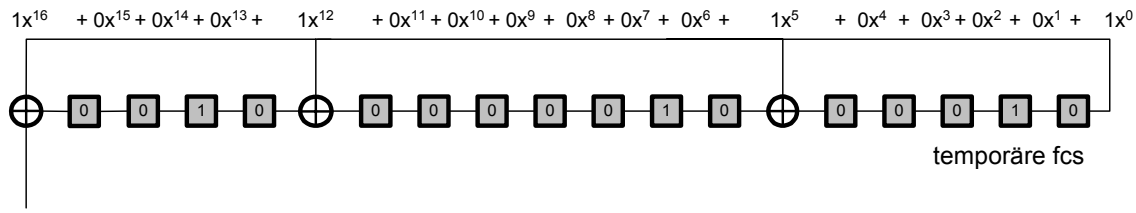


Bild 89e: Temporäre fcs nach dem zweiten Schritt

Die crc-Simulationssoftware kommt zu dem gleichen Ergebnis.

Gemäß Situation nach Bild 89d gilt:

1,

fcs	=	0001 0000 0010 0001
^c	=	**0101 1010** 0000 0000
fcs^c	=	0100 1010 0010 0001

2,

fcs^c	=	0100 1010 0010 0001
& 8000	=	1000 0000 0000 0000
(fcs^c) & 8000	=	**0**000 0000 0000 0000

Ergebnis: MSB=0

Wegen MSB=0 folgt: **fcs<<1**; (vergleiche mit Bild 88)

3,

fcs	=	0001 0000 0010 0001
fcs<<1	=	**0010 0000 0100 0010**

Die Funktion crc setzt ihre Arbeit fort, und bearbeitet nach c<<=1 das nächste Bit im Bitstrom. Dies geschieht solange bis die for-Schleife beendet ist. Danach liefert crc die endgültige fcs für 1 Oktet**: return fcs;** (vgl. Bild 88).

4.7.2 frame check sequence-Berechnung für n Oktets

Angenommen, ein Bitstrom besteht aus n=64 Oktets. Dann sendet die Nachrichtenquelle zuerst Oktet 0 und zuletzt Oktet 63. Und in jedem Oktet überträgt der Sender zuerst das Bit 2^0 und zuletzt das Bit 2^7. In der gleichen Reihenfolge in der die Oktets den Sender verlassen kommen sie beim Empfänger an, also zuerst Oktet 0 und zuletzt Oktet 63. Siehe Bild 90.

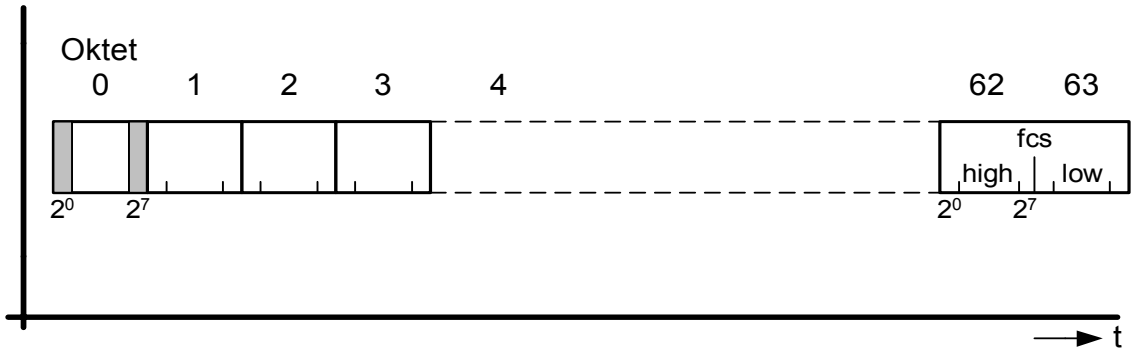


Bild 90: Zeitliches Eintreffen der Oktets beim Empfänger

Benutzt der Empfänger für die fcs-Berechnung die modulo2 Divisions-Hardware, dann bedeutet dies, dass der Empfänger die individuellen Oktets der Divisions-Hardware in der gleichen Reihenfolge zuführen muss wie er sie empfangen hat. Also zuerst Oktet 0, dann Oktet 1 und zum Schluss Oktet 63. Mit anderen Worten, die einzelnen Oktets stellen sich in Summe wie ein **einziger** Bitstrom mit 64•8 Bits = 512 Bits dar.
Die modulo2 Divisions-Hardware ermittelt die fcs für jedes einzelne Oktet und beginnt mit dem höchstwertigen Bit 2^7 und endet mit dem niederwertigen Bit 2^0.

So nimmt im Bitstrom mit 512 Bits das Bit 2^7 im Oktet 0 die höchstwertige Stelle 2^{511} und Bit 2^0 im Oktet 63 die niederwertige Stelle 2^0 ein.

Zur Bestimmung der MSBs für n=64 Oktets wird deshalb mit dem höchstwertigen Bit 2^{511} begonnen und mit dem niederwertigsten Bit 2^0 aufgehört. Siehe Bild 91.
Anmerkung: Die fcs ≠ 0 ermittelt die Nachrichtenquelle und speichert die High- und Low-Bytes der fcs in der richtigen Reihenfolge (zuerst High-Byte, dann Low-Byte) in den Oktets 62 und 63.

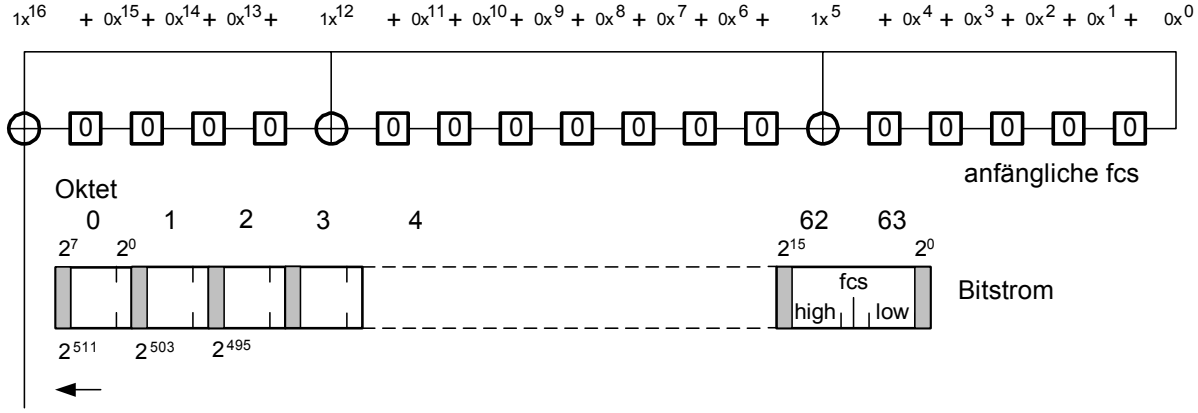


Bild 91: Bitstrom mit n=64 Oktets

Die modulo2 Divisions-Hardware beginnt ihre Arbeit mit dem anfänglichen Schieberegisterinhalt fcs=0 und ermittelt ab Bit 2^{511} des Bitstroms
1, für Oktet 0 die frame check sequence **fcs_0**.

Danach bearbeitet sie Oktet 1, und benutzt als anfänglichen Wert den derzeitigen Schieberegisterinhalt fcs_0. Sie ermittelt ab Bit 2^{503} des Bitstroms
2, für Oktet 0 und Oktet 1 die frame check sequence **fcs_1**.

Anschließend bearbeitet sie Oktet 2, und benutzt als anfänglichen Wert den derzeitigen Schieberegisterinhalt fcs_1. Sie ermittelt ab Bit 2^{495} des Bitstroms
3, für Oktet 0, Oktet 1 und Oktet 2 **fcs_2** .....usw.

Nach der Bearbeitung von 64 Oktets ist schließlich die endgültige fcs bestimmt und **hoffentlich 0**.
Der beschriebene Algorithmus ist leicht durch Software zu simulieren. Es ist lediglich die Funktion crc in eine for-Schleife zu „packen“, die 64mal durchlaufen wird. Siehe Bild 92.

```
fcs=0;                                    /*  fcs berechnen  */
for(i=0;i< max;i++)
     fcs=crc(fcs, puffer[i],mask);
```

Bild 92: fcs-Berechnung für 64 Oktets

In der Simulations-Software sind mit **#define max 64** (im Bild nicht dargestellt) die Anzahl der Oktets=64 bestimmt. Außerdem gibt es dort einen puffer[i], der die empfangenen Oktets aufsammelt und somit den gesamten Bitstrom enthält. Oktet für Oktet, beginnend mit Oktet 0 (i=0), arbeitet die Simulations-Software den Bitstrom ab, und liefert am Ende die frame check sequence fcs des gesamten übertragenen Bitstroms.