1.3.1.3 Cyclic Redundancy Code (CRC)

Hat die Receive Machine die MAC PDU empfangen, ist nicht garantiert, daß alle Bits unbeschädigt angekommen sind. So ist die hardware-basierte Fehlererkennung durch den UART, speziell durch die Schwächen des Priortätsüberwachungsschemas, in seiner Leistungsfähigkeit eingeschränkt (siehe 1.3.1.1).
In der Praxis wird daher oft ein anderes oder auch zusätzliches Modell benutzt, nämlich der **Polynomcode**. Er ist auch als zyklischer Redundanzcode oder CRC (cyclic redundancy code) bekannt. Polynomcodes basieren 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

$$\begin{matrix} 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{matrix}$$

als Polynom folgendermaßen 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 ein UART einen Frame (MAC PDU) überträgt, dann kann der korrespondierende Bitstrom als Polynom T(x) betrachtet werden, mit dem einige Berechnungen durchführbar sind. Von Interesse ist dabei 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 Restpolynmom R(x). Ist R(x) = 0, dann ist der Dividend durch den Divisor teilbar. Ist $R(x) \neq 0$, dann ist der Dividend durch den Divisor nicht teilbar. Was kann man mit dieser Erkenntnis anfangen? Sie ist der Schlüssel zu einer Methode, mit der sich Übertragungsfehler erkennen lassen. Betrachten wir dazu ein Beispiel aus dem Zehnersystem. Siehe Bild 24.

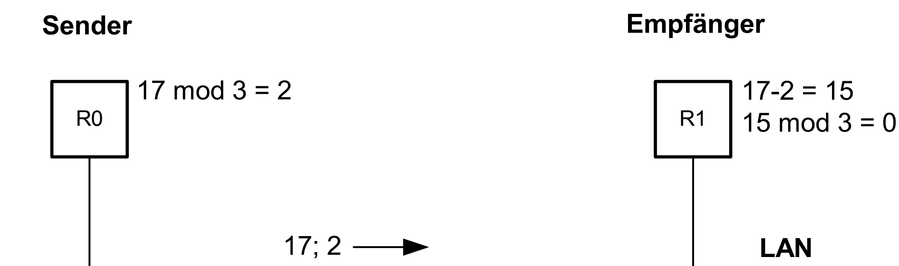

Bild 24: Methode zur Erkennung von Übertragungsfehlern

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 über 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 Rechnung. Eine allgemeine Betrachtung über die modulo m Rechnung ist daher von Vorteil.

1.3.1.3.1 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 aus.

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.

modulo 10 Rechnung:

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 Zahl r unterschieden. So gilt für das angegebene Beispiel

6 = {...,-14,-4,6,16,26,...}

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

r = r+k•m; k =...,-2,-1,0,+1,+2,...

Verifizierung:

6 = 6-2•10 = -14 (k = -2)
6 = 6-1•10 = -4 (k = -1)
6 = 6+0•10 = 6 (k = 0)
6 = 6+1•10 = 16 (k =+1)
6 = 6+2•10 = 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) sind es die Zahlen **0,1,.......9**. Im GF(2) sind es die Zahlen **0,1**.

1.3.1.3.2 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):

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

$$1x^4+1x^3+0x^2+1x^1+1x^0 : 1x^2+1x^1+1x^0 = 1x^2+0x^1+1x^0 \quad + \quad 0$$

$\underline{1x^4+1x^3+1x^2} \;\downarrow\;\downarrow$
$\quad\quad \mathbf{1x^2}+1x^1+1x^0$
$\quad\quad \underline{1x^2+1x^1+1x^0}$
$\quad\quad$ Rest: 0

Im Beispiel 1 ist wegen R(x)=0 der Dividend T(x) durch den Divisor G(x) teilbar. Außerdem ist im Sinne der modulo m Rechnung zwischen dem „Polynom 0“ und dem Polynom $T(x)=1x^4+1x^3+0x^2+1x^1+1x^0$ nicht zu unterscheiden.

Beispiel 2:

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

Division T(x) durch G(x):

T(x) : G(x) = I(x) + R(x)

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

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

1.3.1.3.3 Fehlererkennung nach der Polynomcodemethode

Die Polynomcodemethode benutzt die Polynomdivision, wie wir sie anhand von zwei Beispielen kennengelernt haben. In Anlehnung an die Methode nach Bild 24, 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) teilen läßt:

Der detailierte 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:

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

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

Der entstandene Rest x+1 ist die gesuchte Prüfsumme, die auch als **frame check sequence** (fcs) bekannt ist.

- Ziehe den Rest, 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 klären. **Hinweis**: Weil in der modulo2 Arithmetik zwischen Subtraktion und Addition nicht unterschieden wird, erscheint in der folgenden Rechnung +R(x).

Beispiel Fortsetzung:

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

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

Das vom Sender berechnete Polynom C(x) schickt nun sein UART als Bitstrom durch das LAN zum Empfänger.

2, Empfänger:

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

Beispiel Fortsetzung:

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

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

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) $\neq$ 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 F(x)=G(x), dann 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 $\frac{C(x)}{G(x)}$ ist immer 0, und die modulo2 Division $\frac{F(x)}{G(x)} = \frac{G(x)}{G(x)}$ ebenfalls.
Das heißt, Fehler die dem Polynom G(x) entsprechen, können nicht entdeckt werden, alle anderen werden dagegen erkannt. Doch wie oft kommt dies vor? Trotz geringer Wahrscheinlichkeit ist es gut, zusätzliche Fehlerüberwachungssysteme zu benutzen (z.B. die Paritätsüberprüfung des UART).

1.3.1.3.4 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

$$C(x) \rightarrow C^{(1)}(x)$$

$$1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0 \rightarrow \mathbf{0x^6+1x^5+0x^4+0x^3+1x^2+1x^1+1x^0}$$

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

Beispiel: Fortsetzung

$$C^{(1)}(x) \quad : \quad G(x) \quad = \quad I(x) \quad + \quad R(x)$$

$$1x^5+0x^4+0x^3+1x^2+1x^1+1x^0 : 1x^3+0x^2+1x^1+1x^0 = 1x^2+0x^1+1x^0 + 0$$
$$\underline{1x^5+0x^4+1x^3+1x^2} \downarrow \downarrow$$
$$\mathbf{1x^3+0x^2}+1x^1+1x^0$$
$$\underline{1x^3+0x^2+1x^1+1x^0}$$
Rest: 0

$C^{(1)}(x)$ ist offensichtlich 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 Rotation 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 wohl der Grund für die Bezeichnung cyclic redundancy code. 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: $C^{(7)}(x) = \mathbf{C(x) = 1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0}$

1.3.1.3.5 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:

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

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

$$\underline{1x^6+0x^5+1x^4+1x^3} \downarrow \quad \downarrow \quad \downarrow$$

$$\mathbf{1x^3}+0x^2+1x^1+1x^0$$

$$\underline{1x^3+0x^2+1x^1+1x^0}$$

Rest: 0

Im Bild 25 ist diese Logik zu sehen.

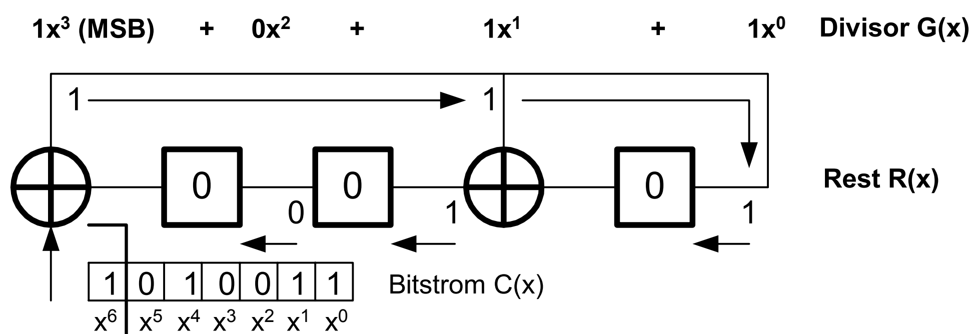

Bild 25: Modulo2 Divisions-Hardware

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, reiht 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 25. Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 1te Bearbeitungsschritt ist beendet. Siehe Bild 25a.

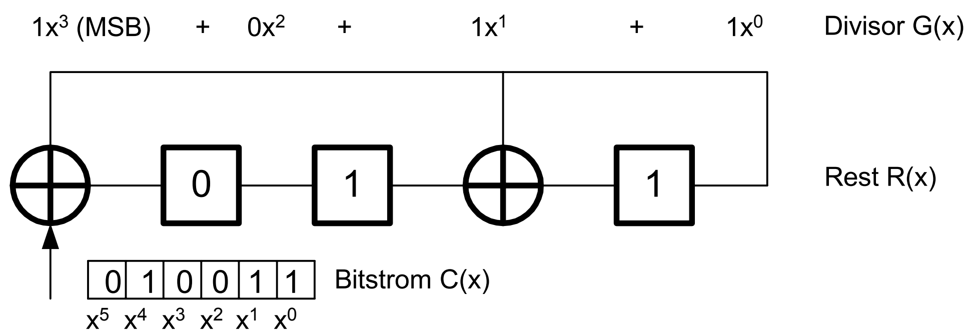

Bild 25a: Situation **nach** der Bearbeitung von $1x^6$ des Bitstroms C(x)

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 25b.

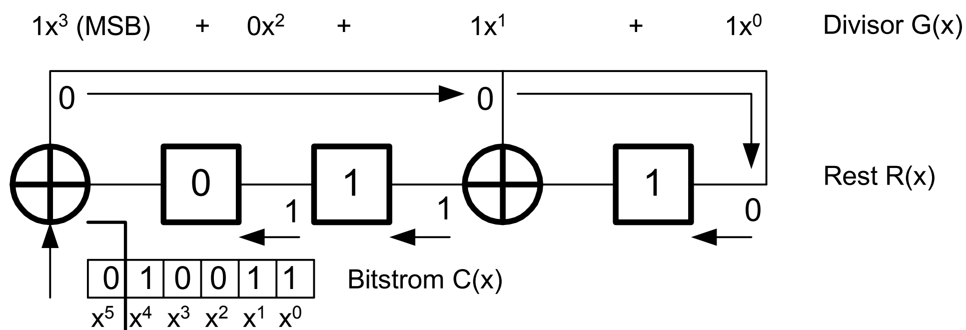

Bild 25b: Bearbeitung von Bit $0x^5$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 2te Bearbeitungsschritt ist beendet. Siehe Bild 25c.

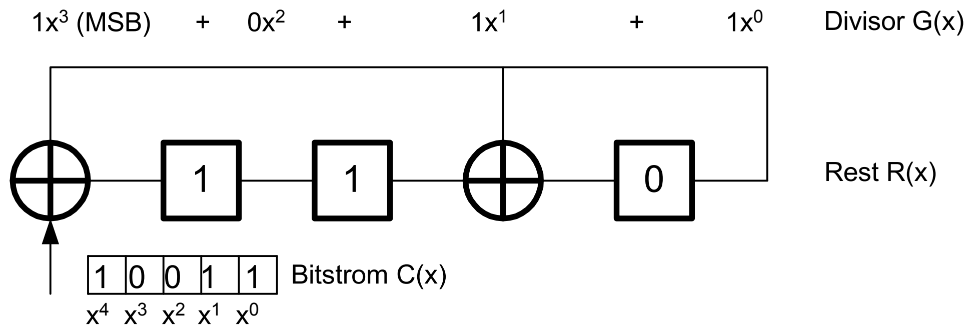

Bild 25c: Situation **nach** der Bearbeitung von $0x^5$ des Bitstroms C(x)

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)=$\mathbf{1x^2}$+$\mathbf{0x^1}$+$\mathbf{0x^0}$. Siehe Bild 25d.

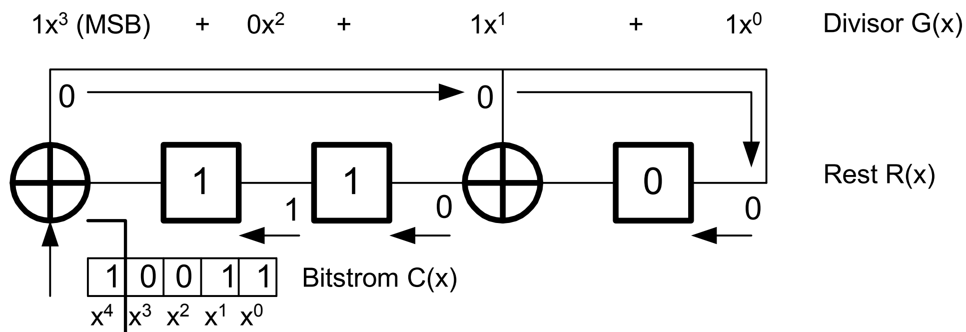

Bild 25d: Bearbeitung von Bit $1x^4$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 3te Bearbeitungsschritt ist beendet. Siehe Bild 25e.

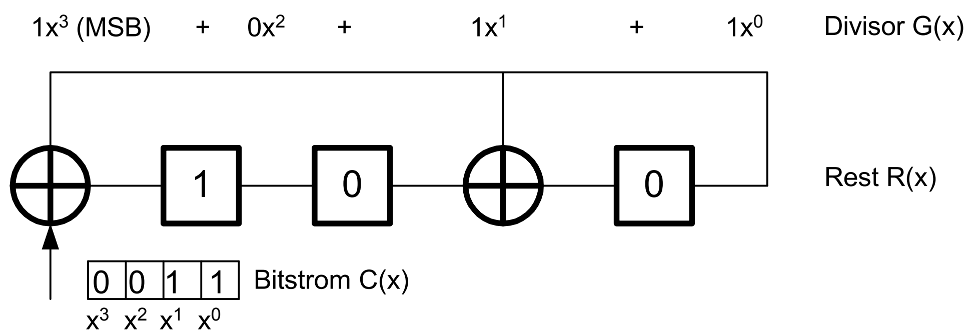

Bild 25e: Situation **nach** der Bearbeitung von $1x^4$ des Bitstroms C(x)

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 25f.

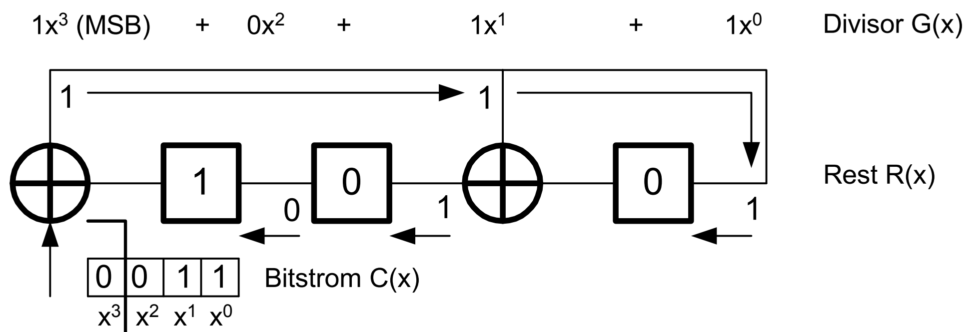

Bild 25f: Bearbeitung von Bit $0x^3$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 3te Bearbeitungsschritt ist beendet. Siehe Bild 25g.

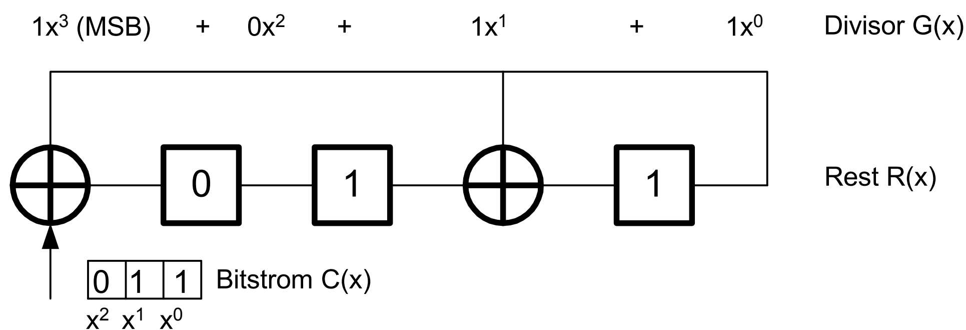

Bild 25g: Situation **nach** der Bearbeitung von $0x^3$ des Bitstroms C(x)

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 25h.

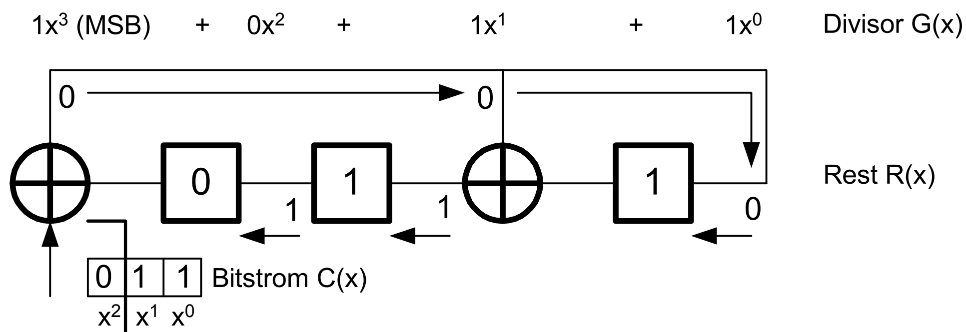

Bild 25h: Bearbeitung von Bit $0x^2$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 5te Bearbeitungsschritt ist beendet. Siehe Bild 25i.

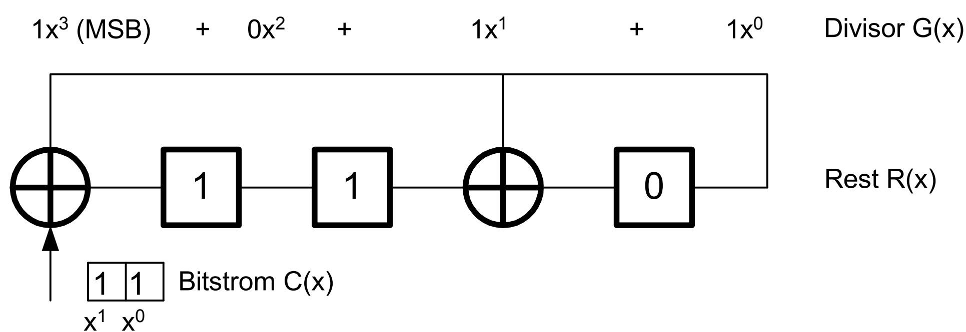

Bild 25i: Situation **nach** der Bearbeitung von $0x^2$ des Bitstroms C(x)

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 25j.

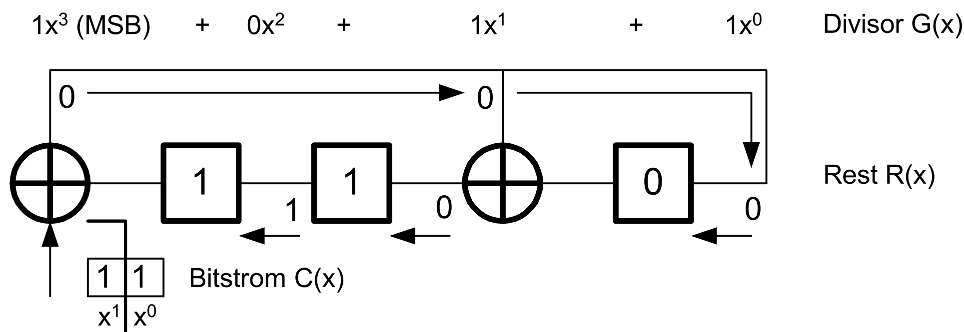

Bild 25j: Bearbeitung von Bit $1x^1$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 6te Bearbeitungsschritt ist beendet. Siehe Bild 25k.

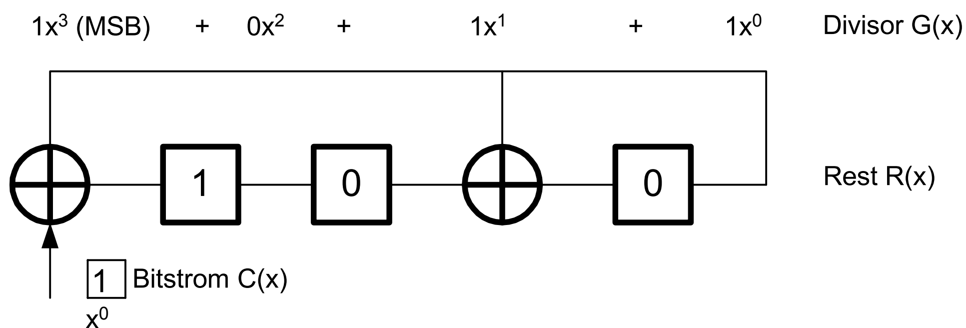

Bild 25k: Situation nach der Bearbeitung von $1x^1$ des Bitstroms C(x)

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)=$\mathbf{0x^2+0x^1+0x^0}$. Siehe Bild 25l.

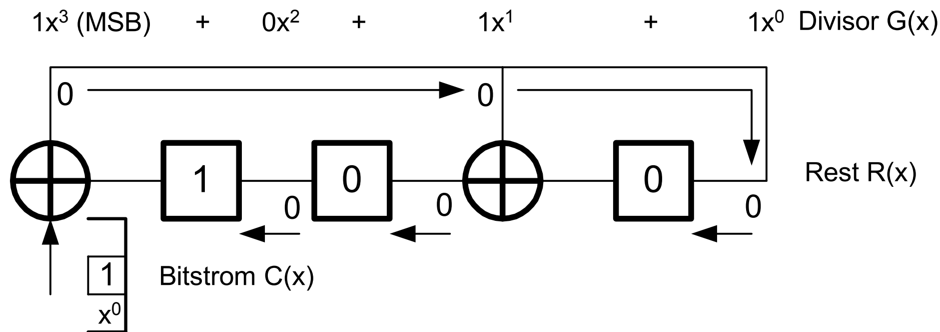

Bild 25l: Bearbeitung von Bit $1x^0$ des Bitstroms C(x)

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 25m.

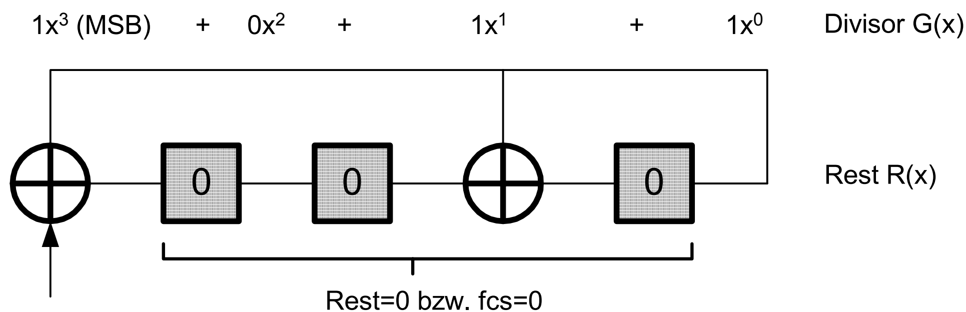

Bild 25m: Das Ergebnis der Hardware-Division C(x) modulo G(x)=0

1.3.1.3.6 Modulo2 Division und Software

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

1, Vorbereitung:

Bilde 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, ersetze den Koeffizienten 1 im höchstwertigen Polynomglied des Divisors durch 0.

2, Algorithmus:

Ermittle das MSB.

a, Wenn **MSB=1** berechne:

- 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: $R(x) = 0x^2+0x^1+0x^0$;augenblickliches R(x), vergleiche mit **Bild 25**
$R'(x) = 0x^2+0x^1+0x^0$;R(x)<<

$$R'(x) + m(x) = \begin{array}{r} 0x^2+0x^1+0x^0 \\ +\ \underline{0x^2+1x^1+1x^0} \\ \mathbf{0x^2+1x^2+1x^2} \end{array}$$;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 25a**.

b, Wenn **MSB=0**

- schiebe den augenblicklichen Inhalt des Schieberegisters um 1 Bitposition nach links.

Beispiel: $R(x) = 0x^2+1x^1+1x^0$;augenblickliches R(x), vergleiche mit **Bild 25b**
$R(x)<< \rightarrow 1x^2+1x^1+0x^0$;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 25c**.

1.3.1.3.7 fcs-Berechnung für 1 Oktet

Das folgende C-Programm (siehe Bild 26) wird sowohl von der Transmit Machine als auch von der Receive Machine des MAC-Layers benutzt. Es realisiert den oben beschriebenen Algorithmus mit dem realen Generator-Polynom CRC-16, und liefert eine 16-Bit frame check sequence 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} 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 (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 26: 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 27 und 27a:

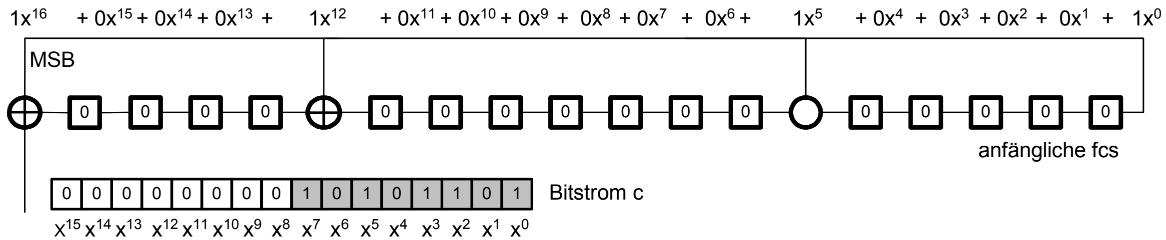

Bild 27: Bitstrom c vor dem Linksschieben

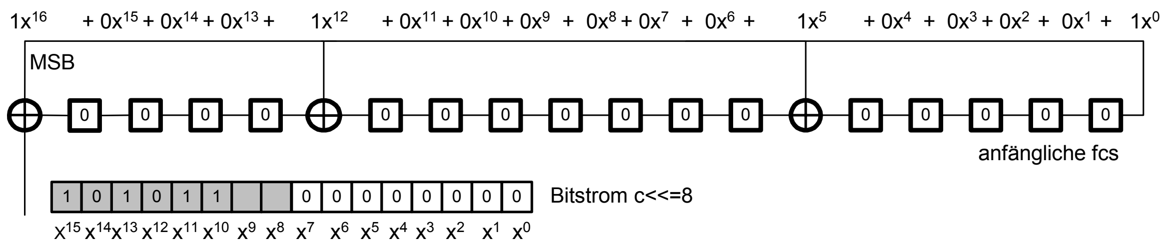

Bild 27a: Bitstrom c nach dem Linksschieben

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 27a:

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. Vergleiche mit Bild 27b.

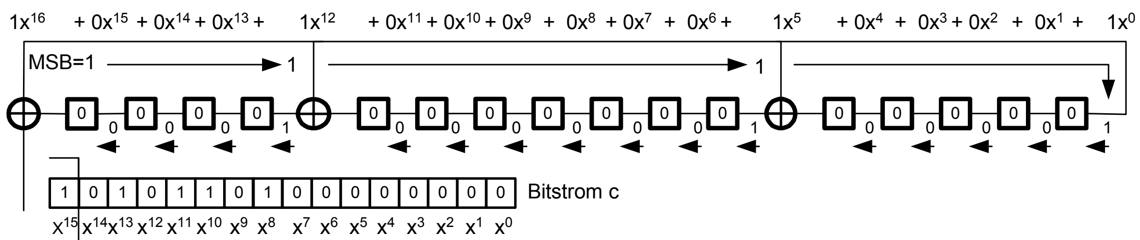

Bild 27b: Die modulo2 Divisions-Hardware liefert MSB=1

Die Divisions-Hardware schließt den ersten Schritt ab, und liefert durch Linksschieben der neuen Informationen das erste temporäre fcs: Siehe Bild 27c.

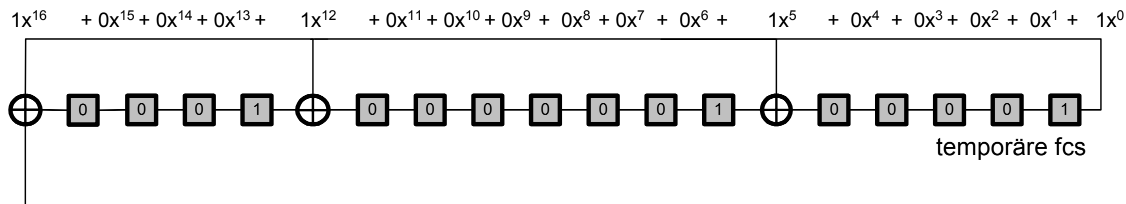

Bild 27c: Temporäre fcs nach dem ersten Schritt

Zum gleichen Ergebnis kommt die crc-Simulationssoftware. Wegen MSB=1 folgt:
fcs = (fcs<<1)^mask; (vergleiche mit Bild 26)

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 26.

Die modulo2 Divisions-Hardware bearbeitet ebenfalls das nächste Bit im Bitstrom, nämlich $0x^{14}$, und liefert MSB=0. Siehe Bild27d.

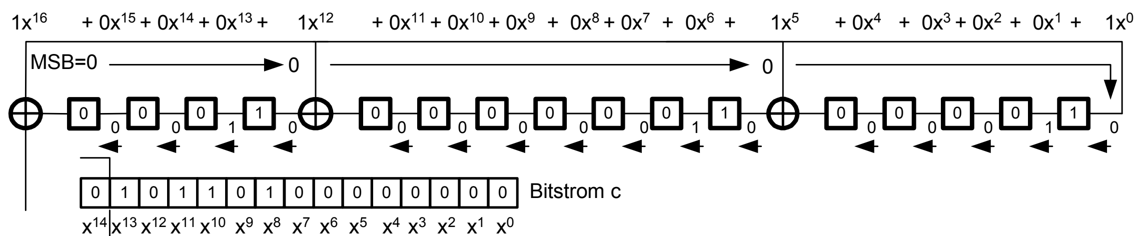

Bild 27d: Die modulo2 Divisions-Hardware liefert MSB=0

Durch das übliche Linksschieben um eine Bitposition kommt sie zur nächsten temporären fcs. Siehe Bild 27e.

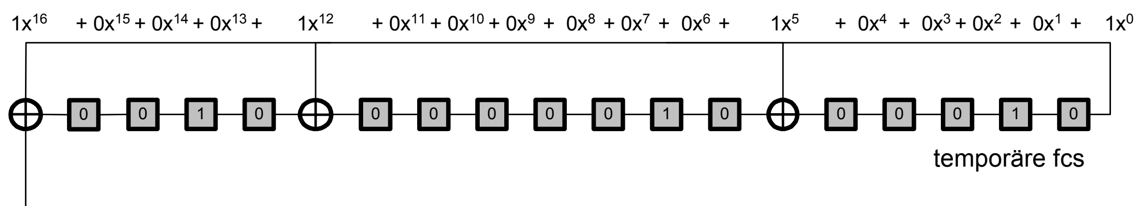

Bild 27e: Temporäre fcs nach dem zweiten Schritt

Die crc-Simulationssoftware kommt zu dem gleichen Ergebnis.

Gemäß Situation nach Bild 27d 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 26)

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;** (vergleiche Bild 26).

1.3.1.3.8 fcs-Berechnung für n Oktets

Wir erinnern uns, die Receive Machine empfängt die MAC PDU und speichert sie im rxm_puffer[14]. In welcher zeitlichen Reihenfolge kommen die Oktets beim Empfänger an? Sie werden von der Transmit Machine des Senders in das LAN eingespeist, und zwar zuerst Oktet 0 und zuletzt Oktet 13. Also kommen sie auch in dieser Reihenfolge beim Empfänger an. Siehe Bild 28.

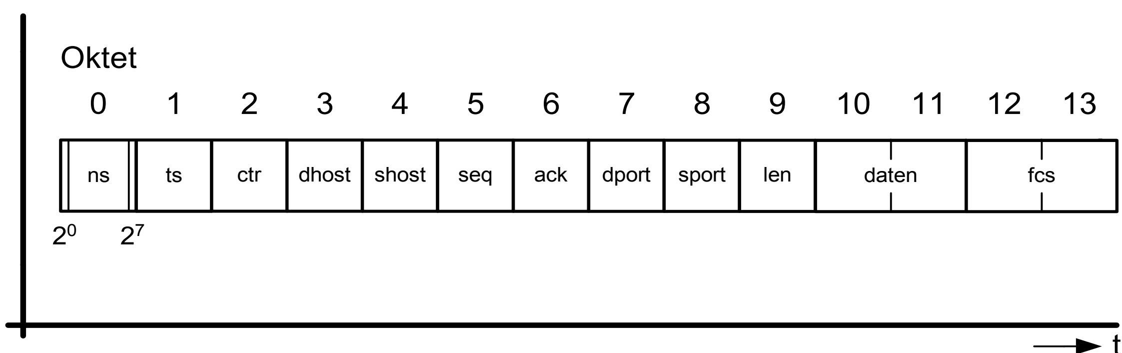

Bild 28: Zeitliches Eintreffen der Oktets beim Empfänger

Benutzt die Receive Machine für die fcs-Berechnung die modulo2 Divisions-Hardware, dann bedeutet dies, daß sie die individuellen Oktets der Hardware in der gleichen Reihenfolge zuführen muß. Also zuerst Oktet 0 und zuletzt Oktet 13. Mit anderen Worten, die MAC PDU stellt sich als Bitstrom mit 14•8 Bits = 112 Bits dar, wobei Bit 2^7 in Oktet 0 die höchstwertige Stelle 2^{111} einnimmt, und Bit 2^0 in Oktet 13 die niederwertigste Stelle 2^0.

Für die Ermittlung der MSBs für n=14 Oktets wird deshalb mit dem höchstwertigen Bit 2^{111} begonnen und mit dem niederwertigsten Bit 2^0 aufgehört. So ist es einsichtig, warum das High-Byte der fcs im niederwertigeren Oktet 12 und das Low-Byte der fcs im höherwertigeren Oktet 13 der MAC PDU hinterlegt ist. Siehe Bild 29.
Anmerkung: Die fcs ≠ 0 ermittelt die Transmit Machine des Senders und speichert die High- und Low-Bytes der fcs in den richtigen Oktets 12 und 13.

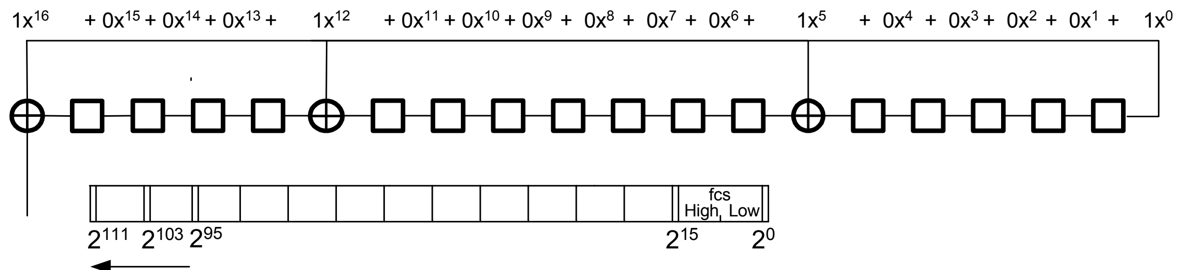

Bild 29: Bitstrom für n=14 Oktets

Die modulo2 Divisions-Hardware beginnt ihre Arbeit mit dem anfänglichen Schieberegisterinhalt fcs=0 und ermittelt ab Bit 2^{111} des Bitstroms

1, für Oktet 0 **fcs_0**.

Danach bearbeitet sie Oktet 1, und benutzt als anfänglichen Wert den derzeitigen Schieberegisterinhalt fcs_0. Sie ermittelt ab Bit 2^{103} des Bitstroms

2, für Oktet 0 und Oktet 1 **fcs_1**.

Anschließend bearbeitet sie Oktet 2, und benutzt als anfänglichen Wert den derzeitigen Schieberegisterinhalt fcs_1. Sie ermittelt ab Bit 2^{95} des Bitstroms

3, für Oktet 0, Oktet 1 und Oktet 2 **fcs_2** usw.

Nach der Bearbeitung von 14 Oktets ist schließlich die endgültige fcs für die MAC PDU 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“. Siehe Bild 30.

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

Bild 30: fcs-Berechnung für 14 Oktets

Das gleiche Simulations-Software benutzt die Receive Machine (vergleiche mit Bild 11). Dort ist mit **#define rxm_max 14** die Anzahl der Oktets=14 bestimmt. Oktet für Oktet, beginnend mit Oktet 0 (i=0), arbeitet die Simulations-Software den Bitstrom ab, und liefert am Ende die fcs der MAC PDU.