CRC Einführung

Version: 0.0.1
Datum: 04.03.2013
Autor: Werner Dichler

Inhalt

Polynom-Division

Allgemein

$$p(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \cdots + a_1 \cdot x^1 + a_0 \cdot x^0$$
$$q(x) = b_m \cdot x^m + b_{m-1} \cdot x^{m-1} + \cdots + b_1 \cdot x^1 + b_0 \cdot x^0$$
$$p(x) : q(x) = s(x) + \frac{p(x)}{r(x)}$$

Formel 1 – Polynom-Division

Der Dividend p(x) ist ein Polynom mit dem Grad n. Der Divisor q(x) ist ein Polynom mit dem Grad m. Das Ergebnis besteht aus einem "Ganzzahl"-Quotienten und einem Rest, der sich durch den Divisor nicht mehr teilen lässt.

Beispiel

$$\frac{p(x)}{q(x)} = \frac{4x^5 - x^4 + 2x^3 + x^2 - x^0}{x^2 + x^0}$$

$$\begin{array}{rl}
4x^5 - x^4 + 2x^3 + x^2 - x^0 : x^2 + x^0 = 4x^3 - x^2 - 2x^1 + 2x^0 & \\
\underline{-4x^5 - 4x^3} & \\
-x^4 - 2x^3 + x^2 - x^0 & \\
\underline{x^4 + x^2} & \\
-2x^3 + 2x^2 - x^0 & \\
\underline{2x^3 + 2x^1} & \\
2x^2 + 2x^1 - x^0 & \\
\underline{-2x^2 - 2x^0} & \\
2x^1 - 3x^0 & \text{Rest}
\end{array}$$

Formel 2 – Beispiel einer Polynom-Division

CRC Grundlagen

Allgemein

Die CRC-Berechnung (Cyclic Redundancy Code) wird für die Fehlererkennung bei der Datenübertragung verwendet. Das Ergebnis einer Berechnung liefert einen Hash-Wert über die gesendeten Daten. Der Empfänger erkennt eine fehlerhafte Übertragung aufgrund des ungleichen Hash-Wertes.

Als Grundlage der Hash-Wert-Berechnung dient die Polynom-Division. Die zu übertragenden Bits können als Polynom dargestellt werden. Wird dieses Polynom mit einem Generator-Polynom dividiert, so erhält man einen Divisions-Rest. Dieser Rest ist für "viele" Eingangs-Bit-Kombinationen unterschiedlich und kann daher als sehr gut als Hash-Wert verwendet werden.

$$\begin{aligned} 0x5C \;=\; 0101.1100 \;&=\; 0x^7 + 1x^6 + 0x^5 + 1x^4 + 1x^3 + 1x^2 + 0x^1 + 0x^0 \\ &=\; x^6 + x^4 + x^3 + x^2 \end{aligned}$$

Formel 3 – Polynom-Darstellung eines Bytes

Dyadische Polynom-Division

Für die CRC-Berechnung wird ein Generator-Polynom mit einem bestimmten Grad k festgelegt, z.B. $x^8 + x^5 + x^4 + x^1$. Die Eingangs-Daten werden als Polynom betrachtet und mit x^k multipliziert. Danach wird die Polynom-Division durchgeführt.

$$\begin{aligned} 0101.1100.0000.0000 \;&=\; (x^6 + x^4 + x^3 + x^2) \cdot x^8 \\ &=\; x^{14} + x^{12} + x^{11} + x^{10} \end{aligned}$$

Formel 5 – erweiterte Daten-Byte

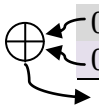
$$\begin{array}{lll} x^{14} + x^{12} + x^{11} + x^{10} & & : x^8 + x^5 + x^4 + x^1 \;=\; x^6 + x^4 - x^1 - x^0 \\ \underline{-x^{14} \quad - x^{11} - x^{10} \qquad\qquad - x^7} & & \\ \quad x^{12} \qquad\qquad - x^7 & & \\ \underline{-x^{12} \qquad - x^9 - x^8 \qquad\quad - x^5} & & \\ \quad -x^9 - x^8 - x^7 \quad - x^5 & & \\ \underline{+x^9 \qquad\quad + x^6 + x^5 \qquad + x^2} & & \\ \quad -x^8 - x^7 + x^6 \qquad + x^2 & & \\ \underline{\quad x^8 \qquad\quad + x^5 + x^4 \qquad + x^1} & & \\ \quad -x^7 + x^6 + x^5 + x^4 + x^2 + x^1 & \text{Rest} & \end{array}$$

Formel 6 – CRC Berechnung

Der CRC wird durch die binäre Darstellung des Restes gebildet. Man erhält $1111.0110 = 0xF6$.

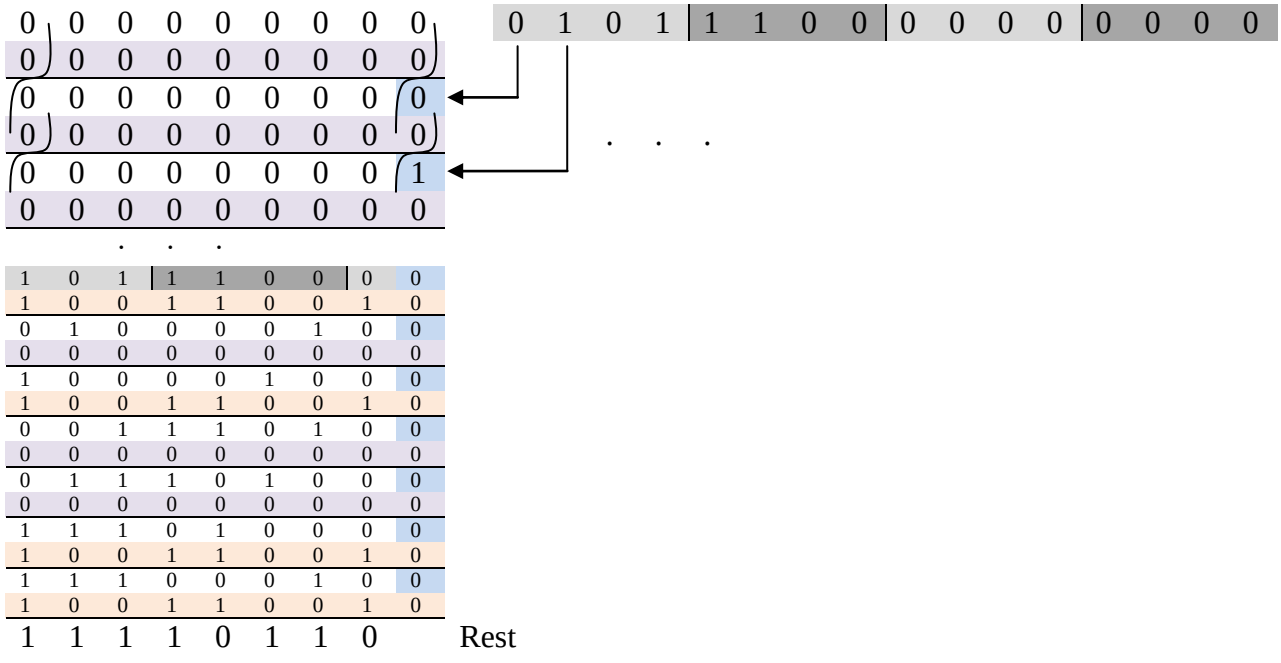
Vereinfachte Berechnung

Die dyadische Polynom-Division kann mit einfachen binären Rechenschritten durchgeführt werden. Man schreibt die Datenbits und deren Erweiterung tabellarisch an. Danach beginnt man von links (MSB) zu prüfen, ob der Divisor bereits enthalten ist. Ist er nicht enthalten, so schreibt man 0en an. Falls er enthalten ist, schreibt man den Divisor an. Danach werden die betrachteten Stellen mit XOR verknüpft. Das Zwischenergebnis wird an der rechten Seite mit einer Stelle vom Dividenden erweitert. Diesen Vorgang wiederholt man, bis der Rest ermittelt wurde (keine weiteren Stellen im Dividenden).



0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	:	1	0	0	1	1	0	0	1	0
0	0	0	0	0	0	0	0	0																
	1	0	1	1	1	0	0	0	0							=	0	1	0	1	0	0	1	1
	1	0	0	1	1	0	0	1	0															
		0	1	0	0	0	0	1	0	0														
		0	0	0	0	0	0	0	0	0														
			1	0	0	0	0	1	0	0	0													
			1	0	0	1	1	0	0	1	0													
				0	0	1	1	1	0	1	0	0												
				0	0	0	0	0	0	0	0	0												
					0	1	1	1	0	1	0	0	0											
					0	0	0	0	0	0	0	0	0											
						1	1	1	0	1	0	0	0	0										
						1	0	0	1	1	0	0	1	0										
							1	1	1	0	0	0	1	0	0									
							1	0	0	1	1	0	0	1	0									
								1	1	1	0	0	0	1	0	0								
								1	0	0	1	1	0	0	1	0								
									1	1	1	1	0	1	1	0	Rest							

Da für die CRC-Berechnung nur der Rest wichtig ist, kann der "Ganzzahlen"-Quotient vernachlässigt werden und muss nicht gespeichert werden. Die Berechnung erfolgt wie oberhalb. Es wurde nur eine andere Darstellung gewählt.



0	1	0	1	1	1	0	0	0	0	0	0	0	0	0										
0	0	0	0	0	0	0	0	0																
0	0	0	0	0	0	0	0	0																
0	0	0	0	0	0	0	0	0																
0	0	0	0	0	0	0	0	0																
0	0	0	0	0	0	0	0	1																
0	0	0	0	0	0	0	0	0																
			.	.	.																			
1	0	1	1	1	0	0	0	0																
1	0	0	1	1	0	0	1	0																
0	1	0	0	0	0	1	0	0																
0	0	0	0	0	0	0	0	0																
1	0	0	0	0	1	0	0	0																
1	0	0	1	1	0	0	1	0																
0	0	1	1	1	0	1	0	0																
0	0	0	0	0	0	0	0	0																
0	1	1	1	0	1	0	0	0																
0	0	0	0	0	0	0	0	0																
1	1	1	0	1	0	0	0	0																
1	0	0	1	1	0	0	1	0																
1	1	1	0	0	0	1	0	0																
1	0	0	1	1	0	0	1	0																
1	1	1	1	0	1	1	0		Rest															

Hardware Implementierung

Shift-Register

Die Polynom-Division kann mit Hilfe von Schieberegistern einfach implementiert werden. Die Anzahl der Register ist durch den Grad des Generator-Polynoms definiert. Der Ausgang des letzten Registers "aktiviert" die XOR-Verknüpfung mit dem Generator-Polynom. Die Datenbits werden von rechts nach links hinein geschoben (zu Vergleichen mit der obigen tabellarischen Berechnung). Sind alle Datenbits mit angehängten Nullen verarbeitet, so steht der CRC innerhalb der Register.

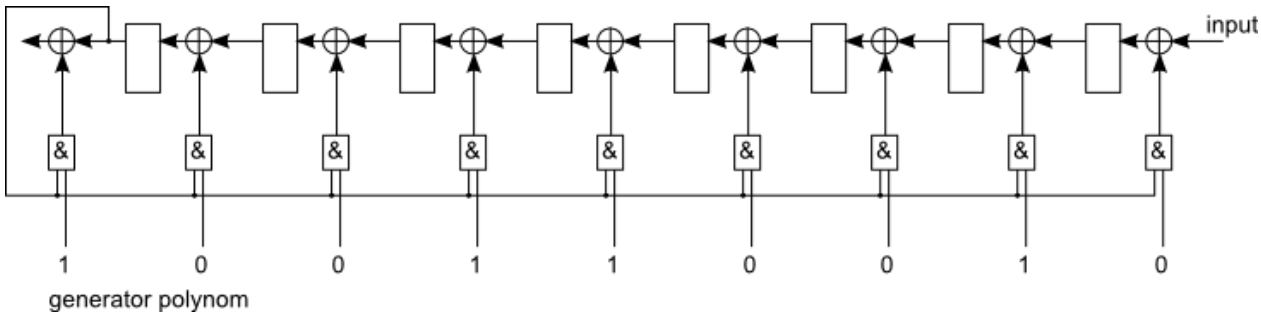


Bild 1 – Generelle Schiebe-Register Implementierung

Bei einem fixiertem Generator-Polynom kann die Implementierung noch weiter vereinfacht werden.

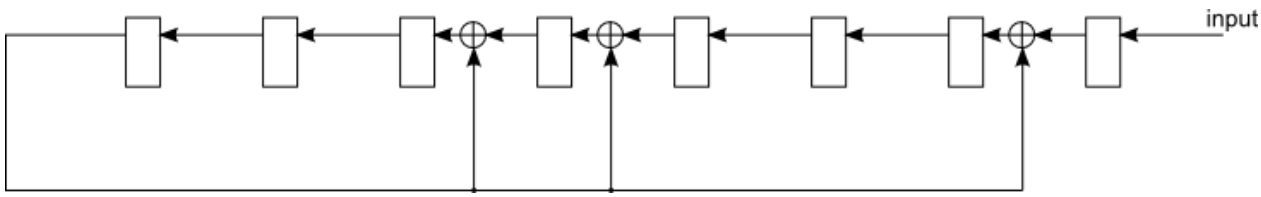


Bild 2 – Vereinfache Schiebe-Register Implementierung

		↓	↓			↓		
								XOR Verknüpfungen
0	0	0	0	0	0	0	0	Initialwerte der Register
0	0	0	0	0	0	0	0	E
0	0	0	0	0	0	0	1	i
0	0	0	0	0	0	1	0	n
0	0	0	0	0	1	0	1	g
0	0	0	0	1	0	1	1	a
0	0	0	1	0	1	1	1	n
0	0	1	0	1	1	1	0	g
0	1	0	1	1	1	0	0	
1	0	1	1	1	0	0	0	
0	1	0	0	0	0	1	0	
1	0	0	0	0	1	0	0	
0	0	1	1	1	0	1	0	
0	1	1	1	0	1	0	0	
1	1	1	0	1	0	0	0	
1	1	1	0	0	0	1	0	
1	1	1	1	0	1	1	0	Rest → CRC

Linear-Feedback-Shift-Register

Die Hardware Realisierung kann noch weiter vereinfacht werden, indem man eine LFSR-Struktur verwendet. Das hat den Vorteil, dass die Angehängten Nullen nicht mehr nötig werden.

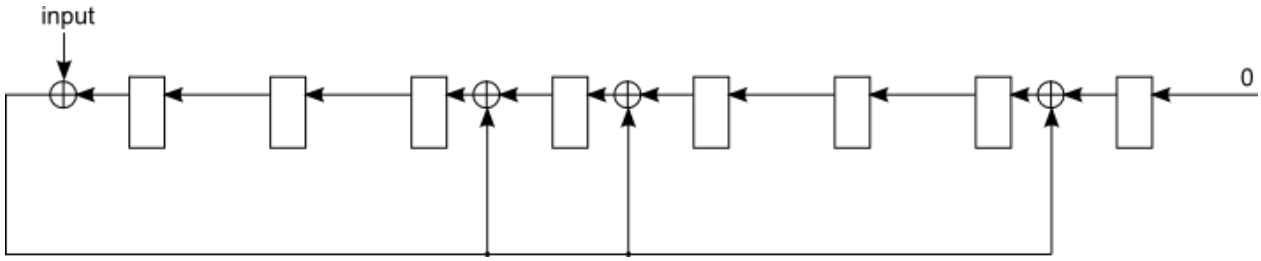


Bild 3 – Linear-Feedback-Shift-Register Implementierung

				↓	↓			↓		XOR Verknüpfungen
Eingang	Eingang XOR letzte Register	0	0	0	0	0	0	0	0	Initialwerte der Register
0	0	0	0	0	0	0	0	0	0	
1	1	0	0	1	1	0	0	1	0	
0	0	0	1	1	0	0	1	0	0	
1	1	1	1	1	1	1	0	1	0	
1	0	1	1	1	1	0	1	0	0	
1	0	1	1	1	0	1	0	0	0	
0	1	1	1	1	0	0	0	1	0	
0	1	1	1	1	1	0	1	1	0	Rest → CRC

Umgekehrte Implementierung

Bei den bisherig betrachteten Implementierungen wird bei der CRC-Ermittlung mit dem MSB der Eingangsdaten begonnen. Da einige Übertragungskanäle das LSB zuvor übertragen, müsste man den Datenstrom vor der CRC-Ermittlung puffern.
Da bei der CRC-Berechnung der tatsächliche Wert der Eingangsdaten irrelevant ist, kann grundsätzlich auch beim LSB begonnen werden. Damit die Schiebe-Richtung mit der Eingangsdaten-Richtung übereinstimmt, wird die Implementierung umgekehrt. Diese Umkehrung hat keinen Einfluss auf die Berechnung. Würde man wieder mit dem MSB beginnen, so erhält man den selben CRC-Wert wie bei der bisherigen Implementierung. Er ist nur umgekehrt in den Registern abgelegt.
Das wiederum heißt, dass eine Implementierung, welche beim LSB anfängt, immer einen anderen CRC liefert, als würde man mit dem MSB beginnen.

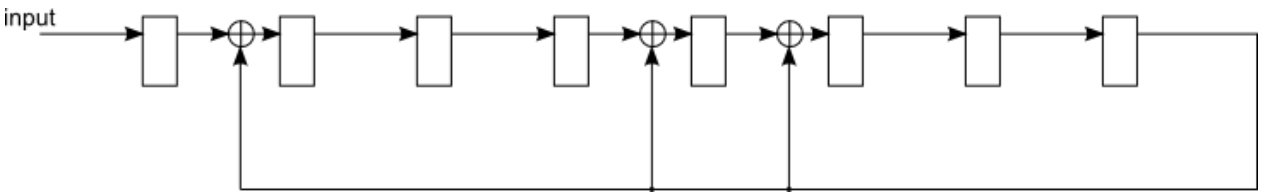


Bild 4 – Reverse Shift-Register Implementierung

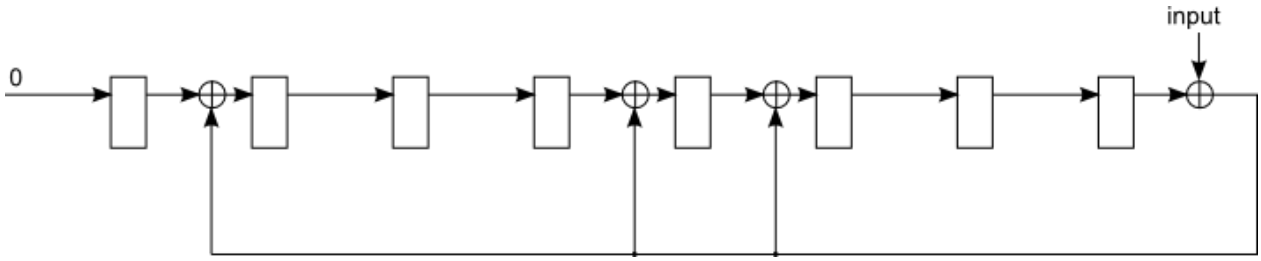


Bild 5 – Reverse Linear-Feedback-Shift-Register Implementierung

Ein Generator-Polynom mit 0x32 (x^8 wird nicht angegeben) hat ein umgekehrtes Generator-Polynom mit 0x4C (1.0011.0010 → 0100.1100.1).

Software Implementierung

Bit-Für-Bit Implementierung

Dieser Implementierung der CRC Berechnung folgt dem gleichen Ansatz, wie die HW Realisierung durch Schiebe-Register. Für sämtliche Strukturen der Schiebe-Register, kann auch eine Software-Funktion abgeleitet werden. Dabei wird ebenso jedes Eingangs-Bit einzeln verarbeitet.

Forward-CRC – SR

Bei der einfachen Schiebe-Register Implementierung, müssen die Daten ebenso mit 0en erweitert werden, um die CRC-Berechnung abzuschließen. Das Erweitern wird bereits in der Funktion durchgeführt (letzte for-Schleife).

```
#include <cstdlib>

/*****************************************************************************/
/* shift register implementation (forward)                                   */
/*****************************************************************************/
unsigned __int8 calc_forward_CRC8_SR(unsigned __int8 const *data, int length)
{
    unsigned __int8 reg         = 0x00;
    unsigned __int8 poly        = 0x32;
    unsigned __int8 polyOrder   = 8;
    unsigned __int8 currentByte = 0x00;

    for(int i=0; i<length*8; i++)
    {
        if (i%8 == 0)
        {
            currentByte = *(data + i/8);
        }
        unsigned __int8 inputBit = ((currentByte & 0x80) >> 7);
        unsigned __int8 highBit  = (reg & 0x80);

        currentByte = currentByte << 1;
        reg         = (reg << 1) | inputBit;

        if (highBit)
        {
            reg = (reg ^ poly);
        }
    }

    for (int i=0; i<polyOrder; i++)
    {
        unsigned __int8 highBit  = (reg & 0x80);

        reg = (reg << 1);

        if (highBit)
        {
            reg = (reg ^ poly);
        }
    }

    return reg;
}
```

Quelltext 1 – Forward-CRC mit Schiebe-Register Implementierung

Forward-CRC – LFSR

Bei der Linear-Feedback-Shift-Register Implentierung müssen nur die Eingangsbits, ohne weitere 0en, verarbeitet werden. Aufgrund dieser Zeitersparniss, wird üblicherweise dieser Algorithmus verwendet.

```
#include <cstdlib>

/*****************************************************************************/
/* linear feedback shift register implementation (forward)                   */
/*****************************************************************************/
unsigned __int8 calc_forward_CRC8_LFSR(unsigned __int8 const *data, int length)
{
    unsigned __int8 reg         = 0x00;
    unsigned __int8 poly        = 0x32;
    unsigned __int8 currentByte = 0x00;

    for(int i=0; i<length*8; i++)
    {
        if (i%8 == 0)
        {
            currentByte = *(data + i/8);
        }
        unsigned __int8 inputBit = currentByte & 0x80;
        currentByte              = currentByte << 1;

        if ((reg & 0x80)^(inputBit))
        {
            reg = (reg << 1)^(poly);
        }
        else
        {
            reg = (reg << 1);
        }
    }

    return reg;
}
```

Quelltext 2 – Forward-CRC mit LFSR-Implementierung

Reverse-CRC – LFSR

Werden die Daten mit LSB zuerst übertragen, so ist es sinnvoll die umgekehrte Implementierung zu verwenden. Die einfache Schiebe-Register-Implementierung, welche wiederum die abschließenden 0en benötigen würde, wird überlicherweise auch nicht verwendet. Deshalb ist unterhalb nur die LFSR-Implementierung zu finden.

```
#include <cstdlib>

/*************************************************************************/
/* reverse linear feedback shift register implementation                 */
/*************************************************************************/
unsigned __int8 calc_reverse_CRC8_LFSR(unsigned __int8 const *data, int length)
{
    unsigned __int8 reg         = 0x00;
    unsigned __int8 poly        = 0x4C;
    unsigned __int8 currentByte = 0x00;

    for(int i=0; i<length*8; i++)
    {
        if (i%8 == 0)
        {
            currentByte = *(data + ((((length-1)*8)-i)>>3));
        }
        unsigned __int8 inputBit = currentByte & 0x01;
        currentByte              = currentByte >> 1;

        if ((reg & 0x01)^(inputBit))
        {
            reg = (reg >> 1)^(poly);
        }
        else
        {
            reg = (reg >> 1);
        }
    }

    return reg;
}
```

Quelltext 3 – Reverse-CRC mit LFSR-Implementierung

Tabellen Implementierung

Die Tabellen Implementierung nutz die Eigenschaft aus, dass die XOR-Verknüpfungen zusammengelegt werden können. Statt dem Ausdruck ((a XOR b) XOR c) kann auch der Ausdruck (a XOR (b XOR c)) berechnet werden. Somit können mehrere XOR-Verknüpfungen zusammengefasst und diese auf einmal abgearbeitet werden. Die zusammengefassten XOR-Verknüpfungen werden in Tabellen abgelegt.

0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	:	1	0	0	1	1	0	0	1	0
0	0	0	0	0	0	0	0	0																
	1	0	1	1	1	0	0	0	0															
	1	0	0	1	1	0	0	1	0															
		0	1	0	0	0	0	1	0	0														

0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	:	1	0	0	1	1	0	0	1	0
0	0	0	0	0	0	0	0	0																
	1	0	0	1	1	0	0	1	0															
		0	1	0	0	0	0	1	0	0														

Bei diesem Vergleich sieht man, dass die ersten zwei Bits die XOR-Verknüpfung für die nächsten Bits vorgeben. Die Anzahl der zusammengefassten Bits kann je nach verfügbarem Speicher ausgewählt werden. Sinnvolle werte sind 2, 4, 8. Größere Werte als 8 machen nur sinn, wenn die Eingangsdaten nicht ein vielfaches eines Bytes sind, sondern von z.B. 2 Byte.

$r7_{alt}$	$r6_{alt}$	r7	r6	r5	r4	r3	r2	r1	r0	…
g8	g7	g6	g5	g4	g3	g2	g1	g0		
	g8	g7	g6	g5	g4	g3	g2	g1	g0	

r7_1 = (r7_0 AND g7) XOR r6_0
r6_1 = (r7_0 AND g6) XOR r5_0
r5_1 = (r7_0 AND g5) XOR r4_0
…

r7_2 = (r7_1 AND g7) XOR r6_1
= (((r7_0 AND g7) XOR r6_0) AND g7) XOR ((r7_0 AND g6) XOR r5_0)
r6_2 = …
…

Forward-LUT

Die Look-Up-Tabelle für 2 Bits mit dem Generatorpolynom 0x32 kann wie folgt berechnet werden:

0	0	.	.	.					
0	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	0
		0	0	0	0	0	0	0	0

0x00

0	1	.	.	.					
0	0	0	0	0	0	0	0	0	
	1	0	0	1	1	0	0	1	0
		0	0	1	1	0	0	1	0

0x32

1	0	.	.	.					
1	0	0	1	1	0	0	1	0	
	0	0	0	0	0	0	0	0	0
		0	1	1	0	0	1	0	0

0x64

1	1	.	.	.					
1	0	0	1	1	0	0	1	0	
	1	0	0	1	1	0	0	1	0
		0	1	0	1	0	1	1	0

0x56

```
#include <cstdlib>
#include <iostream>

using namespace std;

/*****************************************************************************/
/* forward lookup table implementation                                       */
/*****************************************************************************/
unsigned __int8 const bitCount        = 2;
unsigned __int16 const twoPowBitCount = 4;
unsigned __int8 LUT_forward[twoPowBitCount];

void create_forward_LUT()
{
    unsigned __int8 poly         = 0x32;
    unsigned __int16 tableIndex  = 0;
    unsigned __int8 currentIndex = 0;
    unsigned __int8 XORVal       = 0;

    printf("forward look up table: \n");
    for (tableIndex=0; tableIndex<(twoPowBitCount); tableIndex++)
    {
        currentIndex = tableIndex;
        XORVal       = 0;

        for (int i=0; i<bitCount; i++)
        {
            unsigned __int8 currentBit = (currentIndex & (1<<(bitCount-1)))
                                         << (8-bitCount);
            currentIndex               = currentIndex << 1;

            if (currentBit ^ (XORVal & 0x80))
            {
                XORVal = (XORVal << 1) ^ poly;
            }
            else
            {
                XORVal = (XORVal << 1);
            }
```

```
        }

        printf("LUT[%i] = %x \n", tableIndex, XORVal);
        LUT_forward[tableIndex] = XORVal;
    }
}

unsigned __int8 calc_forward_CRC8_SR_LUT(unsigned __int8 const *data, int length)
{
    unsigned __int8 reg         = 0x00;
    unsigned __int8 currentByte = 0x00;
    unsigned __int8 polyOrder   = 8;

    for(int i=0; i<length*8; i+=bitCount)
    {
        if (i%8 == 0)
        {
            currentByte = *(data + i/8);
        }
        unsigned __int8 inputBits = currentByte >> (8-bitCount);
        currentByte               = currentByte << bitCount;

        reg = ((reg << bitCount) | inputBits) ^ LUT_forward[reg >> (8-bitCount)];
    }

    for (int i=0; i<polyOrder; i+=bitCount)
    {
        reg = (reg << bitCount) ^ LUT_forward[reg >> (8-bitCount)];
    }

    return reg;
}

unsigned __int8 calc_forward_CRC8_LFSR_LUT(unsigned __int8 const *data, int length)
{
    unsigned __int8 reg         = 0x00;
    unsigned __int8 currentByte = 0x00;

    for(int i=0; i<length*8; i+=bitCount)
    {
        if (i%8 == 0)
        {
            currentByte = *(data + i/8);
        }
        unsigned __int8 inputBits = currentByte >> (8-bitCount);
        currentByte               = currentByte << bitCount;

        reg = ((reg << bitCount) ^ LUT_forward[(reg >> (8-bitCount)) ^ inputBits]);
    }

    return reg;
}
```

Quelltext 4 – Forward-CRC-Berechnung mittels LUT

Reverse-LUT

Die Look-Up-Tabelle für 2 Bits mit dem Generatorpolynom 0x4C kann wie folgt berechnet werden:

					.	.	.	0	0
	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0		

0x00

					.	.	.	0	1
	0	1	0	0	1	1	0	0	1
0	0	0	0	0	0	0	0	0	
0	0	1	0	0	1	1	0		

0x26

					.	.	.	1	0
	0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	1	
0	1	0	0	1	1	0	0		

0x4C

					.	.	.	1	1
	0	1	0	0	1	1	0	0	1
0	1	0	0	1	1	0	0	1	
0	1	1	0	1	0	1	0		

0x6A

```
#include <cstdlib>
#include <iostream>

using namespace std;

/*****************************************************************************/
/* reverse lookup table implementation                                       */
/*****************************************************************************/
unsigned __int8 const bitCount        = 2;
unsigned __int16 const twoPowBitCount = 4;
unsigned __int8 LUT_reverse[twoPowBitCount];

void create_reverse_LUT()
{
    unsigned __int8 poly         = 0x4C;
    unsigned __int16 tableIndex  = 0;
    unsigned __int8 currentIndex = 0;
    unsigned __int8 XORVal       = 0;

    printf("reverse look up table: \n");
    for (tableIndex=0; tableIndex<(twoPowBitCount); tableIndex++)
    {
        currentIndex = tableIndex;
        XORVal       = 0;

        for (int i=0; i<bitCount; i++)
        {
            unsigned __int8 currentBit = (currentIndex & 1);
            currentIndex               = currentIndex >> 1;

            if (currentBit ^ (XORVal & 0x01))
            {
                XORVal = (XORVal >> 1) ^ poly;
            }
            else
            {
                XORVal = (XORVal >> 1);
            }
        }
```

```
        printf("LUT[%i] = %x \n", tableIndex, XORVal);
        LUT_reverse[tableIndex] = XORVal;
    }
}

unsigned __int8 calc_reverse_CRC8_SR_LUT(unsigned __int8 const *data, int length)
{
    unsigned __int8 reg         = 0x00;
    unsigned __int8 currentByte = 0x00;
    unsigned __int8 polyOrder   = 8;

    for(int i=0; i<(length*8); i+=bitCount)
    {
        if (i%8 == 0)
        {
            currentByte = *(data + ((((length-1)*8)-i)>>3));
        }
        unsigned __int8 inputBits = currentByte << (8-bitCount);
        currentByte               = currentByte >> bitCount;

        reg = ((reg >> bitCount) | inputBits) ^
                    LUT_reverse[((unsigned __int8)(reg << (8-bitCount)))
                                    >> (8-bitCount)];
    }

    for (int i=0; i<polyOrder; i+=bitCount)
    {
        reg = (reg >> bitCount) ^
                    LUT_reverse[((unsigned __int8)(reg << (8-bitCount)))
                                    >> (8-bitCount)];
    }

    return reg;
}

unsigned __int8 calc_reverse_CRC8_LFSR_LUT(unsigned __int8 const *data, int length)
{
    unsigned __int8 reg         = 0x00;
    unsigned __int8 currentByte = 0x00;

    for(int i=0; i<length*8; i+=bitCount)
    {
        if (i%8 == 0)
        {
            currentByte = *(data + ((((length-1)*8)-i)>>3));
        }
        unsigned __int8 inputBits = currentByte << (8-bitCount);
        currentByte               = currentByte >> bitCount;

        reg = ((reg >> bitCount) ^
                    LUT_reverse[((unsigned __int8)((reg << (8-bitCount)) ^ inputBits))
                                    >> (8-bitCount)]);
    }

    return reg;
}
```

Quelltext 5 – Reverse-CRC-Berechnung mittels LUT

Unit-Tests

Um die unterschiedlichen Methoden der CRC-Berechnung zu testen, werden Unit-Tests durchgeführt. Dabei kann sogleich das Handling mit unterschiedlichen Implementierung getestet werden.
Bei der Forward-Implementierung wird die Berechnung z.B. beim MSB begonnen. Möchte man die Reverse-Implementierung mit dieser Vergleichen, so kann man die Daten Bit für Bit umkehren. Somit startet die Reverse-Implementierung auch mit dem MSB und als Ergebnis erhält man den umgekehrten Forward-CRC.
Eine fehlerhafte Implementierung ist leicht zu entdecken. Man muss nur den CRC über ein Byte berechnen. Ist die Implementierung falsch, so ergibt es immer ein unterschiedliches CRC-Ergebnis (auch bei kleinen Implementierungsfehlern). Man muss das Ergebnis mit einem anderen vergleichen können.

```
#include <cstdlib>
#include <iostream>

using namespace std;


int main()
{
    unsigned __int8 crc = 0;

/*****************************************************************************/
/* compare shift register and linear feedback shift register implementation  */
/*****************************************************************************/
    printf("__compare SR with LFSR implementation_____________________\n");


    // data length: 1 byte
    printf("____data length: 1 byte______\n");
    unsigned __int8 data1[] = {0x5C, 0x00};

    crc = calc_forward_CRC8_SR(data1, 1);
    printf("CRC8-SR of [%x]: %x\n", data1[0], crc);

    data1[1] = crc;
    crc = calc_forward_CRC8_SR(data1, 2);
    printf("CRC8-SR of [%x]+crc: %x\n", data1[0], crc);


    crc = calc_forward_CRC8_LFSR(data1, 1);
    printf("CRC8-LFSR of [%x]: %x\n", data1[0], crc);

    data1[1] = crc;
    crc = calc_forward_CRC8_LFSR(data1, 2);
    printf("CRC8-LFSR of [%x]+crc: %x\n", data1[0], crc);


    // data length: 2 byte
    printf("\n");
    printf("____data length: 2 byte______\n");
    unsigned __int8 data2[] = {0x5C, 0xA1, 0x00};

    crc = calc_forward_CRC8_SR(data2, 2);
    printf("CRC8-SR of [%x %x]: %x\n", data2[0], data2[1], crc);

    data2[2] = crc;
    crc = calc_forward_CRC8_SR(data2, 3);
    printf("CRC8-SR of [%x %x]+crc: %x\n", data2[0], data2[1], crc);


    crc = calc_forward_CRC8_LFSR(data2, 2);
    printf("CRC8-LFSR of [%x %x]: %x\n", data2[0], data2[1], crc);

    data2[2] = crc;
    crc = calc_forward_CRC8_LFSR(data2, 3);
    printf("CRC8-LFSR of [%x %x]+crc: %x\n", data2[0], data2[1], crc);
```

```
/*****************************************************************************/
/* compare forward and reverse CRC calculation                               */
/*****************************************************************************/
    printf("\n");
    printf("__compare forward with reverse calculation_____________\n");


    // data length: 1 byte
    printf("____data length: 1 byte______\n");
    unsigned __int8 data3[] = {0x5C, 0x00};

    crc = calc_forward_CRC8_LFSR(data3, 1);
    printf("forward-CRC of [%x]: %x\n", data3[0], crc);

    data3[1] = crc;
    crc = calc_forward_CRC8_LFSR(data3, 2);
    printf("forward-CRC of [%x]+crc: %x\n", data3[0], crc);


    data3[0] = 0x3A;
    crc = calc_reverse_CRC8_LFSR(data3, 1);
    printf("reverse-CRC of [%x]: %x\n", data3[0], crc);

    data3[0] = crc;
    data3[1] = 0x3A;
    crc = calc_reverse_CRC8_LFSR(data3, 2);
    printf("reverse-CRC of crc+[%x]: %x\n", data3[0], crc);


    // data length: 2 byte
    printf("\n");
    printf("____data length: 2 byte______\n");
    unsigned __int8 data4[] = {0x5C, 0xA1, 0x00};

    crc = calc_forward_CRC8_LFSR(data4, 2);
    printf("forward-CRC of [%x %x]: %x\n", data4[0], data4[1], crc);

    data4[2] = crc;
    crc = calc_forward_CRC8_LFSR(data4, 3);
    printf("forward-CRC of [%x %x]+crc: %x\n", data4[0], data4[1], crc);


    data4[0] = 0x85;
    data4[1] = 0x3A;
    crc = calc_reverse_CRC8_LFSR(data4, 2);
    printf("reverse-CRC of [%x %x]: %x\n", data4[0], data4[1], crc);

    data4[0] = crc;
    data4[1] = 0x85;
    data4[2] = 0x3A;
    crc = calc_reverse_CRC8_LFSR(data4, 3);
    printf("reverse-CRC of crc+[%x %x]: %x\n", data4[0], data4[1], crc);


/*****************************************************************************/
/* compare lookup table implementations                                      */
/*****************************************************************************/
    printf("\n");
    printf("__compare lookup table implementations_____________________\n");
    create_forward_LUT();
    create_reverse_LUT();


    // data length: 1 byte
    printf("____data length: 1 byte______\n");
    unsigned __int8 data5[] = {0x5C, 0x00};

    crc = calc_forward_CRC8_SR_LUT(data5, 1);
    printf("forward-CRC-LUT-SR of [%x]: %x\n", data5[0], crc);

    data5[1] = crc;
    crc = calc_forward_CRC8_SR_LUT(data5, 2);
    printf("forward-CRC-LUT-SR of [%x]+crc: %x\n", data5[0], crc);


    crc = calc_forward_CRC8_LFSR_LUT(data5, 1);
    printf("forward-CRC-LUT-LFSR of [%x]: %x\n", data5[0], crc);
```

```
    data5[1] = crc;
    crc = calc_forward_CRC8_LFSR_LUT(data5, 2);
    printf("forward-CRC-LUT-LFSR of [%x]+crc: %x\n", data5[0], crc);


    data5[0] = 0x3A;
    crc = calc_reverse_CRC8_SR_LUT(data5, 1);
    printf("reverse-CRC-LUT-SR of [%x]: %x\n", data5[0], crc);

    data5[0] = crc;
    data5[1] = 0x3A;
    crc = calc_reverse_CRC8_SR_LUT(data5, 2);
    printf("reverse-CRC-LUT-SR of crc+[%x]: %x\n", data5[0], crc);


    data5[0] = 0x3A;
    crc = calc_reverse_CRC8_LFSR_LUT(data5, 1);
    printf("reverse-CRC-LUT-LFSR of [%x]: %x\n", data5[0], crc);

    data5[0] = crc;
    data5[1] = 0x3A;
    crc = calc_reverse_CRC8_LFSR_LUT(data5, 2);
    printf("reverse-CRC-LUT-LFSR of crc+[%x]: %x\n", data5[0], crc);

    // data length: 2 byte
    printf("\n");
    printf("____data length: 2 byte______\n");
    unsigned __int8 data6[] = {0x5C, 0xA1, 0x00};

    crc = calc_forward_CRC8_SR_LUT(data6, 2);
    printf("forward-CRC-LUT-SR of [%x %x]: %x\n", data6[0], data6[1], crc);

    data6[2] = crc;
    crc = calc_forward_CRC8_SR_LUT(data6, 3);
    printf("forward-CRC-LUT-SR of [%x %x]+crc: %x\n", data6[0], data6[1], crc);


    crc = calc_forward_CRC8_LFSR_LUT(data6, 2);
    printf("forward-CRC-LUT-LFSR of [%x %x]: %x\n", data6[0], data6[1], crc);

    data6[1] = crc;
    crc = calc_forward_CRC8_LFSR_LUT(data6, 7);
    printf("forward-CRC-LUT-LFSR of [%x %x]+crc: %x\n", data6[0], data6[1], crc);


    data6[0] = 0x85;
    data6[1] = 0x3A;
    crc = calc_reverse_CRC8_SR_LUT(data6, 2);
    printf("reverse-CRC-LUT-SR of [%x %x]: %x\n", data6[0], data6[1], crc);

    data6[0] = crc;
    data6[1] = 0x85;
    data6[2] = 0x3A;
    crc = calc_reverse_CRC8_SR_LUT(data6, 3);
    printf("reverse-CRC-LUT-SR of crc+[%x %x]: %x\n", data6[0], data6[1], crc);


    data6[0] = 0x85;
    data6[1] = 0x3A;
    crc = calc_reverse_CRC8_LFSR_LUT(data6, 2);
    printf("reverse-CRC-LUT-LFSR of [%x %x]: %x\n", data6[0], data6[1], crc);

    data6[0] = crc;
    data6[1] = 0x85;
    data6[2] = 0x3A;
    crc = calc_reverse_CRC8_LFSR_LUT(data6, 3);
    printf("reverse-CRC-LUT-LFSR of crc+[%x %x]: %x\n", data6[0], data6[1], crc);


    return 0;
}
```

Quelltext 6 – Unit-Test der unterschiedlichen Implementierungen

```
__compare SR with LFSR implementation_____________________
____data length: 1 byte______
CRC8-SR of [5c]: f6
CRC8-SR of [5c]+crc: 0
CRC8-LFSR of [5c]: f6
CRC8-LFSR of [5c]+crc: 0

____data length: 2 byte______
CRC8-SR of [5c a1]: 2
CRC8-SR of [5c a1]+crc: 0
CRC8-LFSR of [5c a1]: 2
CRC8-LFSR of [5c a1]+crc: 0

__compare forward with reverse calculation_____________
____data length: 1 byte______
forward-CRC of [5c]: f6
forward-CRC of [5c]+crc: 0
reverse-CRC of [3a]: 6f
reverse-CRC of crc+[6f]: 0

____data length: 2 byte______
forward-CRC of [5c a1]: 2
forward-CRC of [5c a1]+crc: 0
reverse-CRC of [85 3a]: 40
reverse-CRC of crc+[40 85]: 0

__compare lookup table implementations_____________________
forward look up table:
LUT[0] = 0
LUT[1] = 32
LUT[2] = 64
LUT[3] = 56
reverse look up table:
LUT[0] = 0
LUT[1] = 26
LUT[2] = 4c
LUT[3] = 6a
____data length: 1 byte______
forward-CRC-LUT-SR of [5c]: f6
forward-CRC-LUT-SR of [5c]+crc: 0
forward-CRC-LUT-LFSR of [5c]: f6
forward-CRC-LUT-LFSR of [5c]+crc: 0
reverse-CRC-LUT-SR of [3a]: 6f
reverse-CRC-LUT-SR of crc+[6f]: 0
reverse-CRC-LUT-LFSR of [3a]: 6f
reverse-CRC-LUT-LFSR of crc+[6f]: 0

____data length: 2 byte______
forward-CRC-LUT-SR of [5c a1]: 2
forward-CRC-LUT-SR of [5c a1]+crc: 0
forward-CRC-LUT-LFSR of [5c a1]: 2
forward-CRC-LUT-LFSR of [5c 2]+crc: 0
reverse-CRC-LUT-SR of [85 3a]: 40
reverse-CRC-LUT-SR of crc+[40 85]: 0
reverse-CRC-LUT-LFSR of [85 3a]: 40
reverse-CRC-LUT-LFSR of crc+[40 85]: 0
```

Ausgabe 1 – Unit-Test Ausgabe

Weitere Eigenschaften

Ein zyklischer Code detektiert eine Vielzahl von Übertragungsfehlern:

- sämtliche 1 Bit Fehler
- sämtliche 2 Bit Fehler
- sämtliche Fehler mit ungeraden Anzahl
- sämtliche Burst-Fehler (Burst kleiner CRC)
- viele großen Burst-Fehler

Startwert-Problem

Ist der Initialwert der Register auf 0 festgelegt, so entspricht die Implementierung der einer Polynom-Division. Problem dieser Implementierung ist, dass der CRC-Wert bei vorangestellten 0en nicht beeinflusst wird. Sollten diese den CRC-Wert auch verändert, so verwendet man unterschiedliche Register-Startwerte (z.B. 0xFF).

Nullen-Problem

Wird der CRC-Wert einfach nach den Daten angefügt, so entstehen zwei zusätzliche Fehlerquellen:

- Ist z.B. der CRC-Wert zufällig gleich 0, so ist dieser auch weiterhin 0, wenn bei den Daten irrtümlicherweise 0en angefügt werden.
- Das Zielsystem überprüft die Daten üblicherweise indem es einen CRC-Wert der Daten inklusive dem übertragenen CRC-Wert berechnet. Also eine CRC-Wert-Berechnung über sämtliche empfangenen Daten. Diese Berechnung sollte als Ergebnis 0 haben. Sind nach dem CRC-Wert irrtümlicherweise weitere 0en vorhanden, können diese nicht entdeckt werden.

Um diese Fehler zu verhindern wird der CRC invertiert übertragen. Berechnet das Zielsystem einen CRC-Wert für die empfangenen Daten mit invertiertem CRC, so ergibt es immer einen bestimmten Wert. Dieser Wert wird auch als "Magic-Number" bezeichnet.