The CRC implementation

HMS800 C Compiler for the HMS800 series MCU

DESCRIPTION

This application note describes how to implement the CRC with HMS800 C Compiler for HMS800 series MCU. It is helpful to understand the CRC implementation with HMS800 C Compiler.

The CRC is a very powerful and easy to implement technique to obtain data reliability. The CRC technique is used to protect blocks of data called Frames. Using this technique, the transmitter appends an extra n-bit sequence to every frame called Frame Check Sequence (FCS). The FCS holds redundant information about the frame that helps the transmitter detect errors in the frame.

Introduction

The CRC is one of the most used techniques for error detection in data communications. The technique gained its popularity because it combines three advantages:

1. Extreme error detection capabilities.
2. Little overhead.
3. Ease of implementation.

The CRC algorithm works above the binary field. The algorithm treats all bit streams as binary polynomials. Given the original frame, the transmitter generates the FCS for that frame. The FCS is generated so that the resulting frame (the cascade of the original frame and the FCS) is exactly devisable by some pre-defined polynomial. This pre-defined polynomial is called the devisor or CRC Polynomial.

The CRC working

For the formal explanation we will define the following :

- M - The original frame to be transmitted, before adding the FCS. It is k bits long.
- F - The resulting FCS to be added to M. It is n bits long.
- T - The cascading of M and F. This is the resulting frame that will be transmitted. It is k+n bits long.
- P - The pre-defined CRC Polynomial. A pattern of n+1 bits.

For example, let's assume that:

$$M = 1010001101_{bin}(k = 10bit)$$
$$P = 110101_{bin}(n + 1 = 6)$$

Then the FCS to be calculated by the transmitter will be n = 5 bits in length. Let's assume that the transmitter has calculated the FCS to be:

$$T = 101000110111110_{bin}$$

The main idea behind the CRC algorithm is that the FCS is generated so that the reminder of "T / P" is zero. Its clear that:

$$T = M \times x^n + F \qquad (1)$$

This is caused by cascading F to M we have shifted T by n bits to the left and then added F to the result. We want the transmitted frame, T, to be exactly divisible by the pre-defined polynomial P, so we would have to find a suitable Frame Check Sequence (F) for every raw message (M).

Suppose we divided only $M \times x^n$ by P, we would get:

$$\frac{M \times x^n}{P} = Q + \frac{R}{P}$$

There is a quotient and a reminder. We will use this reminder, R, as our FCS (F). Returning to Eq. (1):

$$T = M \times x^n + R$$

We will now show that this selection of the FCS makes the transmitted frame (T) exactly divisible by P:

$$\frac{T}{P} = \frac{(M \times x^n + R)}{P} = \frac{M \times x^n}{P} + \frac{R}{P}$$
$$= Q + \frac{R}{P} + \frac{R}{P} = Q + \frac{(R + R)}{P}$$

but any binary number added to itself in a modulo 2 field yields zero so:

$$\frac{T}{P} = Q \qquad \text{,with no reminder}$$

Following is a review of the CRC creation process:

> ***1. Get the raw frame***
> ***2. Left shift the raw frame by n bits and the divide it by P***
> ***3. The reminder of the last action is the FCS.***
> ***4. Append the FCS to the raw frame. The result is the frame to transmit.***

And a review of the CRC check process:

> ***1. Receive the frame.***
> ***2. Divide it by P.***
> ***3. Check the reminder. If not zero then there is an error in the frame.***

It can be easily seen that the CRC algorithm must compute the reminder of the division of two polynomials. The process is described in the following.

Hardware CRC implementation

The hardware CRC implementation is shown in the next figure 1. The implementation shown is for a specific set of parameters:

$M = 1010001101_{bin}$

$P = 110101_{bin}$

$F = 1110_{bin}$

The figure 1 circuit is implemented as follows:

> ***1. The register contains n bits, equal to the length of the FCS.***
> ***2. There are up to n XOR gates.***
> ***3. The presence or absence of a gate corresponds to the presence or absence of a term in the divisor polynomial P(X).***

The same circuit is used for both creation and check of the CRC. Refer to figure 2. When creating the FCS, the circuit accepts the bits of the raw frame and then a sequence of zeros. The length of the sequence is the same as the length of the FCS. The contents of the shift register will be the FCS to append. When checking the FCS, the circuit accepts the bits of the received frame (raw frame appended by FCS and perhaps corrupted by errors). The contents of the shift register should be zero or else there are errors.

Software CRC implementation

The simplest way of implementing the CRC algorithm in

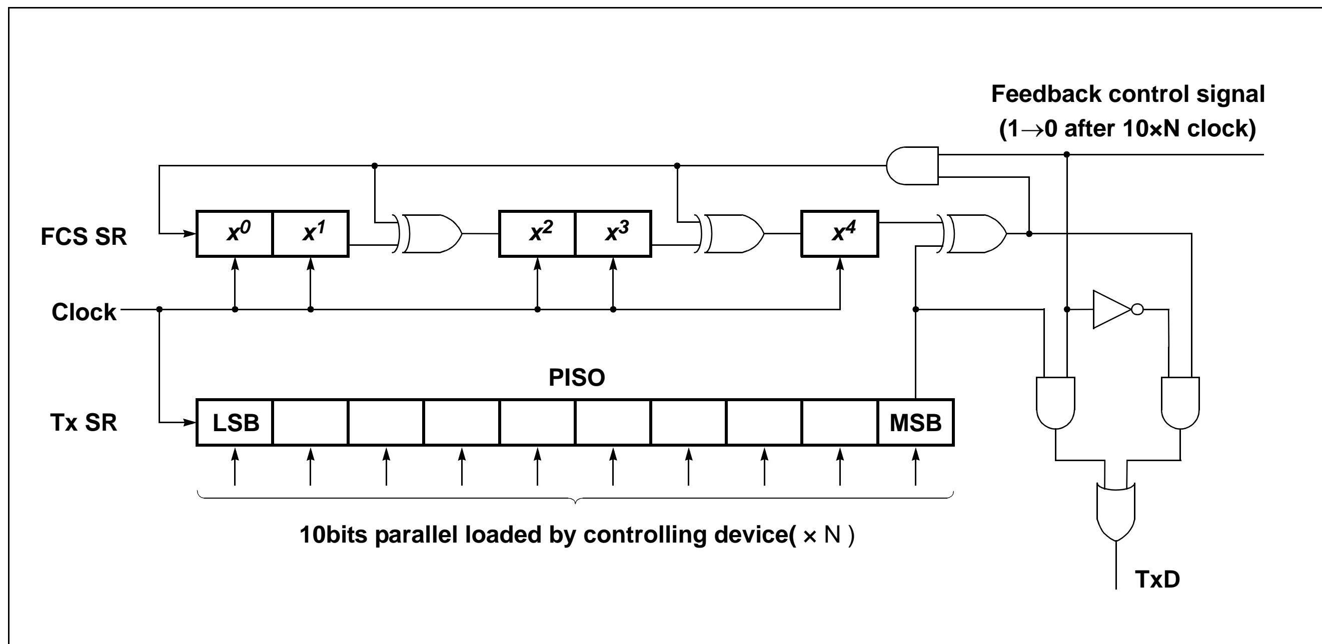

Figure 1. The hardware CRC generation

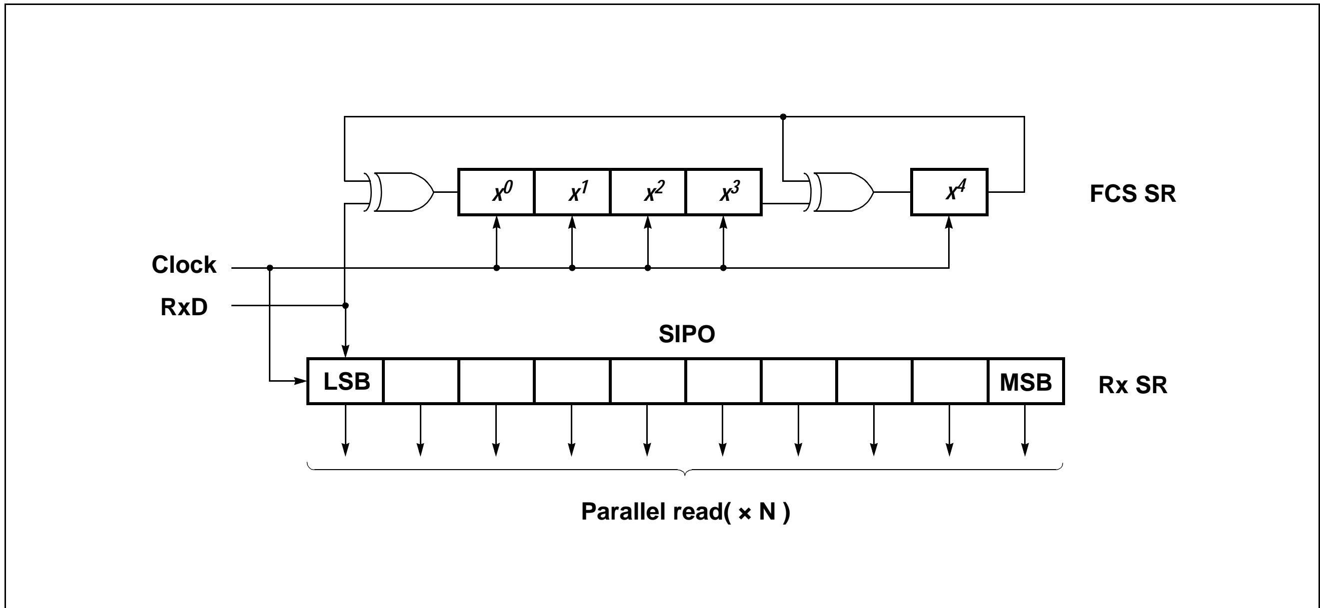

Figure 2. The hardware CRC check

software is to take the hardware implementation and write the appropriate code, replacing the shift register with a variable and the XOR gates with the xor operator. This method is the easiest to implement and consumes very little memory. It's performance, however is poor. When the time of the CRC process is of importance, faster software algorithms should be used. Those algorithms process one byte at a time and not one bit at a time like the hardware implementation does. Those algorithms have one main disadvantage: they have to keep tables in memory and so need more memory than the bitwise algorithm does.

The CRC polynomials

Following is a list of the most used the CRC polynomials:

$$CRC-12 = x^{12} + x^{11} + x^3 + x^2 + x + 1$$

$$CRC-16 = x^{16} + x^{15} + x^2 + 1$$

$$CRC-CCITT = x^{16} + x^{12} + x^5 + 1$$

$$CRC-32 = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

The CRC-12 is used for transmission of streams of 6-bit characters and generates 12-bit FCS. Both CRC-16 and CCRC-CCITT are used for 8 bit transmission streams and both result in 16 bit FCS. The last two are widely used in the USA and Europe respectively and give adequate protection for most applications. Applications that need extra protection can make use of the CRC-32 which generates 32 bit FCS. The CRC-32 is used by the local network standards committee (IEEE-802) and in some DOD applications.

Programming

The C program of this application note is implemented for the CRC-CCITT polynomial, truncated 1 byte. These functions for the CRC-CCITT generation is handled here. The implementation of the software CRC-CCITT generation refer to appendix A: Software CRC generation.

Descriptions of the program:

A) Structure of the program

1. Definition statements
- Definition of the macros.
2. GenerationHalfCRC function
- 1 byte CRC-CCITT generation.
3. GenerationCRC_CCITT function
- Whole 2 byte CRC-CCITT generation.

B) Notification of the program

The “GenerationHalfCRC( )” function will compute and check a true 16-bit Cyclic Redundancy Check for a message of arbitrary continuous data.

Author: *Nicholas Oh*
MCU application team
e-mail: jesu.oh@hynix.com

Appendix A: Functions.

```
/*******************************************************************************************
 Title : CRC implementation
 Programming date : 2004.6.16.
 Present filename : CRC.c
*******************************************************************************************
 Program description
   CRC-CCITT(16 bit) implemented
*******************************************************************************************/
// Type redefine.
typedef unsigned char  uchar;
typedef unsigned long  ushort;
typedef unsigned char  BYTE;
typedef unsigned short WORD;

// Bit-accessable structure statement of 1byte.
union BitAccess
{
                uchar  ToByte;
                struct P8Bit
                {
                   char pb0:1 ; // 1
                   char pb1:1 ; // 2
                   char pb2:1 ; // 3
                   char pb3:1 ; // 4
                   char pb4:1 ; // 5
                   char pb5:1 ; // 6
                   char pb6:1 ; // 7
                   char pb7:1 ; // 8
                }  bitn;
};

// Word-accessable structure statement of 1byte.
union WordAccess
{
                ushort ToWord;
                struct DoubleByte
                {
                   uchar b0;  // Lower byte
                   uchar b1;  // Higher byte
                }  wacc;
};

volatile union BitAccess f1;        // Data RAM for flag bits.
volatile union BitAccess cCRCd1;    // Temporary data RAM1 for the CRC function.
volatile union BitAccess cCRCd2;    // Temporary data RAM2 for the CRC function.
volatile union BitAccess cCRCd3;    // Temporary data RAM3 for the CRC function.
volatile union WordAccess usGenCrc; // 1 word data generated CRC-CCITT.

#define CRCCarryFlag      f1.bitn.pb0  // CRC carry flag
#define CRCInterCarryFlag f1.bitn.pb1  // CRC first inter-carry flag
#define CRCLastCarryFlag  f1.bitn.pb2  // CRC last inter-carry flag
#define ucCrcLowByte      usGenCrc.wacc.b0  // CRC Lower byte data.
#define ucCrcHighByte     usGenCrc.wacc.b1  // CRC Higher byte data.

// Proto-type functions.
void GenerationHalfCRC(uchar);
void GenerationCRC_CCITT(uchar);
```

```
/*********************************************************************
 Function : CRC-CCITT Half generation process
 Description : These subroutines will compute and check a true 16-bit
               Cyclic Redundancy Check for a message of arbitary length.
 Polynomial = X^16 + X^12 + X^5 + 1
*********************************************************************/
void GenerationHalfCRC(uchar data)
{
                uchar uc;
                cCRCd1.ToByte = data;
                cCRCd2.ToByte = ucCrcLowByte;
                cCRCd3.ToByte = ucCrcHighByte;

                for(uc = 0;uc < 8;++uc)
                {
                   CRCCarryFlag = cCRCd1.bitn.pb7;
                   cCRCd1.ToByte <<= 1;
                   cCRCd1.bitn.pb0 = CRCCarryFlag;

                   CRCInterCarryFlag = cCRCd2.bitn.pb7;
                   cCRCd2.ToByte <<= 1;
                   cCRCd2.bitn.pb0 = CRCCarryFlag;

                   CRCLastCarryFlag = cCRCd3.bitn.pb7;
                   cCRCd3.ToByte <<= 1;
                   cCRCd3.bitn.pb0 = CRCInterCarryFlag;

                   if(CRCLastCarryFlag)
                   {
                      cCRCd2.ToByte ^= 0x21;
                      cCRCd3.ToByte ^= 0x10;
                   }
                }
                ucCrcLowByte = cCRCd2.ToByte;
                ucCrcHighByte = cCRCd3.ToByte;
}


/*********************************************************************
 Function : CRC-CCITT whole generation process
 Description : "GenerationHalfCRC()" double call
*********************************************************************/
void GenerationCRC_CCITT(uchar data)
{
                GenerationHalfCRC(data);
                GenerationHalfCRC(data);
}
```