



AN1200.18

Implementing Data Whitening and CRC Calculation
in Software on SX12xx Devices

Table of Contents

Table of Contents	2
Index of Figures	2
1 Preamble.....	3
2 Introduction	3
3 Principle of operation	3
4 CCITT Whitening	4
4.1 Structure	4
4.2 CCIT Data Whitening example.....	4
4.3 Software Implementation	6
5 IBM Whitening.....	7
5.1 Structure.....	7
5.2 IBM Data Whitening example.....	7
5.3 Software Implementation	9
6 CRC Calculation	10
7 Device Implementation	11
7.1 Transmitter setup	11
7.2 Receiver setup	12
8 Errata Note.....	13

Index of Figures

Figure 1: CCIT Data whitening LFSR	4
Figure 2: Output of the CCIT LFSR	5
Figure 3: CCIT Software Whitening function.....	6
Figure 4: IBM Data whitening LFSR	7
Figure 5: Output of the IBM LFSR	8
Figure 6: IBM Software Whitening function.....	9
Figure 7: IBM Whitening Software implementation for compatibility mode	13

1 Preamble

This document describes how to implement the Whitening and the CRC computations in software on the SX12xx devices.

2 Introduction

In an RF system and in all communication system in general, the data to be transmitted are grouped into packets. These packets may often contain long sequences of 1's and 0's which introduces a DC bias in the transmitted signal. This DC bias produces a non-uniform power distribution over the occupied channel bandwidth and data dependences demodulator operations. To remedy to these issues, it is necessary to randomize the data and to make sure the data transmitted are DC free.

DC free data can be obtained by using Manchester encoding, which ensures that there are no more than two consecutive 1's or 0's. However, this reduces the effective bitrate of the system because it doubles the amount of data to be transmitted, and thus halving the effective bit-rate.

Another technique called whitening or scrambling is widely used for randomizing the user data before radio transmission. The data is whitened using a random sequence on the Tx side and de-whitened on the Rx side using the same sequence. This whitening approach is nowadays widely used and we will describe how it can be implemented in software through this document.

In combination to the whitening, it is often mandatory to have a CRC checksum at the end of the payload so that it is possible to check the validity of the data received. There are two fairly similar main algorithms which allow this checksum verification and we will present them across this document.

3 Principle of operation

Most of the SX12xx devices already support whitening and CRC verification of the payload in hardware but it may be necessary, in certain circumstances, to implement them in software.

The whitening process is basically built around a 9-bit LFSR which is used to generate a random sequence and the payload (and 2-byte CRC checksum) is then XORed with this random sequence to generate the whitened payload. The data is de-whitened on the receiver side by XORing with the same random sequence. This setup limits the number of consecutive 1's or 0's to 9. Note that the data whitening is only used when the user data has high correlation with long strings of 0's and 1's. If the data is already random then the whitening is not required. For example a random source generating the Transmit data, when whitened could produce longer strings of 1's and 0's, thus it's not required to randomize an already random sequence.

4 CCITT Whitening

4.1 Structure

The CCITT whitening is based around the 9-bit LFSR polynomial $x^9 + x^5 + 1$. With this structure, the LSBit at the output of the LFSR is XORed with the MSBit of the data. At the initial stage, each flip-flop of the LFSR is set to "1". The figure 1 below presents the CCIT whitening structure.

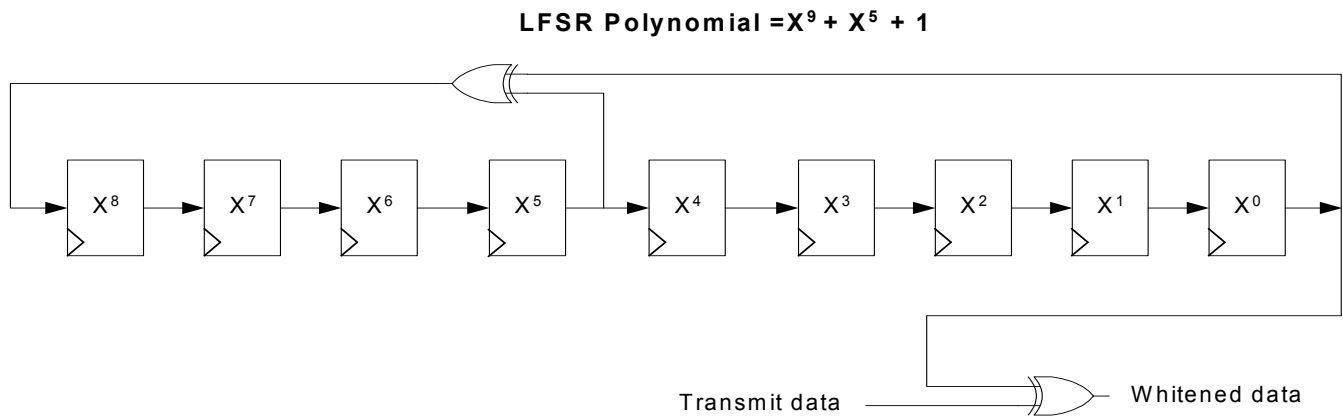


Figure 1: CCIT Data whitening LFSR

4.2 CCIT Data Whitening example

Let's assume that we have a four byte payload that we want to whiten as show in the table below:

Address	Data
0	0x01
1	0x9A
2	0x55
3	0x29

At the same time, we can predict the output of the LFSR as it shifts left. The output of the LFSR is given below in the Figure 2.

Bit Number	Value
1	1
2	1
3	1
4	1
5	1
6	1
7	1

8	1
9	1
10	0
11	0
12	0
13	0
14	1
15	1
16	1
17	1
18	0
19	1
20	1
21	1
22	0
23	0
24	0
25	0
26	1
27	0
28	1
29	1
30	0
31	0
32	1

Figure 2: Output of the CCIT LFSR

This output if arranged into bytes would look like

Byte Number	Value
0	0xFF
1	0x87
2	0xB8
3	0x59

Now, each data byte from the FIFO is transmitted MSB first and this data is XORed with the above output of the whitening LFSR bit-by-bit before being transmitted.

Thus the XOR operation of the FIFO contents with the LFSR output would produce the following sequence of output bytes.

Byte Number	FIFO	LFSR	Whitened Data (XOR)
0	0x01	0xFF	0xFE
1	0x9A	0x87	0x1D
2	0x55	0xB8	0xED
3	0x29	0x59	0x70



4.3 Software Implementation

From this stage, it is fairly simple to implement this LFSR structure in C code. The idea is to simply shift the LFSR for every new bit of the data and to XOR the LSBit of the last flip-flop with the MSBit of the coming data. The example below is a simple function taking a data buffer and performing the whitening in the same buffer.

```
void SX1232RadioComputeWhitening( uint8_t *buffer, uint16_t bufferSize )
{
    uint8_t i = 0;
    uint16_t j = 0;
    uint8_t WhiteningKeyMSBPrevious = 0;
    uint8_t revertedWhiteningKeyLSB = 0;

    revertedWhiteningKeyLSB = WhiteningKeyLSB;    // WhiteningKeyLSB is 0xFF at init

    for( j = 0; j < bufferSize - 1; j++ )
    {
        buffer[j] ^= revertedWhiteningKeyLSB;

        for( i = 0; i < 8; i++ )
        {
            WhiteningKeyMSBPrevious = WhiteningKeyMSB;
            WhiteningKeyMSB = ( WhiteningKeyLSB & 0x01 ) ^ ( ( WhiteningKeyLSB >> 5 ) & 0x01 );
            WhiteningKeyLSB = ( ( ( WhiteningKeyMSBPrevious << 7 ) & 0x80 | ( WhiteningKeyLSB >> 1 ) & 0xFF ) );
        }
        revertedWhiteningKeyLSB = (WhiteningKeyLSB & 0xF0) >> 4 | (WhiteningKeyLSB & 0x0F) << 4;
        revertedWhiteningKeyLSB = (revertedWhiteningKeyLSB & 0xCC) >> 2 | (revertedWhiteningKeyLSB & 0x33) << 2;
        revertedWhiteningKeyLSB = (revertedWhiteningKeyLSB & 0xAA) >> 1 | (revertedWhiteningKeyLSB & 0x55) << 1;
    }
}
```

Figure 3: CCIT Software Whitening function

5 IBM Whitening

5.1 Structure

While the CCIT whitening is handling byte per byte, the IBM whitening is handling the packet bit per bit. The process is almost identical but the result whitening sequence is completely different and care should be taken to implement the proper algorithm depending of the whitening the user is trying to achieve. The Figure 4 presents the LFSR as used in the IBM whitening.

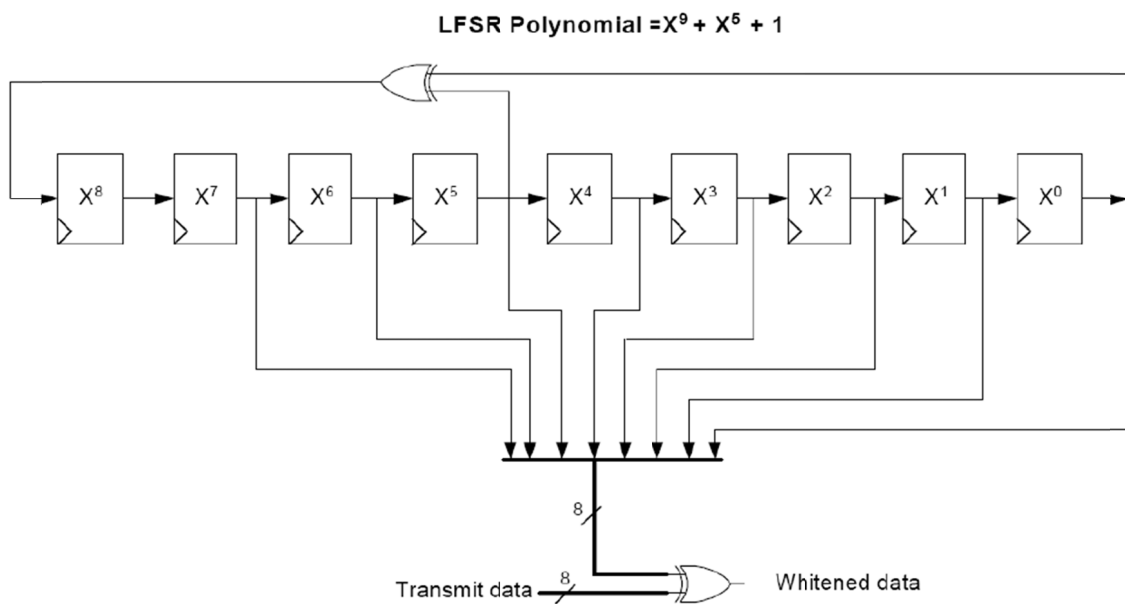


Figure 4: IBM Data whitening LFSR

5.2 IBM Data Whitening example

Let's assume that we have a four byte payload that we want to whiten as show in the table below:

Address	Data
0	0x01
1	0x9A
2	0x55
3	0x29

Again, we can predict the output of the LFSR as it shifts left. The output of the LFSR is given below in the Figure 5.

Bit Number	Value
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1
11	1
12	0
13	0
14	0
15	0
16	1
17	0
18	0
19	0
20	1
21	1
22	1
23	0
24	1
25	1
26	0
27	0
28	1
29	1
30	0
31	1
32	0

Figure 5: Output of the IBM LFSR

This output if arranged into bytes would look like

Byte Number	Value
0	0xFF
1	0xE1
2	0x1D
3	0x9A

Now each data byte from the FIFO is transmitted MSB first and this data is XORed with the above output of the whitening LFSR bit-by-bit before being transmitted.

Thus the XOR operation of the FIFO contents with the LFSR output would produce the following sequence of output bytes.

Byte Number	FIFO	LFSR	Whitened Data (XOR)
0	0x01	0xFF	0xFE
1	0x9A	0xE1	0x7B
2	0x55	0x1D	0x48
3	0x29	0x85	0xB3

5.3 Software Implementation

The IBM whitening software implementation is presented below. The implementation is symmetrical and can be used to either whiten or de-whiten the data.

```

static uint8_t WhiteningKeyMSB;           // Global variable so the value is kept after starting the
static uint8_t WhiteningKeyLSB;           // de-whitening process

WhiteningKeyMSB = 0x01;                   // Init value for the LFSR, these values should be initialise only
WhiteningKeyLSB = 0xFF;                   // at the start of a whitening or a de-whitening process

// *buffer is a char pointer indicating the data to be whiten / de-whiten
// buffersize is the number of char to be whiten / de-whiten

// >> The whitened / de-whitened data are directly placed into the pointer

void SX1232RadioComputeWhitening( uint8_t *buffer, uint16_t bufferSize )
{
    uint8_t i = 0;
    uint16_t j = 0;
    uint8_t WhiteningKeyMSBPrevious = 0;   // 9th bit of the LFSR

    for( j = 0; j < bufferSize; j++ )      // byte counter
    {
        buffer[j] ^= WhiteningKeyLSB;      // XOR between the data and the whitening key

        for( i = 0; i < 8; i++ )           // 8-bit shift between each byte
        {
            WhiteningKeyMSBPrevious = WhiteningKeyMSB;
            WhiteningKeyMSB = ( WhiteningKeyLSB & 0x01 ) ^ ( ( WhiteningKeyLSB >> 5 ) & 0x01 );
            WhiteningKeyLSB = ( ( WhiteningKeyLSB >> 1 ) & 0xFF ) | ( ( WhiteningKeyMSBPrevious << 7 ) & 0x80 );
        }
    }
}

```

Figure 6: IBM Software Whitening function



6 CRC Calculation

```
// CRC types
#define CRC_TYPE_CCITT          0
#define CRC_TYPE_IBM           1

// Polynomial = X^16 + X^12 + X^5 + 1
#define POLYNOMIAL_CCITT       0x1021
// Polynomial = X^16 + X^15 + X^2 + 1
#define POLYNOMIAL_IBM        0x8005

// Seeds
#define CRC_IBM_SEED           0xFFFF
#define CRC_CCITT_SEED        0x1D0F

uint16_t RadioComputeCRC( uint8_t *buffer, uint8_t length, uint8_t crcType )
{
    uint8_t i = 0;
    uint16_t crc = 0;
    uint16_t polynomial = 0;

    polynomial = ( crcType == CRC_TYPE_IBM ) ? POLYNOMIAL_IBM : POLYNOMIAL_CCITT;
    crc = ( crcType == CRC_TYPE_IBM ) ? CRC_IBM_SEED : CRC_CCITT_SEED;

    for( i = 0; i < length; i++ )
    {
        crc = ComputeCrc( crc, buffer[i], polynomial );
    }

    if( crcType == CRC_TYPE_IBM )
    {
        return crc;
    }
    else
    {
        return( ( uint16_t ) ( ~crc ) );
    }
}

uint16_t ComputeCrc( uint16_t crc, uint8_t dataByte, uint16_t polynomial )
{
    uint8_t i;

    for( i = 0; i < 8; i++ )
    {
        if( ( ( ( crc & 0x8000 ) >> 8 ) ^ ( dataByte & 0x80 ) ) != 0 )
        {
            crc <<= 1;          // shift left once
            crc ^= polynomial;  // XOR with polynomial
        }
        else
        {
            crc <<= 1;          // shift left once
        }
        dataByte <<= 1;        // Next data bit
    }
    return crc;
}
```

7 Device Implementation



We are now going to implement the whitening and CRC calculation in software. On the most recent SX12xx devices, these processes are already implemented in the devices hardware.

To start the implementation of the whitening, it is necessary to set the device in Unlimited Length Packet Format. This is done by setting the *Packet Format* bit to 0 (Fixed length packet) and setting the *PayloadLength* to 0. It is also necessary to set the CRC calculation to "OFF" by setting the bit *CrcOn* to 0. From this setup, the complete handling of the packets is left to the user and the packet handler of the device is simply used to detect the Preamble and Sync Word.

7.1 Transmitter setup

Building the packet on the transmitter side is a fairly simple. The idea is to build packets which follow the usual packet structure as shown in the figure xx below

Preamble 0 to 65536 bytes	Sync Word 0 to 8 bytes	Length 1 byte	Address 1 byte	Message Up to 255 bytes	CRC 2 byte
------------------------------	---------------------------	------------------	-------------------	----------------------------	---------------

-  Mandatory packet fields
-  Optional packet fields

In the rest of this document, we will assume the packets have variable length and the CRC checksum as this is the most common setup. If required, an address byte can be added in the same manner as described below.

The first step is to build the packet which is thus composed of 1 byte Payload length and xx byte of actual payload.

The second step is to calculate the CRC on this made up payload (1 byte length plus actual payload)

The last step is to whiten the resulting packet which is now composed of 1 byte payload length, xx bytes actual payload and 2 bytes of CRC.




7.2 Receiver setup

On the receiver side, the process is now reverted and it is necessary to de-whiten the packet on the fly so that the packet length can be extracted from the received data. It is essential here to notice that the device does not know the size of the packet it is going to receive and thus the interrupt CRC OK or payload ready will never be generated.

Preamble 0 to 65536 bytes	Sync Word 0 to 8 bytes	Length 1 byte	Address 1 byte	Message Up to 255 bytes	CRC 2 byte
------------------------------	---------------------------	------------------	-------------------	----------------------------	---------------

 Mandatory packet fields

 Optional packet fields

Because the receiver does not know the length of the packet, it is necessary to set the FIFO Level Threshold fairly low so that at least one FIFO Level interrupt is generated and we can de-whiten the packet length byte and start counting the data bytes until we have received them all.

First step is to de-whiten the data (1 byte length, xx byte payload and 2 bytes CRC)
Then the CRC can be calculated from the made up payload (1 byte length plus xxx bytes payload) and compared with the received payload. From this point the actual payload can be extracted by removing the Payload length and CRC from the data.

8 Errata Note

In all versions prior to the SX1276, there was a bug in the initialization of the polynomial used for the IBM whitening. This resulted in compatibility problem when an SX12xx was used in conjunction with a competitive device. If you are implementing the data whitening to be compatible with one of these previous devices, the code below should be implemented in software.

```
void Sx1272RadioComputeWhitening( uint8_t *buffer, uint16_t bufferSize )
{
    uint8_t i;
    uint16_t j;
    uint8_t WhiteningKeyMSBPrevious;

    j = 0;

    buffer[j] ^= WhiteningKeyLSB;

    for( i = 0; i < 9; i++ )
    {
        WhiteningKeyMSBPrevious = WhiteningKeyMSB;
        WhiteningKeyMSB = ( WhiteningKeyLSB & 0x01 ) ^ ( ( WhiteningKeyLSB >> 5 ) & 0x01 );
        WhiteningKeyLSB = ( ( WhiteningKeyLSB >> 1 ) & 0xFF ) | ( ( WhiteningKeyMSBPrevious << 7 ) & 0x80 );
    }

    for( j = 1; j < bufferSize; j++ )
    {
        buffer[j] ^= WhiteningKeyLSB;

        for( i = 0; i < 8; i++ )
        {
            WhiteningKeyMSBPrevious = WhiteningKeyMSB;
            WhiteningKeyMSB = ( WhiteningKeyLSB & 0x01 ) ^ ( ( WhiteningKeyLSB >> 5 ) & 0x01 );
            WhiteningKeyLSB = ( ( WhiteningKeyLSB >> 1 ) & 0xFF ) | ( ( WhiteningKeyMSBPrevious << 7 ) & 0x80 );
        }
    }
}
```

Figure 7: IBM Whitening Software implementation for compatibility mode



© Semtech 2013

All rights reserved. Reproduction in whole or in part is prohibited without the prior written consent of the copyright owner. The information presented in this document does not form part of any quotation or contract, is believed to be accurate and reliable and may be changed without notice. No liability will be accepted by the publisher for any consequence of its use. Publication thereof does not convey nor imply any license under patent or other industrial or intellectual property rights. Semtech assumes no responsibility or liability whatsoever for any failure or unexpected operation resulting from misuse, neglect improper installation, repair or improper handling or unusual physical or electrical stress including, but not limited to, exposure to parameters beyond the specified maximum ratings or operation outside the specified range.

SEMTECH PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF SEMTECH PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE UNDERTAKEN SOLELY AT THE CUSTOMER'S OWN RISK. Should a customer purchase or use Semtech products for any such unauthorized application, the customer shall indemnify and hold Semtech and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs damages and attorney fees

Contact Information

Semtech Corporation
Wireless & Sensing Products Division
200 Flynn Road, Camarillo, CA 93012
Phone: (805) 498-2111 Fax: (805) 498-3804
E-mail: sales@semtech.com
support_rf@semtech.com
Internet: <http://www.semtech.com>