ETSI/SAGE Specification

Version: 1.5
Date: 4th Jan. 2011

Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification

The ZUC algorithm is the core of the standardised 3GPP Confidentiality and Integrity algorithms 128-EEA3 & 128-EIA3.

Document History		
1.0	**18-06-2010**	**Publication**
1.2	**26-07-2010**	**Improvements to C code**
1.3	**27-07-2010**	**Minor corrections to C code**
1.4	**30-07-2010**	**Corrected preface**
1.5	**04-01-2011**	**A modification of ZUC in the initialization**

Blank Page

PREFACE

This specification has been prepared by the 3GPP Task Force, and gives a detailed specification of the 3GPP algorithm **ZUC**. **ZUC** is a stream cipher that forms the heart of the 3GPP confidentiality algorithm ***128-EEA3*** and the 3GPP integrity algorithm ***128-EIA3***. This document is the second of three, which between them form the entire specification of the 3GPP Confidentiality and Integrity Algorithms:

- Specification of the 3GPP Confidentiality and Integrity Algorithms ***128-EEA3*** & ***128-EIA3***.
  Document 1: ***128-EEA3*** and ***128-EIA3*** Specifications.

- Specification of the 3GPP Confidentiality and Integrity Algorithms ***128-EEA3*** & ***128-EIA3***.
  Document 2: **ZUC** Specification.

- Specification of the 3GPP Confidentiality and Integrity Algorithms ***128-EEA3*** & ***128-EIA3***.
  Document 3: Implementor's Test Data.

The normative part of the specification of **ZUC** is in the main body of this document. Annex A, which is purely informative, contains an implementation program listing of the cryptographic algorithm specified in the main body of this document, written in the programming language C.

TABLE OF CONTENTS

NORMATIVE SECTION

This part of the document contains the normative specification of the **ZUC** algorithm.

1 Introduction

ZUC is a word-oriented stream cipher. It takes a 128-bit initial key and a 128-bit initial vector (IV) as input, and outputs a keystream of 32-bit words (where each 32-bit word is hence called a *key-word*). This keystream can be used for encryption/decryption.

The execution of **ZUC** has two stages: initialization stage and working stage. In the first stage, a key/IV initialization is performed, i.e., the cipher is clocked without producing output (see section 3.6.1). The second stage is a working stage. In this stage, with every clock pulse, it produces a 32-bit word of output (see section 3.6.2).

2 Notations and conventions

2.1 Radix

In this document, integers are represented as decimal numbers unless specified otherwise. We use the prefix "0x" to indicate hexadecimal numbers, and the subscript "2" to indicate a number in binary representation.

Example 1 Integer *a* can be written in different representations:

a = 1234567890	decimal representation
= 0x499602D2	hexadecimal representation
= $1001001100101100000001011010010_2$	binary representation

2.2 Bit ordering

In this document, all data variables are presented with the most significant bit(byte) on the left hand side and the least significant bit(byte) on the right hand side.

Example 2 Let a=$1001001100101100000001011010010_2$. Then its most significant bit is 1 (the leftmost bit) and its least significant bit is 0 (the rightmost bit).

2.3 Notations

$+$	The addition of two integers.
ab	The product of integers a and b.
$=$	The assignment operator.
mod	The modulo operation of integers.
$\oplus$	The bit-wise exclusive-OR operation of integers.
$\boxplus$	The modulo 2^{32} addition .
$a \| b$	The concatenation of strings a and b.

a_H	The leftmost 16 bits of integer *a*.
a_L	The rightmost 16 bits of integer *a*.
$a <<<_n k$	The *k*-bit cyclic shift of the n bit register *a* to the left.
$a >> 1$	The l-bit right shift of integer *a*.
$(a_1, a_2,\ldots, a_n) \rightarrow (b_1, b_2,\ldots, b_n)$	The assignment of the values of a_i to b_i in parallel.

Example 3 For any two strings *a* and *b*, the presentation of string *c* created by the concatenation of *a* and *b* also follows the rules defined in section 2.2 i.e., the most significant digits are on the left hand side and the least significant digits are on the right hand side. For instance,

$$a\text{=0x1234},$$

$$b\text{=0x5678},$$

Then we have

$$c = a||b = \text{0x12345678}.$$

Example 4 Let

$$a=10010011001011000000001011010010_2$$

Then we have

$$a_H=1001001100101100_2,$$

$$a_L=0000001011010010_2.$$

Example 5 Let

$$a=110010011001011000000001011010010_{2.}$$

Then we have

$$a >> 1=11001001100101100000000101101001_{2.}$$

Example 6 Let $a_0, a_1, \ldots, a_{15}, b_0, b_1, \ldots, b_{15}$ be all integer variables. Then

$$(a_0, a_1, \ldots, a_{15}) \rightarrow (b_0, b_1, \ldots, b_{15})$$

will result in $b_i=a_i$, $0\leq i\leq 15$.

3 Algorithm description

3.1 General structure of the algorithm

ZUC has three logical layers, see Fig. 1. The top layer is a linear feedback shift register (LFSR) of 16 stages, the middle layer is for bit-reorganization (BR), and the bottom layer is a nonlinear function *F*.

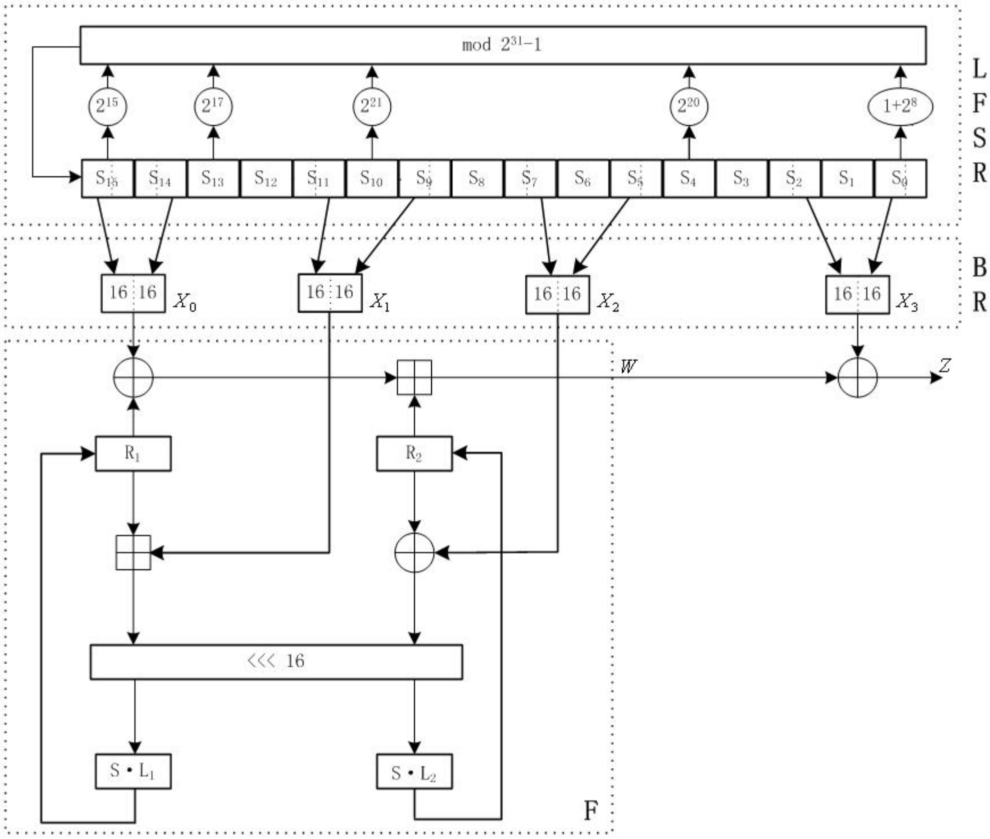

Figure 1. General structure of ZUC

3.2 The linear feedback shift register (LFSR)

The linear feedback shift register (LFSR) has 16 of 31-bit cells ($s_0, s_1, \ldots, s_{15}$). Each cell s_i ($0 \leq i \leq 15$) is restricted to take values from the following set

$$\{1,2,3, \ldots, 2^{31}-1\}.$$

The LFSR has 2 modes of operations: the initialization mode and the working mode.

In the initialization mode, the LFSR receives a 31-bit input word *u*, which is obtained by removing the rightmost bit from the 32-bit output *W* of the nonlinear function *F*, i.e., $u=W>>1$. More specifically, the initialization mode works as follows:

LFSRWithInitialisationMode(*u*)
{

1. $v=2^{15}s_{15}+2^{17}s_{13}+2^{21}s_{10}+2^{20}s_4+(1+2^8)s_0 \bmod (2^{31}-1)$;

2. $s_{16}=(v+u) \bmod (2^{31}-1)$;

3. If $s_{16}=0$, then set $s_{16}=2^{31}-1$;

4. $(s_1,s_2, \ldots,s_{15},s_{16}) \rightarrow (s_0,s_1, \ldots,s_{14},s_{15})$.

}

In the working mode, the LFSR does not receive any input, and it works as follows:

LFSRWithWorkMode()
{

1. $s_{16}=2^{15}s_{15}+2^{17}s_{13}+2^{21}s_{10}+2^{20}s_4+(1+2^8)s_0 \bmod (2^{31}-1)$;

2. If $s_{16}=0$, then set $s_{16}=2^{31}-1$;

3. $(s_1,s_2, \ldots,s_{15},s_{16}) \rightarrow (s_0,s_1, \ldots,s_{14},s_{15})$.

}

Informative note: Since the multiplication of a 31-bit string s by 2^i over GF(2^{31}-1) can be implemented by a cyclic shift of s to the left by i bits, only addition modulo 2^{31}-1 is needed in step 1 of the above functions. More precisely, step 1 of the function LFSRWithInitialisationMode can be implemented by

$$v=(s_{15}<<<_{31}15)+(s_{13}<<<_{31}17)+(s_{10}<<<_{31}21)+(s_4<<<_{31}20)+(s_0<<<_{31}8)+s_0 \bmod (2^{31}-1),$$

and the same implementation is needed for step 1 of the function LFSRWithWorkMode.

Informative note: For two elements a, b over GF(2^{31}-1), the computation of $v=a+b \bmod (2^{31}-1)$ can be done by (1) compute $v=a+b$; and (2) if the carry bit is 1, then set $v=v+1$.

3.3 The bit-reorganization

The middle layer of the algorithm is the bit-reorganization. It extracts 128 bits from the cells of the LFSR and forms 4 of 32-bit words, where the first three words will be used by the nonlinear function F in the bottom layer, and the last word will be involved in producing the keystream.

Let s_0, s_2, s_5, s_7, s_9, s_{11}, s_{14}, s_{15} be 8 cells of LFSR as in section 3.2. Then the bit-reorganization forms 4 of 32-bit words X_0, X_1, X_2, X_3 from the above cells as follows:

Bitreorganization()
{

1 . $X_0=s_{15H} \| s_{14L}$;

2. $X_1=s_{11L} \| s_{9H}$;

3 . $X_2=s_{7L} \| s_{5H}$;

4. $X_3=s_{2L} \|s_{0H}$.

}

Note: The s_i are 31-bit integers, so s_{iH} means bits 30...15 and not 31...16 of s_i, for $0\leq i\leq 15$.

3.4 The nonlinear function *F*

The nonlinear function F has 2 of 32-bit memory cells R_1 and R_2. Let the inputs to F be X_0, X_1 and X_2, which come from the outputs of the bit-reorganization (see section 3.3), then the function F outputs a 32-bit word W. The detailed process of F is as follows:

F (X_0, X_1, X_2)
{

1. $W=(X_0\oplus R_1) \boxplus R_2$;

2. $W_1= R_1\boxplus X_1$;

3. $W_2= R_2\oplus X_2$;

4. $R_1=S(L_1(W_{1L}\|W_{2H}))$;

5. $R_2=S(L_2(W_{2L}\|W_{1H}))$.

}

where S is a 32×32 S-box, see section 3.4.1, L_1 and L_2 are linear transforms as defined in section 3.4.2.

3.4.1 The S-box *S*

The 32×32 S-box S is composed of 4 juxtaposed 8×8 S-boxes, i.e., $S=(S_0,S_1,S_2,S_3)$, where $S_0=S_2$, $S_1=S_3$. The definitions of S_0 and S_1 can be found in table 3.1 and table 3.2 respectively.

Let x be an 8-bit input to S_0 (or S_1). Write x into two hexadecimal digits as $x=h\|l$, then the entry at the intersection of the h-th row and the l-th column in table 3.1 (or table 3.2) is the output of S_0 (or S_1).

Example 7 S_0(0x12)=0xF9 and S_1(0x34)=0xC0.

Let the 32-bit input X and the 32-bit output Y of the S-box S be as follows:

$$X = x_0 \| x_1 \| x_2 \| x_3,$$

$$Y = y_0 \| y_1 \| y_2 \| y_3,$$

where x_i and y_i are all bytes, i=0,1,2,3. Then we have

$$y_i=S_i(x_i),\ i=0,1,2,3.$$

Example 8 Let X=0x12345678 be a 32-bit input to S-box and Y its 32-bit output. Then we have

$$Y=S(X)=S_0(0x12)||S_1(0x34)||S_2(0x56)||S_3(0x78)=0xF9C05A4E.$$

Table 3.1. The S-box S_0

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	3E	72	5B	47	CA	E0	00	33	04	D1	54	98	09	B9	6D	CB
1	7B	1B	F9	32	AF	9D	6A	A5	B8	2D	FC	1D	08	53	03	90
2	4D	4E	84	99	E4	CE	D9	91	DD	B6	85	48	8B	29	6E	AC
3	CD	C1	F8	1E	73	43	69	C6	B5	BD	FD	39	63	20	D4	38
4	76	7D	B2	A7	CF	ED	57	C5	F3	2C	BB	14	21	06	55	9B
5	E3	EF	5E	31	4F	7F	5A	A4	0D	82	51	49	5F	BA	58	1C
6	4A	16	D5	17	A8	92	24	1F	8C	FF	D8	AE	2E	01	D3	AD
7	3B	4B	DA	46	EB	C9	DE	9A	8F	87	D7	3A	80	6F	2F	C8
8	B1	B4	37	F7	0A	22	13	28	7C	CC	3C	89	C7	C3	96	56
9	07	BF	7E	F0	0B	2B	97	52	35	41	79	61	A6	4C	10	FE
A	BC	26	95	88	8A	B0	A3	FB	C0	18	94	F2	E1	E5	E9	5D
B	D0	DC	11	66	64	5C	EC	59	42	75	12	F5	74	9C	AA	23
C	0E	86	AB	BE	2A	02	E7	67	E6	44	A2	6C	C2	93	9F	F1
D	F6	FA	36	D2	50	68	9E	62	71	15	3D	D6	40	C4	E2	0F
E	8E	83	77	6B	25	05	3F	0C	30	EA	70	B7	A1	E8	A9	65
F	8D	27	1A	DB	81	B3	A0	F4	45	7A	19	DF	EE	78	34	60

Table 3.2. The S-box S_1

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	55	C2	63	71	3B	C8	47	86	9F	3C	DA	5B	29	AA	FD	77
1	8C	C5	94	0C	A6	1A	13	00	E3	A8	16	72	40	F9	F8	42
2	44	26	68	96	81	D9	45	3E	10	76	C6	A7	8B	39	43	E1
3	3A	B5	56	2A	C0	6D	B3	05	22	66	BF	DC	0B	FA	62	48
4	DD	20	11	06	36	C9	C1	CF	F6	27	52	BB	69	F5	D4	87
5	7F	84	4C	D2	9C	57	A4	BC	4F	9A	DF	FE	D6	8D	7A	EB
6	2B	53	D8	5C	A1	14	17	FB	23	D5	7D	30	67	73	08	09
7	EE	B7	70	3F	61	B2	19	8E	4E	E5	4B	93	8F	5D	DB	A9
8	AD	F1	AE	2E	CB	0D	FC	F4	2D	46	6E	1D	97	E8	D1	E9
9	4D	37	A5	75	5E	83	9E	AB	82	9D	B9	1C	E0	CD	49	89
A	01	B6	BD	58	24	A2	5F	38	78	99	15	90	50	B8	95	E4
B	D0	91	C7	CE	ED	0F	B4	6F	A0	CC	F0	02	4A	79	C3	DE
C	A3	EF	EA	51	E6	6B	18	EC	1B	2C	80	F7	74	E7	FF	21
D	5A	6A	54	1E	41	31	92	35	C4	33	07	0A	BA	7E	0E	34

E	88	B1	98	7C	F3	3D	60	6C	7B	CA	D3	1F	32	65	04	28
F	64	BE	85	9B	2F	59	8A	D7	B0	25	AC	AF	12	03	E2	F2

Note: The entries in the above S-boxes S_0 and S_1 are all in hexadecimal representation.

3.4.2 The linear transforms L_1 and L_2

Both L_1 and L_2 are linear transforms from 32-bit words to 32-bit words, and are defined as follows:

$$L_1(X)=X\oplus (X<<<_{32}2)\oplus (X<<<_{32}10)\oplus (X<<<_{32}18)\oplus (X<<<_{32}24),$$

$$L_2(X)=X\oplus (X<<<_{32}8)\oplus (X<<<_{32}14)\oplus (X<<<_{32}22)\oplus (X<<<_{32}30).$$

3.5 Key loading

The key loading procedure will expand the initial key and the initial vector into 16 of 31-bit integers as the initial state of the LFSR. Let the 128-bit initial key k and the 128-bit initial vector iv be

$$k=k_0||k_1||k_2||\ldots||k_{15}$$

and

$$iv= iv_0|| iv_1|| iv_2||\ldots|| iv_{15}$$

respectively, where k_i and iv_i, $0\le i\le 15$, are all bytes. Then k and iv are loaded to the cells s_0, s_1, …, s_{15} of LFSR as follows:

1. Let D be a 240-bit long constant string composed of 16 substrings of 15 bits:

$$D= d_0||d_1 ||\ldots||d_{15},$$

where

$d_0 = 100010011010111_2$,

$d_1 = 010011010111100_2$,

$d_2 = 110001001101011_2$,

$d_3 = 001001101011110_2$,

$d_4 = 101011110001001_2$,

$d_5 = 011010111100010_2$,

$d_6 = 111000100110101_2$,

$d_7 = 000100110101111_2$,

$d_8= 100110101111000_2$,

$d_9 = 010111100010011_2$,

$d_{10} = 110101111000100_2$,

$d_{11} = 001101011110001_2$,

$d_{12} = 101111000100110_2$,

$d_{13} = 011110001001101_2$,

$d_{14} = 111100010011010_2$,

$d_{15} = 100011110101100_2$.

2. For $0 \leq i \leq 15$, let $s_i = k_i \| d_i \| iv_i$.

3.6 The execution of ZUC

The execution of ZUC has two stages: the initialization stage and the working stage.

3.6.1 The initialization stage

During the initialization stage, the algorithm calls the key loading procedure (see section 3.5) to load the 128-bit initial key k and the 128-bit initial vector iv into the LFSR, and set the 32-bit memory cells R_1 and R_2 to be all 0. Then the cipher runs the following operations 32 times:

1. Bitreorganization(); // see section 3.3
2. $w=F(X_0, X_1, X_2)$; // see section 3.4
3. LFSRWithInitialisationMode(w>>1). // see section 3.2

3.6.2 The working stage

After the initialization stage, the algorithm moves into the working stage. At the working stage, the algorithm executes the following operations once, and discards the output W of F:

1. Bitreorganization(); // see section 3.3
2. $F(X_0, X_1, X_2)$; //output discarded, see section 3.4
3. LFSRWithWorkMode(). // see section 3.2

Then the algorithm goes into the stage of producing keystream, i.e., for each iteration, the following operations are executed once, and a 32-bit word Z is produced as an output:

1. Bitreorganization(); // see section 3.3
2. $Z= F(X_0, X_1, X_2) \oplus X_3$; // for the definition of X_3, see section 3.3
3. LFSRWithWorkMode() . // see section 3.2

Appendix A: A C implementation of ZUC

```
/* ——————————————————————- */
typedef unsigned char u8;
typedef unsigned int u32;
/* ——————————————————————- */
/* the state registers of LFSR */
u32 LFSR_S0;
u32 LFSR_S1;
u32 LFSR_S2;
u32 LFSR_S3;
u32 LFSR_S4;
u32 LFSR_S5;
u32 LFSR_S6;
u32 LFSR_S7;
u32 LFSR_S8;
u32 LFSR_S9;
u32 LFSR_S10;
u32 LFSR_S11;
u32 LFSR_S12;
u32 LFSR_S13;
u32 LFSR_S14;
u32 LFSR_S15;

/* the registers of F */
u32 F_R1;
u32 F_R2;

/* the outputs of BitReorganization */
u32 BRC_X0;
u32 BRC_X1;
u32 BRC_X2;
u32 BRC_X3;

/* the s-boxes */
u8 S0[256] = {
    0x3e,0x72,0x5b,0x47,0xca,0xe0,0x00,0x33,0x04,0xd1,0x54,0x98,0x09,0xb9,0x6d,0xcb,
    0x7b,0x1b,0xf9,0x32,0xaf,0x9d,0x6a,0xa5,0xb8,0x2d,0xfc,0x1d,0x08,0x53,0x03,0x90,
    0x4d,0x4e,0x84,0x99,0xe4,0xce,0xd9,0x91,0xdd,0xb6,0x85,0x48,0x8b,0x29,0x6e,0xac,
    0xcd,0xc1,0xf8,0x1e,0x73,0x43,0x69,0xc6,0xb5,0xbd,0xfd,0x39,0x63,0x20,0xd4,0x38,
    0x76,0x7d,0xb2,0xa7,0xcf,0xed,0x57,0xc5,0xf3,0x2c,0xbb,0x14,0x21,0x06,0x55,0x9b,
    0xe3,0xef,0x5e,0x31,0x4f,0x7f,0x5a,0xa4,0x0d,0x82,0x51,0x49,0x5f,0xba,0x58,0x1c,
    0x4a,0x16,0xd5,0x17,0xa8,0x92,0x24,0x1f,0x8c,0xff,0xd8,0xae,0x2e,0x01,0xd3,0xad,
    0x3b,0x4b,0xda,0x46,0xeb,0xc9,0xde,0x9a,0x8f,0x87,0xd7,0x3a,0x80,0x6f,0x2f,0xc8,
    0xb1,0xb4,0x37,0xf7,0x0a,0x22,0x13,0x28,0x7c,0xcc,0x3c,0x89,0xc7,0xc3,0x96,0x56,
    0x07,0xbf,0x7e,0xf0,0x0b,0x2b,0x97,0x52,0x35,0x41,0x79,0x61,0xa6,0x4c,0x10,0xfe,
    0xbc,0x26,0x95,0x88,0x8a,0xb0,0xa3,0xfb,0xc0,0x18,0x94,0xf2,0xe1,0xe5,0xe9,0x5d,
    0xd0,0xdc,0x11,0x66,0x64,0x5c,0xec,0x59,0x42,0x75,0x12,0xf5,0x74,0x9c,0xaa,0x23,
    0x0e,0x86,0xab,0xbe,0x2a,0x02,0xe7,0x67,0xe6,0x44,0xa2,0x6c,0xc2,0x93,0x9f,0xf1,
    0xf6,0xfa,0x36,0xd2,0x50,0x68,0x9e,0x62,0x71,0x15,0x3d,0xd6,0x40,0xc4,0xe2,0x0f,
    0x8e,0x83,0x77,0x6b,0x25,0x05,0x3f,0x0c,0x30,0xea,0x70,0xb7,0xa1,0xe8,0xa9,0x65,
    0x8d,0x27,0x1a,0xdb,0x81,0xb3,0xa0,0xf4,0x45,0x7a,0x19,0xdf,0xee,0x78,0x34,0x60
};

u8 S1[256] =  {
    0x55,0xc2,0x63,0x71,0x3b,0xc8,0x47,0x86,0x9f,0x3c,0xda,0x5b,0x29,0xaa,0xfd,0x77,
    0x8c,0xc5,0x94,0x0c,0xa6,0x1a,0x13,0x00,0xe3,0xa8,0x16,0x72,0x40,0xf9,0xf8,0x42,
    0x44,0x26,0x68,0x96,0x81,0xd9,0x45,0x3e,0x10,0x76,0xc6,0xa7,0x8b,0x39,0x43,0xe1,
    0x3a,0xb5,0x56,0x2a,0xc0,0x6d,0xb3,0x05,0x22,0x66,0xbf,0xdc,0x0b,0xfa,0x62,0x48,
    0xdd,0x20,0x11,0x06,0x36,0xc9,0xc1,0xcf,0xf6,0x27,0x52,0xbb,0x69,0xf5,0xd4,0x87,
    0x7f,0x84,0x4c,0xd2,0x9c,0x57,0xa4,0xbc,0x4f,0x9a,0xdf,0xfe,0xd6,0x8d,0x7a,0xeb,
    0x2b,0x53,0xd8,0x5c,0xa1,0x14,0x17,0xfb,0x23,0xd5,0x7d,0x30,0x67,0x73,0x08,0x09,
    0xee,0xb7,0x70,0x3f,0x61,0xb2,0x19,0x8e,0x4e,0xe5,0x4b,0x93,0x8f,0x5d,0xdb,0xa9,
    0xad,0xf1,0xae,0x2e,0xcb,0x0d,0xfc,0xf4,0x2d,0x46,0x6e,0x1d,0x97,0xe8,0xd1,0xe9,
    0x4d,0x37,0xa5,0x75,0x5e,0x83,0x9e,0xab,0x82,0x9d,0xb9,0x1c,0xe0,0xcd,0x49,0x89,
    0x01,0xb6,0xbd,0x58,0x24,0xa2,0x5f,0x38,0x78,0x99,0x15,0x90,0x50,0xb8,0x95,0xe4,
    0xd0,0x91,0xc7,0xce,0xed,0x0f,0xb4,0x6f,0xa0,0xcc,0xf0,0x02,0x4a,0x79,0xc3,0xde,
    0xa3,0xef,0xea,0x51,0xe6,0x6b,0x18,0xec,0x1b,0x2c,0x80,0xf7,0x74,0xe7,0xff,0x21,
    0x5a,0x6a,0x54,0x1e,0x41,0x31,0x92,0x35,0xc4,0x33,0x07,0x0a,0xba,0x7e,0x0e,0x34,
    0x88,0xb1,0x98,0x7c,0xf3,0x3d,0x60,0x6c,0x7b,0xca,0xd3,0x1f,0x32,0x65,0x04,0x28,
    0x64,0xbe,0x85,0x9b,0x2f,0x59,0x8a,0xd7,0xb0,0x25,0xac,0xaf,0x12,0x03,0xe2,0xf2
};

/* the constants D */
u32 EK_d[16] = {
0x44D7, 0x26BC, 0x626B, 0x135E, 0x5789, 0x35E2, 0x7135, 0x09AF,
0x4D78, 0x2F13, 0x6BC4, 0x1AF1, 0x5E26, 0x3C4D, 0x789A, 0x47AC
};
```

```
/* ———————————————————————- */
/* c = a + b mod 2^31 - 1 */
u32 AddM(u32 a, u32 b)
{
    u32 c = a + b;
    if (c & 0x80000000)
    {
        c = (c & 0x7FFFFFFF) + 1;
    }
    return c;
}
/* LFSR with initialization mode */
#define MulByPow2(x, k) ((((x) << k) | ((x) >> (31 - k))) & 0x7FFFFFFF)
void LFSRWithInitialisationMode(u32 u)
{
    u32 f, v;

    f = LFSR_S0;
    v = MulByPow2(LFSR_S0, 8);
    f = AddM(f, v);

    v = MulByPow2(LFSR_S4, 20);
    f = AddM(f, v);

    v = MulByPow2(LFSR_S10, 21);
    f = AddM(f, v);

    v = MulByPow2(LFSR_S13, 17);
    f = AddM(f, v);

    v = MulByPow2(LFSR_S15, 15);
    f = AddM(f, v);

/* update the state */
    LFSR_S0 = LFSR_S1;
    LFSR_S1 = LFSR_S2;
    LFSR_S2 = LFSR_S3;
    LFSR_S3 = LFSR_S4;
    LFSR_S4 = LFSR_S5;
    LFSR_S5 = LFSR_S6;
    LFSR_S6 = LFSR_S7;
    LFSR_S7 = LFSR_S8;
    LFSR_S8 = LFSR_S9;
    LFSR_S9 = LFSR_S10;
    LFSR_S10 = LFSR_S11;
    LFSR_S11 = LFSR_S12;
    LFSR_S12 = LFSR_S13;
    LFSR_S13 = LFSR_S14;
    LFSR_S14 = LFSR_S15;
    LFSR_S15 = AddM(f, u);

/* adjust LFSR_S15 if LFSR_S15 is zero */
    if (LFSR_S15 == 0)
    {
        LFSR_S15 = 0x7FFFFFFF;
    }
}
/* LFSR with work mode */
void LFSRWithWorkMode()
{
    u32 f, v;

    f = LFSR_S0;
    v = MulByPow2(LFSR_S0, 8);
    f = AddM(f, v);

    v = MulByPow2(LFSR_S4, 20);
    f = AddM(f, v);

    v = MulByPow2(LFSR_S10, 21);
    f = AddM(f, v);

    v = MulByPow2(LFSR_S13, 17);
    f = AddM(f, v);

    v = MulByPow2(LFSR_S15, 15);
    f = AddM(f, v);
```

```
/* update the state */
    LFSR_S0 = LFSR_S1;
    LFSR_S1 = LFSR_S2;
    LFSR_S2 = LFSR_S3;
    LFSR_S3 = LFSR_S4;
    LFSR_S4 = LFSR_S5;
    LFSR_S5 = LFSR_S6;
    LFSR_S6 = LFSR_S7;
    LFSR_S7 = LFSR_S8;
    LFSR_S8 = LFSR_S9;
    LFSR_S9 = LFSR_S10;
    LFSR_S10 = LFSR_S11;
    LFSR_S11 = LFSR_S12;
    LFSR_S12 = LFSR_S13;
    LFSR_S13 = LFSR_S14;
    LFSR_S14 = LFSR_S15;
    LFSR_S15 = f;
}

/* BitReorganization */
void BitReorganization()
{
    BRC_X0 = ((LFSR_S15 & 0x7FFF8000) << 1) | (LFSR_S14 & 0xFFFF);
    BRC_X1 = ((LFSR_S11 & 0xFFFF) << 16) | (LFSR_S9 >> 15);
    BRC_X2 = ((LFSR_S7 & 0xFFFF) << 16) | (LFSR_S5 >> 15);
    BRC_X3 = ((LFSR_S2 & 0xFFFF) << 16) | (LFSR_S0 >> 15);
}

#define ROT(a, k) (((a) << k) | ((a) >> (32 - k)))

/* L1 */
u32 L1(u32 X)
{
    return (X ^ ROT(X, 2) ^ ROT(X, 10) ^ ROT(X, 18) ^ ROT(X, 24));
}

/* L2 */
u32 L2(u32 X)
{
    return (X ^ ROT(X, 8) ^ ROT(X, 14) ^ ROT(X, 22) ^ ROT(X, 30));
}

#define MAKEU32(a, b, c, d) (((u32)(a) << 24) | ((u32)(b) << 16)
                 | ((u32)(c) << 8) | ((u32)(d)))
/* F */
u32 F()
{
    u32 W, W1, W2, u, v;
    W = (BRC_X0 ^ F_R1) + F_R2;
    W1 = F_R1 + BRC_X1;
    W2 = F_R2 ^ BRC_X2;
    u = L1((W1 << 16) | (W2 >> 16));
    v = L2((W2 << 16) | (W1 >> 16));
    F_R1 = MAKEU32(S0[u >> 24], S1[(u >> 16) & 0xFF],
    S0[(u >> 8) & 0xFF], S1[u & 0xFF]);
    F_R2 = MAKEU32(S0[v >> 24], S1[(v >> 16) & 0xFF],
    S0[(v >> 8) & 0xFF], S1[v & 0xFF]);
    return W;
}

#define MAKEU31(a, b, c) (((u32)(a) << 23) | ((u32)(b) << 8) | (u32)(c))

/* initialize */
void Initialization(u8* k, u8* iv)
{
    u32 w, nCount;

/* expand key */
    LFSR_S0 = MAKEU31(k[0], EK_d[0], iv[0]);
    LFSR_S1 = MAKEU31(k[1], EK_d[1], iv[1]);
    LFSR_S2 = MAKEU31(k[2], EK_d[2], iv[2]);
    LFSR_S3 = MAKEU31(k[3], EK_d[3], iv[3]);
    LFSR_S4 = MAKEU31(k[4], EK_d[4], iv[4]);
    LFSR_S5 = MAKEU31(k[5], EK_d[5], iv[5]);
    LFSR_S6 = MAKEU31(k[6], EK_d[6], iv[6]);
    LFSR_S7 = MAKEU31(k[7], EK_d[7], iv[7]);
```

```
    LFSR_S8 = MAKEU31(k[8], EK_d[8], iv[8]);
    LFSR_S9 = MAKEU31(k[9], EK_d[9], iv[9]);
    LFSR_S10 = MAKEU31(k[10], EK_d[10], iv[10]);
    LFSR_S11 = MAKEU31(k[11], EK_d[11], iv[11]);
    LFSR_S12 = MAKEU31(k[12], EK_d[12], iv[12]);
    LFSR_S13 = MAKEU31(k[13], EK_d[13], iv[13]);
    LFSR_S14 = MAKEU31(k[14], EK_d[14], iv[14]);
    LFSR_S15 = MAKEU31(k[15], EK_d[15], iv[15]);

/* set F_R1 and F_R2 to zero */
    F_R1 = 0;
    F_R2 = 0;
    nCount = 32;
    while (nCount > 0)
    {
        BitReorganization();
        w = F();
        LFSRWithInitialisationMode(w >> 1);
        nCount --;
    }
}

void GenerateKeystream(u32* pKeystream, int KeystreamLen)
{
    int i;

    {

        BitReorganization();
        F();            /* discard the output of F */
        LFSRWithWorkMode();
    }
    for (i = 0; i < KeystreamLen; i ++)
    {
        BitReorganization();
        pKeystream[i] = F() ^ BRC_X3;
        LFSRWithWorkMode();
    }
}
```