INTERNET DRAFT
Document: draft-thomas-w7cipher-00.ps
Category: Informational

S. Thomas, D. Anthony
Wave7 Optics
T. Berson
Anagram Laboratories
G. Gong
University of Waterloo
October 2001

The W7 Stream Cipher Algorithm

Status of this Memo

This document is an Internet-Draft and is NOT offered in accordance with Section 10 of RFC2026, and the author does not provide the IETF with any rights other than to publish as an Internet-Draft

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts. Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet- Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

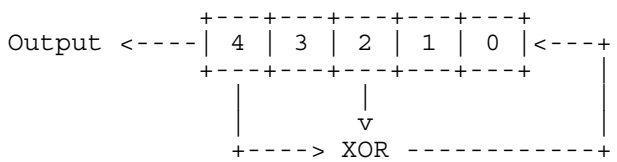
Abstract

This document specifies the W7 cipher algorithm. The W7 algorithm is a byte-wide, synchronous stream cipher optimized for efficient hardware implementation at very high data rates. It is a symmetric key algorithm supporting key lengths of 128 bits. In addition to the algorithm specification, this document contains test vectors and a representative software implementation in the C language. The .ps version includes a graphical illustration of the algorithm.

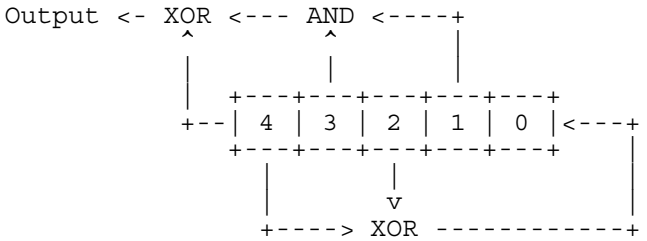
1. Notation

The W7 cipher uses linear feedback shift registers (LFSRs) as its basic building block. The figure below shows a simple LFSR, not one that is used in the actual algorithm. The figure shows a 5-bit shift register. The bits are numbered 4 to 0 from left to right, and the output is taken from bit 4.

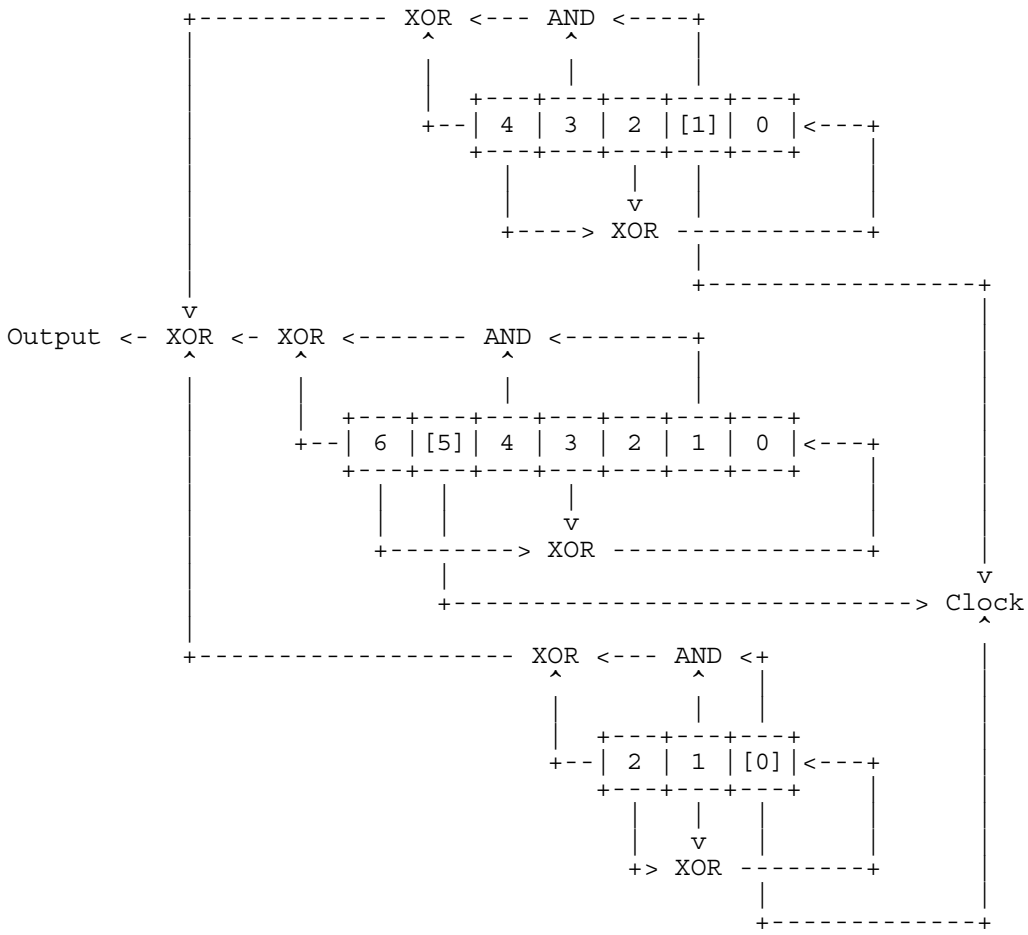
At each clock, the bits shift from right to left, and the bit shifted into bit 0 is the exclusive-OR of bit 4 and bit 2.



The W7 cipher does not use the output bit directly. Instead, it uses a non-linear filtering function that is a combination of several bits in the shift register. The figure below shows a simple example of this approach. The actual output is the exclusive-OR of two quantities: (a) the shift register output and (b) the logical-AND of bits 3 and 1.



For additional security, the W7 cipher combines three of these output-filtered LFSRs. The actual keystream output is taken as the exclusive-OR of the three individual LFSRs. The three LFSRs together also determine when each shift register is clocked. One bit in each register is designated as the clock tap for that register. At each clock cycle, the majority value for those three taps determines which shift registers actually advance; only those shift registers whose clock taps agree with the majority advance. The following figure shows an example of this structure. In the figure, the clock taps are bits 1, 5, and 0.



2. The W7 Implementation

The W7 cipher uses eight combined LFSRs operating in parallel. Each combined LFSR generates one bit of the keystream. Each byte of plaintext is exclusive-ORed with the eight parallel keystreams to generate a byte of ciphertext. Decryption uses the exact same procedure to recover the plaintext from the ciphertext.

The LFSRs for each of the eight streams are 38, 43, and 47 bits long. Their initial state--which is identical for all eight keystreams--is the symmetric encryption key. Specifically, the 128 bits of the symmetric encryption key map to the LFSR initial states according to the following table. (Bit 0 is the least significant bit of the key.)

```
LFSRa (38 bits)  bit  0 = key bit 0
                 bit  1 = key bit 1
                 ...
                 bit 36 = key bit 36
                 bit 37 = key bit 37

LFSRb (43 bits)  bit  0 = key bit 38
                 bit  1 = key bit 39
                 ...
                 bit 41 = key bit 79
                 bit 42 = key bit 80

LFSRc (47 bits)  bit  0 = key bit 81
                 bit  1 = key bit 82
                 ...
                 bit 45 = key bit 126
                 bit 46 = key bit 127
```

The next table lists the parameters for each of the eight LFSR combinations. The table shows the size of each LFSR, its feedback taps, clock tap, and output filter taps.

The LFSR number in the table represents the bit position in the plaintext and ciphertext that the LFSR affects. For example, LFSR combination 0 modifies the least significant bit of the plaintext or ciphertext, while LFSR combination 7 modifies the most significant bit.

The output filter specification in the table contains a series of terms, enclosed in parenthesis. The bits represented within each term are logically-ANDed together, and the resulting four values are exclusive-ORed to produce the final shift register output.

The output of each keystream is the exclusive-OR of the outputs of each of the three shift registers.

The first 1031 bytes of keystream are discarded. The 1032nd byte of keystream is exclusive-ORed with the first byte of plaintext to create the first byte of ciphertext.

```
LFSR Combination 0
    LFSR 0a (38 bits)
       Feedback Taps 37 32 29 27 26 21 20 14 12 11 10 9 8 5 2 0
       Clock Tap     22
       Output Filter 37 (36, 33) (32, 29) (28, 25, 22)
    LFSR 0b (43 bits)
       Feedback Taps 42 5 3 2
       Clock Tap     25
       Output Filter 42 (41, 39) (38, 36) (35, 33, 31)
    LFSR 0c (47 bits)
       Feedback Taps 46 4
       Clock Tap     27
       Output Filter 46 (45, 40) (39, 34) (33, 28, 23)
LFSR Combination 1
    LFSR 1a (38 bits)
       Feedback Taps 37 36 34 31 28 27 26 25 24 22 16 15 10 9 7 4
       Clock Tap     15
       Output Filter 37 (3, 0) (7, 4) (14, 11, 8)
    LFSR 1b (43 bits)
       Feedback Taps 42 39 38 36
       Clock Tap     18
       Output Filter 42 (2, 0) (5, 3) (10, 8, 6)
    LFSR 1c (47 bits)
       Feedback Taps 46 41
       Clock Tap     20
       Output Filter 46 (5, 0) (11, 6) (22, 17, 12)
LFSR Combination 2
    LFSR 2a (38 bits)
       Feedback Taps 37 23 21 18 17 16 14 10 9 7 4 0
       Clock Tap     21
       Output Filter 37 (35, 32) (31, 28) (27, 24, 21)
    LFSR 2b (43 bits)
       Feedback Taps 42 29 16 5 4 3 2 0
       Clock Tap     24
       Output Filter 42 (40, 38) (37, 35) (34, 32, 30)
    LFSR 2c (47 bits)
       Feedback Taps 46 32 18 4
       Clock Tap     26
       Output Filter 46 (44, 39) (38, 33) (32, 27, 22)
LFSR Combination 3
    LFSR 3a (38 bits)
       Feedback Taps 37 36 32 29 27 26 22 20 19 18 15 13
       Clock Tap     16
       Output Filter 37 (4, 1) (8, 5) (15, 12, 9)
    LFSR 3b (43 bits)
       Feedback Taps 42 41 39 38 37 36 25 12
       Clock Tap     19
       Output Filter 42 (3, 1) (6, 4) (11, 9, 7)
    LFSR 3c (47 bits)
       Feedback Taps 46 41 27 13
       Clock Tap     21
       Output Filter 46 (4, 1) (12, 7) (21, 18, 13)
```

```
LFSR Combination 4
    LFSR 4a (38 bits)
       Feedback Taps 37 24 22 11 7 5 3 1
       Clock Tap     20
       Output Filter 37 (34, 31) (30, 27) (26, 23, 20)
    LFSR 4b (43 bits)
       Feedback Taps 42 34 26 19 18 17 12 5 4 3
       Clock Tap     23
       Output Filter 42 (39, 37) (36, 34) (33, 31, 29)
    LFSR 4c (47 bits)
       Feedback Taps 46 4 3 0
       Clock Tap     25
       Output Filter 46 (43, 38) (37, 32) (31, 26, 21)
LFSR Combination 5
    LFSR 5a (38 bits)
       Feedback Taps 37 35 33 31 29 25 14 12
       Clock Tap     17
       Output Filter 37 (5, 2) (9, 6) (16, 13, 10)
    LFSR 5b (43 bits)
       Feedback Taps 42 38 37 36 29 24 23 22 15 7
       Clock Tap     20
       Output Filter 42 (4, 2) (7, 5) (12, 10, 8)
    LFSR 5c (47 bits)
       Feedback Taps 46 45 42 41
       Clock Tap     22
       Output Filter 46 (5, 2) (13, 8) (22, 19, 14)
LFSR Combination 6
    LFSR 6a (38 bits)
       Feedback Taps 37 5 4 0
       Clock Tap     19
       Output Filter 37 (33, 30) (29, 26) (25, 22, 19)
    LFSR 6b (43 bits)
       Feedback Taps 42 29 28 25 17 14 13 9 4 3
       Clock Tap     22
       Output Filter 42 (38, 36) (35, 33) (32, 30, 28)
    LFSR 6c (47 bits)
       Feedback Taps 46 32 18 10 7 4
       Clock Tap     24
       Output Filter 46 (42, 37) (36, 31) (30, 25, 20)
LFSR Combination 7
    LFSR 7a (38 bits)
       Feedback Taps 37 36 32 31
       Clock Tap     18
       Output Filter 37 (6, 3) (10, 7) (17, 14, 11)
    LFSR 7b (43 bits)
       Feedback Taps 42 38 37 32 28 27 24 16 13 12
       Clock Tap     21
       Output Filter 42 (5, 3) (8, 6) (13, 11, 9)
    LFSR 7c (47 bits)
       Feedback Taps 46 41 38 35 27 13
       Clock Tap     23
       Output Filter 46 (6, 3) (14, 9) (23, 20, 15)
```

3. W7 Properties

The W7 cipher has several mathematical properties that provide bounds on its resistance to various attacks. Those properties include

- The period of the output sequence of each LFSR combination is between 3E38 and 1E77.
- The linear span of each LFSR combination is at least 2E42.
- The output of each filtered LFSR is balanced with respect to the number of 0s and 1s in each period.

4. Security Considerations

The W7 cipher is a traditional synchronous stream cipher. All standard precautions for using stream ciphers apply to W7. In particular:

- A key consisting entirely of zero bits should not be used. Such a key causes the cipher algorithm to have no effect, so that the plaintext and ciphertext are equal.
- The same initialized keystream should never be used to encrypt more than one plaintext message. An attacker possessing two such ciphertexts could trivially recover the difference between the plaintexts by subtracting the two ciphertexts. (Note: there is no problem using the same keystream to encrypt successive messages, so long as the keystream is not re-initialized between messages.)

Because W7 uses a total of eight shift register combinations, each of which is initialized with the same 128-bit key, it may be tempting to increase the key size and initialize different shift register combinations with different parts of the lengthened initial key. This approach represents only a very modest improvement in the security of the algorithm, however. Because each shift register combination operates in isolation of the others, it is possible to attack each bit separately. Lengthening the key size to 1024 bits in this manner, for example, only increases the difficulty of a brute force attack by a factor of eight. The resulting cipher has an effective key length of 131 bits, therefore, rather than 1024.

5. Author's Addresses

Stephen Thomas
Wave7 Optics
1075 Windward Ridge Parkway Suite 170
Alpharetta, GA 30005
USA
Phone: +1 678 339 1040
Email: stephen.thomas@wave7optics.com

Deven Anthony
Wave7 Optics
1075 Windward Ridge Parkway Suite 170
Alpharetta, GA 30005
USA
Phone: +1 678 339 1040
Email: deven.anthony@wave7optics.com

Tom Berson
Anagram Laboratories
P.O. Box 791
Palo Alto, CA 94301
USA
Phone: +1 650 324 0100
Email: berson@anagram.com

Guang Gong
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
Canada
Phone: +1 519 888 4567 x5650
Email: ggong@cacr.math.uwaterloo.ca

Appendix A. Test Vectors

Key (hexadecimal, most significant byte first):

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

Plaintext (hexadecimal, first byte first):

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

Ciphertext (hexadecimal, first byte first):

```
B6 FD 65 03 38 08 B6 0A 6F 47 10 CF DA FB 1F 7E
71 66 40 5E 38 1D 26 07 D4 7D 58 07 5F B7 93 83
EC 28 FC B7 45 7E 56 EF 3D 32 12 57 F0 6C 9D 85
A6 11 66 02 1A BC 51 B3 D8 AD 11 0D A3 FE ED F0
D6 E4 69 7C 3F 24 9F E1 14 9E 87 E3 9B 11 5A E1
B4 AE 27 74 BE 66 02 A1 59 F8 EA 70 04 39 DA 54
0A F3 B1 13 9D 3A 99 D6 9C 18 18 3C 32 EA 7F 72
89 68 4A D4 77 4E F8 25 9D 94 8C B4 6F 50 F2 33
2B F4 72 AA 1B 62 EC F3 64 B0 6F 87 07 B0 31 7E
EC 11 C0 7E 82 6B 60 7C 17 73 A1 71 1E 7F 2D C7
6C E6 35 9C DC 1F 0B 55 18 C8 32 08 19 0C 19 03
8B FE 54 8D A3 C8 69 FD 5E 09 96 53 66 D3 1B 6A
F2 9D D0 96 02 DF 9A BE 72 42 59 1E CA D1 A2 33
AE EA 65 4D D1 BB 76 CC 13 70 E1 37 0D F4 F2 23
8E A9 09 71 89 4C 16 C1 DC B0 9F 5E 94 68 AA 0F
1F D9 0C 70 F5 02 CD 67 17 6C 14 78 0B 0E 6B 1B
```

Appendix B. Representative Implementation

```
/*
 * This file contains a sample implementation of the W7 stream
 * cipher. It is not a complete program, but rather a collection
 * of callable functions.
 *
 * For convenience, this file includes both header definitions
 * and function definitions. In actual projects these would
 * likely be separate.
 *
 * Although developed strictly in ANSI C, this implementation is
 * organized according to object-oriented principles. The primary
 * (and only public) object is the W7 object, which is used for
 * both encryption and decryption. The public interface to that
 * object consists of the following five methods.
 *
 * W7New()      -  Create a new W7 object for subsequent use
 * W7Rekey()    -  Reset the key for the W7 object
 * W7Encrypt()  -  Encrypt an array of bytes
 * W7Decrypt()  -  Decrypt an array of bytes
 * W7Delete()   -  Destroy a previously created W7 object
 */
```

```
/*
 * The first part of this file is the private definition of the
 * W7 object. Normally, this part would be in a separate, private
 * header file. Functions that use the W7 object do not need to
 * know the structure of that object.
 *
 * The W7 object is built using a hierarchy of lower layer objects.
 * From the top-down, that hierarchy looks like:
 *
 *   W7 object (privately defined as W7CPHR)
 *   |
 *   +- W7STRM object (an individual bit stream, 8 total per W7CPHR)
 *      |
 *      +- W7LFSR (an individual shift register, 3 total per W7STRM)
 *         |
 *         + W7FLTR (an output filter term, 3 total per W7LFSR)
 *
 *
 * The W7 cipher is optimally implemented with 64-bit words;
 * however, this implementation assumes native support for
 * only 32-bit words. It is straightforward (though tedious) to
 * extend the implementation to 16-bit or 8-bit processors. For
 * native 64-bit processors, considerable, though easily
 * implemented, simplification is possible.
```

```
 *
 * Note that these definitions use many "magic number" values
 * rather than C-language constants. Although contrary to standard
 * programming practice, the approach is deliberate here. In this
 * case, these numbers are truly magic numbers, as all facets of the
 * W7 algorithm have been carefully designed. None of these values
 * should be changed; doing so may well compromise the security
 * of the cipher. The use of hard-coded magic numbers is intended
 * to discourage such changes.
 */



/*----- general types, normally defined elsewhere -----*/

typedef unsigned long UINT32;
typedef unsigned char UINT8;


/*----- W7FILT  a single term in the output filter -----*/

/*
 * A single term in an LFSR output filter. In the output
 * filter algorithm, each term is the logical AND of either
 * two or three individual bits. This object represents
 * those bits with a bit mask. Mask bits that are set to
 * one are included in the term; zero bits are not.
 */

typedef struct
{
    UINT32  filtMS;      /* most significant filter mask */
    UINT32  filtLS;      /* least significant filter mask */
    int     filtBits;    /* how many bits (2 or 3) included */
}
W7FILT;


/*----- W7LFSR  a single linear feedback shift register -----*/

/*
 * The linear feedback shift register which is the basic
 * building block of the W7 cipher. In addition to the
 * shift register itself, this object holds a bit mask
 * that represents the feedback taps, a bit mask that
 * holds the clock tap, and the three output filter terms.
 * As with the W7FILT object, these 64-bit masks are
 * divided into most significant and least significant
 * 32-bit words for easy operation on 32-bit processors.
 */
```

```
typedef struct
{
    unsigned lfsrSize;    /* must be between 33-64 */
    UINT32   lfsrMS;      /* bits 63-32 */
    UINT32   lfsrLS;      /* bits 31-0  */
    UINT32   lfsrTapMS;   /* most significant tap mask */
    UINT32   lfsrTapLS;   /* least significant tap mask */
    UINT32   lfsrClockMS; /* most significant clock mask */
    UINT32   lfsrClockLS; /* least significant clock mask */
    W7FILT   lfsrFilt[3]; /* output filter terms */
}
W7LFSR;


/*----- W7STRM  a single-bit keystream -----*/

/*
 * A single-bit keystream using three linear feedback
 * shift registers in combination. The details of how
 * each LFSR contributes to the keystream are properties
 * of the LFSRs themselves, not this object.
 */

typedef struct
{
    W7LFSR  strmLfsr[3];  /* shift registers */
}
W7STRM;


/*----- W7CPHR  the full keystream with all 8 bits -----*/

/*
 * Nothing too fancy here, the full keystream is simply
 * eight individual single-bit keystreams operating in
 * parallel.
 */

typedef struct
{
    W7STRM cphrStrm[8];   /* bit streams */
}
W7CPHR;


#define W7 W7CPHR  /* for the private interface, W7 = W7CPHR */
```

```
/*
 * Here's the public interface to the W7 object. This section would
 * normally be placed in a public header file. Note that in the
 * public header file the W7 object is defined as a void so its
 * structure remains opague to callers. The #ifndef directives
 * can turn off that definition when the private header file has
 * been included prior to this section; that allows the actual
 * implementation to include the public header file as well so
 * it can verify function prototypes.
 */



#ifndef W7
#define W7 void  /* for public interface, the W7 object is opague */
#endif



/*----- W7New()  create a new W7 object for subsequent use -----*/

W7 *                          /* returns pointer to object or null */
W7New
(
    unsigned char *pKey       /* array of octets containing key */
);


/*----- W7Rekey()  resets the key for the W7 object -----*/

unsigned                      /* returns false (0) on error */
W7Rekey
(
    W7            *pW7,       /* W7 object to rekey */
    unsigned char *pKey       /* array of octets containing key */
);


/*----- W7Encrypt()  encrypt plaintext into ciphertext -----*/

unsigned                         /* returns false (0) on error */
W7Encrypt
(
    W7            *pW7,          /* W7 object for encryption */
    unsigned       dataLen,      /* number of octets of data */
    unsigned char *pPlaintext,   /* plaintext input */
    unsigned char *pCiphertext   /* place for ciphertext output */
);
```

```
/*----- W7Decrypt()  decrypt ciphertext to recover plaintext -----*/

unsigned                        /* returns false (0) on error */
W7Decrypt
(
    W7            *pW7,         /* W7 object for decryption */
    unsigned       dataLen,     /* number of octets of data */
    unsigned char *pCiphertext, /* ciphertext input */
    unsigned char *pPlaintext   /* place to store plaintext output
*/
);


/*----- W7Delete()  destroy a W7 object that's no longer needed ----
-*/

unsigned                    /* returns false (0) on error */
W7Delete
(
    W7            *pW7      /* W7 object to destroy */
);









/*
 * Now we get to the actual implementation. This section would
 * normally be alone in a .c file. Start with function prototypes
 * so we can develop the code itself in a more natural, top-down
 * manner.
 */


/*----- standard header files -----*/

/*
 * The functions below use malloc() and free(), and the debugging
 * functions use printf(), so pick up their prototypes to make
 * sure we're using them correctly.
 */

#include <stdio.h>
#include <stdlib.h>
```

```
/*----- prototypes for private functions -----*/

/*
 * Only prototypes for private functions are needed here; public
 * prototypes are defined earlier. Also, because these functions
 * are defined in this file, documentation is omitted.
 */

void     W7CPHRInit (W7CPHR *pCphr);
void     W7CPHRSetKey (W7CPHR *pCphr, UINT8 *pKey);
UINT8    W7CPHRNext (W7CPHR *pCphr);
void     W7CPHRDebug (W7CPHR *pCphr);

unsigned W7STRMAdvance (W7STRM *pStrm);
void     W7STRMDebug (W7STRM *pStrm);

void     W7LFSRInit (W7LFSR *pLfsr, unsigned size, unsigned clock);
void     W7LFSRAddTap (W7LFSR *pLfsr, unsigned tapNum);
void     W7LFSRSetKey (W7LFSR *pLfsr, UINT32 keyMS, UINT32 keyLS);
void     W7LFSRShift (W7LFSR *pLfsr);
unsigned W7LFSRGetClock (W7LFSR *pLfsr);
unsigned W7LFSRGetOutput (W7LFSR *pLfsr);
void     W7LFSRDebug (W7LFSR *pLfsr);

void     W7FILTSet2Taps (W7FILT *pFilt, int tap1, int tap2);
void     W7FILTSet3Taps (W7FILT *pFilt, int tap1, int tap2, int
tap3);
void     W7FILTDebug (W7FILT *pFilt);

unsigned bit_count (UINT32 x);
```

```
/*----- W7New()  create a new W7 object for subsequent use -----*/

/*
 * Allocate memory for a W7CPHR object, initialize it, and set its
 * key. We use W7Rekey to set the keys (instead of W7CPHRSetKey) to
 * take advantage of the public function's error checking.
 */

W7 *                            /* returns pointer to object or null */
W7New
(
    unsigned char *pKey         /* array of octets containing key */
)
{
    W7 *pW7;

    if (0 != (pW7 = malloc(sizeof(W7))))
    {
        W7CPHRInit(pW7);
        if (!W7Rekey(pW7, pKey))
        {
            free(pW7);
            pW7 = 0;
        }
    }

    return(pW7);
}
```

```
/*----- W7Rekey()  resets the key for the W7 object -----*/

unsigned                    /* returns false (0) on error */
W7Rekey
(
    W7             *pW7,    /* W7 object to rekey */
    unsigned char *pKey     /* array of octets containing key */
)
{
    int       i;            /* loop counter */
    unsigned  okay=0;       /* return value, assume error */

    /* make sure the key isn't all zero */
    for (i=0; i<16; i++)
        if (0 != pKey[i]) break;

    /* if we found a non-zero byte, key is okay */
    if (i != 16)
    {
        W7CPHRSetKey(pW7, pKey);
        /* discard the first 100 bytes of keystream */
        for (i=0; i<100; i++) (void)W7CPHRNext(pW7);
        okay = 1;
    }
    return(okay);
}



/*----- W7Encrypt()  encrypt plaintext into ciphertext -----*/

/*
 * Encryption is trivial; just exclusive-OR the plaintext with
 * successive bytes of the full keystream.
 */

unsigned                        /* returns false (0) on error */
W7Encrypt
(
    W7             *pW7,        /* W7 object for encryption */
    unsigned       dataLen,     /* number of octets of data */
    unsigned char *pPlaintext,  /* plaintext input */
    unsigned char *pCiphertext  /* place for ciphertext output */
)
{
    while (dataLen-- > 0)
    {
        *pCiphertext++ = *pPlaintext++ ^ W7CPHRNext(pW7);
    }

    return(1);                  /* no errors defined so far */
}
```

```
/*----- W7Decrypt()  decrypt ciphertext to recover plaintext -----*/

/*
 * Since encryption is simply an exclusive-OR with the keystream,
 * reversing it is the same: exclusive-OR with the keystream.
 */

unsigned                        /* returns false (0) on error */
W7Decrypt
(
    W7             *pW7,        /* W7 object for decryption */
    unsigned        dataLen,    /* number of octets of data */
    unsigned char *pCiphertext, /* ciphertext input */
    unsigned char *pPlaintext   /* place for plaintext output */
)
{
    /* decryption is same as encryption */
    return(W7Encrypt(pW7, dataLen, pCiphertext, pPlaintext));
}


/*----- W7Delete()  destroy a W7 object no longer needed -----*/

/*
 * Nothing to do here but free the memory.
 */

unsigned                        /* returns false (0) on error */
W7Delete
(
    W7             *pW7         /* W7 object to destroy */
)
{
    free(pW7);
    return(1);                  /* no errors defined so far */
}
```

```
/*----- W7CPHRInit  initialize the W7CPHR object -----*/

/*
 * This function is *the* key function for the cipher, as it sets up
 * the W7CPHR object with appropriate parameters. It's also pretty
 * boring; just feeding a bunch of (seemingly) arbitrary numbers
 * into the initial initialization functions. The LFSR numbering
 * scheme in the comments refers to the W7 algorithm documentation.
 *
 * Note that this function initializes the cipher parameters but
 * it does not set the key.
 */

void                    /* nothing to return */
W7CPHRInit
(
    W7CPHR *pCphr       /* object to initialize */
)
{
    /* LFSR 0a */
    W7LFSRInit(&(pCphr->cphrStrm[0].strmLfsr[0]), 38, 22);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 37);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 32);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 29);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 27);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 26);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 21);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 20);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 14);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 12);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 11);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 10);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 9);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 8);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 5);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 2);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[0]), 0);
    W7FILTSet2Taps(&(pCphr->cphrStrm[0].strmLfsr[0].lfsrFilt[0]),
                   36, 33);
    W7FILTSet2Taps(&(pCphr->cphrStrm[0].strmLfsr[0].lfsrFilt[1]),
                   32, 29);
    W7FILTSet3Taps(&(pCphr->cphrStrm[0].strmLfsr[0].lfsrFilt[2]),
                   28, 25, 22);

    /* LFSR 0b */
    W7LFSRInit(&(pCphr->cphrStrm[0].strmLfsr[1]), 43, 25);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[1]), 42);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[1]), 5);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[1]), 3);
    W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[1]), 2);
    W7FILTSet2Taps(&(pCphr->cphrStrm[0].strmLfsr[1].lfsrFilt[0]),
                   41, 39);
```

```
        W7FILTSet2Taps(&(pCphr->cphrStrm[0].strmLfsr[1].lfsrFilt[1]),
                       38, 36);
        W7FILTSet3Taps(&(pCphr->cphrStrm[0].strmLfsr[1].lfsrFilt[2]),
                       35, 33, 31);

        /* LFSR 0c */
        W7LFSRInit(&(pCphr->cphrStrm[0].strmLfsr[2]), 47, 27);
        W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[2]), 46);
        W7LFSRAddTap(&(pCphr->cphrStrm[0].strmLfsr[2]), 4);
        W7FILTSet2Taps(&(pCphr->cphrStrm[0].strmLfsr[2].lfsrFilt[0]),
                       45, 40);
        W7FILTSet2Taps(&(pCphr->cphrStrm[0].strmLfsr[2].lfsrFilt[1]),
                       39, 34);
        W7FILTSet3Taps(&(pCphr->cphrStrm[0].strmLfsr[2].lfsrFilt[2]),
                       33, 28, 23);

        /* LFSR 1a */
        W7LFSRInit(&(pCphr->cphrStrm[1].strmLfsr[0]), 38, 15);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 37);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 36);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 34);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 31);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 28);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 27);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 26);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 25);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 24);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 22);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 16);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 15);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 10);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 9);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 7);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[0]), 4);
        W7FILTSet2Taps(&(pCphr->cphrStrm[1].strmLfsr[0].lfsrFilt[0]),
                       3, 0);
        W7FILTSet2Taps(&(pCphr->cphrStrm[1].strmLfsr[0].lfsrFilt[1]),
                       7, 4);
        W7FILTSet3Taps(&(pCphr->cphrStrm[1].strmLfsr[0].lfsrFilt[2]),
                       14, 11, 8);

        /* LFSR 1b */
        W7LFSRInit(&(pCphr->cphrStrm[1].strmLfsr[1]), 43, 18);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[1]), 42);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[1]), 39);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[1]), 38);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[1]), 36);
        W7FILTSet2Taps(&(pCphr->cphrStrm[1].strmLfsr[1].lfsrFilt[0]),
                       2, 0);
        W7FILTSet2Taps(&(pCphr->cphrStrm[1].strmLfsr[1].lfsrFilt[1]),
                       5, 3);
        W7FILTSet3Taps(&(pCphr->cphrStrm[1].strmLfsr[1].lfsrFilt[2]),
                       10, 8, 6);
```

```
        /* LFSR 1c */
        W7LFSRInit(&(pCphr->cphrStrm[1].strmLfsr[2]), 47, 20);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[2]), 46);
        W7LFSRAddTap(&(pCphr->cphrStrm[1].strmLfsr[2]), 41);
        W7FILTSet2Taps(&(pCphr->cphrStrm[1].strmLfsr[2].lfsrFilt[0]),
                       5, 0);
        W7FILTSet2Taps(&(pCphr->cphrStrm[1].strmLfsr[2].lfsrFilt[1]),
                       11, 6);
        W7FILTSet3Taps(&(pCphr->cphrStrm[1].strmLfsr[2].lfsrFilt[2]),
                       22, 17, 12);

        /* LFSR 2a */
        W7LFSRInit(&(pCphr->cphrStrm[2].strmLfsr[0]), 38, 21);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 37);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 23);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 21);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 18);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 17);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 16);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 14);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 10);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 9);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 7);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 4);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[0]), 0);
        W7FILTSet2Taps(&(pCphr->cphrStrm[2].strmLfsr[0].lfsrFilt[0]),
                       35, 32);
        W7FILTSet2Taps(&(pCphr->cphrStrm[2].strmLfsr[0].lfsrFilt[1]),
                       31, 28);
        W7FILTSet3Taps(&(pCphr->cphrStrm[2].strmLfsr[0].lfsrFilt[2]),
                       27, 24, 21);

        /* LFSR 2b */
        W7LFSRInit(&(pCphr->cphrStrm[2].strmLfsr[1]), 43, 24);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[1]), 42);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[1]), 29);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[1]), 16);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[1]), 5);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[1]), 4);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[1]), 3);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[1]), 2);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[1]), 0);
        W7FILTSet2Taps(&(pCphr->cphrStrm[2].strmLfsr[1].lfsrFilt[0]),
                       40, 38);
        W7FILTSet2Taps(&(pCphr->cphrStrm[2].strmLfsr[1].lfsrFilt[1]),
                       37, 35);
        W7FILTSet3Taps(&(pCphr->cphrStrm[2].strmLfsr[1].lfsrFilt[2]),
                       34, 32, 30);

        /* LFSR 2c */
        W7LFSRInit(&(pCphr->cphrStrm[2].strmLfsr[2]), 47, 26);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[2]), 46);
```

```
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[2]), 32);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[2]), 18);
        W7LFSRAddTap(&(pCphr->cphrStrm[2].strmLfsr[2]), 4);
        W7FILTSet2Taps(&(pCphr->cphrStrm[2].strmLfsr[2].lfsrFilt[0]),
                       44, 39);
        W7FILTSet2Taps(&(pCphr->cphrStrm[2].strmLfsr[2].lfsrFilt[1]),
                       38, 33);
        W7FILTSet3Taps(&(pCphr->cphrStrm[2].strmLfsr[2].lfsrFilt[2]),
                       32, 27, 22);

        /* LFSR 3a */
        W7LFSRInit(&(pCphr->cphrStrm[3].strmLfsr[0]), 38, 16);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 37);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 36);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 32);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 29);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 27);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 26);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 22);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 20);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 19);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 18);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 15);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[0]), 13);
        W7FILTSet2Taps(&(pCphr->cphrStrm[3].strmLfsr[0].lfsrFilt[0]),
                       4, 1);
        W7FILTSet2Taps(&(pCphr->cphrStrm[3].strmLfsr[0].lfsrFilt[1]),
                       8, 5);
        W7FILTSet3Taps(&(pCphr->cphrStrm[3].strmLfsr[0].lfsrFilt[2]),
                       15, 12, 9);

        /* LFSR 3b */
        W7LFSRInit(&(pCphr->cphrStrm[3].strmLfsr[1]), 43, 19);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[1]), 42);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[1]), 41);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[1]), 39);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[1]), 38);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[1]), 37);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[1]), 36);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[1]), 25);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[1]), 12);
        W7FILTSet2Taps(&(pCphr->cphrStrm[3].strmLfsr[1].lfsrFilt[0]),
                       3, 1);
        W7FILTSet2Taps(&(pCphr->cphrStrm[3].strmLfsr[1].lfsrFilt[1]),
                       6, 4);
        W7FILTSet3Taps(&(pCphr->cphrStrm[3].strmLfsr[1].lfsrFilt[2]),
                       11, 9, 7);

        /* LFSR 3c */
        W7LFSRInit(&(pCphr->cphrStrm[3].strmLfsr[2]), 47, 21);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[2]), 46);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[2]), 41);
```

```
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[2]), 27);
        W7LFSRAddTap(&(pCphr->cphrStrm[3].strmLfsr[2]), 13);
        W7FILTSet2Taps(&(pCphr->cphrStrm[3].strmLfsr[2].lfsrFilt[0]),
                       4, 1);
        W7FILTSet2Taps(&(pCphr->cphrStrm[3].strmLfsr[2].lfsrFilt[1]),
                       12, 7);
        W7FILTSet3Taps(&(pCphr->cphrStrm[3].strmLfsr[2].lfsrFilt[2]),
                       21, 18, 13);

        /* LFSR 4a */
        W7LFSRInit(&(pCphr->cphrStrm[4].strmLfsr[0]), 38, 20);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[0]), 37);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[0]), 24);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[0]), 22);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[0]), 11);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[0]), 7);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[0]), 5);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[0]), 3);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[0]), 1);
        W7FILTSet2Taps(&(pCphr->cphrStrm[4].strmLfsr[0].lfsrFilt[0]),
                       34, 31);
        W7FILTSet2Taps(&(pCphr->cphrStrm[4].strmLfsr[0].lfsrFilt[1]),
                       30, 27);
        W7FILTSet3Taps(&(pCphr->cphrStrm[4].strmLfsr[0].lfsrFilt[2]),
                       26, 23, 20);

        /* LFSR 4b */
        W7LFSRInit(&(pCphr->cphrStrm[4].strmLfsr[1]), 43, 23);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[1]), 42);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[1]), 34);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[1]), 26);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[1]), 19);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[1]), 18);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[1]), 17);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[1]), 12);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[1]), 5);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[1]), 4);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[1]), 3);
        W7FILTSet2Taps(&(pCphr->cphrStrm[4].strmLfsr[1].lfsrFilt[0]),
                       39, 37);
        W7FILTSet2Taps(&(pCphr->cphrStrm[4].strmLfsr[1].lfsrFilt[1]),
                       36, 34);
        W7FILTSet3Taps(&(pCphr->cphrStrm[4].strmLfsr[1].lfsrFilt[2]),
                       33, 31, 29);

        /* LFSR 4c */
        W7LFSRInit(&(pCphr->cphrStrm[4].strmLfsr[2]), 47, 25);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[2]), 46);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[2]), 4);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[2]), 3);
        W7LFSRAddTap(&(pCphr->cphrStrm[4].strmLfsr[2]), 0);
```

```
        W7FILTSet2Taps(&(pCphr->cphrStrm[4].strmLfsr[2].lfsrFilt[0]),
                       43, 38);
        W7FILTSet2Taps(&(pCphr->cphrStrm[4].strmLfsr[2].lfsrFilt[1]),
                       37, 32);
        W7FILTSet3Taps(&(pCphr->cphrStrm[4].strmLfsr[2].lfsrFilt[2]),
                       31, 26, 21);

        /* LFSR 5a */
        W7LFSRInit(&(pCphr->cphrStrm[5].strmLfsr[0]), 38, 17);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[0]), 37);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[0]), 35);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[0]), 33);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[0]), 31);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[0]), 29);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[0]), 25);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[0]), 14);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[0]), 12);
        W7FILTSet2Taps(&(pCphr->cphrStrm[5].strmLfsr[0].lfsrFilt[0]),
                       5, 2);
        W7FILTSet2Taps(&(pCphr->cphrStrm[5].strmLfsr[0].lfsrFilt[1]),
                       9, 6);
        W7FILTSet3Taps(&(pCphr->cphrStrm[5].strmLfsr[0].lfsrFilt[2]),
                       16, 13, 10);

        /* LFSR 5b */
        W7LFSRInit(&(pCphr->cphrStrm[5].strmLfsr[1]), 43, 20);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[1]), 42);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[1]), 38);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[1]), 37);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[1]), 36);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[1]), 29);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[1]), 24);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[1]), 23);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[1]), 22);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[1]), 15);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[1]), 7);
        W7FILTSet2Taps(&(pCphr->cphrStrm[5].strmLfsr[1].lfsrFilt[0]),
                       4, 2);
        W7FILTSet2Taps(&(pCphr->cphrStrm[5].strmLfsr[1].lfsrFilt[1]),
                       7, 5);
        W7FILTSet3Taps(&(pCphr->cphrStrm[5].strmLfsr[1].lfsrFilt[2]),
                       12, 10, 8);

        /* LFSR 5c */
        W7LFSRInit(&(pCphr->cphrStrm[5].strmLfsr[2]), 47, 22);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[2]), 46);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[2]), 45);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[2]), 42);
        W7LFSRAddTap(&(pCphr->cphrStrm[5].strmLfsr[2]), 41);
        W7FILTSet2Taps(&(pCphr->cphrStrm[5].strmLfsr[2].lfsrFilt[0]),
                       5, 2);
```

```
        W7FILTSet2Taps(&(pCphr->cphrStrm[5].strmLfsr[2].lfsrFilt[1]),
                       13, 8);
        W7FILTSet3Taps(&(pCphr->cphrStrm[5].strmLfsr[2].lfsrFilt[2]),
                       22, 19, 14);

        /* LFSR 6a */
        W7LFSRInit(&(pCphr->cphrStrm[6].strmLfsr[0]), 38, 19);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[0]), 37);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[0]), 5);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[0]), 4);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[0]), 0);
        W7FILTSet2Taps(&(pCphr->cphrStrm[6].strmLfsr[0].lfsrFilt[0]),
                       33, 30);
        W7FILTSet2Taps(&(pCphr->cphrStrm[6].strmLfsr[0].lfsrFilt[1]),
                       29, 26);
        W7FILTSet3Taps(&(pCphr->cphrStrm[6].strmLfsr[0].lfsrFilt[2]),
                       25, 22, 19);

        /* LFSR 6b */
        W7LFSRInit(&(pCphr->cphrStrm[6].strmLfsr[1]), 43, 22);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[1]), 42);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[1]), 29);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[1]), 28);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[1]), 25);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[1]), 17);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[1]), 14);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[1]), 13);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[1]), 9);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[1]), 4);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[1]), 3);
        W7FILTSet2Taps(&(pCphr->cphrStrm[6].strmLfsr[1].lfsrFilt[0]),
                       38, 36);
        W7FILTSet2Taps(&(pCphr->cphrStrm[6].strmLfsr[1].lfsrFilt[1]),
                       35, 33);
        W7FILTSet3Taps(&(pCphr->cphrStrm[6].strmLfsr[1].lfsrFilt[2]),
                       32, 30, 28);

        /* LFSR 6c */
        W7LFSRInit(&(pCphr->cphrStrm[6].strmLfsr[2]), 47, 24);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[2]), 46);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[2]), 32);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[2]), 18);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[2]), 10);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[2]), 7);
        W7LFSRAddTap(&(pCphr->cphrStrm[6].strmLfsr[2]), 4);
        W7FILTSet2Taps(&(pCphr->cphrStrm[6].strmLfsr[2].lfsrFilt[0]),
                       42, 37);
        W7FILTSet2Taps(&(pCphr->cphrStrm[6].strmLfsr[2].lfsrFilt[1]),
                       36, 31);
        W7FILTSet3Taps(&(pCphr->cphrStrm[6].strmLfsr[2].lfsrFilt[2]),
                       30, 25, 20);
```

```
    /* LFSR 7a */
    W7LFSRInit(&(pCphr->cphrStrm[7].strmLfsr[0]), 38, 18);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[0]), 37);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[0]), 36);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[0]), 32);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[0]), 31);
    W7FILTSet2Taps(&(pCphr->cphrStrm[7].strmLfsr[0].lfsrFilt[0]),
                   6, 3);
    W7FILTSet2Taps(&(pCphr->cphrStrm[7].strmLfsr[0].lfsrFilt[1]),
                   10, 7);
    W7FILTSet3Taps(&(pCphr->cphrStrm[7].strmLfsr[0].lfsrFilt[2]),
                   17, 14, 11);

    /* LFSR 7b */
    W7LFSRInit(&(pCphr->cphrStrm[7].strmLfsr[1]), 43, 21);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[1]), 42);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[1]), 38);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[1]), 37);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[1]), 32);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[1]), 28);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[1]), 27);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[1]), 24);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[1]), 16);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[1]), 13);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[1]), 12);
    W7FILTSet2Taps(&(pCphr->cphrStrm[7].strmLfsr[1].lfsrFilt[0]),
                   5, 3);
    W7FILTSet2Taps(&(pCphr->cphrStrm[7].strmLfsr[1].lfsrFilt[1]),
                   8, 6);
    W7FILTSet3Taps(&(pCphr->cphrStrm[7].strmLfsr[1].lfsrFilt[2]),
                   13, 11, 9);

    /* LFSR 7c */
    W7LFSRInit(&(pCphr->cphrStrm[7].strmLfsr[2]), 47, 23);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[2]), 46);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[2]), 41);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[2]), 38);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[2]), 35);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[2]), 27);
    W7LFSRAddTap(&(pCphr->cphrStrm[7].strmLfsr[2]), 13);
    W7FILTSet2Taps(&(pCphr->cphrStrm[7].strmLfsr[2].lfsrFilt[0]),
                   6, 3);
    W7FILTSet2Taps(&(pCphr->cphrStrm[7].strmLfsr[2].lfsrFilt[1]),
                   14, 9);
    W7FILTSet3Taps(&(pCphr->cphrStrm[7].strmLfsr[2].lfsrFilt[2]),
                   23, 20, 15);
  }
```

```
/*----- W7CPHRSetKey  install a key into the W7CPHR object -----*/

/*
 * Each single-bit key stream has three shift registers of 38, 43,
 * and 47 bits. The initial state of these shift registers--a total
 * of 128 bits for each bit stream--serves as an encryption /
 * decryption key.
 */

void                    /* nothing to return */
W7CPHRSetKey
(
    W7CPHR   *pCphr,  /* W7CPHR object to key */
    UINT8    *pKey    /* the key itself, MSB first */
)
{
    UINT32 subkeyMS;  /* place to stuff individual LFSR subkeys */
    UINT32 subkeyLS;  /* least significant subkey */

    int    i;         /* temporary loop counter */

    /* LFSR[0] gets 38 bits */
    subkeyLS  =  pKey[15] | (pKey[14]<<8) + (pKey[13]<<16) |
                (pKey[12]<<24);
    subkeyMS  =  pKey[11] & ((1<<6)-1);

    for (i = 0; i < 8; i++)
    {
        W7LFSRSetKey(&(pCphr->cphrStrm[i].strmLfsr[0]),
                     subkeyMS, subkeyLS);
    }

    /* LFSR[1] gets 43 bits */
    subkeyLS  =  (pKey[11] & ~((1<<6)-1)) >> 6;
    subkeyLS |=  pKey[10]<<2 | pKey[9]<<10 | pKey[8]<<18;
    subkeyLS |=  (pKey[7] & ((1<<6)-1)) << 26;
    subkeyMS  =  pKey[7]>>6;
    subkeyMS |=  (pKey[6]<<2 | (pKey[5]&1)<<10);

    for (i = 0; i < 8; i++)
    {
        W7LFSRSetKey(&(pCphr->cphrStrm[i].strmLfsr[1]),
                     subkeyMS, subkeyLS);
    }

    /* LFSR[2] gets 47 bits */
    subkeyLS  =  (pKey[5] & ~(1)) >> 1;
    subkeyLS |=  pKey[4]<<7 | pKey[3]<<15 | pKey[2]<<23;
    subkeyLS |=  (pKey[1]&1)<<31;
    subkeyMS  =  pKey[1]>>1 | pKey[0]<<7;
```

```
    for (i = 0; i < 8; i++)
    {
        W7LFSRSetKey(&(pCphr->cphrStrm[i].strmLfsr[2]),
                     subkeyMS, subkeyLS);
    }

}


/*----- W7CPHRNext  get the next byte of the full keystream -----*/

/*
 * Nothing fancy here; we just advance each of the individual
 * one-bit streams and combine the results into a full byte.
 * Note that getting the next byte advances the cipher core
 * so it's ready for the following byte.
 */

UINT8               /* returns the next byte of the keystream */
W7CPHRNext
(
    W7CPHR *pCphr   /* cipher object to use */
)
{
    UINT8 byte;     /* place to accumulate results */
    int   i;        /* loop counter */

    byte = 0;
    for (i=0; i<8; i++)
      byte |=  W7STRMAdvance(&(pCphr->cphrStrm[i])) << i;

    return(byte);
}


/*----- W7CPHRDebug  print the state of the full cipher -----*/

void                /* nothing to return */
W7CPHRDebug
(
    W7CPHR *pCphr   /* cipher object to debug */
)
{
    int   i;        /* loop counter */

    printf("\n***** W7 Cipher State *****\n");
    for (i=0; i<8; i++)
        W7STRMDebug(&(pCphr->cphrStrm[i]));

}
```

```
/*----- W7STRMAdvance  advance an individual bit stream -----*/

/*
 * This function calculates the output bit for a bit stream
 * by combining the LFSRs while it also handles clocking of
 * the shift registers. The output is simply the exclusive-OR
 * of the outputs of the separate LFSRs. To calculate it, we
 * just add them up modulo 2. LFSR clocking is managed by the
 * majority clock function. We look at the clock taps in each
 * LFSR. The LFSRs whose clock tap value matches the majority
 * get clocked. Others don't. (This is the decimated clocking
 * part of the W7 cipher; and it is implemented so that at least
 * two of the three LFSRs is clocked at every bit.)
 */

unsigned                /* returns next bit in bit stream */
W7STRMAdvance
(
    W7STRM *pStrm       /* bit stream to advance */
)
{
    int      i;         /* loop counter */
    unsigned output;    /* place to accumulate output */
    unsigned clock;     /* majority clock value */

    output = 0;
    clock  = 0;

    /* loop through all LFSRs to get output and check clock */
    for (i=0; i<3; i++)
    {
        /* XOR is equivalent to add mod 2, sum output for now */
        output += W7LFSRGetOutput(&(pStrm->strmLfsr[i]));

        /* for the majority clock, we want to count the one bits */
        clock  += W7LFSRGetClock(&(pStrm->strmLfsr[i]));
    }

    /* if > one LFSR's clock is set, majority clock value is 1 */
    clock = (clock > 1);

    /* loop through the LFSRs again, clocking those in majority */
    for (i=0; i<3; i++)
    {
        if (clock == W7LFSRGetClock(&(pStrm->strmLfsr[i])))
            W7LFSRShift(&(pStrm->strmLfsr[i]));
    }

    /* since output is XOR, mask on lowest bit (i.e. mod 2) */
    return(output % 2);
}
```

```
/*----- W7STRMDebug  print the state of a single bit stream -----*/

/*
 * To calculate all the information about a stream, we do almost the
 * same calculation as in advancing it, except that we don't clock
any
 * of the shift registers.
 */

void                    /* no return */
W7STRMDebug
(
    W7STRM *pStrm       /* stream to debug */
)
{
    int      i;         /* loop counter */
    unsigned output;    /* place to accumulate output */
    unsigned clock;     /* place to calculate majority clock value */

    printf("STRM Debug:\n");

    output = 0;
    clock  = 0;

    /* loop through the registers once to get outputs and clocks */
    for (i=0; i<3; i++)
    {
        output += W7LFSRGetOutput(&(pStrm->strmLfsr[i]));
        clock  += W7LFSRGetClock(&(pStrm->strmLfsr[i]));
    }

    /* if > one LFSR's clock is set, majority clock value is 1 */
    clock = (clock > 1);

    printf("  Majority clock: %d\n", clock);

    /* loop through the LFSRs again to display their contents */
    for (i=0; i<3; i++)
    {
        W7LFSRDebug(&(pStrm->strmLfsr[i]));
        if (clock == W7LFSRGetClock(&(pStrm->strmLfsr[i])))
           printf("  will clock\n");
        else
           printf("  won't clock\n");
    }

    printf("  Output: %d\n", (output%2));
}
```

```
/*----- W7LFSRInit  initialize an individual shift register -----*/

/*
 * Here we (partially) initialize a shift register by setting
 * as much of it as can be easily accommodated in a function
 * call. This function does not set up the feedback taps, since
 * there are a variable number of them.
 */

void                /* nothing to return */
W7LFSRInit
(
    W7LFSR  *pLfsr,  /* shift register to initialize */
    unsigned size,   /* how many bits in the shift register */
    unsigned clock   /* which bit is the clock tap, 0 = LSBit */
)
{
    /* set the size and zero everything else */
    pLfsr->lfsrSize    = size;
    pLfsr->lfsrMS      = 0;
    pLfsr->lfsrLS      = 0;
    pLfsr->lfsrTapMS   = 0;
    pLfsr->lfsrTapLS   = 0;
    pLfsr->lfsrClockMS = 0;
    pLfsr->lfsrClockLS = 0;

    /* now convert the clock position into a bit mask */
    if (clock > 31)
        pLfsr->lfsrClockMS = (1 << (clock-32));
    else
        pLfsr->lfsrClockLS = (1 << clock);
}


/*----- W7LFSRAddTap  add a tap to a shift register -----*/

/*
 * Nothing fancy, convert a tap position into a bit mask and
 * OR it into the LFSR tap attribute.
 */

void                    /* nothing to return */
W7LFSRAddTap
(
    W7LFSR   *pLfsr,    /* shift register for new tap */
    unsigned  tapNum    /* position of tap, 0 = LSBit */
)
{
    if (tapNum > 31)
        pLfsr->lfsrTapMS |= (1 << (tapNum-32));
    else
        pLfsr->lfsrTapLS |= (1 << tapNum);
}
```

```
/*----- W7LFSRSetKey  set the state of a shift register -----*/

/*
 * No surprises, just set the attribute appropriately. Because
 * we use 32-bit words, we have to break the subkey up into two
 * parameters. Also, since our subkeys are at most 47 bits long,
 * we won't actually use all 64 bits. The high order bits are
 * just ignored.
 */

void                    /* nothing to return */
W7LFSRSetKey
(
    W7LFSR *pLfsr,      /* shift register to set */
    UINT32  keyMS,      /* most significant 32 bits */
    UINT32  keyLS       /* least significant 32 bits */
)
{
    pLfsr->lfsrMS = keyMS;
    pLfsr->lfsrLS = keyLS;
}



/*----- W7LFSRShift  shift a shift register -----*/

/*
 * Since we break up the register into 32-bit words, we have
 * to manually handle the shifting of the MSBit in the LSWord
 * to the LSBit position of the MSWord. Also, we need to know
 * what new value to shift into the LSBit of the LSWord. That's
 * an exclusive-OR of all the bits in the tap positions. To
 * calculate it, we logical-AND the register contents with the
 * tap mask, sum the resulting bits (i.e. count the number of
 * one bits) and modulo 2 the result. (XOR is the same as
 * addition mod 2.)
 */

void                    /* nothing to return */
W7LFSRShift
(
    W7LFSR *pLfsr       /* shift register to shift */
)
{
    unsigned bits;      /* count of ones in tap position */
    unsigned xbit;      /* bit that moves from LS to MS word */

    /* count how many tap bits are set */
    bits  = bit_count(pLfsr->lfsrMS & pLfsr->lfsrTapMS);
    bits += bit_count(pLfsr->lfsrLS & pLfsr->lfsrTapLS);
```

```
    /* save the transition bit */
    xbit = ((pLfsr->lfsrLS & (1<<31)) != 0);

    /* shift MS word, bring in the xbit, and mask off to size */
    pLfsr->lfsrMS <<=  1;
    pLfsr->lfsrMS  |=  xbit;
    pLfsr->lfsrMS  &=  (1 << (pLfsr->lfsrSize - 32)) - 1;

    /* shift LS word, LS bit is count of tap bits mod 2 */
    pLfsr->lfsrLS <<=  1;
    pLfsr->lfsrLS  |=  bits % 2;
}


/*----- W7LFSRGetClock  return the value of the clock tap bit -----
*/

/*
 * To find the value of the clock tap, we mask the register
 * contents with the clock tap mask. If the result is non-zero
 * then the clock tap bit is set.
 */

unsigned                /* returns clock tap bit (0 or 1) */
W7LFSRGetClock
(
    W7LFSR *pLfsr       /* shift register to check */
)
{
    unsigned clock;
    clock  = bit_count(pLfsr->lfsrMS & pLfsr->lfsrClockMS);
    clock += bit_count(pLfsr->lfsrLS & pLfsr->lfsrClockLS);
    return(clock != 0);
}
```

```
/*----- W7LFSRGetOutput  return the output bit for an LFSR -----*/

/*
 * The shift register output is the exclusive-OR of its most
 * significant bit with three filter terms. Each filter term is the
 * logical-AND of 2 or 3 bit positions. To perform the XOR, we use
 * addition modulo 2.vFor each AND, we mask the register with the
 * term's bits, count the number of one bits, and see if that's
 * equal to the number of one bits in the term's mask.
 */

unsigned                /* returns output bit (0 or 1) */
W7LFSRGetOutput
(
    W7LFSR *pLfsr       /* shift register to check */
)
{
    int      i;         /* loop counter */
    unsigned term;      /* holds results of each term (AND) */
    unsigned output;

    output = 0;

    /* three terms in each filter */
    for (i=0; i<3; i++)
    {
        /* mask off MS part and count one bits */
        term  = bit_count(pLfsr->lfsrMS &
                          pLfsr->lfsrFilt[i].filtMS);

        /* mask off LS part and count one bits, adding to total */
        term += bit_count(pLfsr->lfsrLS &
                          pLfsr->lfsrFilt[i].filtLS);

        /* does resulting count equal number of bits in term? */
        term  = (term == pLfsr->lfsrFilt[i].filtBits);

        /* accumulate each term */
        output += term;
    }

    /* also add in the register's most significant bit */
    output += ((pLfsr->lfsrMS & (1 << (pLfsr->lfsrSize-33))) != 0);

    /* return value is mod 2 of the resulting accumulation */
    return(output % 2);
}
```

```
/*----- W7LFSRDebug  print the state of a shift register -----*/

void                    /* no return */
W7LFSRDebug
(
    W7LFSR *pLfsr    /* shift register to debug */
)
{
    int i;           /* loop counter */

    printf("--LFSR Debug:\n");
    printf("    size: %d\n", pLfsr->lfsrSize);
    printf("    register:  %08X %08X\n", pLfsr->lfsrMS,
                                         pLfsr->lfsrLS);
    printf("    shift tap: %08X %08X\n", pLfsr->lfsrTapMS,
                                         pLfsr->lfsrTapLS);
    printf("    clock tap: %08X %08X\n", pLfsr->lfsrClockMS,
                                         pLfsr->lfsrClockLS);
    for (i=0; i<3; i++)
    {
        W7FILTDebug(&pLfsr->lfsrFilt[i]);
    }
    printf("    data out:  %d\n", W7LFSRGetOutput(pLfsr));
    printf("    clock out: %d\n", W7LFSRGetClock(pLfsr));

};
```

```
/*----- W7FILTSet2Taps  initialize a filter term with 2 taps-----*/

/*
 * Set the tap values in one filter term. Each filter term can
 * have either 2 or 3 taps. This function handles those with
 * two taps.
 *
 * Each filter term is just a bit mask. We simply set the
 * appropriate bits here. We also count the number of taps
 * and store that in the filter term.
 */

void                    /* nothing to return */
W7FILTSet2Taps
(
    W7FILT *pFilt,      /* filter term to initialize */
    int     tap1,       /* first tap position, 0 = LSBit */
    int     tap2        /* second tap, 0 = LSBit */
)
{
    pFilt->filtMS = 0;
    pFilt->filtLS = 0;

    if (tap1 > 31)
        pFilt->filtMS |= (1 << (tap1-32));
    else
        pFilt->filtLS |= (1 << tap1);
    if (tap2 > 31)
        pFilt->filtMS |= (1 << (tap2-32));
    else
        pFilt->filtLS |= (1 << tap2);

    pFilt->filtBits = 2;
}
```

```
/*----- W7FILTSet3Taps  initialize a filter term with 3 taps -----*/

/*
 * Set the tap values in one filter term. Each filter term can
 * have either 2 or 3 taps. This function handles those with
 * three taps.
 *
 * Each filter term is just a bit mask. We simply set the
 * appropriate bits here. We also count the number of taps
 * and store that in the filter term.
 */

void                    /* nothing to return */
W7FILTSet3Taps
(
    W7FILT *pFilt,      /* filter term to initialize */
    int     tap1,       /* first tap position, 0 = LSBit */
    int     tap2,       /* second tap, 0 = LSBit */
    int     tap3        /* third tap, if present */
)
{
    pFilt->filtMS = 0;
    pFilt->filtLS = 0;

    if (tap1 > 31)
        pFilt->filtMS |= (1 << (tap1-32));
    else
        pFilt->filtLS |= (1 << tap1);
    if (tap2 > 31)
        pFilt->filtMS |= (1 << (tap2-32));
    else
        pFilt->filtLS |= (1 << tap2);
    if (tap3 > 31)
        pFilt->filtMS |= (1 << (tap3-32));
    else
        pFilt->filtLS |= (1 << tap3);

    pFilt->filtBits = 3;
}


/*----- W7FILTDebug  print the state of a filter term -----*/

void                    /* nothing to return */
W7FILTDebug
(
    W7FILT *pFilt       /* filter term to debug */
)
{
    printf("    filter:  %08X %08X (%d)\n", pFilt->filtMS,
                                            pFilt->filtLS,
                                            pFilt->filtBits);
}
```

```
/*----- bit_count  how many one bits are in a value -----*/

/*
 * This is cut-and-paste from www.snippets.org, but I think the
 * algorithm originally came from Efficient C, by Plum and Brodie.
 */

unsigned                /* returns count of one bits */
bit_count
(
    UINT32 x            /* value to count */
)
{
    int n = 0;
    if (x) do n++;
    while (0 != (x = x&(x-1)));
    return(n);
}
```

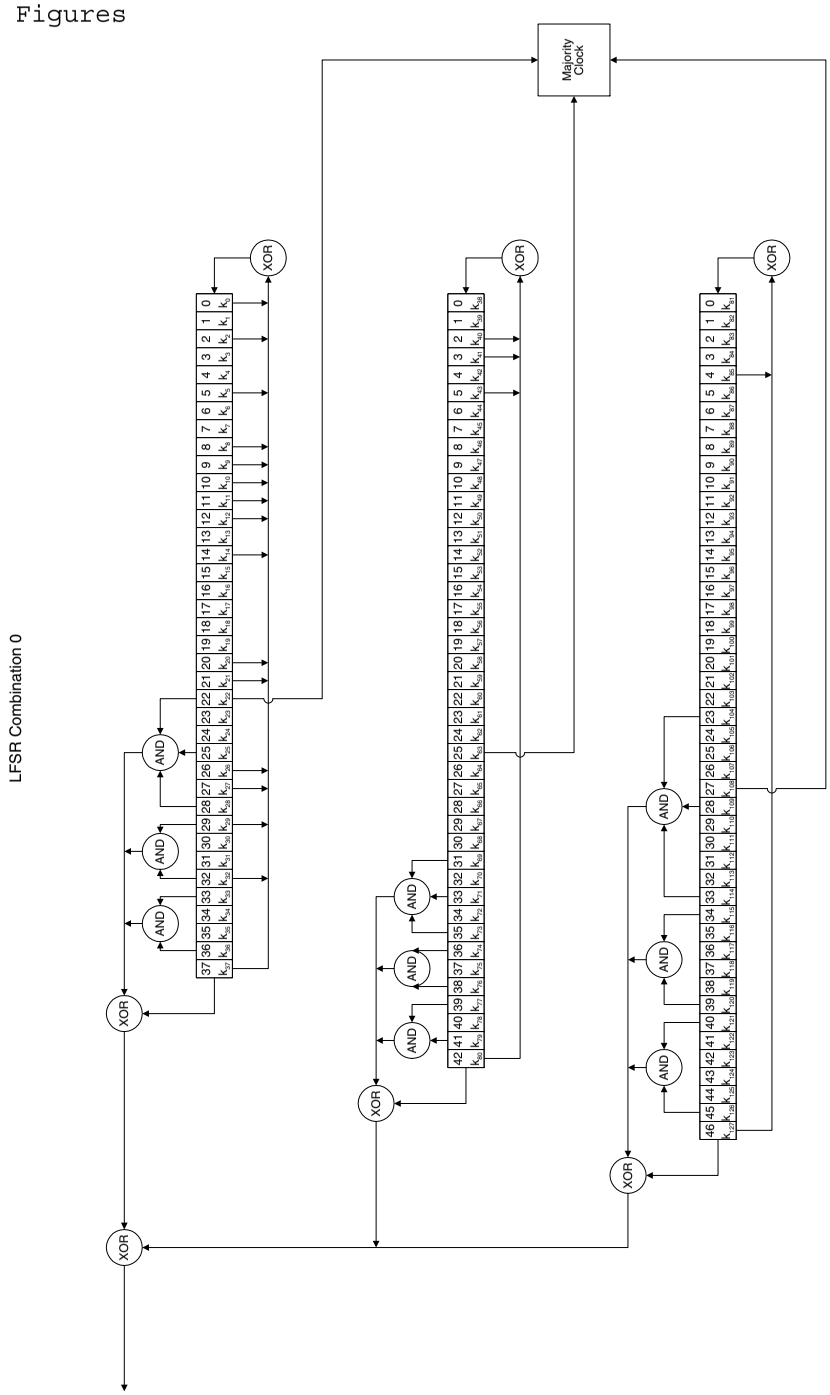
```
/*----- main  used for standalone testing of the functions -----*/

#define W7TEST 1
#ifdef  W7TEST
main()
{
    W7 *pW7;
    int i;
    static UINT8 zkey[16] =
    {
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    };
    static UINT8 key[16]  =
    {
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
    };
    static UINT8 plain[256] =
    {
          0,   1,   2,   3,   4,   5,   6,   7,
          8,   9,  10,  11,  12,  13,  14,  15,
         16,  17,  18,  19,  20,  21,  22,  23,
         24,  25,  26,  27,  28,  29,  30,  31,
         32,  33,  34,  35,  36,  37,  38,  39,
         40,  41,  42,  43,  44,  45,  46,  47,
         48,  49,  50,  51,  52,  53,  54,  55,
         56,  57,  58,  59,  60,  61,  62,  63,
         64,  65,  66,  67,  68,  69,  70,  71,
         72,  73,  74,  75,  76,  77,  78,  79,
         80,  81,  82,  83,  84,  85,  86,  87,
         88,  89,  90,  91,  92,  93,  94,  95,
         96,  97,  98,  99, 100, 101, 102, 103,
        104, 105, 106, 107, 108, 109, 110, 111,
        112, 113, 114, 115, 116, 117, 118, 119,
        120, 121, 122, 123, 124, 125, 126, 127,
        128, 129, 130, 131, 132, 133, 134, 135,
        136, 137, 138, 139, 140, 141, 142, 143,
        144, 145, 146, 147, 148, 149, 150, 151,
        152, 153, 154, 155, 156, 157, 158, 159,
        160, 161, 162, 163, 164, 165, 166, 167,
        168, 169, 170, 171, 172, 173, 174, 175,
        176, 177, 178, 179, 180, 181, 182, 183,
        184, 185, 186, 187, 188, 189, 190, 191,
        192, 193, 194, 195, 196, 197, 198, 199,
        200, 201, 202, 203, 204, 205, 206, 207,
        208, 209, 210, 211, 212, 213, 214, 215,
        216, 217, 218, 219, 220, 221, 222, 223,
        224, 225, 226, 227, 228, 229, 230, 231,
        232, 233, 234, 235, 236, 237, 238, 239,
        240, 241, 242, 243, 244, 245, 246, 247,
        248, 249, 250, 251, 252, 253, 254, 255
    };
    static UINT8 cipher[256];
```

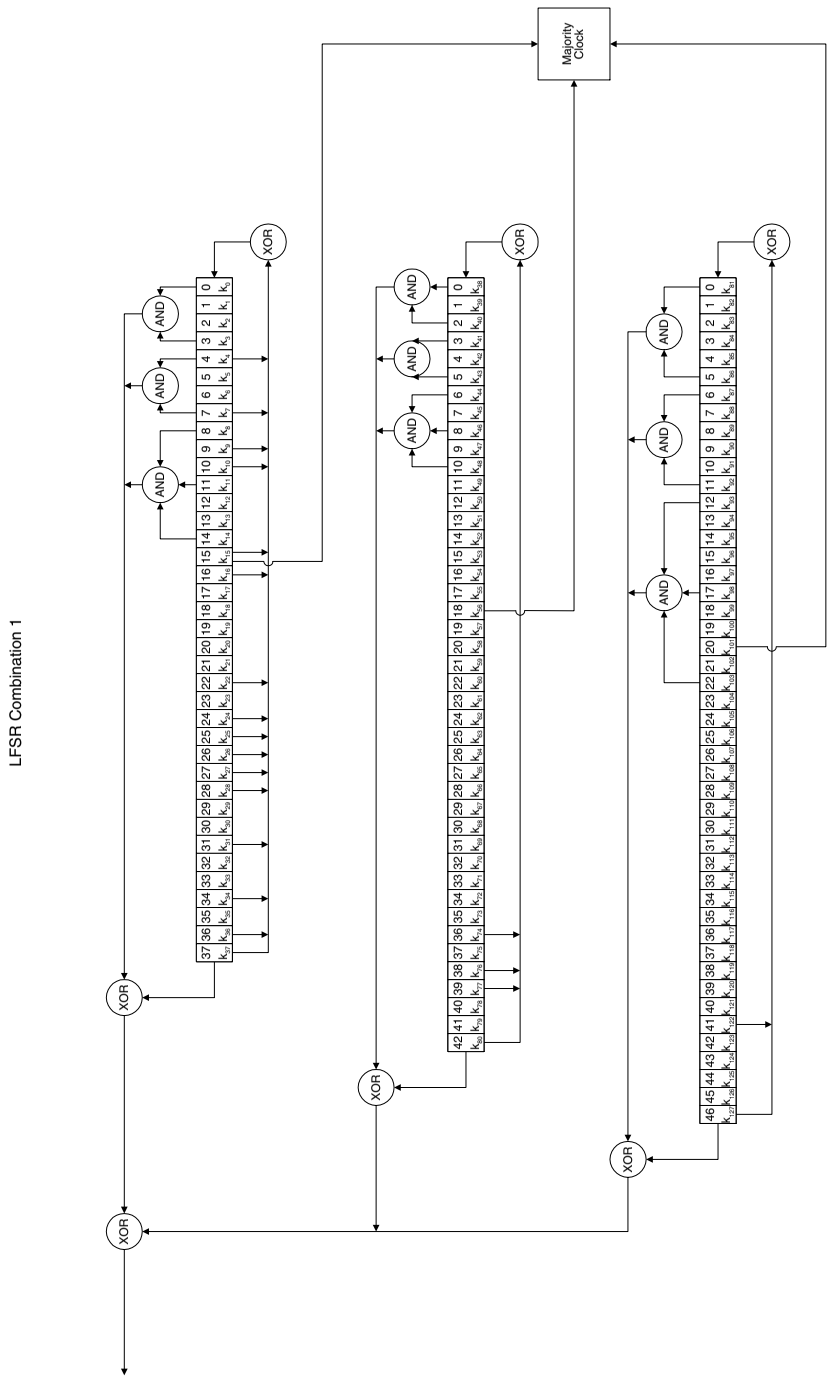
```
static UINT8 ciphercheck[256] =
{
    0xB6, 0xFD, 0x65, 0x03, 0x38, 0x08, 0xB6, 0x0A,
    0x6F, 0x47, 0x10, 0xCF, 0xDA, 0xFB, 0x1F, 0x7E,
    0x71, 0x66, 0x40, 0x5E, 0x38, 0x1D, 0x26, 0x07,
    0xD4, 0x7D, 0x58, 0x07, 0x5F, 0xB7, 0x93, 0x83,
    0xEC, 0x28, 0xFC, 0xB7, 0x45, 0x7E, 0x56, 0xEF,
    0x3D, 0x32, 0x12, 0x57, 0xF0, 0x6C, 0x9D, 0x85,
    0xA6, 0x11, 0x66, 0x02, 0x1A, 0xBC, 0x51, 0xB3,
    0xD8, 0xAD, 0x11, 0x0D, 0xA3, 0xFE, 0xED, 0xF0,
    0xD6, 0xE4, 0x69, 0x7C, 0x3F, 0x24, 0x9F, 0xE1,
    0x14, 0x9E, 0x87, 0xE3, 0x9B, 0x11, 0x5A, 0xE1,
    0xB4, 0xAE, 0x27, 0x74, 0xBE, 0x66, 0x02, 0xA1,
    0x59, 0xF8, 0xEA, 0x70, 0x04, 0x39, 0xDA, 0x54,
    0x0A, 0xF3, 0xB1, 0x13, 0x9D, 0x3A, 0x99, 0xD6,
    0x9C, 0x18, 0x18, 0x3C, 0x32, 0xEA, 0x7F, 0x72,
    0x89, 0x68, 0x4A, 0xD4, 0x77, 0x4E, 0xF8, 0x25,
    0x9D, 0x94, 0x8C, 0xB4, 0x6F, 0x50, 0xF2, 0x33,
    0x2B, 0xF4, 0x72, 0xAA, 0x1B, 0x62, 0xEC, 0xF3,
    0x64, 0xB0, 0x6F, 0x87, 0x07, 0xB0, 0x31, 0x7E,
    0xEC, 0x11, 0xC0, 0x7E, 0x82, 0x6B, 0x60, 0x7C,
    0x17, 0x73, 0xA1, 0x71, 0x1E, 0x7F, 0x2D, 0xC7,
    0x6C, 0xE6, 0x35, 0x9C, 0xDC, 0x1F, 0x0B, 0x55,
    0x18, 0xC8, 0x32, 0x08, 0x19, 0x0C, 0x19, 0x03,
    0x8B, 0xFE, 0x54, 0x8D, 0xA3, 0xC8, 0x69, 0xFD,
    0x5E, 0x09, 0x96, 0x53, 0x66, 0xD3, 0x1B, 0x6A,
    0xF2, 0x9D, 0xD0, 0x96, 0x02, 0xDF, 0x9A, 0xBE,
    0x72, 0x42, 0x59, 0x1E, 0xCA, 0xD1, 0xA2, 0x33,
    0xAE, 0xEA, 0x65, 0x4D, 0xD1, 0xBB, 0x76, 0xCC,
    0x13, 0x70, 0xE1, 0x37, 0x0D, 0xF4, 0xF2, 0x23,
    0x8E, 0xA9, 0x09, 0x71, 0x89, 0x4C, 0x16, 0xC1,
    0xDC, 0xB0, 0x9F, 0x5E, 0x94, 0x68, 0xAA, 0x0F,
    0x1F, 0xD9, 0x0C, 0x70, 0xF5, 0x02, 0xCD, 0x67,
    0x17, 0x6C, 0x14, 0x78, 0x0B, 0x0E, 0x6B, 0x1B
};
```

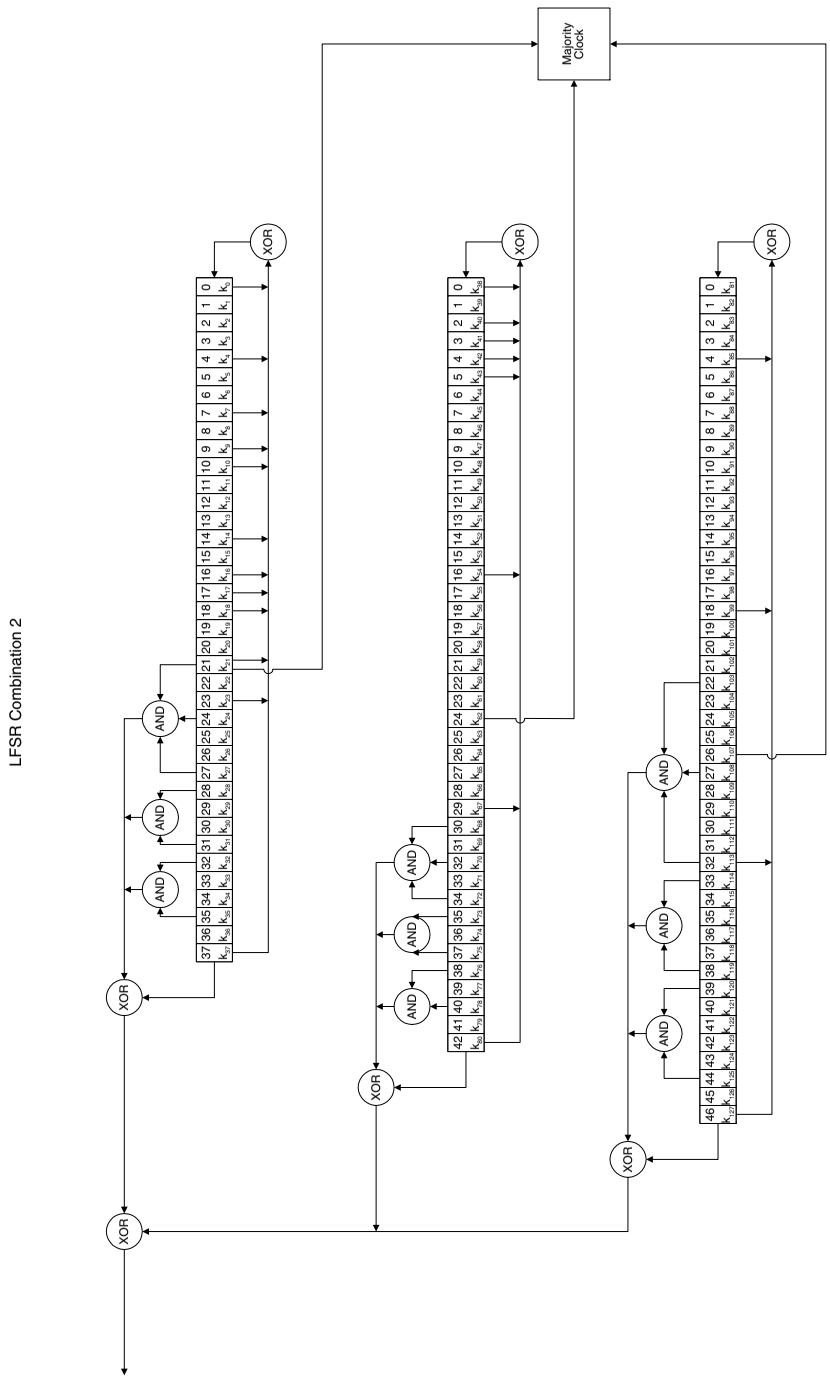
```
static UINT8 plaincheck[256] =
{
      0,   1,   2,   3,   4,   5,   6,   7,
      8,   9,  10,  11,  12,  13,  14,  15,
     16,  17,  18,  19,  20,  21,  22,  23,
     24,  25,  26,  27,  28,  29,  30,  31,
     32,  33,  34,  35,  36,  37,  38,  39,
     40,  41,  42,  43,  44,  45,  46,  47,
     48,  49,  50,  51,  52,  53,  54,  55,
     56,  57,  58,  59,  60,  61,  62,  63,
     64,  65,  66,  67,  68,  69,  70,  71,
     72,  73,  74,  75,  76,  77,  78,  79,
     80,  81,  82,  83,  84,  85,  86,  87,
     88,  89,  90,  91,  92,  93,  94,  95,
     96,  97,  98,  99, 100, 101, 102, 103,
    104, 105, 106, 107, 108, 109, 110, 111,
    112, 113, 114, 115, 116, 117, 118, 119,
    120, 121, 122, 123, 124, 125, 126, 127,
    128, 129, 130, 131, 132, 133, 134, 135,
    136, 137, 138, 139, 140, 141, 142, 143,
    144, 145, 146, 147, 148, 149, 150, 151,
    152, 153, 154, 155, 156, 157, 158, 159,
    160, 161, 162, 163, 164, 165, 166, 167,
    168, 169, 170, 171, 172, 173, 174, 175,
    176, 177, 178, 179, 180, 181, 182, 183,
    184, 185, 186, 187, 188, 189, 190, 191,
    192, 193, 194, 195, 196, 197, 198, 199,
    200, 201, 202, 203, 204, 205, 206, 207,
    208, 209, 210, 211, 212, 213, 214, 215,
    216, 217, 218, 219, 220, 221, 222, 223,
    224, 225, 226, 227, 228, 229, 230, 231,
    232, 233, 234, 235, 236, 237, 238, 239,
    240, 241, 242, 243, 244, 245, 246, 247,
    248, 249, 250, 251, 252, 253, 254, 255
};
```

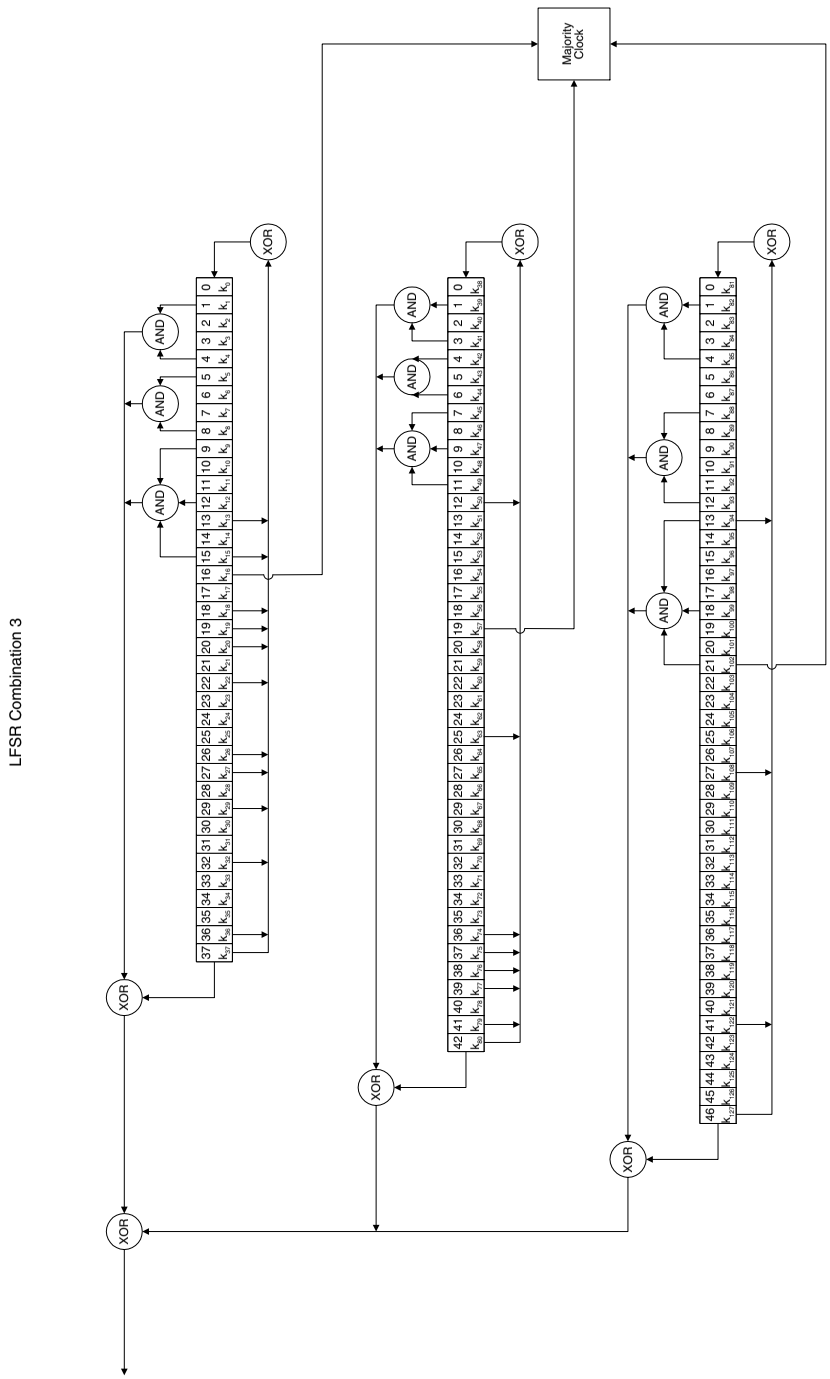
```
    if (0 != (pW7 = W7New(zkey)))
    {
        printf("error detecting zero key\n");
        exit(1);
    }
    if (0 == (pW7 = W7New(key)))
    {
        printf("error creating W7 object\n");
        exit(2);
    }
    if (0 == W7Encrypt(pW7, 256, plain, cipher))
    {
        printf("error encrypting\n");
        exit(3);
    }
    for (i=0; i<256; i++)
    {
        if (cipher[i] != ciphercheck[i])
        {
          printf("encrypt failure\n");
          exit(4);
        }
    }
    if (0 == W7Rekey(pW7, key))
    {
        printf("error re-keying W7 object\n");
        exit(5);
    }
    if (0 == W7Decrypt(pW7, 256, cipher, plain))
    {
        printf("error decrypting\n");
        exit(6);
    }
    for (i=0; i<256; i++)
    {
        if (plain[i] != plaincheck[i])
        {
          printf("decrypt failure\n");
          exit(7);
        }
    }
    if (0 == W7Delete(pW7))
    {
        printf("error deleting W7 object\n");
        exit(8);
    }

    printf("all tests passed\n");
    exit(0);
}
#endif /* W7TEST */
```

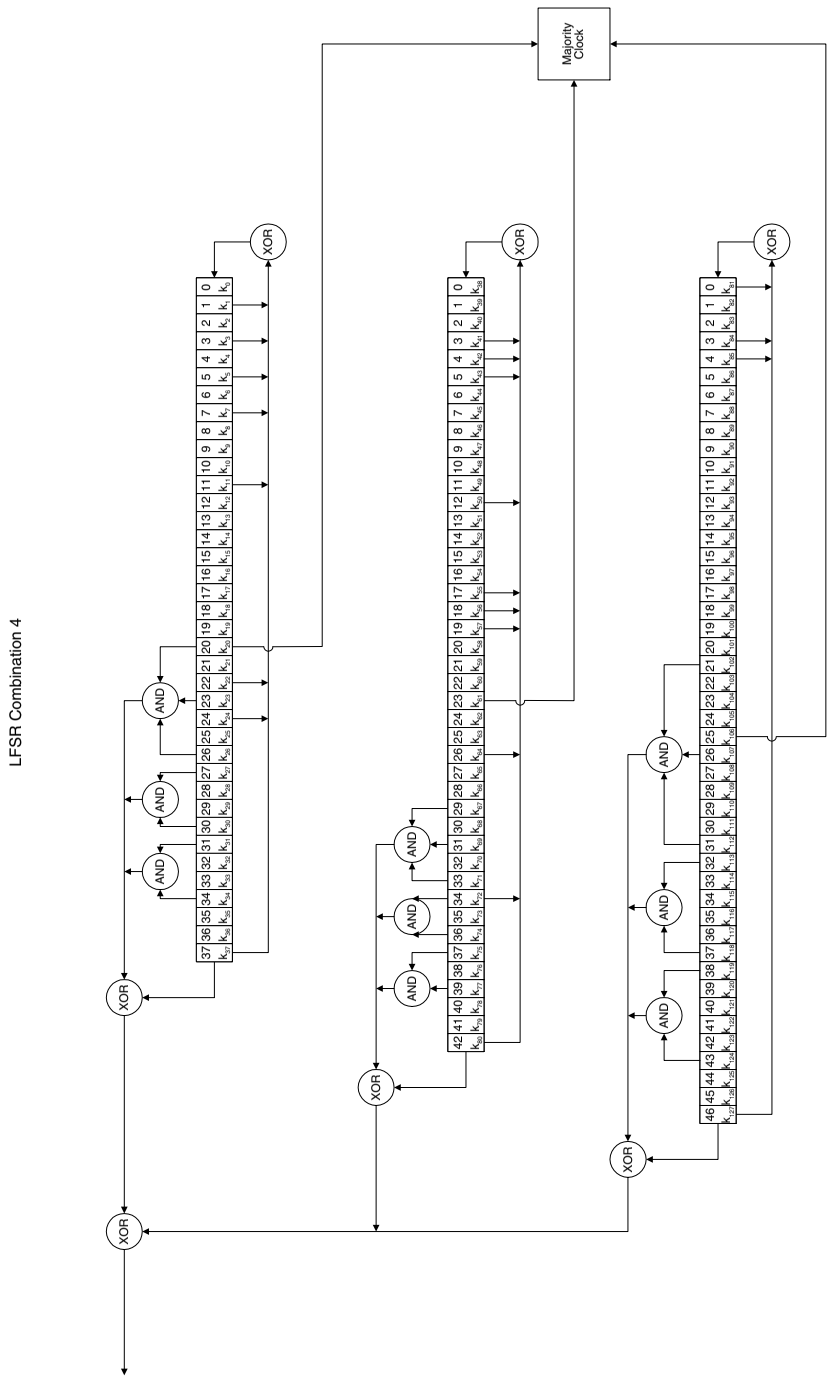
Appendix C. Figures

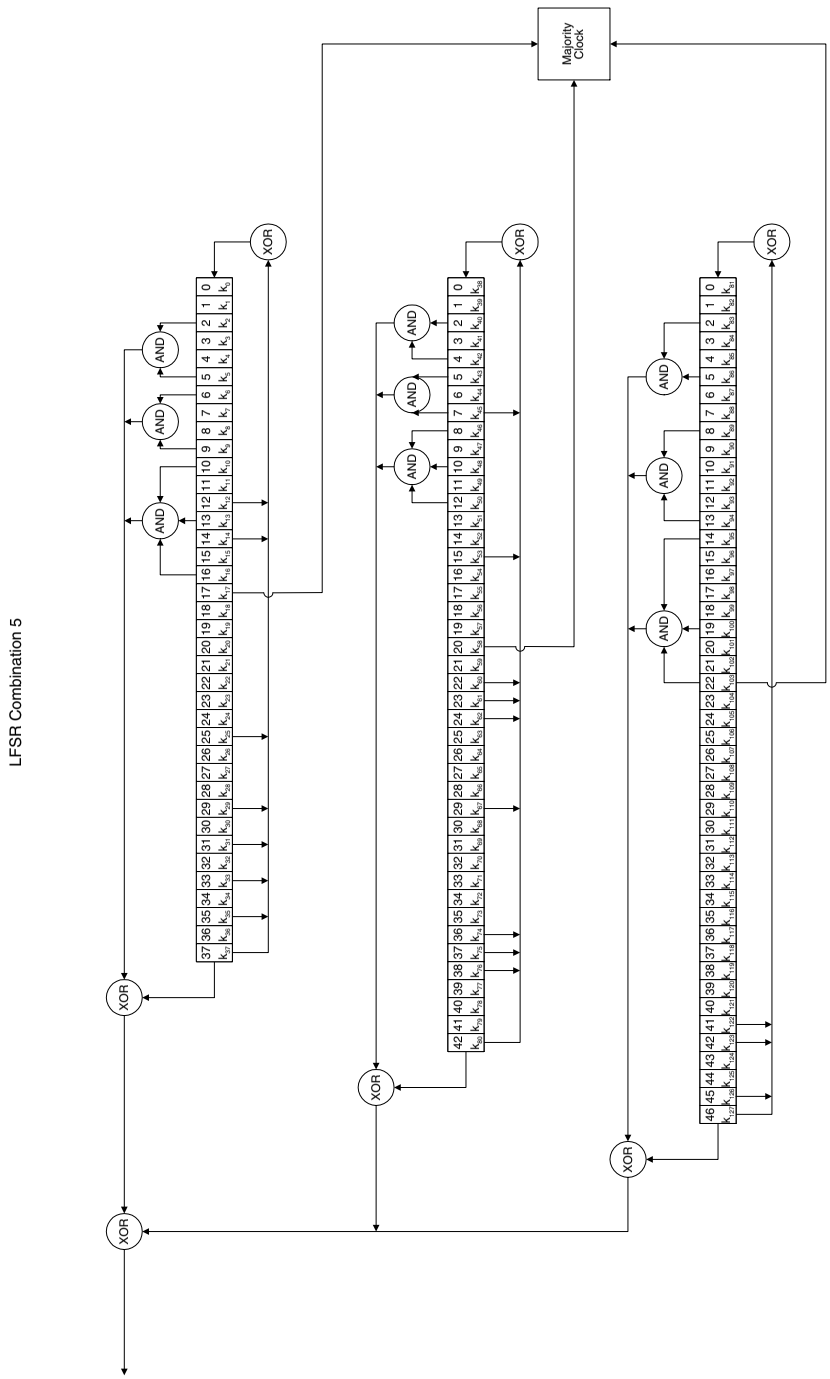


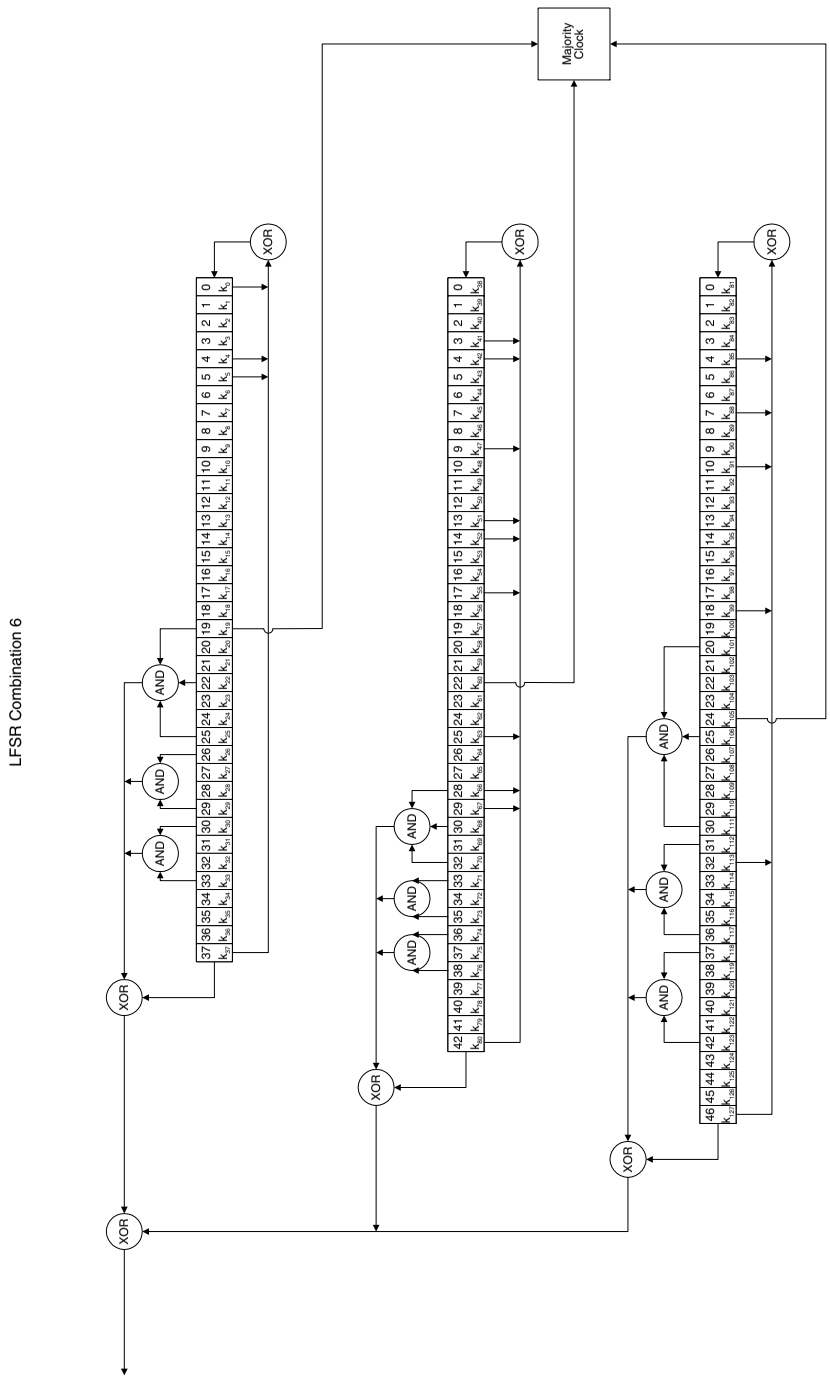

LFSR Combination 1
Majority Clock
XOR
AND

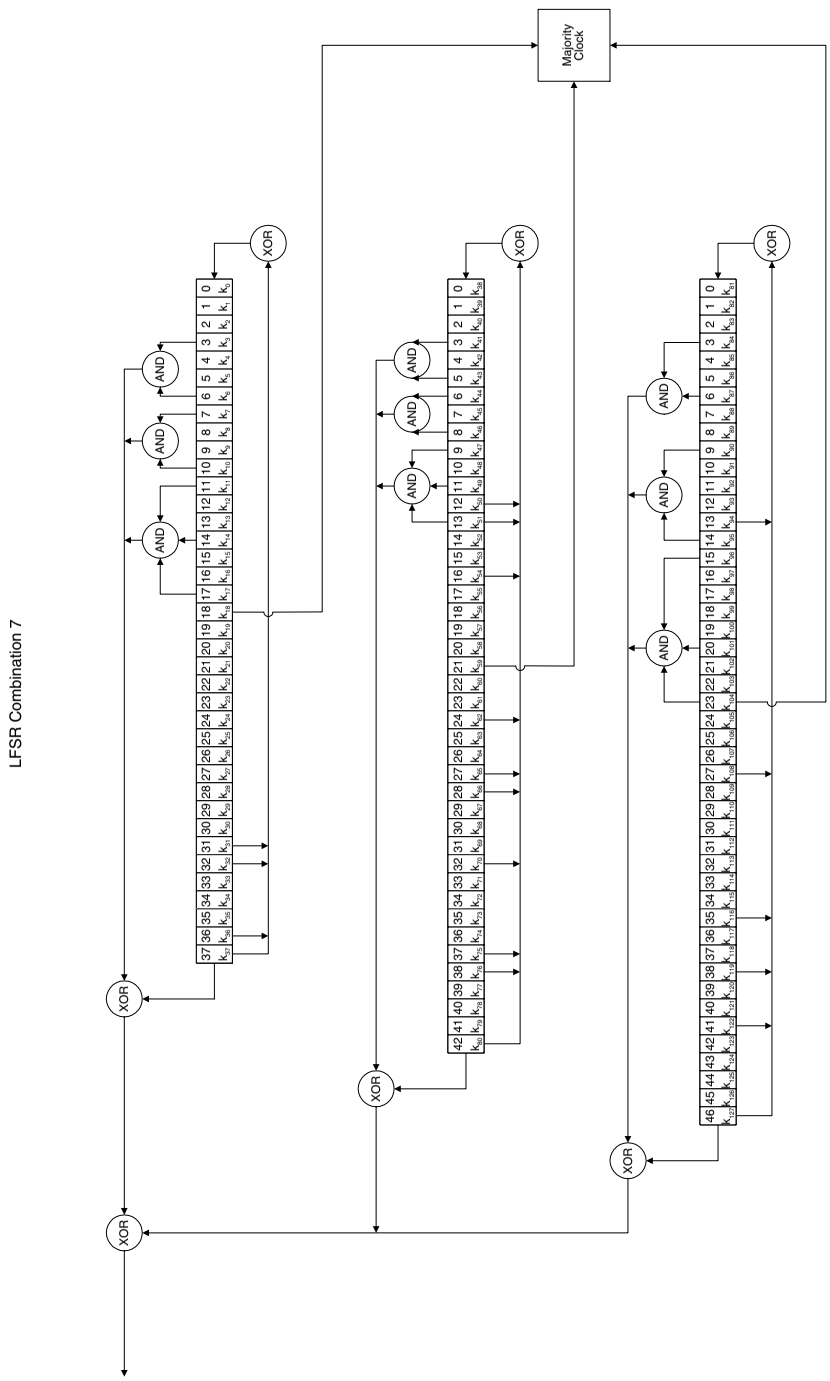

LFSR Combination 2
Majority Clock
XOR
AND


LFSR Combination 3
Majority Clock
XOR
AND


LFSR Combination 4
Majority Clock
XOR
AND


Majority Clock
XOR
AND
LFSR Combination 5


LFSR Combination 6
Majority Clock
XOR
AND


LFSR Combination 7
Majority Clock
XOR
AND

Full Copyright Statement

"Copyright (C) The Internet Society (date). All Rights Reserved. This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages.