3GPP2 S.S0053
Version 1.0
Version Date: 21 January 2002

Common Cryptographic Algorithms

EDITOR

Frank Quick
Qualcomm Incorporated
5775 Morehouse Drive
San Diego, CA 92121 USA
fquick@qualcomm.com

REVISION HISTORY

REVISION HISTORY		
Revision number	*Content changes.*	*Date*
0.1	*First draft*	*01-21-02*

Table of Contents

List of Exhibits

1. Introduction

This document describes detailed cryptographic procedures for wireless system applications. These procedures are used to perform the security services of mobile station authentication, subscriber message encryption, and encryption key and subscriber voice privacy key generation within wireless equipment.

This document is organized as follows:

§2 describes the Cellular Authentication, Voice Privacy and Encryption (CAVE) algorithm used for authentication of mobile subscriber equipment and for generation of cryptovariables to be used in other procedures.

§2.2 describes the procedure to verify the manual entry of the subscriber authentication key (A-key).

§2.3 describes the generation of intermediate subscriber cryptovariables, Shared Secret Data (SSD), from the unique and private subscriber A-key.

§2.4 describes the authentication signature calculation procedure.

§2.5 describes the procedures used for generating cryptographic keys. These keys include the Voice Privacy Mask (VPM), the Cellular Message Encryption Algorithm (CMEA) key, and Enhanced Cellular Message Encryption Algorithm (ECMEA) secrets and keys. The VPM is used to provide forward link and reverse link voice confidentiality over the air interface. The CMEA key is used with the CMEA algorithm for protection of digital data exchanged between the mobile station and the base station. The ECMEA secrets and keys are used with the ECMEA algorithm for enhanced protection of signaling messages.

§2.6 describes the Cellular Message Encryption Algorithm (CMEA) and the Enhanced Cellular Message Encryption Algorithm (ECMEA), used for enciphering and deciphering subscriber data exchanged between the mobile station and the base station.

§2.7 describes the procedures for key and authentication signature generation for wireless residential extension applications.

§2.8 describes the ORYX algorithm and procedures for key and mask generation for encryption and decryption in wireless data services.

§2.9 describes the SCEMA algorithm, which may be used for voice and data privacy.

§3 provides test data (vectors) that may be employed to verify the correct operation of the cryptographic algorithms described in this document.

Manufacturers are cautioned that no mechanisms should be provided for the display at the ACRE, PB or mobile station (or any other equipment that may be interfaced with it) of valid A-key, SSD_A, SSD_B, MANUFACT_KEY, WIKEY, WRE_KEY or other cryptovariables associated with the cryptographic functions described in this document. The invocation of test mode in the ACRE, PB or mobile station must not alter the operational values of A-key, SSD_A, SSD_B MANUFACT_KEY, WIKEY, WRE_KEY or other cryptovariables.

1.1. Notations

The notation 0x indicates a hexadecimal (base 16) number.

Binary numbers are expressed as a string of zero(s) and/or one(s) followed by a lower-case “b”.

Data arrays are indicated by square brackets, as Array[ ]. Array indices start at zero (0). Where an array is loaded using a quantity that spans several array elements, the most significant bits of the quantity are loaded into the element having the lowest index. Similarly, where a quantity is loaded from several array elements, the element having the lowest index provides the most significant bits of the quantity.

For example, Exhibit 2-1 shows the mixing registers R[00] through R[15] and the linear feedback shift register (LFSR). In this exhibit, the mixing registers are loaded from left (most significant bit) to right (least significant bit). Similarly, the LFSR is loaded with the most significant bits in its leftmost octet (LFSR A7-A0) and the least significant bits into its rightmost octet (LFSR D7-D0).

This document uses ANSI C language programming syntax to specify the behavior of the cryptographic algorithms (see ANSI/ISO 9899-1990, “Programming Languages - C”). This specification is not meant to constrain implementations. Any implementation that demonstrates the same behavior at the external interface as the algorithm specified herein, by definition, complies with this standard.

1.2. Definitions

AAV Authentication Algorithm Version, an 8-bit constant equal to hexadecimal 0xC7, used in the CAVE algorithm. Use of different values for this constant in some future version would allow other “versions” or “flavors” of the basic CAVE algorithm.

ACRE Authorization and Call Routing Equipment. A network device which authorizes the Personal Base and provides automatic call routing.

ACRE_PHONE_NUMBER	A 24-bit pattern comprised of the last 6 digits of the ACRE's directory number.
A-key	A 64-bit cryptographic key variable stored in the semi-permanent memory of the mobile station and also known to the Authentication Center (AC or HLR/AC) of the wireless system. It is entered when the mobile station is first put into service with a particular subscriber, and usually will remain unchanged unless the operator determines that its value has been compromised. The A-key is used in the SSD generation procedure.
AND	Bitwise logical AND function.
Boolean	Describes a quantity whose value is either TRUE or FALSE.
CAVE	Cellular Authentication and Voice Encryption algorithm.
CaveTable	A lookup table consisting of 256 8-bit quantities. The table, partitioned into table0 and table1, is used in the CAVE algorithm.
CMEA	Cellular Message Encryption Algorithm.
CMEAKEY	A 64-bit cryptographic key stored in eight 8-bit registers identified separately as k0, k1, ... k7 or CMEAKEY[0 through 7]. The data in these registers results from the action of the CAVE algorithm and is used to encrypt certain messages.
DataKey	A 32-bit cryptographic key used for generation of masks for encryption and decryption in wireless data services.
Data_type	A one-bit value indicating whether the financial or non-financial data encryption parameters are used.
Directory Number	The telephone network address.
ECMEA	Enhanced Cellular Message Encryption Algorithm.
ECMEA_KEY	A 64-bit cryptographic key stored in eight 8-bit registers identified separately as ecmea_key[0 through 7]. The data in these registers results from the action of the CAVE algorithm and is used to encrypt financial messages.
ECMEA_NF_KEY	A 64-bit cryptographic key stored in eight 8-bit registers identified separately as ecmea_nf_key[0 through 7]. The data in these registers results from the action of the CAVE algorithm and is used to encrypt non-financial messages.
ESN	The 32-bit electronic serial number of the mobile station.
Internal Stored Data	Stored data that is defined locally within the cryptographic procedures and is not accessible for examination or use outside those procedures.
Iteration	Multi-round execution of the CAVE algorithm. All applications of CAVE throughout this document use either four or eight rounds per iteration.
k0,k1...k7	Eight 8-bit registers whose contents constitute the CMEA key.
LFSR	A 32-bit Linear Feedback Shift Register used in the CAVE algorithm, which is composed of four 8-bit registers.
LFSR_A	The A register, a synonym for bits 31-24 of the LFSR.
LFSR_B	The B register, a synonym for bits 23-16 of the LFSR.
LFSR_C	The C register, a synonym for bits 15-8 of the LFSR.

LFSR_D The D register, a synonym for bits 7-0 of the LFSR.

LFSR-Cycle An LFSR-cycle consists of the following steps:

1. Compute the value of bit A7 using the formula A7 = B6 XOR D2 XOR D1 XOR D0. Save this value temporarily without changing the prior value of the A7 bit in the A register.
2. Perform a linked 1-bit right shift on the 32-bit LFSR, and discard the D0 bit which has been shifted out.
3. Use the previously computed and stored value of bit A7 from the first of these three statements.

LSB Least Significant Bit.

MSB Most Significant Bit.

OR Bitwise logical inclusive OR function.

Offset1 An 8-bit quantity that points to one of the 256 4-bit values in table0. Arithmetic operations on Offset1 are performed modulo 256. Also called offset_1.

Offset2 An 8-bit quantity that points to one of the 256 4-bit values in table1. Arithmetic operations on Offset2 are performed modulo 256. Also called offset_2.

offset_key A 32-bit cryptographic key stored in four 8-bit registers identified separately as offset_key[0 through 3] whose contents are used to create offsets that are passed to ECMEA.

offset_nf_key A 32-bit cryptographic key stored in four 8-bit registers identified separately as offset_nf_key[0 through 3] whose contents are used to create offsets that are passed to ECMEA for use in encryption of non-financial data.

PB Personal Base. A fixed device which provides cordless like service to a mobile station.

PBID Personal Base Identification Code.

RAND_ACRE A 32-bit random number which is generated by the PB.

RAND_PB A 32-bit random number which is generated by the ACRE.

RAND_WIKEY A 56-bit random number which is generated by the ACRE.

RAND_WRE A 19-bit random number which is generated by the PB.

Round A round is one individual execution of the CAVE mixing function.

R00-R15 Sixteen separate 8-bit mixing registers used in the CAVE algorithm. Also called register[0 through 15].

SEED_NF_KEY Five 8-bit registers whose content constitutes the 40-bit binary quantity generated after the CMEA key and used to initialize the CAVE algorithm for generation of the ECMEA_NF key and offset_nf keys.

SSD SSD is an abbreviation for Shared Secret Data. It consists of two quantities, SSD_A and SSD_B.

SSD_A A 64-bit binary quantity in the semi-permanent memory of the mobile station and also known to the Authentication Center. It may be shared with the serving MSC. It is used in the computation of the authentication response.

SSD_A_NEW	The revised 64-bit quantity held separately from SSD_A, generated as a result of the SSD generation process.
SSD_B	A 64-bit binary quantity in the semi-permanent memory of the mobile station and also known to the Authentication Center. It may be shared with the serving MSC. It is used in the computation of the CMEA key, VPM and DataKey.
SSD_B_NEW	The revised 64-bit quantity held separately from SSD_B, generated as a result of the SSD generation process.
Sync	A 16-bit value provided by the air interface used to generate offsets for ECMEA.
table0	The low-order four bits of the 256-octet lookup table used in the CAVE algorithm. Computed as `CaveTable[] AND 0x0F`.
table1	The high-order four bits of the 256-octet lookup table used in the CAVE algorithm. Computed as `CaveTable[] AND 0xF0`.
VPM	Voice Privacy Mask. This name describes a 520-bit entity that may be used for voice privacy functions as specified in wireless system standards.
WIKEY	Wireline Interface key. A 64-bit pattern stored in the PB and the ACRE (in semi-permanent memory).
WIKEY_NEW	A 64-bit pattern stored in the PB and the ACRE. It contains the value of an updated WIKEY.
WRE_KEY	Wireless Residential Extension key. A 64-bit pattern stored in the PB and the MS (in semi-permanent memory).
XOR	Bitwise logical exclusive OR function.

2. Procedures

2.1. CAVE

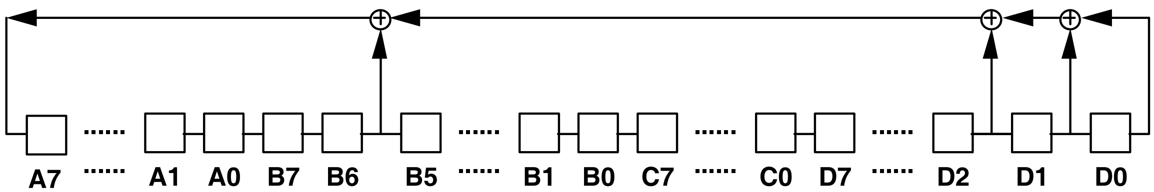
CAVE is a software-compatible non-linear mixing function shown in Exhibit 2-1. Its primary components are a 32-bit linear-feedback shift register (LFSR), sixteen 8-bit mixing registers, and a 256-entry lookup table. The table is organized as two (256 x 4 bit) tables. The 256-octet table is listed in Exhibit 2-5. The low order four bits of the entries comprise *table0* and the high order four bits of the entries comprise *table1*.

The pictorial arrangement of Exhibit 2-1 shows that the linear-feedback shift register (LFSR) consists of the 8-bit register stages A, B, C, and D. The CAVE process repeatedly uses the LFSR and table to randomize the contents of the 8-bit mixing register stages R00, R01, R02, R03, R04, R05, R06, R07, R08, R09, R10, R11, R12, R13, R14, and R15. Two lookup table pointer offsets further randomize table access. The registers are shifted one bit to the right. Finally, eight 16-entry permutation recipes are embedded in the lookup tables to "shuffle" registers R00 through R15 after each computational "round" through the algorithm.

The algorithm operation consists of three steps: an initial loading, a repeated randomization consisting of four or eight "rounds", and processing of the output. Initial loading consists of filling the LFSR, register stages R00 through R15, and the pointer offsets with information that is specific to the application. The randomization process is common to all cases that will be described in the later sections. Randomization is a detailed operation; it is described below by means of Exhibit 2-1, Exhibit 2-2, and Exhibit 2-5. The output processing utilizes the final (randomized) contents of R00 through R15 in a simple function whose result is returned to the calling process.

The CAVE Algorithm may be applied in a number of different cases. In each, there are different initialization requirements, and different output processing. All cases are detailed in §2.2 through §2.9 of this document.

Exhibit 2-1 CAVE Elements



A7(input) = B6 XOR D2 XOR D1 XOR D0

Mixing Registers:

R00	R01	R02	R03	R04	R05	R06	R07	R08	R09	R10	R11	R12	R13	R14	R15

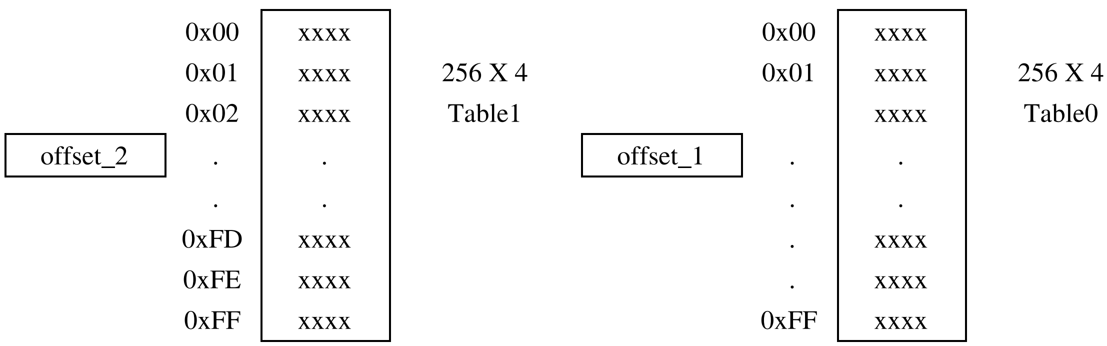


Exhibit 2-2 CAVE Algorithm External Header

```
#ifndef CAVE_H
#define CAVE_H

/* external header for CAVE and related procedures */

/* function declarations */

void CAVE(const int number_of_rounds,
          int *offset_1,
          int *offset_2);

void A_Key_Checksum(const char A_KEY_DIGITS[20],
                    char A_KEY_CHECKSUM[6]);

int A_Key_Verify(const char A_KEY_DIGITS[26]);

void SSD_Generation(const unsigned char RANDSSD[7]);

void SSD_Update(void);

unsigned long Auth_Signature(const unsigned char RAND_CHALLENGE[4],
                             const unsigned char AUTH_DATA[3],
                             const unsigned char *SSD_AUTH,
                             const int SAVE_REGISTERS);

void Key_VPM_Generation(void);

void CMEA(unsigned char *msg_buf, const int octet_count);

/* global variable definitions */

#ifdef CAVE_SOURCE_FILE
#define CAVE_GLOBAL
#else
#define CAVE_GLOBAL extern
#endif

/* externally available results */

CAVE_GLOBAL
unsigned char     cmeakey[8];

CAVE_GLOBAL
unsigned char     VPM[65];

CAVE_GLOBAL
unsigned char     SAVED_LFSR[4];
CAVE_GLOBAL
int               SAVED_OFFSET_1;
CAVE_GLOBAL
int               SAVED_OFFSET_2;
CAVE_GLOBAL
unsigned char     SAVED_RAND[4];
CAVE_GLOBAL
unsigned char     SAVED_DATA[3];
```

```
/* global constant definitions */

#ifndef CAVE_SOURCE_FILE

CAVE_GLOBAL
unsigned char     CaveTable[256];

CAVE_GLOBAL
unsigned char     ibox[256];

#endif // ifndef CAVE_SOURCE_FILE

#endif // ifndef CAVE_H

/* end of CAVE external header */
/**********************************************************************/
```

Exhibit 2-3 CAVE Algorithm Internal Header

```
/* internal header for CAVE, used by all cryptographic source files */

#include "cave.h" /* see Exhibit 2-2 */

/* authentication algorithm version (fixed) */

#define AAV 0xC7

#define LOMASK       0x0F
#define HIMASK       0xF0
#define TRUE         1
#define FALSE        0

/* NAM stored data */

extern
unsigned char     ESN[4];

extern
unsigned char     A_key[8];

extern
unsigned char     SSD_A_NEW[8], SSD_A[8];

extern
unsigned char     SSD_B_NEW[8], SSD_B[8];

/* saved outputs */

CAVE_GLOBAL
unsigned char     LFSR[4];

#define LFSR_A LFSR[0]
#define LFSR_B LFSR[1]
#define LFSR_C LFSR[2]
#define LFSR_D LFSR[3]

CAVE_GLOBAL
unsigned char     Register[16];
```

Exhibit 2-4 CAVE Algorithm

```
#define CAVE_SOURCE_FILE
#include "cavei.h" /* see Exhibit 2-3 */

/******************************************************************/

/* table0 is the 4 lsbs of the array,
   table1 is the 4 msbs of the array */

unsigned char     CaveTable[256] =

   {  0xd9, 0x23, 0x5f, 0xe6, 0xca, 0x68, 0x97, 0xb0,
      0x7b, 0xf2, 0x0c, 0x34, 0x11, 0xa5, 0x8d, 0x4e,
      0x0a, 0x46, 0x77, 0x8d, 0x10, 0x9f, 0x5e, 0x62,
      0xf1, 0x34, 0xec, 0xa5, 0xc9, 0xb3, 0xd8, 0x2b,
      0x59, 0x47, 0xe3, 0xd2, 0xff, 0xae, 0x64, 0xca,
      0x15, 0x8b, 0x7d, 0x38, 0x21, 0xbc, 0x96, 0x00,
      0x49, 0x56, 0x23, 0x15, 0x97, 0xe4, 0xcb, 0x6f,
      0xf2, 0x70, 0x3c, 0x88, 0xba, 0xd1, 0x0d, 0xae,
      0xe2, 0x38, 0xba, 0x44, 0x9f, 0x83, 0x5d, 0x1c,
      0xde, 0xab, 0xc7, 0x65, 0xf1, 0x76, 0x09, 0x20,
      0x86, 0xbd, 0x0a, 0xf1, 0x3c, 0xa7, 0x29, 0x93,
      0xcb, 0x45, 0x5f, 0xe8, 0x10, 0x74, 0x62, 0xde,
      0xb8, 0x77, 0x80, 0xd1, 0x12, 0x26, 0xac, 0x6d,
      0xe9, 0xcf, 0xf3, 0x54, 0x3a, 0x0b, 0x95, 0x4e,
      0xb1, 0x30, 0xa4, 0x96, 0xf8, 0x57, 0x49, 0x8e,
      0x05, 0x1f, 0x62, 0x7c, 0xc3, 0x2b, 0xda, 0xed,
      0xbb, 0x86, 0x0d, 0x7a, 0x97, 0x13, 0x6c, 0x4e,
      0x51, 0x30, 0xe5, 0xf2, 0x2f, 0xd8, 0xc4, 0xa9,
      0x91, 0x76, 0xf0, 0x17, 0x43, 0x38, 0x29, 0x84,
      0xa2, 0xdb, 0xef, 0x65, 0x5e, 0xca, 0x0d, 0xbc,
      0xe7, 0xfa, 0xd8, 0x81, 0x6f, 0x00, 0x14, 0x42,
      0x25, 0x7c, 0x5d, 0xc9, 0x9e, 0xb6, 0x33, 0xab,
      0x5a, 0x6f, 0x9b, 0xd9, 0xfe, 0x71, 0x44, 0xc5,
      0x37, 0xa2, 0x88, 0x2d, 0x00, 0xb6, 0x13, 0xec,
      0x4e, 0x96, 0xa8, 0x5a, 0xb5, 0xd7, 0xc3, 0x8d,
      0x3f, 0xf2, 0xec, 0x04, 0x60, 0x71, 0x1b, 0x29,
      0x04, 0x79, 0xe3, 0xc7, 0x1b, 0x66, 0x81, 0x4a,
      0x25, 0x9d, 0xdc, 0x5f, 0x3e, 0xb0, 0xf8, 0xa2,
      0x91, 0x34, 0xf6, 0x5c, 0x67, 0x89, 0x73, 0x05,
      0x22, 0xaa, 0xcb, 0xee, 0xbf, 0x18, 0xd0, 0x4d,
      0xf5, 0x36, 0xae, 0x01, 0x2f, 0x94, 0xc3, 0x49,
      0x8b, 0xbd, 0x58, 0x12, 0xe0, 0x77, 0x6c, 0xda  };
```

```
unsigned char ibox[256] =

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

```c
/* CAVE local functions */

static unsigned char bit_val(const unsigned char octet, const int bit)
{
   return((octet << (7 - bit)) & 0x80);
}

static void LFSR_cycle(void)
{
   unsigned char temp;
   int i;

   temp  = bit_val(LFSR_B,6);
   temp ^= bit_val(LFSR_D,2);
   temp ^= bit_val(LFSR_D,1);
   temp ^= bit_val(LFSR_D,0);

   /* Shift right LFSR, Discard LFSR_D[0] bit */

   for (i = 3; i > 0; i--)
   {
      LFSR[i] >>= 1;
      if (LFSR[i-1] & 0x01)
         LFSR[i] |= 0x80;
   }
   LFSR[0] >>= 1;

   LFSR_A |= temp;
}

static void Rotate_right_registers(void)
{
   unsigned int temp_reg;
   int i;

   temp_reg = Register[15]; /* save lsb */

   for (i = 15; i > 0; i--)
   {
      Register[i] >>= 1;
      if (Register[i-1] & 0x01)
         Register[i] |= 0x80;
   }

   Register[0] >>= 1;
   if (temp_reg & 0x01)
      Register[0] |= 0x80;
}
```

```
void CAVE(const int number_of_rounds,
          int *offset_1,
          int *offset_2)
{
   unsigned char     temp_reg0;
   unsigned char     lowNibble;
   unsigned char     hiNibble;
   unsigned char     temp;
   int               round_index;
   int               R_index;
   int               fail_count;
   unsigned char     T[16];

   for (round_index = number_of_rounds - 1;
        round_index >= 0;
        round_index--)
   {
      /* save R0 for reuse later */
      temp_reg0 = Register[0];

      for (R_index = 0; R_index < 16; R_index++)
      {
         fail_count = 0;
         while(1)
         {
            *offset_1 += (LFSR_A ^ Register[R_index]);
            /* will overflow; mask to prevent */
            *offset_1 &= 0xff;
            lowNibble = CaveTable[*offset_1] & LOMASK;
            if (lowNibble == (Register[R_index] & LOMASK))
            {
               LFSR_cycle();
               fail_count++;
               if (fail_count == 32)
               {
                  LFSR_D++; /* no carry to LFSR_C */
                  break;
               }
            }
            else break;
         }
```

```
            fail_count = 0;
            while(1)
            {
               *offset_2 += (LFSR_B ^ Register[R_index]);
               /* will overflow; mask to prevent */
               *offset_2 &= 0xff;
               hiNibble = CaveTable[*offset_2] & HIMASK;
               if (hiNibble == (Register[R_index] & HIMASK))
               {
                  LFSR_cycle();
                  fail_count++;
                  if (fail_count == 32)
                  {
                     LFSR_D++; /* no carry to LFSR_C */
                     break;
                  }
               }
               else
                  break;
            }


            temp = lowNibble | hiNibble;
            if (R_index == 15)
               Register[R_index] = temp_reg0 ^ temp;
            else
               Register[R_index] = Register[R_index+1] ^ temp;

            LFSR_cycle();
         }

         Rotate_right_registers();

         /* shuffle the mixing registers */
         for (R_index = 0; R_index < 16; R_index++)
         {
            temp = CaveTable[16*round_index + R_index] & LOMASK;
            T[temp] = Register[R_index];
         }
         for (R_index = 0; R_index < 16; R_index++)
            Register[R_index] = T[R_index];
      }
   }
```

Exhibit 2-5 CAVE Table

table0 is comprised by the 4 LSBs of the array
table1 is comprised by the 4 MSBs of the array

This table is read by rows, e.g. CaveTable[0x12] = 0x77.

hi/lo	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	D9	23	5F	E6	CA	68	97	B0	7B	F2	0C	34	11	A5	8D	4E
1	0A	46	77	8D	10	9F	5E	62	F1	34	EC	A5	C9	B3	D8	2B
2	59	47	E3	D2	FF	AE	64	CA	15	8B	7D	38	21	BC	96	00
3	49	56	23	15	97	E4	CB	6F	F2	70	3C	88	BA	D1	0D	AE
4	E2	38	BA	44	9F	83	5D	1C	DE	AB	C7	65	F1	76	09	20
5	86	BD	0A	F1	3C	A7	29	93	CB	45	5F	E8	10	74	62	DE
6	B8	77	80	D1	12	26	AC	6D	E9	CF	F3	54	3A	0B	95	4E
7	B1	30	A4	96	F8	57	49	8E	05	1F	62	7C	C3	2B	DA	ED
8	BB	86	0D	7A	97	13	6C	4E	51	30	E5	F2	2F	D8	C4	A9
9	91	76	F0	17	43	38	29	84	A2	DB	EF	65	5E	CA	0D	BC
A	E7	FA	D8	81	6F	00	14	42	25	7C	5D	C9	9E	B6	33	AB
B	5A	6F	9B	D9	FE	71	44	C5	37	A2	88	2D	00	B6	13	EC
C	4E	96	A8	5A	B5	D7	C3	8D	3F	F2	EC	04	60	71	1B	29
D	04	79	E3	C7	1B	66	81	4A	25	9D	DC	5F	3E	B0	F8	A2
E	91	34	F6	5C	67	89	73	05	22	AA	CB	EE	BF	18	D0	4D
F	F5	36	AE	01	2F	94	C3	49	8B	BD	58	12	E0	77	6C	DA

2.2. Authentication Key (A-Key) Procedures

2.2.1. A-Key Checksum Calculation

Procedure name:	
A_Key_Checksum	
Inputs from calling process:	
A_KEY_DIGITS	20 decimal digits
ESN	32 bits
Inputs from internal stored data:	
AAV	8 bits
Outputs to calling process:	
A_KEY_CHECKSUM	6 decimal digits
Outputs to internal stored data:	
None.	

This procedure computes the checksum for an A-key to be entered into a mobile station. In a case where the number of digits to be entered is less than 20, the leading most significant digits will be set equal to zero.

The generation of the A-key is the responsibility of the service provider. A-keys should be chosen and managed using procedures that minimize the likelihood of compromise.

The checksum provides a check for the accuracy of the A-Key when entered into a mobile station. The 20 A-Key digits are converted into a 64-bit representation to serve as an input to CAVE, along with the mobile station's ESN. CAVE is then run in the same manner as for the Auth_Signature procedure, and its 18-bit response is the A-Key checksum. The checksum is returned as 6 decimal digits for entry into the mobile station.

The first decimal digit of the A-Key to be entered is considered to be the most significant of the 20 decimal digits, followed in succession by the other nineteen. A decimal to binary conversion process converts the digit sequence into its equivalent mod-2 representation. For example, the 20 digits

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0

have a hexadecimal equivalent of

A B 5 4 A 9 8 C E B 1 F 0 A D 2.

CAVE will be initialized as shown in Exhibit 2-6. First, the 32 most significant bits of the 64-bit entered number will be loaded into the LFSR. If this 32-bit pattern fills the LFSR with all zeros, then the LFSR will be loaded with the ESN. Then, in all instances, the entire 64-bit entered number will be put into R00 through R07. The least significant 24 bits will be repeated into R09, R10, and R11. Authentication Algorithm Version (hexadecimal C7) will occupy R08, and ESN will be loaded into R12 through R15. CAVE will then be performed for eight rounds, as described in §2.1. The checksum is obtained from the final value of CAVE registers R00, R01, R02, R13, R14, and R15. The two most significant bits of the checksum are equal to the two least significant bits of R00 XOR R13. The next eight bits of the checksum are equal to R01 XOR R14. Finally, the least significant bits of the checksum are equal to R02 XOR R15.

The 18-bit checksum is returned as 6 decimal digits for entry into the mobile station.

Exhibit 2-6 CAVE Initial Loading for A-key Checksum

CAVE Element	Source Identifier		Size (Bits)
	32 MSBs of A-key all zeros	32 MSBs of A-key not all zeros	
LFSR	ESN	32 MSBs of A-key	32
Register [0-7]	A-key	A-key	64
Register [8]	AAV	AAV	8
Register [9-11]	24 LSBs of A-key	24 LSBs of A-key	24
Register [12-15]	ESN	ESN	32

Exhibit 2-7 A-key Checksum

```
/* A_Key_Checksum has the same header as CAVE (see Exhibit 2-4) */

static void mul10(unsigned char i64[8], unsigned int carry)
{
   int i;
   unsigned int temp;

   for (i = 7; i >= 0; i--)
   {
      temp = ((unsigned int)(i64[i]) * 10) + carry;
      i64[i] = temp & 0xFF;
      carry =  temp >> 8;
   }
}

static unsigned long Calc_Checksum(const unsigned char A_key[8])
{
   int i,offset_1,offset_2;
   unsigned long A_key_checksum;

   /* see if 32 MSB are zero */

   if ((A_key[0] | A_key[1] | A_key[2] | A_key[3]) != 0)
   {
      /* put 32 MSB into LFSR */
      for (i = 0; i < 4; i++)
         LFSR[i] = A_key[i];
   }
   else
   {
      /* put ESN into LFSR */
      for (i = 0; i < 4; i++)
         LFSR[i] = ESN[i];
   }

   /* put A_key into r0-r7 */

   for (i = 0; i < 8; i++)
      Register[i] = A_key[i];

   Register[8] = AAV;

   /* put ls 24 bits of A_key into r9-r11 */

   for (i = 9; i < 12; i++)
      Register[i] = A_key[5+i-9];

   /* put ESN into r12-r15 */
   for (i = 12; i < 16; i++)
      Register[i] = ESN[i-12];

   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);
```

```
   A_key_checksum =
        ( ((unsigned long)(Register[0] ^ Register[13]) << 16) +
          ((unsigned long)(Register[1] ^ Register[14]) << 8)  +
          ((unsigned long)(Register[2] ^ Register[15]) )) & 0x3ffff;

   return (A_key_checksum);
}

/* A_KEY_DIGITS contains the ASCII digits in the order to be entered */

void A_Key_Checksum(const char A_KEY_DIGITS[20],
                    char A_KEY_CHECKSUM[6])
{
   int i;
   unsigned char temp_A_key[8];
   unsigned long A_key_checksum;

   /* convert digits to 64-bit representation in temp_A_key */

   for (i = 0; i < 8; i++)
      temp_A_key[i] = 0;

   for (i = 0; i < 20; i++)
   {
      mul10(temp_A_key,(unsigned int)(A_KEY_DIGITS[i] - '0'));
   }

   A_key_checksum = Calc_Checksum(temp_A_key);

   /* convert checksum to decimal digits */

   for (i = 0; i < 6; i++)
   {
      A_KEY_CHECKSUM[5-i] = '0' + (char)(A_key_checksum % 10);
      A_key_checksum /= 10;
   }
}
```

2.2.2. A-Key Verification

Procedure name:	
A_Key_Verify	
Inputs from calling process:	
A_KEY_DIGITS	from 6 to 26 decimal digits
ESN	32 bits
Inputs from internal stored data:	
AAV	8 bits
Outputs to calling process:	
A_KEY_VERIFIED	Boolean
Outputs to internal stored data:	
A-key	64 bits
SSD_A	64 bits (set to zero)
SSD_B	64 bits (set to zero)

The A-key may be entered into the mobile station by any of several methods. These include direct electronic entry, over-the-air procedures, and manual entry via the mobile station's keypad. This procedure verifies the A-key entered into a mobile station via the keypad.

The default value of the A-key when the mobile station is shipped from the factory will be all binary zeros. The value of the A-key is specified by the operator and is to be communicated to the subscriber according to the methods specified by each operator. A multiple NAM mobile station will require multiple A-keys, as well as multiple sets of the corresponding cryptovariables per A-key.

While A-key digits are being entered from a keypad, the mobile station transmitter shall be disabled.

When the A-key digits are entered from a keypad, the number of digits entered is to be at least 6, and may be any number of digits up to and including 26 digits. In a case where the number of digits entered is less than 26, the leading most significant digits will be set equal to zero, in order to produce a 26-digit quantity called the "entry value".

The verification procedure checks the accuracy of the 26 decimal digit entry value. If the verification is successful, the 64-bit pattern determined by the first 20 digits of the entry value will be written to the subscriber's semi-permanent memory as the A-key. Furthermore, the SSD_A and the SSD_B will be set to zero. The return value A_KEY_VERIFIED will be set to TRUE. In the case of a mismatch, A_KEY_VERIFIED is set to FALSE, and no internal data is updated.

The first decimal digit of the “entry value” is considered to be the most significant of the 20 decimal digits, followed in succession by the other nineteen. The twenty-first digit is the most significant of the check digits, followed in succession by the remaining five. For example, the 26 digits

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0, 1 3 1 1 3 6

has a hexadecimal equivalent of

A B 5 4 A 9 8 C E B 1 F 0 A D 2, 2 0 0 4 0.

Exhibit 2-8 A-key Verification

```
/* A_Key_Verify has the same header as CAVE (see Exhibit 2-4) */

/* A_KEY_DIGITS contains the ASCII digits in the order entered */

int A_Key_Verify(const char A_KEY_DIGITS[26])
{
   int i;
   unsigned char temp_A_key[8];
   unsigned long entered_checksum;

   /* convert first 20 digits to 64-bit representation in temp_A_key */

   for (i = 0; i < 8; i++)
      temp_A_key[i] = 0;

   for (i = 0; i < 20; i++)
   {
      mul10(temp_A_key,(unsigned int)(A_KEY_DIGITS[i] - '0'));
   }

   /* convert last 6 digits to entered checksum */

   entered_checksum = 0;
   for (i = 20; i < 26; i++)
   {
      entered_checksum = (entered_checksum * 10)
         + (A_KEY_DIGITS[i] - '0');
   }

   if(Calc_Checksum(temp_A_key) == entered_checksum)
   {
      for (i = 0; i < 8; i++)
      {
         A_key[i] = temp_A_key[i];
         SSD_A[i] = SSD_B[i] = 0;
      }
      return TRUE;
   }
   else
   {
      return FALSE;
   }
}
```

2.3. SSD Generation and Update

2.3.1. SSD Generation Procedure

Procedure name:	
SSD_Generation	
Inputs from calling process:	
RANDSSD	56 bits
ESN	32 bits
Inputs from internal stored data:	
AAV	8 bits
A-key	64 bits
Outputs to calling process:	
None.	
Outputs to internal stored data:	
SSD_A_NEW	64 bits
SSD_B_NEW	64 bits

This procedure performs the calculation of Shared Secret Data. The result is held in memory as SSD_A_NEW and SSD_B_NEW until the SSD_Update procedure (§2.3.2) is invoked. Exhibit 2-9 shows the process graphically. Exhibit 2-10 indicates the operations in ANSI C.

The input variables for this procedure are: RANDSSD (56 bits), Authentication Algorithm Version (8 bits), ESN (32 bits), and A-key (64 bits). CAVE will be initialized as follows. First, the LFSR will be loaded with the 32 least significant bits of RANDSSD XOR'd with the 32 most significant bits of A-key XOR'd with the 32 least significant bits of A-key. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be loaded with the 32 least significant bits of RANDSSD to prevent a trivial null result.

Registers R00 through R07 will be initialized with A-key, R08 will be the 8-bit Authentication Algorithm Version (11000111). R09, R10, and R11 will be the most significant bits of RANDSSD, and the ESN will be loaded into R12 through R15. Offset1 and Offset2 will initially be set to 128.

CAVE will be run for 8 rounds as previously described in §2.1. When this is complete, registers R00 through R07 will become SSD_A_NEW and Registers R08 through R15 will become SSD_B_NEW.

Exhibit 2-9 Generation of SSD_A_NEW and SSD_B_NEW

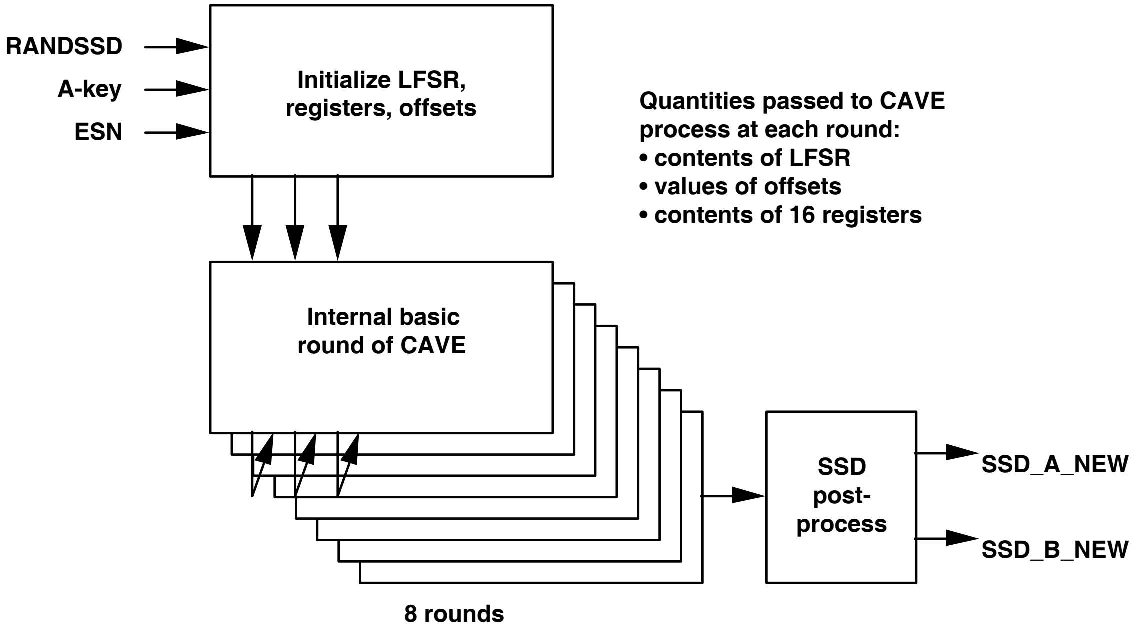


Exhibit 2-10 SSD Generation

```
/* SSD_Generation has the same header as CAVE (see Exhibit 2-4) */

void SSD_Generation(const unsigned char RANDSSD[7])
{
   int i,offset_1,offset_2;

   for (i = 0; i < 4; i++)
   {
      LFSR[i] = RANDSSD[i+3] ^ A_key[i] ^ A_key[i+4];
   }

   if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
   {
      for (i = 0; i < 4; i++)
         LFSR[i] = RANDSSD[i+3];
   }

   for (i = 0; i < 8; i++)
      Register[i] = A_key[i];

   Register[8] = AAV;

   for (i = 9; i < 12; i++)
      Register[i] = RANDSSD[i-9];

   for (i = 12; i < 16; i++)
      Register[i] = ESN[i-12];

   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);

   for (i = 0; i < 8; i++)
   {
      SSD_A_NEW[i] = Register[i];
      SSD_B_NEW[i] = Register[i+8];
   }
}
```

2.3.2. SSD Update Procedure

Procedure name:

SSD_Update

Inputs from calling process:

None.

Inputs from internal stored data:

SSD_A_NEW	64 bits
SSD_B_NEW	64 bits

Outputs to calling process:

None.

Outputs to internal stored data:

SSD_A	64 bits
SSD_B	64 bits

This procedure copies the values SSD_A_NEW and SSD_B_NEW into the stored SSD_A and SSD_B. Exhibit 2-11 indicates the operations in ANSI C.

The values SSD_A_NEW and SSD_B_NEW calculated by the SSD_Generation procedure (§2.3.1) should be validated prior to storing them permanently as SSD_A and SSD_B. The base station and the mobile station should exchange validation data sufficient to determine that the values of the Shared Secret Data are the same in both locations. When validation is completed successfully, the SSD_Update procedure is invoked, setting SSD_A to SSD_A_NEW and setting SSD_B to SSD_B_NEW.

Exhibit 2-11 SSD Update

```
/* SSD_Update has the same header as CAVE (see Exhibit 2-4) */

void SSD_Update(void)
{
   int i;

   for (i = 0; i < 8; i++)
   {
      SSD_A[i] = SSD_A_NEW[i];
      SSD_B[i] = SSD_B_NEW[i];
   }
}
```

2.4. Authentication Signature Calculation Procedure

Procedure name:	
Auth_Signature	
Inputs from calling process:	
RAND_CHALLENGE	32 bits
ESN	32 bits
AUTH_DATA	24 bits
SSD_AUTH	64 bits
SAVE_REGISTERS	Boolean
Inputs from internal stored data:	
AAV	8 bits
Outputs to calling process:	
AUTH_SIGNATURE	18 bits
Outputs to internal stored data:	
SAVED_LFSR	32 bits
SAVED_OFFSET_1	8 bits
SAVED_OFFSET_2	8 bits
SAVED_RAND	32 bits
SAVED_DATA	24 bits

This procedure is used to calculate 18-bit signatures used for verifying the authenticity of messages used to request wireless system services, and for verifying Shared Secret Data.

The initial loading of CAVE for calculation of authentication signatures is given in Exhibit 2-12.

AAV is as defined in §1.1.

For authentication of mobile station messages and for base station challenges of a mobile station, RAND_CHALLENGE should be selected by the authenticating entity (normally the HLR or VLR). RAND_CHALLENGE must be received by the mobile station executing this procedure. Results returned by the mobile station should include check data that can be used to verify that the RAND_CHALLENGE value used by the mobile station matches that used by the authenticating entity.

For mobile station challenges of a base station, as performed during the verification of Shared Secret Data, the mobile station should select RAND_CHALLENGE. The selected value of RAND_CHALLENGE must be received by the base station executing this procedure.

When this procedure is used to generate an authentication signature for a message, AUTH_DATA should include a part of the message to be authenticated. The contents should be chosen to minimize the possibility that other messages would produce the same authentication signature.

SSD_AUTH should be either SSD_A or SSD_A_NEW computed by the SSD_Generation procedure, or SSD_A as obtained from the HLR/AC.

Exhibit 2-12 CAVE Initial Loading for Authentication Signatures

CAVE Item	Source Identifier	Size (Bits)
LFSR	RAND_CHALLENGE	32
Reg [0-7]	SSD_AUTH	64
Reg [8]	AAV	8
Reg [9-11]	AUTH_DATA	24
Reg [12-15]	ESN	32

CAVE is run for eight rounds. The 18-bit result is AUTH_SIGNATURE. Exhibit 2-13 shows the process in graphical form, while ANSI C for the process is given in Exhibit 2-14.

The LFSR will initially be loaded with RAND_CHALLENGE. This value will be XOR'd with the 32 most significant bits of SSD_AUTH XOR'd with the 32 least significant bits of SSD_AUTH, then reloaded into the LFSR. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be reloaded with RAND_CHALLENGE to prevent a trivial null result.

The 18-bit authentication result AUTH_SIGNATURE is obtained from the final value of CAVE registers R00, R01, R02, R13, R14, and R15. The two most significant bits of AUTH_SIGNATURE are equal to the two least significant bits of R00 XOR R13. The next eight bits of AUTH_SIGNATURE are equal to R01 XOR R14. Finally, the least significant bits of AUTH_SIGNATURE are equal to R02 XOR R15.

If the calling process sets SAVE_REGISTERS to TRUE, the RAND_CHALLENGE, ESN and AUTH_DATA and the contents of the LFSR, offsets and CAVE registers are saved in internal storage. If the calling process sets SAVE_REGISTERS to FALSE, the contents of internal storage are not changed. A means should be provided to indicate whether the internal storage contents are valid.

Exhibit 2-13 Calculation of AUTH_SIGNATURE

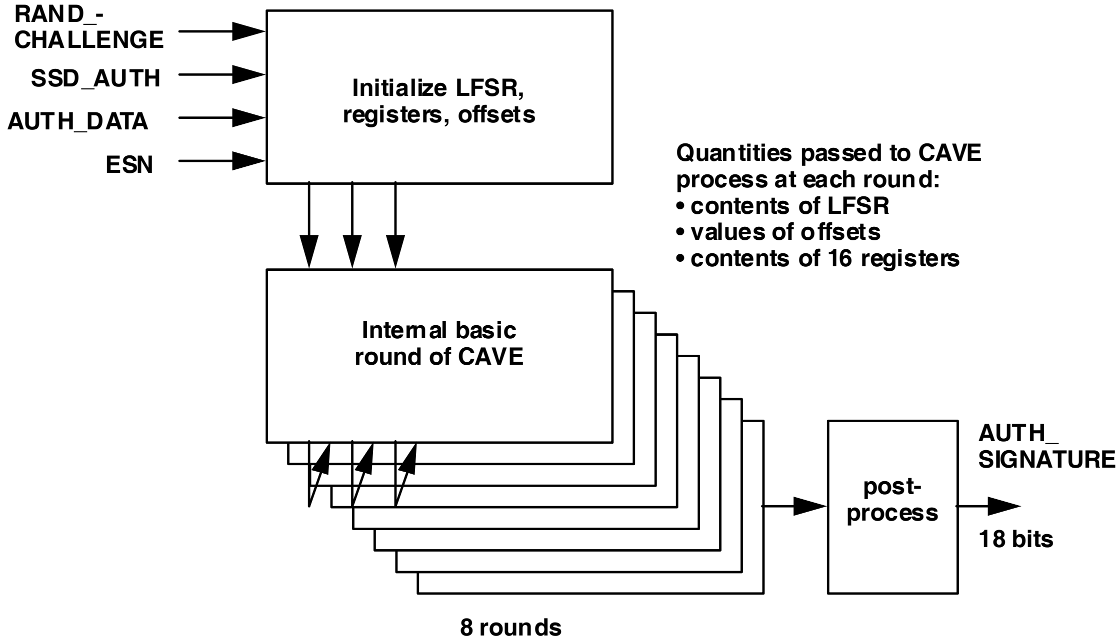


Exhibit 2-14 Code for Calculation of AUTH_SIGNATURE

```
/* Auth_Signature has the same header as CAVE (see Exhibit 2-4) */

unsigned long Auth_Signature(const unsigned char RAND_CHALLENGE[4],
                             const unsigned char AUTH_DATA[3],
                             const unsigned char *SSD_AUTH,
                             const int SAVE_REGISTERS)
{
   int i,offset_1,offset_2;
   unsigned long AUTH_SIGNATURE;

   for (i = 0; i < 4; i++)
   {
      LFSR[i] = RAND_CHALLENGE[i] ^ SSD_AUTH[i] ^ SSD_AUTH[i+4];
   }

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      for (i = 0; i < 4; i++)
         LFSR[i] = RAND_CHALLENGE[i];
   }

   /* put SSD_AUTH into r0-r7 */

   for (i = 0; i < 8; i++)
      Register[i] = SSD_AUTH[i];

   Register[8] = AAV;

   /* put AUTH_DATA into r9-r11 */

   for (i = 9; i < 12; i++)
      Register[i] = AUTH_DATA[i-9];

   /* put ESN into r12-r15 */

   for (i = 12; i < 16; i++)
      Register[i] = ESN[i-12];

   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);

   AUTH_SIGNATURE =
      ( ((unsigned long)(Register[0] ^ Register[13]) << 16) +
        ((unsigned long)(Register[1] ^ Register[14]) << 8)  +
        ((unsigned long)(Register[2] ^ Register[15]) )) & 0x3ffff;
```

```
    if (SAVE_REGISTERS)
    {
       /* save LFSR and offsets */

       SAVED_OFFSET_1 = offset_1;
       SAVED_OFFSET_2 = offset_2;
       for (i = 0; i < 4; i++)
       {
          SAVED_LFSR[i] = LFSR[i];
          SAVED_RAND[i] = RAND_CHALLENGE[i];
          if (i < 3)
          {
             SAVED_DATA[i] = AUTH_DATA[i];
          }
       }
    }

    return(AUTH_SIGNATURE);
}
```

2.5. Secret Key and Secret Parameter Generation

This section describes four procedures used for generating secret keys and other secret parameters for use in CMEA, Enhanced CMEA (ECMEA) and the voice privacy mask. The generation of distinct secrets for ECMEA-encryption of financial and non-financial messages (e.g. user data) is addressed.

The first procedure uses SSD_B and other parameters to generate

- the secret CMEA key for message encryption, and
- the voice privacy mask.

The second procedure uses the secret CMEA key produced in the first procedure to generate the secrets used by ECMEA to encrypt financial messages.

The third procedure uses the secret CMEA key produced in the first procedure to generate the secret non-financial seed key needed to start the fourth procedure.

The fourth procedure uses the secret non-financial seed key produced in the third procedure to generate the secrets used by ECMEA to encrypt non-financial messages.

For backward compatibility with CMEA, the first procedure will always be executed. The secret CMEA key will exist in both the infrastructure and the mobile station.

When ECMEA is implemented, the second, third, and fourth procedures will be executed to produce the secret keys and parameters needed to encrypt both financial and non-financial messages.

2.5.1. CMEA Encryption Key and VPM Generation Procedure

Procedure name:

Key_VPM_Generation

Inputs from calling process:

None.

Inputs from internal stored data:

SAVED_LFSR	32 bits
SAVED_OFFSET_1	8 bits
SAVED_OFFSET_2	8 bits
SAVED_RAND	32 bits
SAVED_DATA	24 bits
SSD_B	64 bits
AAV	8 bits

Outputs to calling process:

None.

Outputs to internal stored data:

CMEAKEY[0-7]	64 bits
VPM	520 bits

This procedure computes the CMEA key for message encryption and the voice privacy mask. Prior to invoking this procedure, the authentication signature calculation procedure (§2.4) must have been invoked with SAVE_REGISTERS set to TRUE. This procedure must be invoked prior to execution of the encryption procedure (§2.5.2).

The processes for generation of the CMEA key and the voice privacy mask (VPM) will generally be most efficient when concatenated as described in the following sections (§2.5.1.1 and §2.5.1.2). The post-authentication cryptovariables to be used are those from the last authentication signature calculation for which the calling process set SAVE_REGISTERS to true. This should generally be the authentication calculation for the message that establishes the call for which encryption and/or voice privacy is to be invoked. See Exhibit 2-13 and Exhibit 2-14 for graphical detail of the generation process.

2.5.1.1. CMEA key Generation

CMEA key generation is depicted in Exhibit 2-16 and Exhibit 2-17. Eight octets of CMEA session key are derived by running CAVE through an 8-round iteration and then two 4-round iterations following an authentication. This is shown in the upper portion of Exhibit 2-16 and Exhibit 2-17. The post-authentication initialization and output processing requirements are as follows (for analog phones iterations 4 - 14 are omitted:

- First, the LFSR will be re-initialized to the exclusive-or sum of SAVED_LFSR and both halves of SSD_B. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be loaded with SAVED_RAND.
- Second, registers R00 through R07 will be initialized with SSD_B instead of SSD_A.
- Third, Registers R09, R10, and R11 will be loaded with SAVED_DATA.
- Fourth, Registers R12 through R15 will be loaded with ESN.
- Fifth, the offset table pointers will begin this process at their final authentication value (SAVED_OFFSET_1 and SAVED_OFFSET_2), rather than being reset to a predetermined state.
- Sixth, the LFSR is loaded before the second and third post-authentication iterations with a “roll-over RAND” comprised of the contents of R00, R01, R14, and R15. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be loaded with SAVED_RAND.

The CMEA key octets drawn from iterations two and three are labeled:

- k0 = register[4] XOR register[8]; (iteration 2)
- k1 = register[5] XOR register[9]; (iteration 2)
- k2 = register[6] XOR register[10]; (iteration 2)
- k3 = register[7] XOR register[11]; (iteration 2)
- k4 = register[4] XOR register[8]; (iteration 3)
- k5 = register[5] XOR register[9]; (iteration 3)
- k6 = register[6] XOR register[10]; (iteration 3)
- k7 = register[7] XOR register[11]; (iteration 3)

2.5.1.2. Voice Privacy Mask Generation

VPM generation is a continuation of the CMEA key generation and should be performed at the same time under the same conditions as the CMEA key. CAVE is run for eleven iterations beyond those that produced the CMEA octets. Each iteration consists of four rounds. The CAVE registers R00 through R15 are not reset between iterations, but the LFSR is reloaded between iterations with the "rollover RAND" as described in §2.5.1.1.

Exhibit 2-15 CMEA Key and VPM Generation

```
/* Key_VPM_Generation has the same header as CAVE (see Exhibit 2-4) */

static void roll_LFSR(void)
{
   int i;

   LFSR_A = Register[0];
   LFSR_B = Register[1];
   LFSR_C = Register[14];
   LFSR_D = Register[15];

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      for (i = 0; i < 4; i++)
         LFSR[i] = SAVED_RAND[i];
   }
}

void Key_VPM_Generation(void)
{
   int i,j,r_ptr,offset_1,offset_2,vpm_ptr;

   /* iteration 1, first pass through CAVE */

   for (i = 0; i < 4; i++)
      LFSR[i] = SAVED_LFSR[i] ^ SSD_B[i] ^ SSD_B[i+4];

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      for (i = 0; i < 4; i++)
         LFSR[i] = SAVED_RAND[i];
   }

   for (i = 0; i < 8; i++)
      Register[i] = SSD_B[i];

   Register[8] = AAV;

   /* put SAVED_DATA into r9-r11 */

   for (i = 9; i < 12; i++)
      Register[i] = SAVED_DATA[i-9];
```

```
    /* put ESN into r12-r15 */

    for (i = 12; i < 16; i++)
       Register[i] = ESN[i-12];

    offset_1 = SAVED_OFFSET_1;
    offset_2 = SAVED_OFFSET_2;

    CAVE(8, &offset_1, &offset_2);

    /* iteration 2, generation of first CMEA key parameters */

    roll_LFSR();
    CAVE(4, &offset_1, &offset_2);
    for (i = 0; i < 4; i++)
       cmeakey[i] = Register[i+4] ^ Register[i+8];

    /* iteration 3, generation of second CMEA key parameters */

    roll_LFSR();
    CAVE(4, &offset_1, &offset_2);
    for (i = 4; i < 8; i++)
       cmeakey[i] = Register[i] ^ Register[i+4];

    /* iterations 4-13, generation of VPM */

    vpm_ptr = 0;
    for (i = 0; i < 10; i++)
    {
       roll_LFSR();
       CAVE(4, &offset_1, &offset_2);
       for (r_ptr = 0; r_ptr < 6; r_ptr++)
       {
          VPM[vpm_ptr] = Register[r_ptr+2] ^ Register[r_ptr+8];
          vpm_ptr++;
       }
    }

    /* iteration 14, generation of last VPM bits */

    roll_LFSR();
    CAVE(4, &offset_1, &offset_2);
    for (j = 0; j < 5; j++)
    {
       VPM[vpm_ptr] = Register[j+2] ^ Register[j+8];
       vpm_ptr++;
    }
}
```

Exhibit 2-16 Generation of CMEA Key and VPM

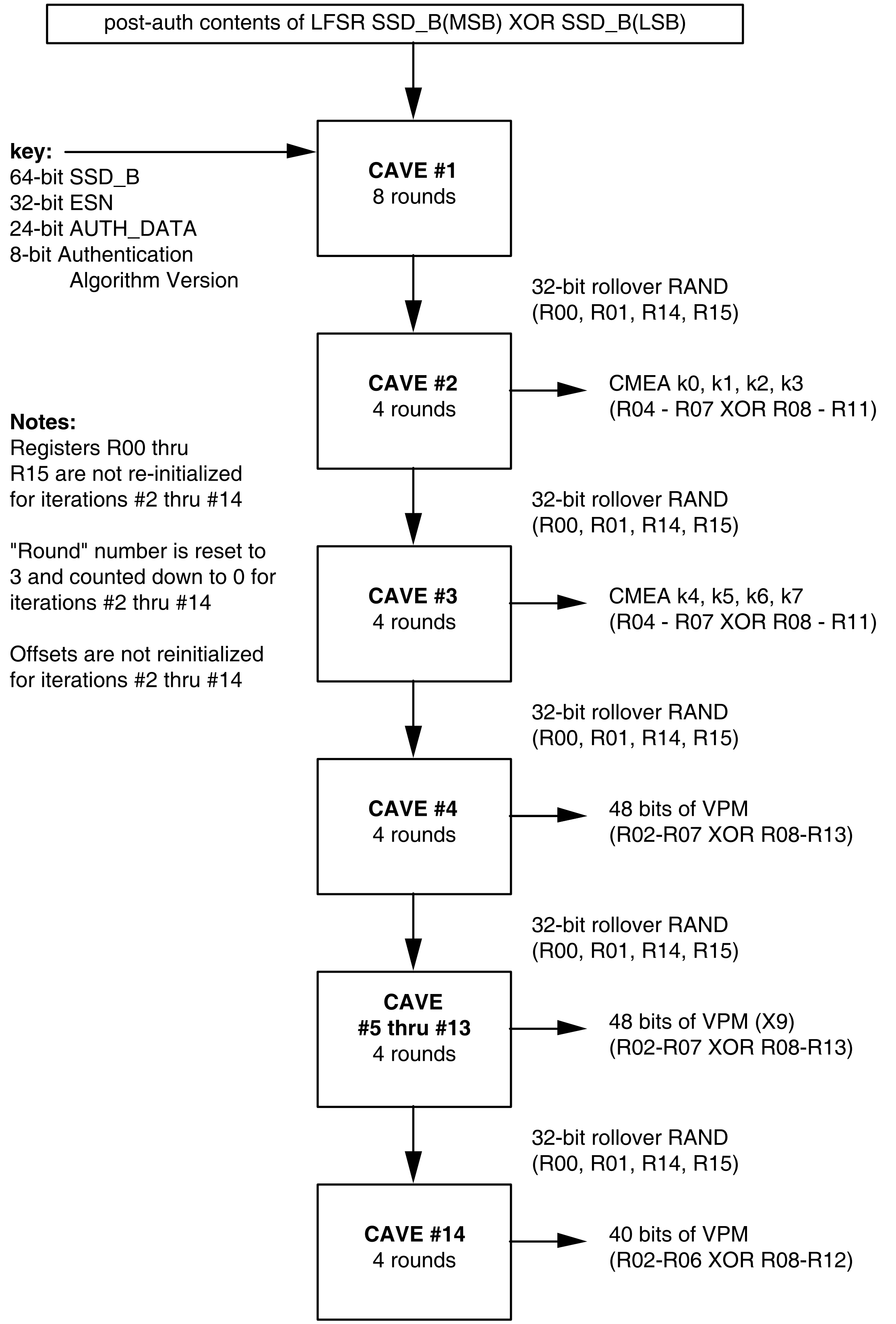
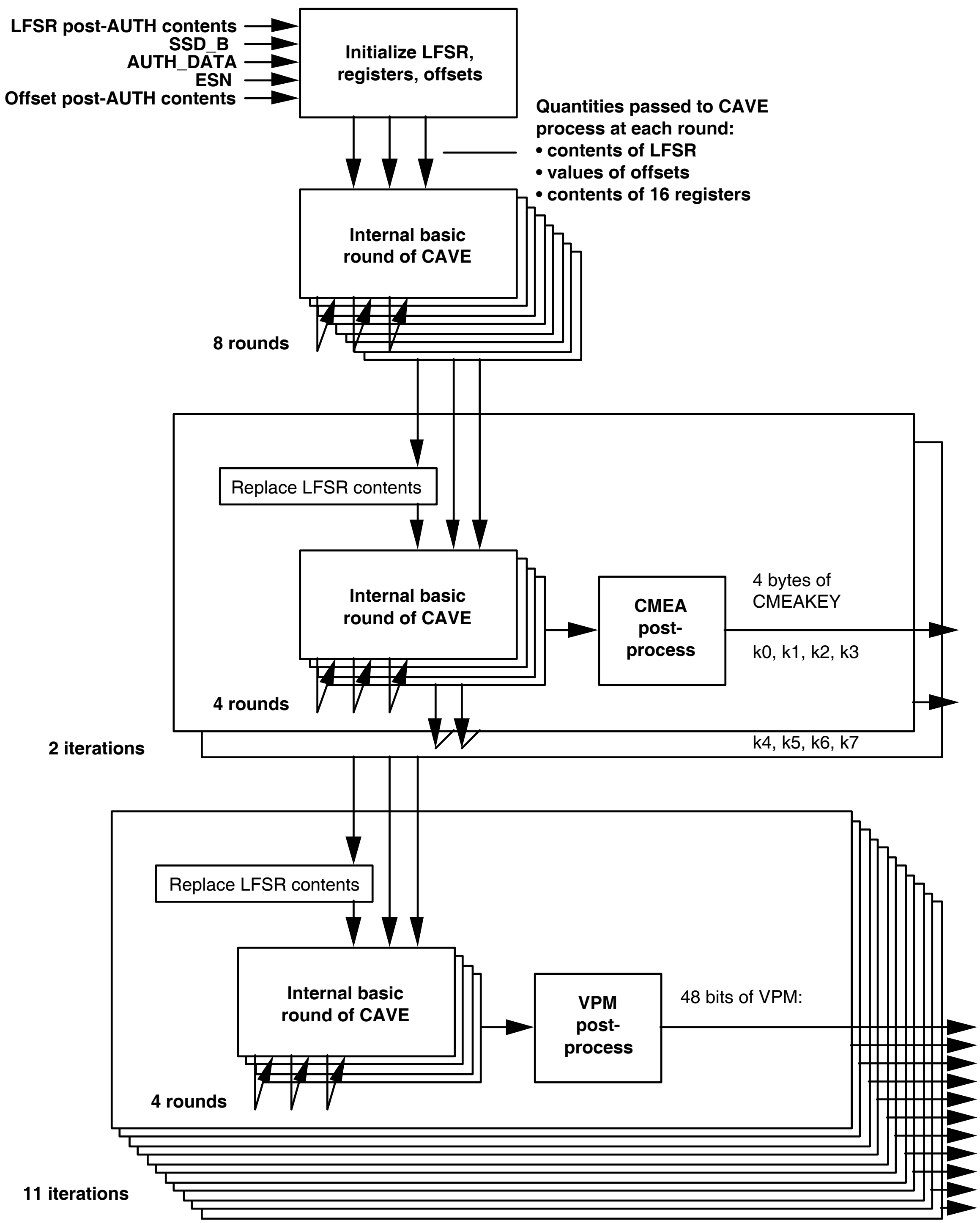


Exhibit 2-17 Detailed Generation of CMEA Key and VPM



2.5.2. ECMEA Secrets Generation for Financial Messages Procedure

Procedure name:

ECMEA_Secret_Generation

Inputs from calling process:

None.

Inputs from internal stored data:

CMEAKEY[0-7] 64 bits

Outputs to calling process:

None.

Outputs to internal stored data:

ECMEA_KEY [0-7] 64 bits
OFFSET_KEY[0-3] 32 bits

The CMEA Encryption Key and VPM Generation Procedure defined in §2.5.1 is used to generate a CMEA key on a per-call basis. ECMEA for financial messages requires additional secret values to be generated on a per-call basis. This procedure accomplishes this by running the CAVE algorithm initialized by the original CMEA key (64 bits). The generation procedure is depicted in Exhibit 2-18.

- First, the LFSR will be loaded with the 32 MSBs of the CMEA key. If these MSBs are all zero, then a constant, 0x31415926, will be loaded instead.
- Second, registers R00 through R07 will be loaded with the CMEA key.
- Third, registers R08 through R15 will be loaded with the one's-complement of the CMEA key.
- Fourth, the offset table pointers will be reset to all zeros.
- Fifth, the LFSR is loaded before each of the second through fourth iterations with a "roll-over RAND" comprised of the contents of R00, R01, R14, and R15 at the end of the previous iteration. If the resulting bit pattern fills the LFSR with all zeros, then the LFSR will be loaded with the constant, 0x31415926.

The ECMEA key octets drawn from iterations two and three are labelled:

- ecmea_key[0] = register[4] XOR register[8]; (iteration 2)
- ecmea_key[1] = register[5] XOR register[9]; (iteration 2)
- ecmea_key [2] = register[6] XOR register[10]; (iteration 2)
- ecmea_key[3] = register[7] XOR register[11]; (iteration 2)

- ecmea_key[4] = register[4] XOR register[8]; (iteration 3)
- ecmea_key[5] = register[5] XOR register[9]; (iteration 3)
- ecmea_key[6] = register[6] XOR register[10]; (iteration 3)
- ecmea_key[7] = register[7] XOR register[11]; (iteration 3)

Note: if, during this process, any of the octets of ECMEA_KEY as defined above are zero, that octet is replaced by the next nonzero octet generated. Additional iterations are performed as necessary to generate eight nonzero octets for ECMEA_KEY.

The offset_key octets drawn from iteration 4 are labeled:

- offset_key[0] = register[4] XOR register[8]; (iteration 4)
- offset_key [1] = register[5] XOR register[9]; (iteration 4)
- offset_key [2] = register[6] XOR register[10]; (iteration 4)
- offset_key [3] = register[7] XOR register[11]; (iteration 4)

Exhibit 2-18 Generation of ECMEA Secrets

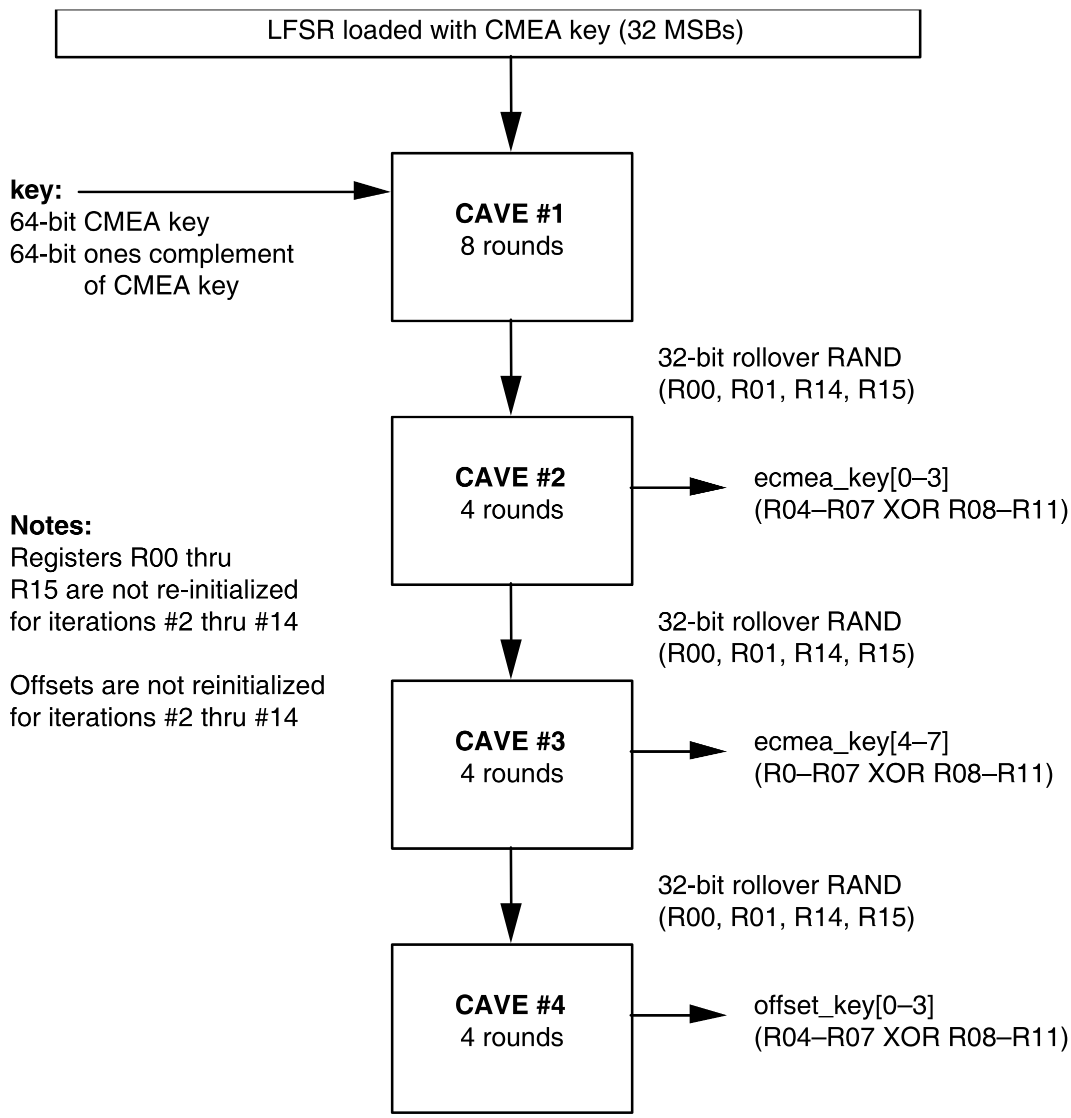


Exhibit 2-19 ECMEA Secret Generation

```
/* ECMEA_Secret_Generation has the same header as ECMEA (see Exhibit 2-
30) */

static void roll_LFSR_2(void)
{
   LFSR_A = Register[0];
   LFSR_B = Register[1];
   LFSR_C = Register[14];
   LFSR_D = Register[15];

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      LFSR_A = 0x31;
      LFSR_B = 0x41;
      LFSR_C = 0x59;
      LFSR_D = 0x26;
   }
}

void ECMEA_Secret_Generation(void)
{
   int i,j,offset_1,offset_2;

   /* iteration 1, first pass through CAVE */

   for (i = 0; i < 4; i++)
      LFSR[i] = cmeakey[i+4];

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      LFSR_A = 0x31;
      LFSR_B = 0x41;
      LFSR_C = 0x59;
      LFSR_D = 0x26;
   }
   for (i = 0; i < 8; i++)
      Register[i] = cmeakey[i];

   for (i = 8; i < 16; i++)
      Register[i] = ~cmeakey[i-8];

   offset_1 = 0x0;
   offset_2 = 0x0;

   CAVE(8, &offset_1, &offset_2);
```

```
    /* Iterations 2 and 3, generation of ECMEA_KEY */

    i = 0; j = 4;
    while (i < 8)
    {
       /* see if new key material needs to be generated */
       if( j == 4 )
       {
          j = 0;
          roll_LFSR_2();
          CAVE(4, &offset_1, &offset_2);
       }

       ecmea_key[i] = Register[j+4] ^ Register[j+8];
       j++;

       /* advance to next octet of ECMEA_KEY if not zero; otherwise
          generate another value */

       if (ecmea_key[i] != 0)
          i++;
    }

    /* iteration 4, generation of ECMEA offset keys */

    roll_LFSR_2();
    CAVE(4, &offset_1, &offset_2);
    for (i = 0; i < 4; i++)
       offset_key[i] = Register[i+4] ^ Register[i+8];
}
```

2.5.3. Non-Financial Seed Key Generation Procedure

Procedure name:
Non-Financial_Seed_Key_Generation
Inputs from calling process:
None.
Inputs from internal stored data:
CMEAKEY[0-7] 64 bits
Outputs to calling process:
None.
Outputs to internal stored data:
SEED_NF_KEY[0-4] 40 bits

The CMEA Encryption Key and VPM Generation Procedure defined in §2.5.1 is used to generate a CMEA key on a per-call basis. A non-financial seed key is required before generating the ECMEA secrets for non-financial messages. This procedure accomplishes this by running the CAVE algorithm initialized by the original CMEA key (64 bits). The generation procedure is depicted in Exhibit 2-20.

- First, the LFSR will be loaded with the 32 LSBs of the CMEA key. If these MSBs are all zero, then a constant, 0x31415926, will be loaded instead.
- Second, registers R00 through R07 will be loaded with the one's-complement of the CMEA key.
- Third, registers R08 through R15 will be loaded with the CMEA key.
- Fourth, the offset table pointers will be reset to all zeros.
- Fifth, the LFSR is loaded before the second iteration with a "roll-over RAND" comprised of the contents of R00, R01, R14, and R15 at the end of the previous iteration. If the resulting bit pattern fills the LFSR with all zeros, then the LFSR will be loaded with the constant, 0x31415926.

The non-financial seed key octets drawn from iteration two are labeled:

- seed_nf_key[0] = register[2] XOR register[8]; (iteration 2)
- seed_nf_key[1] = register[3] XOR register[9]; (iteration 2)
- seed_nf_key[2] = register[4] XOR register[10]; (iteration 2)
- seed_nf_key[3] = register[5] XOR register[11]; (iteration 2)
- seed_nf_key [4] = register[6] XOR register[12]; (iteration 2)

Exhibit 2-20 Generation of Non-Financial Seed Key

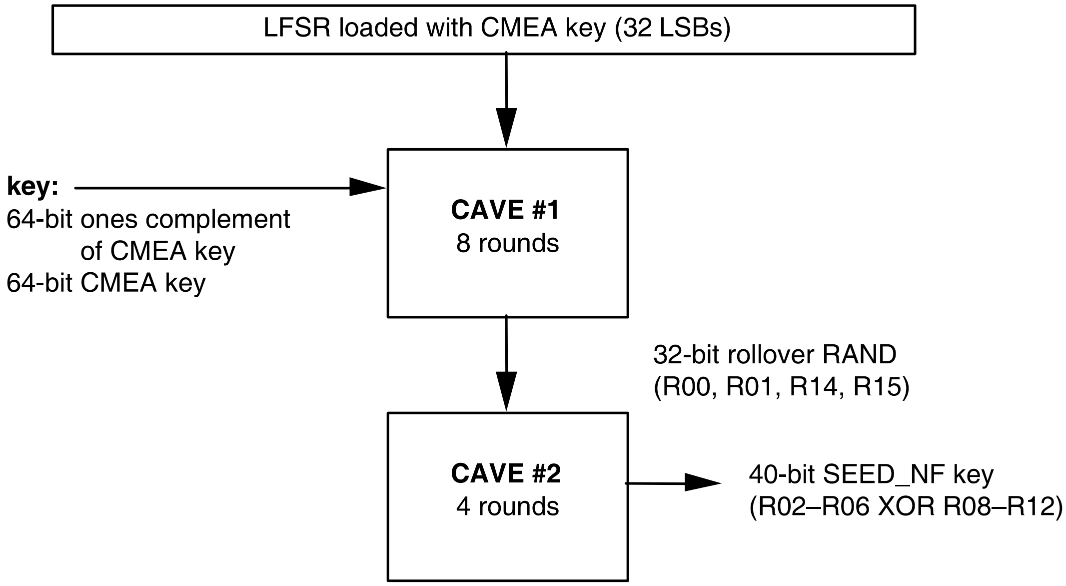


Exhibit 2-21 Non-Financial Seed Key Generation

```
/* Non_Financial_Seed_Key_Generation has the same header as ECMEA (see
Exhibit 2-30) */

void Non_Financial_Seed_Key_Generation(void)
{
   int i,offset_1,offset_2;

   /* iteration 1, first pass through CAVE */

   for (i = 0; i < 4; i++)
      LFSR[i] = cmeakey[i];

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      LFSR_A = 0x31;
      LFSR_B = 0x41;
      LFSR_C = 0x59;
      LFSR_D = 0x26;
   }
   for (i = 0; i < 8; i++)
      Register[i] = ~cmeakey[i];

   for (i = 8; i < 16; i++)
      Register[i] = cmeakey[i-8];

   offset_1 = 0x0;
   offset_2 = 0x0;

   CAVE(8, &offset_1, &offset_2);

   /* iteration 2, generation of seed_nf_key */

   roll_LFSR_2();  /* defined in Exhibit 2-19 */
   CAVE(4, &offset_1, &offset_2);
   for (i = 0; i < 5; i++)
      seed_nf_key[i] = Register[i+2] ^ Register[i+8];
}
```

2.5.4. ECMEA Secrets Generation for Non-Financial Messages Procedure

Procedure name:	
Non-Financial_Secret_Generation	
Inputs from calling process:	
None.	
Inputs from internal stored data:	
SEED_NF_KEY[0-4]	40 bits
Outputs to calling process:	
None.	
Outputs to internal stored data:	
ECMEA_NF_KEY[0-7]	64 bits
OFFSET_NF_KEY[0-3]	32 bits

The Non-Financial Seed Key Generation Procedure defined in §2.5.3 is used to generate a seed key on a per-call basis. ECMEA for non-financial messages requires additional secret values to be generated on a per-call basis. This procedure accomplishes this by running the CAVE algorithm initialized by the original seed key (40 bits). The generation procedure is depicted in Exhibit 2-22.

- First, the LFSR will be loaded with the 32 MSBs of the SEED_NF key. If these MSBs are all zero, then a constant, 0x31415926, will be loaded instead.
- Second, registers R00 through R04 will be loaded with the 40-bit SEED_NF key.
- Third, registers R05 through R07 will be loaded with zeros.
- Fourth, registers R08 through R12 will be loaded with the one's-complement of the 40-bit SEED_NF key.
- Fifth, registers R13 through R15 will be loaded with zeros.
- Sixth, the offset table pointers will be reset to all zeros.
- Seventh, the LFSR is loaded before each of the second through seventh iterations with a "roll-over RAND" comprised of the contents of R00, R01, R14, and R15 at the end of the previous iteration. If the resulting bit pattern fills the LFSR with all zeros, then the LFSR will be loaded with the constant, 0x31415926.

The ECMEA_NF key octets drawn from iterations two and three are labeled:

- ecmea_nf_key[0] = register[4] XOR register[8]; (iteration 2)
- ecmea_nf_key[1] = register[5] XOR register[9]; (iteration 2)
- ecmea_nf_key[2] = register[6] XOR register[10]; (iteration 2)
- ecmea_nf_key[3] = register[7] XOR register[11]; (iteration 2)
- ecmea_nf_key[4] = register[4] XOR register[8]; (iteration 3)
- ecmea_nf_key[5] = register[5] XOR register[9]; (iteration 3)
- ecmea_nf_key[6] = register[6] XOR register[10]; (iteration 3)
- ecmea_nf_key[7] = register[7] XOR register[11]; (iteration 3)

Note: if, during this process, any of the octets of ECMEA_NF_KEY as defined above are zero, that octet is replaced by the next nonzero octet generated. Additional iterations are performed as necessary to generate eight nonzero octets for ECMEA_NF_KEY.

The offset_key octets drawn from iteration 4 are labeled:

- offset_nf_key[0] = register[4] XOR register[8]; (iteration 4)
- offset_nf_key[1] = register[5] XOR register[9]; (iteration 4)
- offset_nf_key[2] = register[6] XOR register[10]; (iteration 4)
- offset_nf_key3] = register[7] XOR register[11]; (iteration 4)

Exhibit 2-22 Generation of Non-Financial Secrets

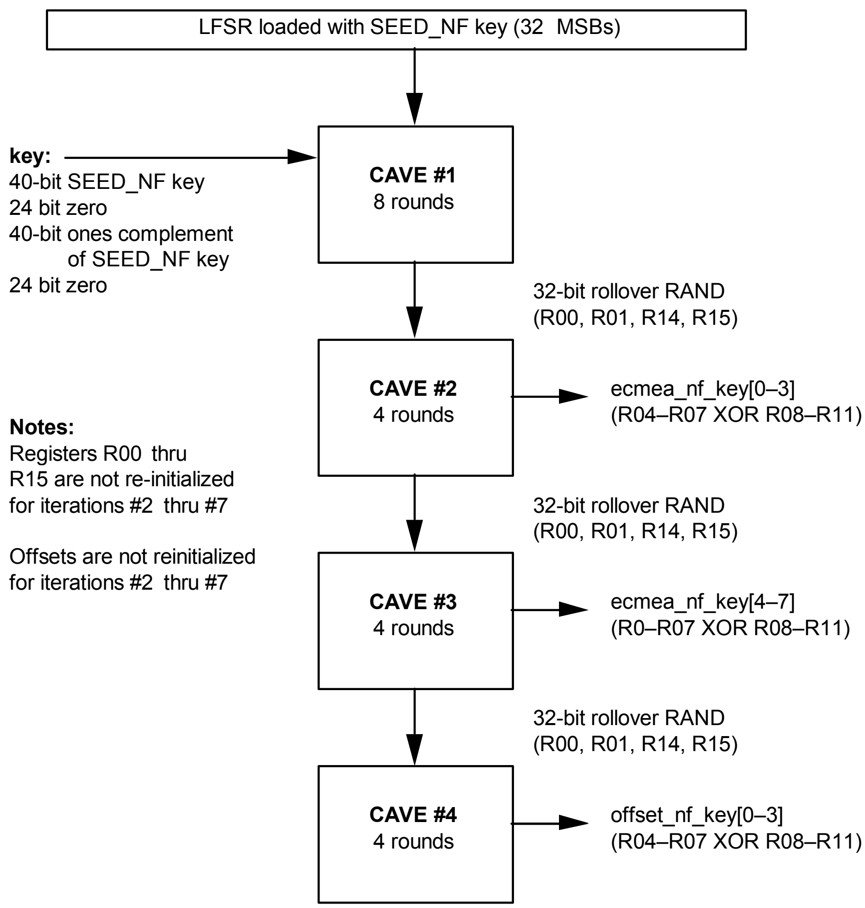


Exhibit 2-23 Non-Financial Secret Generation

```
/* Non_Financial_Secret_Generation has the same header as ECMEA (see
Exhibit 2-30) */

void Non_Financial_Secret_Generation(void)
{
   int i,j,offset_1,offset_2;

   /* iteration 1, first pass through CAVE */

   for (i = 0; i < 4; i++)
      LFSR[i] = seed_nf_key[i+1];

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      LFSR_A = 0x31;
      LFSR_B = 0x41;
      LFSR_C = 0x59;
      LFSR_D = 0x26;
   }
```

```
    for (i = 0; i < 5; i++)
       Register[i] = seed_nf_key[i];

    for (i = 5; i < 8; i++)
       Register[i] = 0;

    for (i = 8; i < 13; i++)
       Register[i] = ~seed_nf_key[i-8];

    for (i = 13; i < 16; i++)
       Register[i] = 0;

    offset_1 = 0x0;
    offset_2 = 0x0;

    CAVE(8, &offset_1, &offset_2);

    /* Iterations 2 and 3, generation of ECMEA_NF_KEY */

    i = 0; j = 4;
    while (i < 8)
    {
       /* see if new key material needs to be generated */
       if( j == 4 )
       {
          j = 0;
          roll_LFSR_2();
          CAVE(4, &offset_1, &offset_2);
       }

       ecmea_nf_key[i] = Register[j+4] ^ Register[j+8];
       j++;

       /* advance to next octet of ECMEA_NF_KEY if not zero; otherwise
          generate another value */

       if (ecmea_nf_key[i] != 0)
          i++;
    }

    /* iteration 4, generation of ECMEA offset_nf_key */

    roll_LFSR_2();  /* defined in Exhibit 2-19 */
    CAVE(4, &offset_1, &offset_2);
    for (i = 0; i < 4; i++)
       offset_nf_key[i] = Register[i+4] ^ Register[i+8];
}
```

2.6. Message Encryption/Decryption Procedures

2.6.1. CMEA Encryption/Decryption Procedure

Procedure name:	
Encrypt	
Inputs from calling process:	
msg_buf[n]	n*8 bits, n > 1
Inputs from internal stored data:	
CMEAKEY[0-7]	64 bits
Outputs to calling process:	
msg_buf[n]	n*8 bits
Outputs to internal stored data:	
None.	

This algorithm encrypts and decrypts messages that are of length n*8 bits, where n > 1. Decryption is performed in the same manner as encryption.

The message is first stored in an n-octet buffer called `msg_buf[]`, such that each octet is assigned to one “`msg_buf[]`” value. `msg_buf[]` will be encrypted by means of three operations before it is ready for transmission.

This process uses the CMEA eight-octet session key to produce enciphered messages via a unique CMEA algorithm. The process of CMEA key generation is described in §2.5.1.

The function tbox( ) is frequently used. This is defined as:

tbox(z) = C(((C(((C(((C((z XOR k0)+k1)+z)XOR k2)+k3)+z)XOR k4)+k5)+z)XOR k6)+k7)+z

where “+” denotes modulo 256 addition,

“XOR” is the XOR function,

“z” is the function argument,

k0,. .,k7 are defined above,

and C( ) is the outcome of a CAVE 8-bit table look-up, (Exhibit 2-5)

Exhibit 2-24 shows ANSI C code for an algorithmic procedure for tbox().

Exhibit 2-24 tbox

```
/* tbox has the same header as CAVE (see Exhibit 2-4) */

static unsigned char tbox(const unsigned char z)
{
   int k_index,i;
   unsigned char result;

   k_index = 0;
   result = z;

   for (i = 0; i < 4; i++)
   {
      result ^= cmeakey[k_index];
      result += cmeakey[k_index+1];
      result = z + CaveTable[result];
      k_index += 2;
   }

   return(result);
}
```

The CMEA algorithm is the message encryption process used for both the encryption and decryption of a message. Each message to which the CMEA algorithm is applied must be a multiple of 8 bits in length. The CMEA algorithm may be divided into three distinct manipulations. See Exhibit 2-25.

Exhibit 2-25 CMEA Algorithm

```
/* CMEA has the same header as CAVE (see Exhibit 2-4) */

void CMEA(unsigned char *msg_buf, const int octet_count)
{
   int msg_index,half;
   unsigned char k,z;

   /* first manipulation (inverse of third) */

   z = 0;
   for (msg_index = 0; msg_index < octet_count; msg_index++)
   {
      k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
      msg_buf[msg_index] += k;
      z += msg_buf[msg_index];
   }

   /* second manipulation (self-inverse) */

   half = octet_count/2;
   for (msg_index = 0; msg_index < half; msg_index++)
   {
      msg_buf[msg_index] ^=
         msg_buf[octet_count - 1 - msg_index] | 0x01;
   }

   /* third manipulation (inverse of first) */

   z = 0;
   for (msg_index = 0; msg_index < octet_count; msg_index++)
   {
      k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
      z += msg_buf[msg_index];
      msg_buf[msg_index] -= k;
   }
}
```

2.6.2. ECMEA Encryption/Decryption Procedure

Procedure name:

ECMEA

Inputs from calling process:

msg_buf[n]	n*8 bits, n > 1
Sync[0-1]	16 bits
Decrypt	1 bit
Data_type	1 bit

Inputs from internal stored data:

ECMEA_KEY[0-7]	64 bits
offset_key[0-3]	32 bits

Outputs to calling process:

msg_buf[n]	n*8 bits

Outputs to internal stored data:

None.

This algorithm encrypts and decrypts messages that are of length n*8 bits, where n > 1.

The message is first stored in an n-octet buffer called `msg_buf[]`, such that each octet is assigned to one "`msg_buf[]`" value. The input variable `sync` should have a unique value for each message that is encrypted. The same value of `sync` is used again for decryption.

This process uses the ECMEA eight-octet session key to produce enciphered messages via an enhanced CMEA algorithm. The process of ECMEA key generation is described in §2.5.2.

The `decrypt` variable shall be set to 0 for encryption, and to 1 for decryption.

The `data_type` variable shall be set to 0 for financial messages, and to 1 for non-financial messages.

ECMEA encryption of financial messages uses ECMEA key and offset_key.

ECMEA encryption of non-financial messages uses ECMEA_NF key and offset_nf_key.

The function etbox( ) is frequently used. This is defined as:

etbox(z,k) = I(I(I(I(I(I(I(I(I(I(I(I(I(z+k0)XOR k1)+k2)XOR k3)+k4)XOR k5)+k6)XOR k7)-k6)XOR k5)-k4)XOR k3)-k2)XOR k1)-k0

where "+" denotes modulo 256 addition,

"-" denotes modulo 256 subtraction,

"XOR" is the XOR function,

"z" is the function argument,

k0,. .,k7 are the eight octets of ECMEA key,

and I( ) is the outcome of the ibox 8-bit table look-up, (Exhibit 2-2).

Exhibit 2-26 shows ANSI C code for an algorithmic procedure for tbox().

Exhibit 2-26 Enhanced tbox

```
/* enhanced tbox has the same header as ECMEA (see Exhibit 2-30) */

unsigned char etbox(const unsigned char z,
const unsigned char *ecmea_key)
{

   unsigned char t;

   t = ibox[(z + ecmea_key[0]) & 0xff];
   t = ibox[t ^ ecmea_key[1]];
   t = ibox[(t + ecmea_key[2]) & 0xff];
   t = ibox[t ^ ecmea_key[3]];
   t = ibox[(t + ecmea_key[4]) & 0xff];
   t = ibox[t ^ ecmea_key[5]];
   t = ibox[(t + ecmea_key[6]) & 0xff];
   t = ibox[t ^ ecmea_key[7]];
   t = ibox[(t - ecmea_key[6]) & 0xff];
   t = ibox[t ^ ecmea_key[5]];
   t = ibox[(t - ecmea_key[4]) & 0xff];
   t = ibox[t ^ ecmea_key[3]];
   t = ibox[(t - ecmea_key[2]) & 0xff];
   t = ibox[t ^ ecmea_key[1]];
   t = (t - ecmea_key[0]) & 0xff;

   return t;
}
```

Enhanced CMEA is based on the basic CMEA construct for ease of implementation. It uses a modified CMEA which is passed keying information. The ECMEA encryption algorithm also uses a transformation and its inverse which are called before and after the CMEA block.

For each message encrypted or decrypted with ECMEA, offsets are calculated and then used to permute the tbox values used in CMEA and the transformations. ECMEA uses two offsets which are calculated as follows:

offset12 = ((offset_key[1.0]+1)*(CS[1,0] + 1) mod 65537)
XOR offset_key[3,2]

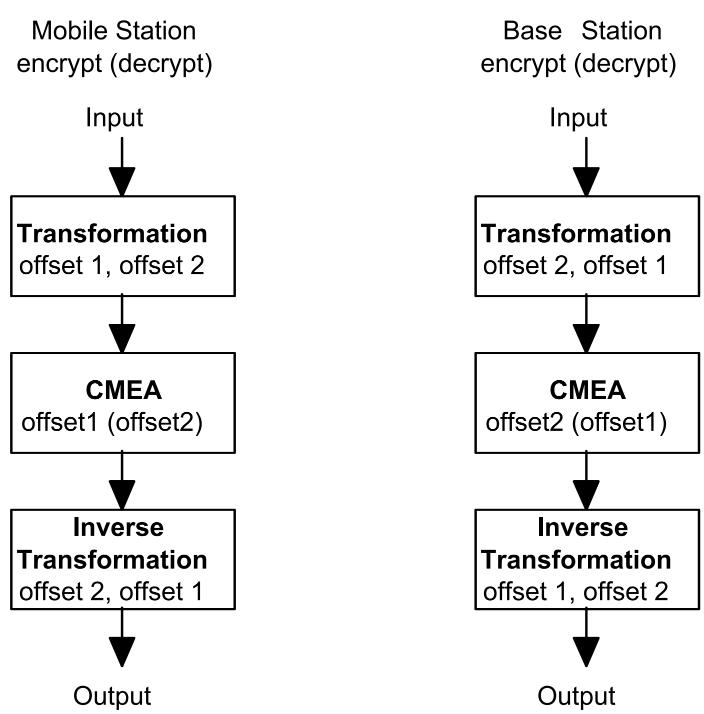
offset1 = (offset12 >> 8) mod 256

offset2 = offset1 XOR MAX(offset12 mod 256, 1)

where XOR stands for logical bitwise exclusive or, offset_key[i,j] means octets i and j of offset_key concatenated to form a 16-bit quantity with the second octet as the least significant, and CS denotes the 16 bits of cryptosynchronizing information for the message.

CMEA uses one offset while the transformation and its inverse use two offsets. The transformations are non-self-inverting and so the entire algorithm is non-self-inverting. For the inverse ECMEA algorithm, the order of passing offsets to the transformations is reversed. ECMEA is configured as shown in Exhibit 2-27.

Exhibit 2-27 ECMEA Structure



The mobile station and the base station implement the same basic algorithm with the only change being the offsets that are used in the transformation, in CMEA and in the inverse transformation. For example, offsets 1 and 2 (in that order) are used in the first transformation in the mobile station while the same offsets in the reverse order are passed to the first transformation in the base station. The inverse transformation always uses the offsets in the reverse order. The transformation and its inverse are given in Exhibit 2-28.

Exhibit 2-28 ECMEA Transformation and its Inverse

```
/* transform and inv_transform have the same header as ECMEA (see
Exhibit 2-30) */

void transform(unsigned char *msg_buf, const int octet_count,
               unsigned char offseta, const unsigned char offsetb,
               const unsigned char *key)
{
   unsigned char k, z;
   int msg_index;

   for (msg_index = 0; msg_index < octet_count; msg_index++)
   {
      /* offseta rotation and involutary lookup of present octet */

      if (msg_index > 0)
         offseta = (offseta >> 1) | (offseta << 7);
      msg_buf[msg_index] = offsetb ^
         etbox((unsigned char)(msg_buf[msg_index] ^ offseta), key);

      /* bit-trade between present octet and the one below */

      if (msg_index > 0)
      {
         k = msg_buf[msg_index - 1] ^ msg_buf[msg_index];
         k &= etbox((unsigned char)(k ^ offseta), key);
         msg_buf[msg_index - 1] ^= k;
         msg_buf[msg_index] ^= k;
      }

      /* random octet permutation */
      /* exchange previous octet with a random one below it */

      if (msg_index > 1)
      {
         k = etbox((unsigned char)(msg_buf[msg_index] ^ offseta),
            key);
         k = ((msg_index) * k) >> 8;
         z = msg_buf[k];
         msg_buf[k] = msg_buf[msg_index - 1];
         msg_buf[msg_index - 1] = z;
      }
   }

   /* final octet permutation */
   /* exchange last octet with a random one below it */

   k = etbox((unsigned char)(0x37 ^ offseta), key);
   k = ((msg_index) * k) >> 8;
   z = msg_buf[k];
   msg_buf[k] = msg_buf[msg_index - 1];
   msg_buf[msg_index - 1] = z;
```

```
    /* final involution and XORing */

    k = etbox(msg_buf[0], key);
    for (msg_index = 1; msg_index < octet_count; msg_index++)
    {
       msg_buf[msg_index] = etbox(msg_buf[msg_index], key);
       k ^= msg_buf[msg_index];
    }

    msg_buf[0] = k;
    for (msg_index = 1; msg_index < octet_count; msg_index++)
       msg_buf[msg_index] ^= k ;

}

/* Inverse Transformation */

void inv_transform(unsigned char *msg_buf, const int octet_count,
                   unsigned char offseta, const unsigned char offsetb,
                   const unsigned char *key)
{
   unsigned char k, z;
   int msg_index;

   /* initial offseta rotation */

   k = (octet_count - 1) & 0x07;
   offseta = (offseta >> k) | (offseta << (8 - k));

   /* inverse of final involution and XORing */

   for (msg_index = 1; msg_index < octet_count; msg_index++)
      msg_buf[msg_index] ^= msg_buf[0];

   for (msg_index = 1; msg_index < octet_count; msg_index++)
   {
      msg_buf[0] ^= msg_buf[msg_index];
      msg_buf[msg_index] = etbox(msg_buf[msg_index], key);
   }
   msg_buf[0] = etbox(msg_buf[0], key);

   /* initial octet permutation */
   /* exchange last octet with a random one below it */

   k = etbox((unsigned char)(0x37 ^ offseta), key);
   k = ((octet_count) * k) >> 8;
   z = msg_buf[k];
   msg_buf[k] = msg_buf[octet_count - 1];
   msg_buf[octet_count - 1] = z;
```

```
    for (msg_index = octet_count - 1; msg_index >= 0; msg_index--)
    {
       /* random octet permutation */
       /* exchange previous octet with a random one below it */

       if (msg_index > 1)
       {
          k = etbox((unsigned char)(msg_buf[msg_index] ^ offseta),
             key);
          k = ((msg_index) * k) >> 8;
          z = msg_buf[k];
          msg_buf[k] = msg_buf[msg_index - 1];
          msg_buf[msg_index - 1] = z;
       }

       /* bit-trade between present octet and the one below */

       if (msg_index > 0)
       {
          k = msg_buf[msg_index - 1] ^ msg_buf[msg_index];
          k &= etbox((unsigned char)(k ^ offseta), key);
          msg_buf[msg_index - 1] ^= k;
          msg_buf[msg_index] ^= k;
       }

       /* involutary lookup of present octet and offset rotation */

       msg_buf[msg_index] = offseta ^
          etbox((unsigned char)(msg_buf[msg_index] ^ offsetb), key);
       offseta = (offseta << 1) | (offseta >> 7);
    }
}
```

Exhibit 2-30 gives the ECMEA algorithm for the mobile station. Each message to which ECMEA is applied must be a multiple of 8 bits in length.

The base station algorithm is the same as the mobile station algorithm except for the two calls to the transformations and the offset used for CMEA. C code for the base station procedure is identical to that in Exhibit 2-30, except the first transformation call is changed to

```
transform(msg_buf, octet_count,
          offset2, offset1, key);
```

the offsets used for CMEA are reversed (i.e., the decryption and encryption offsets are the opposite of those used by the mobile station) and the final inverse transformation call is changed to

```
inv_transform(msg_buf, octet_count,
              offset1, offset2, key);
```

Exhibit 2-29 ECMEA Algorithm Header

```
void ECMEA_Secret_Generation(void);

void Non_Financial_Seed_Key_Generation(void);

void Non_Financial_Secret_Generation(void);

void ECMEA(unsigned char *msg_buf,
           const int octet_count,
           const unsigned char sync[2],
           const unsigned int decrypt,
           const unsigned int data_type);

#ifndef ECMEA_SOURCE_FILE
extern
unsigned char     ecmea_key[8];
extern
unsigned char     ecmea_nf_key[8];
extern
unsigned char     offset_key[4];
extern
unsigned char     offset_nf_key[4];
extern
unsigned char     seed_nf_key[5];
#endif
```

Exhibit 2-30 ECMEA Encryption/Decryption Algorithm for the Mobile Station

```
#define ECMEA_SOURCE_FILE
#include "cavei.h"  /* see Exhibit 2-3 */
#include "ecmea.h"  /* see Exhibit 2-29 */

#define MOBILE 1 /* set to 0 for base station algorithm */

void ECMEA(unsigned char *msg_buf, const int octet_count,
           const unsigned char sync[2],
           const unsigned int decrypt,
           const unsigned int data_type)
{
   unsigned char k, z, offset1, offset2, offsetc;
   unsigned long x1, x2, s;
   int msg_index;
   unsigned char *key, *offset;

   /* select key and offset key */
   if (data_type)
   {
      key = ecmea_nf_key;
      offset = offset_nf_key;
   }
   else
   {
      key = ecmea_key;
      offset = offset_key;
   }

   /* calculate offsets */
   /* offset12 =
      ((offset[1,0]+1)*(CS+1) mod 65537)^offset[3,2] mod 65536 */
   x1 = ((unsigned long)offset[1] << 8) + (unsigned long)offset[0];
   x2 = ((unsigned long)offset[3] << 8) + (unsigned long)offset[2];
   s = ((unsigned long)sync[1] << 8) + (unsigned long)sync[0];
   /* x1 = (((x1 + 1) * (s + 1)) % 65537) ^ x2; in two steps to
      prevent overflow */
   x1 = (x1 * (s + 1)) % 65537;
   x1 = ((x1 + s + 1) % 65537) ^ x2;
   offset1 = (unsigned char) (x1 >> 8);
   offset2 = (unsigned char) (offset1 ^ x1);
   if (offset2 == offset1)
      offset2 ^= 1;

#if MOBILE

   if (decrypt)
      offsetc = offset2;
   else
      offsetc = offset1;
```

```
#else

   if (decrypt)
      offsetc = offset1;
   else
      offsetc = offset2;

#endif

   /* initial transformation */
#if MOBILE
   transform(msg_buf, octet_count, offset1, offset2, key);
#else
   transform(msg_buf, octet_count, offset2, offset1, key);
#endif

   /* CMEA */
   /* first manipulation (inverse of third) */
   z = 0;
   for (msg_index = 0; msg_index < octet_count; msg_index++)
   {
      k = etbox((unsigned char)(z ^ offsetc), key);
      msg_buf[msg_index] += k;
      z = msg_buf[msg_index];
   }

   /* second manipulation (self-inverse) */
   for (msg_index = 0; msg_index < octet_count - 1; msg_index += 2)
      msg_buf[msg_index] ^= msg_buf[msg_index + 1];

   /* third manipulation (inverse of first) */
   z = 0;
   for (msg_index = 0; msg_index < octet_count; msg_index++)
   {
      k = etbox((unsigned char)(z ^ offsetc), key);
      z = msg_buf[msg_index];
      msg_buf[msg_index] -= k;
   }

   /* final inverse transformation */
#if MOBILE
   inv_transform(msg_buf, octet_count, offset2, offset1, key);
#else
   inv_transform(msg_buf, octet_count, offset1, offset2, key);
#endif
}
```

2.7. Wireless Residential Extension Procedures

This section describes detailed cryptographic procedures for wireless mobile telecommunications systems offering auxiliary services. These procedures are used to perform the security services of Authorization and Call Routing Equipment (ACRE), Personal Base (PB) and Mobile Station (MS) authentication. The ANSI C header file for Wireless Residential Extension Procedures is given in

Exhibit 2-31 WRE Header

```
void WIKEY_Generation(const unsigned char MANUFACT_KEY[16],
                      const unsigned char PBID[4]);

void WIKEY_Update(const unsigned char RANDWIKEY[7],
                  const unsigned char PBID[4]);

unsigned long WI_Auth_Signature(const unsigned char RAND_CHALLENGE[4],
                                const unsigned char PBID[4],
                                const unsigned char
ACRE_PHONE_NUMBER[3]);

unsigned long WRE_Auth_Signature(const unsigned char RAND_WRE[3],
                                 const unsigned char PBID[4],
                                 const unsigned char ESN[4]);

#ifndef WRE_SOURCE_FILE
extern
unsigned char     WIKEY[8];
extern
unsigned char     WIKEY_NEW[8];
extern
unsigned char     WRE_KEY[8];
#endif
```

2.7.1. WIKEY Generation

Procedure name:	
WIKEY_Generation	
Inputs from calling process:	
MANUFACT_KEY	122 bits
PBID	30 bits
Inputs from internal stored data:	
AAV	8 bits
Outputs to calling process:	
None.	
Outputs to internal stored data:	
WIKEY	64 bits

This procedure is used to calculate the WIKEY value generated during the manufacturing process. This WIKEY value is stored in semi-permanent memory of the PB.

The initial loading of CAVE for calculation of WIKEY is given in Exhibit 2-32.

MANUFACT_KEY is a 122-bit value that is chosen by the manufacturer. This value is the same for all of the manufacturer's PBs. PB manufactures must provide this number to each ACRE manufacture so that the ACREs can calculate the correct WIKEY values. The 32 MSBs of MANUFACT_KEY must not be all zeroes. There must be at least 40 zeroes and 40 ones in MANUFACT_KEY.

Exhibit 2-32 CAVE Initial Loading for WIKEY Generation

CAVE Item	Source Identifier	Size (Bits)
LFSR	bits 121-90 (32 MSBs) of MANUFACT_KEY	32
Reg [0-7]	bits 89-26 of MANUFACT_KEY	64
Reg [8]	AAV	8
Reg [9-11]	bits 25-2 of MANUFACT_KEY	24
Reg [12] 2 MSBs	bits 1-0 (2 LSBs) of MANUFACT_KEY	2
Reg [12] 6 LSBs	6 MSBs of PBID	6
Reg [13-15]	24 LSBs of PBID	24

CAVE is run for eight rounds. The 64-bit result is WIKEY. Exhibit 2-33 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-34.

The 64-bit WIKEY result is obtained from the final value of CAVE registers R00 through R15. The first 8 CAVE registers are XORed with the last 8 CAVE registers to produce the value for WIKEY.

Exhibit 2-33 Generation of WIKEY

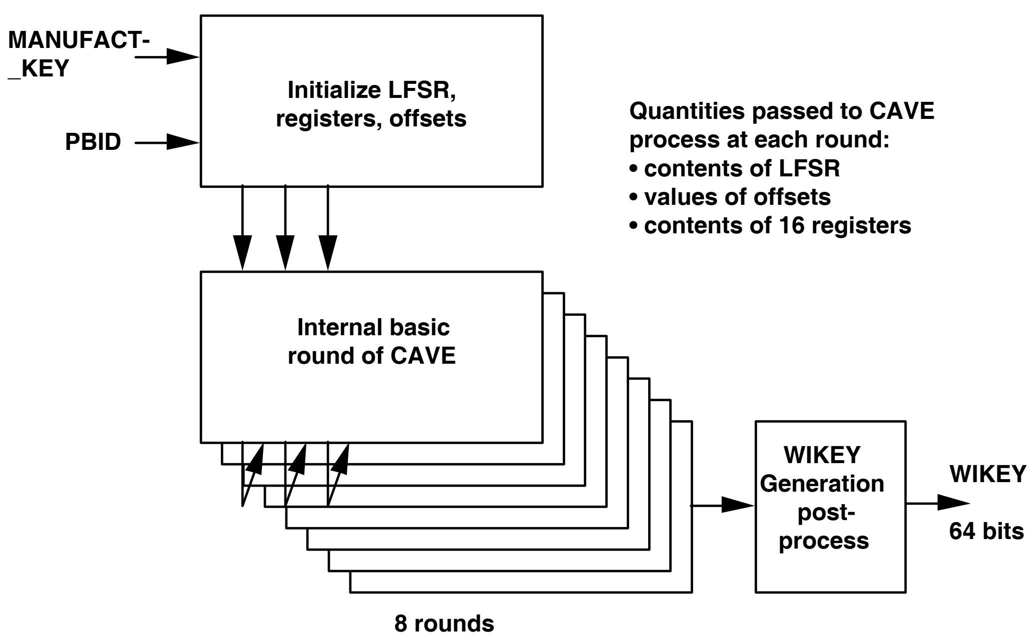


Exhibit 2-34 Code for WIKEY Generation

```
#define WRE_SOURCE_FILE
#include "cavei.h"  /* see Exhibit 2-3 */
#include "wre.h"    /* see Exhibit 2-31 */

unsigned char     WIKEY[8];
unsigned char     WIKEY_NEW[8];
unsigned char     WRE_KEY[8];

/* Note that MANUFACT_KEY is left justified and PBID is right justified.
   This means that the 6 LSBs of MANUFACT_KEY and the 2 MSBs of PBID
   must be set to 0 by the calling routine. */

void WIKEY_Generation(const unsigned char MANUFACT_KEY[16],
                      const unsigned char PBID[4])
{
   int i,offset_1,offset_2;

   for (i = 0; i < 4; i++)
      LFSR[i] = MANUFACT_KEY[i];
   for (i = 0; i < 8; i++)
      Register[i] = MANUFACT_KEY[i+4];
   Register[8] = AAV;
   for (i = 0; i < 4; i++)
      Register[i+9] = MANUFACT_KEY[i+12];
   Register[12] = Register[12] | PBID[0];
   for (i = 0; i < 3; i++)
      Register[i+13] = PBID[i+1];
   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);
   for (i = 0; i < 8; i++)
      WIKEY[i] = Register[i] ^ Register[i+8];
}
```

2.7.2. WIKEY Update Procedure

Procedure name:	
WIKEY_Update	
Inputs from calling process:	
RANDWIKEY	56 bits
PBID	30 bits
Inputs from internal stored data:	
WIKEY	64 bits
AAV	8 bits
Outputs to calling process:	
None.	
Outputs to internal stored data:	
WIKEY_NEW	64 bits

This procedure is used to calculate a new WIKEY value.

The initial loading of CAVE for calculation of WIKEY_NEW is given in Exhibit 2-35.

Exhibit 2-35 CAVE Initial Loading for WIKEY Update

CAVE Item	Source Identifier	Size (Bits)
LFSR	32 LSB of RANDWIKEY	32
Reg [0-7]	WIKEY	64
Reg [8]	AAV	8
Reg [9-11]	24 MSB of RANDWIKEY	24
Reg [12] 2 MSBs	00	2
Reg [12] 6 LSBs	6 MSBs of PBID	6
Reg [13-15]	24 LSBs of PBID	24

CAVE is run for eight rounds. The 64-bit result is WIKEY_NEW. Exhibit 2-36 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-37.

The LFSR will initially be loaded with the 32 LSBs of RANDWIKEY. This value will be XOR'd with the 32 most significant bits of WIKEY XOR'd with the 32 least significant bits of WIKEY, then reloaded into the LFSR. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be reloaded with the 32 LSBs of RANDWIKEY to prevent a trivial null result.

The 64-bit WIKEY_NEW result is obtained from the final value of CAVE registers R00 through R15. The first 8 CAVE registers are XORed with the last 8 CAVE registers to produce the value for WIKEY_NEW.

Exhibit 2-36 Generation of WIKEY_NEW

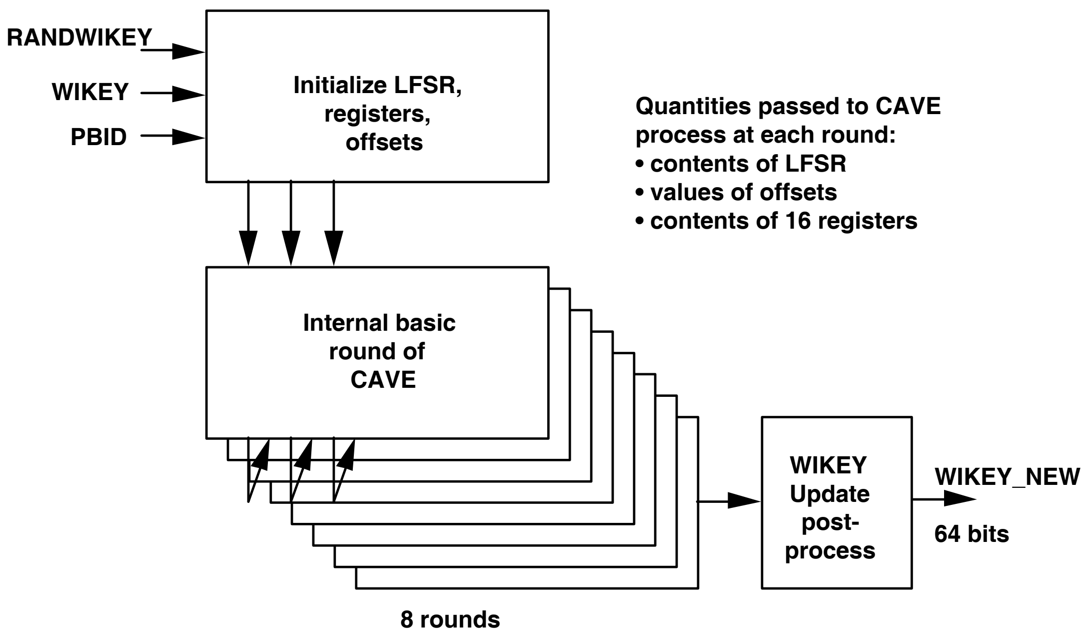


Exhibit 2-37 Code for WIKEY_NEW Generation

```
/* WIKEY_Update has the same header as WIKEY_Generation (see Exhibit 2-
34) */

/* Note that PBID is right justified.  This means that the 2 MSBs of PBID
   must be set to 0 by the calling routine. */

void WIKEY_Update(const unsigned char RANDWIKEY[7],
                  const unsigned char PBID[4])
{
   int i,offset_1,offset_2;

   for (i = 0; i < 4; i++)
      LFSR[i] = RANDWIKEY[i+3] ^ WIKEY[i] ^ WIKEY[i+4];
   if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
      for (i = 0; i < 4; i++)
         LFSR[i] = RANDWIKEY[i+3];
   for (i = 0; i < 8; i++)
      Register[i] = WIKEY[i];
   Register[8] = AAV;
   for (i = 0; i < 3; i++)
      Register[i+9] = RANDWIKEY[i];
   for (i = 0; i < 4; i++)
      Register[i+12] = PBID[i];
   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);
   for (i = 0; i < 8; i++)
      WIKEY_NEW[i] = Register[i] ^ Register[i+8];
}
```

2.7.3. Wireline Interface Authentication Signature Calculation Procedure

Procedure name:	
WI_Auth_Signature	
Inputs from calling process:	
RAND_CHALLENGE	32 bits
PBID	30 bits
ACRE_PHONE_NUMBER	24 bits
Inputs from internal stored data:	
WIKEY	64 bits
AAV	8 bits
Outputs to calling process:	
AUTH_SIGNATURE	18 bits
Outputs to internal stored data:	
None.	

This procedure is used to calculate 18-bit signatures used for verifying WIKEY values.

The initial loading of CAVE for calculation of wireline interface authentication signatures is given in Exhibit 2-38.

For authentication of an ACRE, RAND_CHALLENGE is received from the PB as RAND_ACRE.

For authentication of a PB, RAND_CHALLENGE is received from the ACRE as RAND_PB.

The ACRE_PHONE_NUMBER is 24 bits comprised of the least significant 24 bits of the ACRE's directory number (4 bits per digit). The digits 1 through 9 are represented by their 4-bit binary value (0001b - 1001b), while the digit 0 is represented by 1010b. If the phone number of the acre is less than 6 digits, then the digits are filled on the left with zeros until 6 full digits are reached. Example: If the acre's phone number is (987) 654-3210, ACRE_PHONE_NUMBER is 010101000011001000011010b. If the acre's phone number is 8695, ACRE_PHONE_NUMBER is 000000001000011010010101b.

Exhibit 2-38 CAVE Initial Loading for Wireline Interface Authentication Signatures

CAVE Item	Source Identifier	Size (Bits)
LFSR	RAND_CHALLENGE	32
Reg [0-7]	WIKEY	64
Reg [8]	AAV	8
Reg [9-11]	24 LSBs of ACRE_PHONE_NUMBER	24
Reg [12] 2 MSBs	00	2
Reg [12] 6 LSBs	6 MSBs of PBID	6
Reg [13-15]	24 LSBs of PBID	24

CAVE is run for eight rounds. The 18-bit result is AUTH_SIGNATURE. Exhibit 2-39 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-40.

The LFSR will initially be loaded with RAND_CHALLENGE. This value will be XOR'd with the 32 most significant bits of WIKEY XOR'd with the 32 least significant bits of WIKEY, then reloaded into the LFSR. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be reloaded with RAND_CHALLENGE to prevent a trivial null result.

The 18-bit authentication result AUTH_SIGNATURE is obtained from the final value of CAVE registers R00, R01, R02, R13, R14, and R15. The two most significant bits of AUTH_SIGNATURE are equal to the two least significant bits of R00 XOR R13. The next eight bits of AUTH_SIGNATURE are equal to R01 XOR R14. Finally, the least significant bits of AUTH_SIGNATURE are equal to R02 XOR R15.

Exhibit 2-39 Calculation of AUTH_SIGNATURE

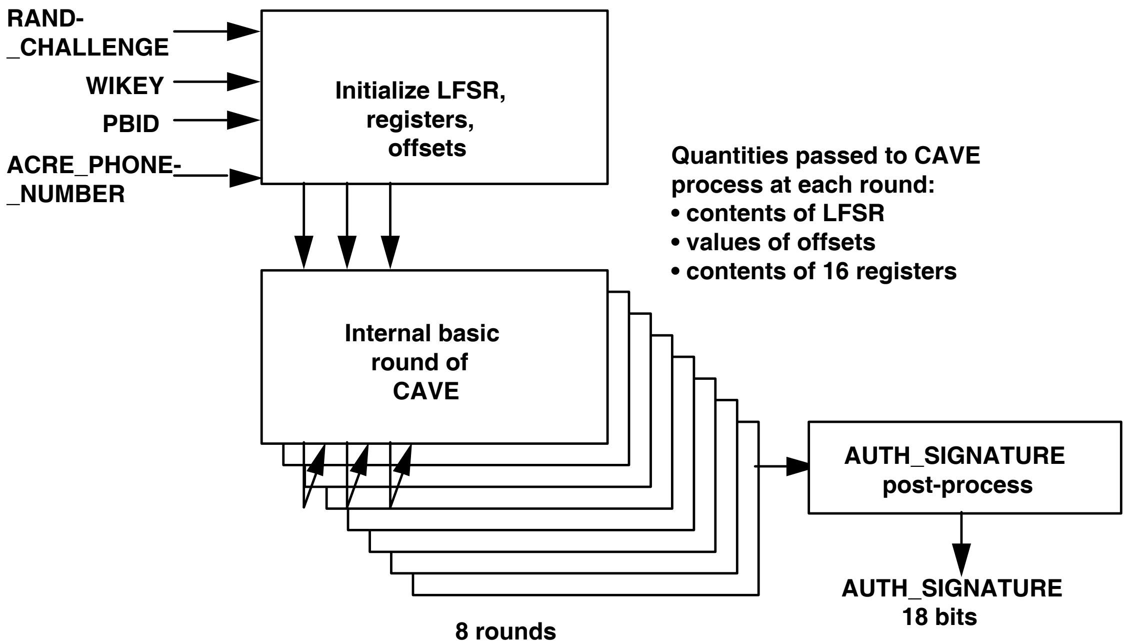


Exhibit 2-40 Code for calculation of AUTH_SIGNATURE

```
/* WI_Auth_Signature has the same header as WIKEY_Generation (see Exhibit 2-34) */

/* Note that PBID is right justified.  This means that the 2 MSBs of PBID
   must be set to 0 by the calling routine. */

unsigned long WI_Auth_Signature(const unsigned char RAND_CHALLENGE[4],
                                const unsigned char PBID[4],
                                const unsigned char ACRE_PHONE_NUMBER[3])
{
   int i,offset_1,offset_2;
   unsigned long AUTH_SIGNATURE;

   for (i = 0; i < 4; i++)
      LFSR[i] = RAND_CHALLENGE[i] ^ WIKEY[i] ^ WIKEY[i+4];
   if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
      for (i = 0; i < 4; i++)
         LFSR[i] = RAND_CHALLENGE[i];
   for (i = 0; i < 8; i++)
      Register[i] = WIKEY[i];
   Register[8] = AAV;
   for (i = 0; i < 3; i++)
      Register[i+9] = ACRE_PHONE_NUMBER[i];
   for (i = 0; i < 4; i++)
      Register[i+12] = PBID[i];
   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);
   AUTH_SIGNATURE =
      (  ((unsigned long)(Register[0] ^ Register[13]) << 16) +
         ((unsigned long)(Register[1] ^ Register[14]) << 8)  +
         ((unsigned long)(Register[2] ^ Register[15]) )        )
      & 0x3ffff;
   return(AUTH_SIGNATURE);
}
```

2.7.4. Wireless Residential Extension Authentication Signature Calculation Procedure

Procedure name:

WRE_Auth_Signature

Inputs from calling process:

RAND_WRE	19 bits
ESN	32 bits
PBID	30 bits

Inputs from internal stored data:

WRE_KEY	64 bits
AAV	8 bits

Outputs to calling process:

AUTH_SIGNATURE	18 bits

Outputs to internal stored data:

None.

This procedure is used to calculate 18-bit signatures used for verifying a mobile station.

The initial loading of CAVE for calculation of wireless residential extension authentication signatures is given in Exhibit 2-41.

Exhibit 2-41 CAVE Initial Loading for Residential Wireless Extension Authentication Signature

CAVE Item	Source Identifier	Size (Bits)
LFSR 19 MSBs	RAND_WRE	19
LFSR 13 LSBs	13 LSBs of PBID	13
Reg [0-7]	WRE_KEY	64
Reg [8]	AAV	8
Reg [9] 2 MSBs	00b	2
Reg [9] 6 LSBs	6 MSBs of PBID	6
Reg [10-11]	bits 23-8 of PBID	16
Reg [12-15]	ESN	32

CAVE is run for eight rounds. The 18-bit result is AUTH_SIGNATURE. Exhibit 2-42 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-43.

The 19 MSBs of LFSR will initially be loaded with RAND_WRE. The 13 LSBs of LFSR will initially be loaded with the 13 LSBs of PBID. LFSR will be XOR'd with the 32 most significant bits of WRE_KEY XOR'd with the 32 least significant bits of WRE_KEY, then reloaded into the LFSR. If the resulting bit pattern fills the LFSR with all zeroes, then the 19 MSBs of LFSR will be reloaded with RAND_WRE, and the 13 LSBs of LFSR will be reloaded with the 13 LSBs of PBID.

The 18-bit authentication result AUTH_SIGNATURE is obtained from the final value of CAVE registers R00, R01, R02, R13, R14, and R15. The two most significant bits of AUTH_SIGNATURE are equal to the two least significant bits of R00 XOR R13. The next eight bits of AUTH_SIGNATURE are equal to R01 XOR R14. Finally, the least significant bits of AUTH_SIGNATURE are equal to R02 XOR R15.

Exhibit 2-42 Calculation of AUTH_SIGNATURE

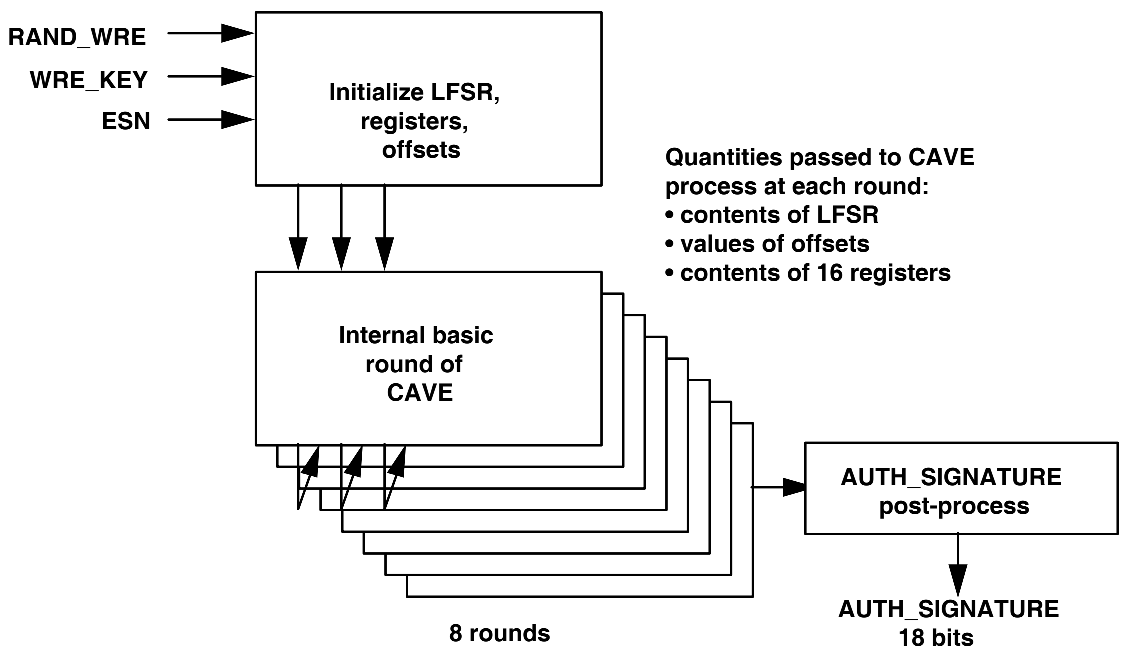


Exhibit 2-43 Code for calculation of AUTH_SIGNATURE

```
/* WRE_Auth_Signature has the same header as WIKEY_Generation (see
Exhibit 2-34) */

/* Note that RAND_WRE is left justified and PBID is right justified.
   This means that the 5 LSBs of RAND_WRE and the 2 MSBs of PBID
   must be set to 0 by the calling routine. */

unsigned long WRE_Auth_Signature(const unsigned char RAND_WRE[3],
                                 const unsigned char PBID[4],
                                 const unsigned char ESN[4])
{
   int i,offset_1,offset_2;
   unsigned long AUTH_SIGNATURE;

   for (i = 0; i < 3; i++)
      LFSR[i] = RAND_WRE[i];
   LFSR[2] = LFSR[2] | (PBID[2] & 0x1F);
   LFSR[3] = PBID[3];
   for (i = 0; i < 4; i++)
      LFSR[i] = LFSR[i] ^ WRE_KEY[i] ^ WRE_KEY[i+4];
   if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
   {
     for (i = 0; i < 3; i++)
        LFSR[i] = RAND_WRE[i];
     LFSR[2] = LFSR[2] | (PBID[2] & 0x1F);
     LFSR[3] = PBID[3];
   }
   for (i = 0; i < 8; i++)
      Register[i] = WRE_KEY[i];
   Register[8] = AAV;
   for (i = 0; i < 3; i++)
      Register[i+9] = PBID[i];
   for (i = 0; i < 4; i++)
      Register[i+12] = ESN[i];
   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);
   AUTH_SIGNATURE =
      (  ((unsigned long)(Register[0] ^ Register[13]) << 16) +
         ((unsigned long)(Register[1] ^ Register[14]) << 8)  +
         ((unsigned long)(Register[2] ^ Register[15]) )        )
      & 0x3ffff;
   return(AUTH_SIGNATURE);
}
```

2.8. Basic Wireless Data Encryption

Data encryption for wireless data services is provided by the ORYX algorithm (as named by its developers) which is described in the following.

- The DataKey Generation Procedure uses the A, B, and K registers to generate a DataKey. SSD_B provides the sole input to this procedure. If the data encryptor has access to SSD_B, DataKey may be generated locally. If not, DataKey is calculated elsewhere, then sent to the encryptor.

  In the network, this procedure executes at the initial serving system if SSD_B is shared or at the authentication center if SSD_B is not shared. DataKey may be precomputed when the mobile station registers.

- The LTable Generation Procedure uses the K register to generate a lookup table. RAND provides the sole input to this procedure. L is generated locally. In the network, this procedure executes at the initial serving system, and after intersystem handoff, it may execute at subsequent serving systems.

- The Data_Mask Procedure provides an encryption mask of the length requested by the calling process. It uses four inputs:

  1. DataKey from the DataKey Generation Procedure via the calling process;
  2. HOOK directly from the calling process;
  3. len directly from the calling process; and
  4. L as stored from the LTable Generation Procedure.

  The encryption mask is generated locally.

ORYX uses 3 Galois shift registers: A, B, and K. ORYX also uses a 256-octet look up table L.

Register K is a 32-bit Galois shift register, with feedback polynomial

$$k(z) = z^{32} + z^{28} + z^{19} + z^{18} + z^{16} + z^{14} + z^{11} + z^{10} + z^{9} + z^{6} + z^{5} + z + 1.$$

This is implemented by shifting the contents of K to the right and XORing the bit shifted out of the right-most position into the bit positions specified by the feedback polynomial.

Before stepping, a check is made to see if all of the bit positions in K are zero. If they are, K is initialized with the hex constant 0x31415926.

The feedback polynomial k(z) is primitive and has Peterson & Weldon octal code 42003247143.[1)]

Registers A and B are 32 bit Galois shift registers, shifting to the left: the leftmost bit is XORed into the bit positions specified by the feedback polynomial. Register A sometimes steps with feedback polynomial

$$a_1(z) = z^{32} + z^{26} + z^{23} + z^{22} + z^{16} + z^{12} + z^{11} + z^{10} + z^{8} + z^{7} + z^{5} + z^{4} + z^{2} + z + 1$$

and sometimes with feedback polynomial

$$a_2(z) = z^{32} + z^{27} + z^{26} + z^{25} + z^{24} + z^{23} + z^{22} + z^{17} + z^{13} + z^{11} + z^{10} + z^{9} + z^{8} + z^{7} + z^{2} + z + 1$$

The decision is based on the current high order bit of K. First K is stepped. If the (new) high order bit of K is set, register A steps according to polynomial $a_1(z)$; if the high order bit of K is clear, register A steps according to polynomial $a_2(z)$.

Register B steps once if the next-to-high order bit of K is clear, or twice if the next-to-high order bit of K is set, with feedback polynomial

$$\begin{aligned} b(z) &= (z + 1)\,(z^{31} + z^{20} + z^{15} + z^{5} + z^{4} + z^{3} + 1) \\ &= z^{32} + z^{31} + z^{21} + z^{20} + z^{16} + z^{15} + z^{6} + z^{3} + z + 1 \end{aligned}$$

This is also implemented with a left shift, XORing the leftmost bit into the bit positions specified by the feedback polynomial.

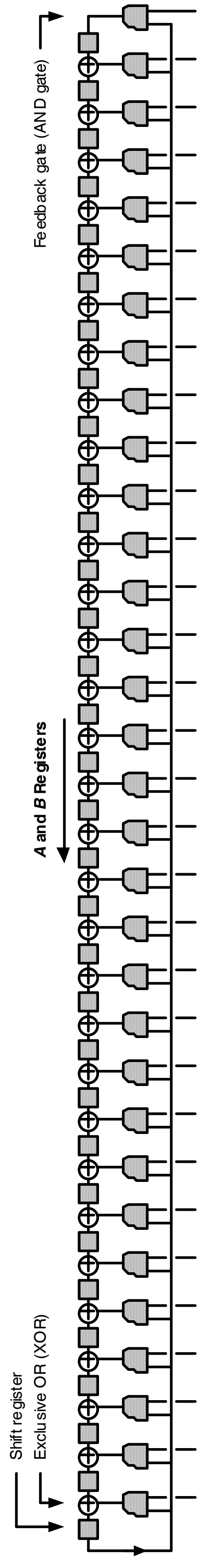
Polynomials $a_1(z)$, $a_2(z)$, and the degree 31 factor of polynomial b(z) are all primitive, with Peterson & Weldon octal codes 40460216667, 41760427607, and 20004100071, respectively.

Exhibit 2-44 illustrates the operation of the three Galois shift registers used in ORYX.

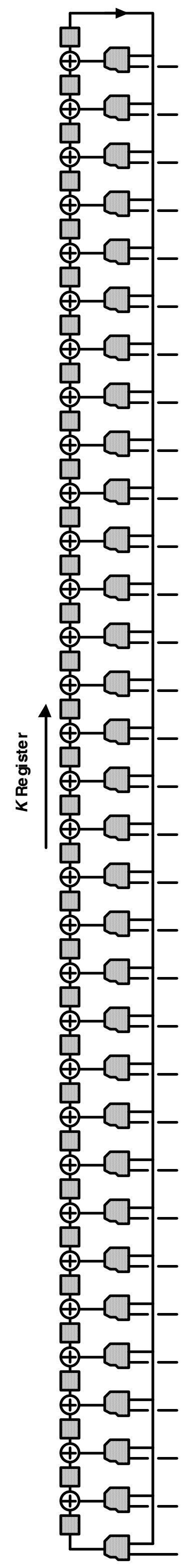
---

1) Since each shift register always has its output connected to its feedback gates, the most-significant bit is not required explicitly in the accompanying C code, hence the leading 4 (octal) is omitted from the representations of the polynomial within the C code.

Exhibit 2-44 Galois Shift Registers



Bit Nbr	A_1	Octal	A_2	Octal	B	Octal
0	1		1		1	
1	1	7	1	7	1	3
2	1		1		0	
3	0		0		1	
4	1	6	0	0	0	1
5	1		0		0	
6	0		0		1	
7	1	6	1	6	0	1
8	1		1		0	
9	0		1		0	
10	1	6	1	7	0	0
11	1		1		0	
12	1		0		0	
13	0	1	1	2	0	0
14	0		0		0	
15	0		0		1	
16	1	2	0	4	1	3
17	0		1		0	
18	0		0		0	
19	0	0	0	0	0	4
20	0		0		1	
21	0		0		1	
22	1	6	1	6	0	1
23	1		1		0	
24	0		1		0	
25	0	4	1	7	0	0
26	1		1		0	
27	0		1		0	
28	0	0	0	1	0	0
29	0		0		0	
30	0		0		0	
31	0	0	0	0	1	2



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

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

2.8.1. Data Encryption Key Generation Procedure

Procedure name:	
DataKey_Generation	
Inputs from calling process:	
None.	
Inputs from internal stored data:	
SSD_B	64 bits
Outputs to calling process:	
DataKey	32 bits
Outputs to internal stored data:	
None.	

This procedure generates DataKey, a key used by the Data_Mask procedure (see 2.8.3).

The calculation of DataKey depends only on SSD_B, therefore DataKey may be computed at the beginning of each call using the current value of SSD_B, or it may be computed and saved when SSD is updated. The value of DataKey shall not change during a call.

Here is how DataKey is formed from SSD_B, using ORYX as a hash function: First, register A is initialized with the first 32 bits of SSD_B, B is initialized with the remaining 32 bits of SSD_B and K is initialized with the XOR of A and B. Then K is stepped 256 times.

After the i-th step, for $0 \le i < 256$, the i-th entry, L[i], in the look up table is initialized with the most-significant octet of K. Then the following three-step procedure is repeated 32 times:

1. ORYX is stepped by calling the `keygen()` procedure, producing a key octet, which is temporarily stored in the variable `temp`. Register A is modified by shifting its contents to the left by 9 bits and adding the contents of `temp`.
2. ORYX is stepped, producing a key octet, which is temporarily stored in the variable `temp`. Register B is modified by shifting its contents to the left by 9 bits and adding the contents of `temp`.
3. ORYX is stepped, producing a key octet, which is temporarily stored in the variable `temp`. The value of the variable `temp` is used to modify K as described in Exhibit 2-45.

The XOR of the final values of K, A, and B is stored in DataKey.

Exhibit 2-45 describes the calculation in ANSI C.

Exhibit 2-45 Header for Basic Data Encryption

```
unsigned long DataKey_Generation(void);

void LTable_Generation(const unsigned char [] );

void Data_Mask(const unsigned long ,
               const unsigned long ,
               const int ,
               unsigned char []);

#ifndef ORYX_SOURCE_FILE
extern
unsigned char L[256];
extern
unsigned long DataKey;
#endif
```

Exhibit 2-46 DataKey Generation

```
#define ORYX_SOURCE_FILE
#include "cavei.h"  /* see Exhibit 2-3 */
#include "oryx.h"   /* see Exhibit 2-45 */

#define high(x) (unsigned char)(0xffU&(x>>24))    /* leftmost octet */
#define FA1 000460216667         /* Peterson & Weldon prim 32 */
#define FA2 001760427607         /* Peterson & Weldon prim 32 */
#define FB 020014300113          /* P&W prim 31 020004100071 times z+1 */
#define FK 030634530010          /* reverse of P&W prim 32 042003247143
*/

static
unsigned long K;                 /* 32-bit K  register */
static
unsigned long A, B;              /* 32-bit LFSRs */

unsigned char L[256];            /* look up table */
unsigned long DataKey;           /* data encryption key */

static
void kstep(void);
static
unsigned char keygen(void);
```

```
unsigned long DataKey_Generation(void)
{
   int i;
   unsigned long temp;

   A = 0;
   for(i=0; i<4; i++)
      A = (A<<8) + (unsigned long)SSD_B[i];
   B = 0;
   for(i=4; i<8; i++)
      B = (B<<8) + (unsigned long)SSD_B[i];

   K = A ^ B;
   for(i=0; i<256; i++)
   {
      kstep();
      L[i] = high(K);
   }
   for(i=0; i<32; i++)
   {
      temp = (unsigned long)keygen();
      A = (A<<9) + temp;
      temp = (unsigned long)keygen();
      B = (B<<9) + temp;
      temp = (unsigned long)keygen();
      K = (0xff00ffffU & K) + (temp << 16);
      K &= 0xffff00ffU + (temp<<8);
   }
   return ( (A ^ B ^ K) & 0xffffffff );
}

static
unsigned char keygen(void)
{
   unsigned char x;
   int  i, trips;

   kstep();
   /*
    * if high bit of K set, use A1 feedback
    * otherwise use A2 feedback
    */
   if((1UL<<31) & A)
   {
      A += A;
      if((1UL<<31) & K)
         A = A ^ FA1;
      else
         A = A ^ FA2;
   }
   else
      A += A;

```

```
    /*
     * if next-high bit of  K set, step B twice
     * otherwise once
     */
    if((1UL<<30) & K)
       trips = 2;
    else
       trips = 1;
    for(i=0; i<trips; i++)
    {
       if((1UL<<31) & B)
       {
          B += B;
          B = B ^ FB;
       }
       else
          B += B;
    }
    x = high(K) + L[high(A)] + L[high(B)];
    x &= 0xffU;                               /* use only 8 bits */
    return x;
}

/*
 * step the K register
 */
static
void kstep(void)
{
    if(K==0) K = 0x31415926;
    if(K&1)
    {
       K = (K>>1) ^ FK;
    }
    else
    {
       K = (K>>1);
    }
    K &= 0xffffffff;
}
```

2.8.2. L-Table Generation Procedure

Procedure name:	
LTable_Generation	
Inputs from calling process:	
RAND	32 bits
Inputs from internal stored data:	
None.	
Outputs to calling process:	
None.	
Outputs to internal stored data:	
L	256*8 bits

This procedure generates L, a table used in the Data_Mask procedure (see 2.8.3).

The LTable_Generation procedure shall be executed at the beginning of each call, and may be executed after intersystem handoff, using the value of RAND in effect at the start of the call. The value of L shall not change during a call.

L is initialized as follows:

K is set equal to RAND.

The i-th cell in the L table, L[i], is initialized with the value i, for 0 ≤ i < 256.

Then the K register is stepped 256 times. After the i-th step, for 0 ≤ i < 256, the value stored in the cell whose index is the most significant octet of K and the value stored in the i-th cell of the L table are interchanged.

Exhibit 2-47 describes the calculation in ANSI C.

Exhibit 2-47 LTable Generation

```
/* The header for LTable_Generation is the same as for
   DataKey_Generation (see Exhibit 2-46).*/

void LTable_Generation(const unsigned char RAND[4])
{
   int i,j;
   unsigned char tempc;

   K = 0;
   for(i=0; i<4; i++)
      K = (K<<8) + (unsigned long)RAND[i];
   for (i=0; i<256; i++)
      L[i] = (unsigned  char)i;

   /* use high octet of K to permute 0 through 255 */
   for (i=0; i< 256; i++)
   {
      kstep();
      j =  high(K);
      tempc = L[i];
      L[i] = L[j];
      L[j] = tempc;
   }
}
```

2.8.3. Data Encryption Mask Generation Procedure

Procedure name:	
Data_Mask	
Inputs from calling process:	
DataKey	32 bits
HOOK	32 bits
len	integer
Inputs from internal stored data:	
L	256*8 bits
Outputs to calling process:	
mask	len*8 bits
Outputs to internal stored data:	
None.	

This procedure generates an encryption mask of length len*8.

Implementations using data encryption shall comply with the following requirements. These requirements apply to all data encrypted during a call.

- The least-significant bits of HOOK shall change most frequently.
- A mask produced using a value of HOOK should be used to encrypt only one set of data.
- A mask produced using a value of HOOK shall not be used to encrypt data in more than one direction of transmission, nor shall it be used to encrypt data on more than one logical channel.

The DataKey and the look up table L must be computed prior to executing Data_Mask.

The key octets in a frame mask are produced by initializing the registers K, A, and B with values derived from DataKey and HOOK as follows.

1. K is set equal to the current value of HOOK. If K_1, K_2, K_3, and K_4 denote the four octets of K, the following assignments are made in turn:

   $K_1 = L[K_1 + K_4]$

$$K_2 = L[K_2 + K_4]$$

$$K_3 = L[K_3 + K_4]$$

$$K_4 = L[K_4]$$

where the additions $K_i + K_4$ are performed modulo 256.

2. K is stepped once, and A is set equal to DataKey XOR-ed with K.
3. K is stepped again, and B is set equal to DataKey XOR-ed with K.
4. K is stepped again, and K is set equal to DataKey XOR-ed with K.

With these values of A, B, and K, the ORYX key generator is stepped n times, and the resulting key octets are the n octets of the frame mask.

Exhibit 2-48 describes the calculation in ANSI C.

Exhibit 2-48 Data Encryption Mask Generation

```
/* Data_Mask has the same header as DataKey_Generation
   (see Exhibit 2-46) */

void Data_Mask(const unsigned long DataKey,
               const unsigned long HOOK,
               const int len,
               unsigned char mask[] )
{
   int i;

   K = (unsigned long)L[HOOK&0xff];
   K += ((unsigned long)L[((HOOK>>8)+HOOK)&0xff])<<8;
   K += ((unsigned long)L[((HOOK>>16)+HOOK)&0xff])<<16;
   K += ((unsigned long)L[((HOOK>>24)+HOOK)&0xff])<<24;
   kstep(); A = DataKey ^ K; /* kstep() is defined in Exhibit 2-45 */
   kstep(); B = DataKey ^ K;
   kstep(); K = DataKey ^ K;

   for(i=0; i<len; i++)
      mask[i] = keygen(); /* keygen() is defined in Exhibit 2-45 */
}
```

2.9. Enhanced Voice and Data Privacy

This section defines key generation and encryption procedures for the following TDMA content: voice, DTC and DCCH messages, and RLP data.

There are three key generation procedures: DTC key schedule generation, DCCH key schedule generation, and a procedure that each of these call termed the SCEMA Secrets Generation. The DCCH key schedule is based on a CMEA Key instance which is generated at Registration and remains for the life of the Registration. The DTC key is generated from the CMEA Key on a per call basis.

The encryption procedures contained herein are grouped into three levels, where the higher level procedures typically call procedures from a lower level. Level 1 has one member: the SCEMA encryption algorithm. Level 2 contains three procedures: a Long Block Encryptor for blocks of 48 bits, a Short Block Encryptor for blocks less than 48 bits, and a KSG used in voice and message encryption. Level 3 contains voice, message, and RLP data encryption procedures which interface directly to TIA/EIA-136-510.

CAVE algorithm code used in this section but defined external to it comprises CAVE header files, "cave.h" (see Exhibit 2-2) and "cavei.h" (see Exhibit 2-3), and CAVE source code (see Exhibit 2-4).

Throughout this section, the source code exhibits will be tagged with file names. While these names are arbitrary, they serve as a visual aid to the reader to flag a source code file and differentiate it from header files.

2.9.1. SCEMA Key Generation Code

This section describes the procedures used for generating secret key schedules for use in Enhanced Privacy and Encryption (EPE). Separate schedules are generated for the TDMA DTC (Digital Traffic Channel) and the DCCH (Digital Control Channel).

2.9.1.1. DTC Key Generation

Procedure name: DTC_Key_Generation Inputs from calling process: None. Inputs from internal stored data: CMEA Key (implicitly) Outputs to calling process: None. Outputs to internal stored data: dtcScheds[ ]   DTC key schedule structure

This procedure creates an array of DTC key schedule structures. Currently, the array contains a single element but allows the option to be extended in the future to accommodate multiple key schedules of different strengths. Each array element is a structure containing *scemaKey, *obox, *offKey, and neededLength The first three elements are pointers to keys (cryptovariables). The fourth, called neededLength, generally corresponds to the true entropy of the key, and is set in "scema.h" (see Exhibit 2-53).

dtcScheds[0] is generated from the CMEA Key. In TIA/EIA-136-510, this 45-octet schedule is termed DTCKey. These 45 octets comprise

dtcScemaKeyCK1	8 octets
dtcOboxCK1	32 octets
dtcOffKeyAuxCK1	4 octets
NeededLengthCK1	1 octet

The suffix “CK1” denotes CaveKey1.

Exhibit 2-49 SCEMA DTC Key Generation

```
/* SCEMA DTC Key Generation "dtcKeyGen.c" */

#include "scema.h" /* see Exhibit 2-53 */

/*
dtcScheds[0] accesses DTC CaveKey1 schedule.
*/

unsigned char  dtcScemaKeyCK1[ScemaKeyLengthCK1];
unsigned int   dtcOboxCK1[16];
unsigned int   dtcOffKeyAuxCK1[2];

keySched dtcScheds[] = {
   {dtcScemaKeyCK1, dtcOboxCK1, dtcOffKeyAuxCK1, NeededLengthCK1},
};


void DTC_Key_Generation(void)
{
   SCEMA_Secret_Generation(dtcScheds);
}

/*
Note: If a key schedule of a different strength is required in the
future,
the following can serve as an example:

/.
dtcScheds[0] will access DTC CaveKey1 schedule.
dtcScheds[1] will access DTC TBD Key2 schedule.
./

unsigned char  dtcScemaKeyCK1[ScemaKeyLengthCK1];
unsigned int   dtcOboxCK1[16];
unsigned int   dtcOffKeyAuxCK1[2];

unsigned char  dtcScemaKeyTbdK2[ScemaKeyLengthTbdK2];
unsigned int   dtcOboxTbdK2[16];
unsigned int   dtcOffKeyAuxTbdK2[2];

keySched dtcScheds[] = {
   {dtcScemaKeyCK1, dtcOboxCK1, dtcOffKeyAuxCK1, NeededLengthCK1},
   {dtcScemaKeyTbdK2, dtcOboxTbdK2, dtcOffKeyAuxTbdK2,
NeededLengthTbdK2}
};
*/
```

2.9.1.2. DCCH Key Generation

Procedure name:	
DCCH_Key_Generation	
Inputs from calling process:	
None.	
Inputs from internal stored data:	
CMEA Key (implicitly)	
Outputs to calling process:	
None.	
Outputs to internal stored data:	
dcchScheds[ ]	DCCH key schedule structure

This procedure creates an array of DCCH key schedule structures. Currently, the array contains a single element but allows the option to be extended in the future to accommodate multiple key schedules of different strengths. Each array element is a structure containing *scemaKey, *obox, *offKey, and neededLength The first three elements are pointers to keys (cryptovariables). The fourth, called neededLength, generally corresponds to the true entropy of the key, and is set in "scema.h" (see Exhibit 2-53).

dcchScheds[0] is generated from the CMEA Key. In TIA/EIA-136-510, this 45-octet schedule is termed DCCHKey. These 45 octets comprise

dcchScemaKeyCK1	8 octets
dcchOboxCK1	32 octets
dcchOffKeyAuxCK1	4 octets
NeededLengthCK1	1 octet

The suffix “CK1” denotes CaveKey1.

Exhibit 2-50 SCEMA DCCH Key Generation

```
/* SCEMA DCCH Key Generation "dcchKeyGen.c" */

#include "scema.h" /* see Exhibit 2-53 */

/*
dcchScheds[0] accesses DCCH CaveKey1 schedule.
*/

unsigned char  dcchScemaKeyCK1[ScemaKeyLengthCK1];
unsigned int   dcchOboxCK1[16];
unsigned int   dcchOffKeyAuxCK1[2];

keySched dcchScheds[] = {
   {dcchScemaKeyCK1, dcchOboxCK1, dcchOffKeyAuxCK1, NeededLengthCK1},
};


void DCCH_Key_Generation(void)
{
   SCEMA_Secret_Generation(dcchScheds);
}


/*
Note: If a key schedule of a different strength is required in the
future,
see the example in dtcKeyGen.c.
*/
```

2.9.1.3. SCEMA Secret Generation

Procedure name: SCEMA_Secret_Generation Inputs from calling process: None. Inputs from internal stored data: CMEAKEY[0-7] 64 bits Outputs to calling process: None. Outputs to internal stored data: SCEMA_KEY [0-7] 64 bits oboxSchedFin[0-15] 16 words (256 bits) offKeyAuxFin[0-1] 2 words (32 bits)

The CMEA Encryption Key and VPM Generation Procedure, defined in section 2.5.1, is used to generate a CMEA key on a per-call basis. SCEMA requires additional secret values to be generated on a per-call or per-registration basis. This procedure accomplishes this by running the CAVE algorithm initialized by the original CMEA key (64 bits).

- First, the LFSR will be loaded with the 32 MSBs of the CMEA key. If these MSBs are all zero, then a constant, 0x31415926, will be loaded instead.
- Second, registers R00 through R07 will be loaded with the CMEA key.
- Third, registers R08 through R15 will be loaded with the one's-complement of the CMEA key.
- Fourth, the offset table pointers will be reset to all zeros.
- Fifth, the LFSR is loaded before all of the remaining iterations with a "roll-over RAND" comprised of the contents of R00, R01, R14, and R15 at the end of the previous iteration. If the resulting bit pattern fills the LFSR with all zeros, then the LFSR will be loaded with the constant, 0x31415926.

The SCEMA key octets are drawn as follows (assuming that none equate to zero):

- scema_key[0] = register[4] XOR register[8]; (iteration 2)
- scema_key[1] = register[5] XOR register[9]; (iteration 2)
- scema_key [2] = register[6] XOR register[10]; (iteration 2)

- scema_key[3] = register[7] XOR register[11]; (iteration 2)
- scema_key[4] = register[4] XOR register[8]; (iteration 3)
- scema_key[5] = register[5] XOR register[9]; (iteration 3)
- scema_key[6] = register[6] XOR register[10]; (iteration 3)
- scema_key[7] = register[7] XOR register[11]; (iteration 3)

Note: If, during this process, any of the octets of SCEMA_KEY as defined above are zero, that octet is replaced by the next nonzero octet generated. Additional iterations are performed as necessary to generate eight nonzero octets for SCEMA_KEY. Thus the output of the CAVE iterations can be viewed as SCEMA_KEY candidates which are then screened to yield the actual SCEMA_KEY.

The Obox table comprises 16 16-bit words. Its values are drawn in a similar manner with the following exceptions: First, the LSB and MSB octets of the words are filled in succession. Second, a different screen is used here which rejects those Obox table candidates where the 4 LSBs of the sum of the table values and its index equals zero.

Finally, the two auxiliary offset keys are derived as follows via a single CAVE iteration:

- offKeyAuxFin[0] (lower octet) = register[4] XOR register[8]
- offKeyAuxFin[0] (upper octet) = register[5] XOR register[9]
- offKeyAuxFin[1] (lower octet) = register[6] XOR register[10]
- offKeyAuxFin[1] (upper octet) = register[7] XOR register[11]

Exhibit 2-51 Generation of SCEMA Secrets

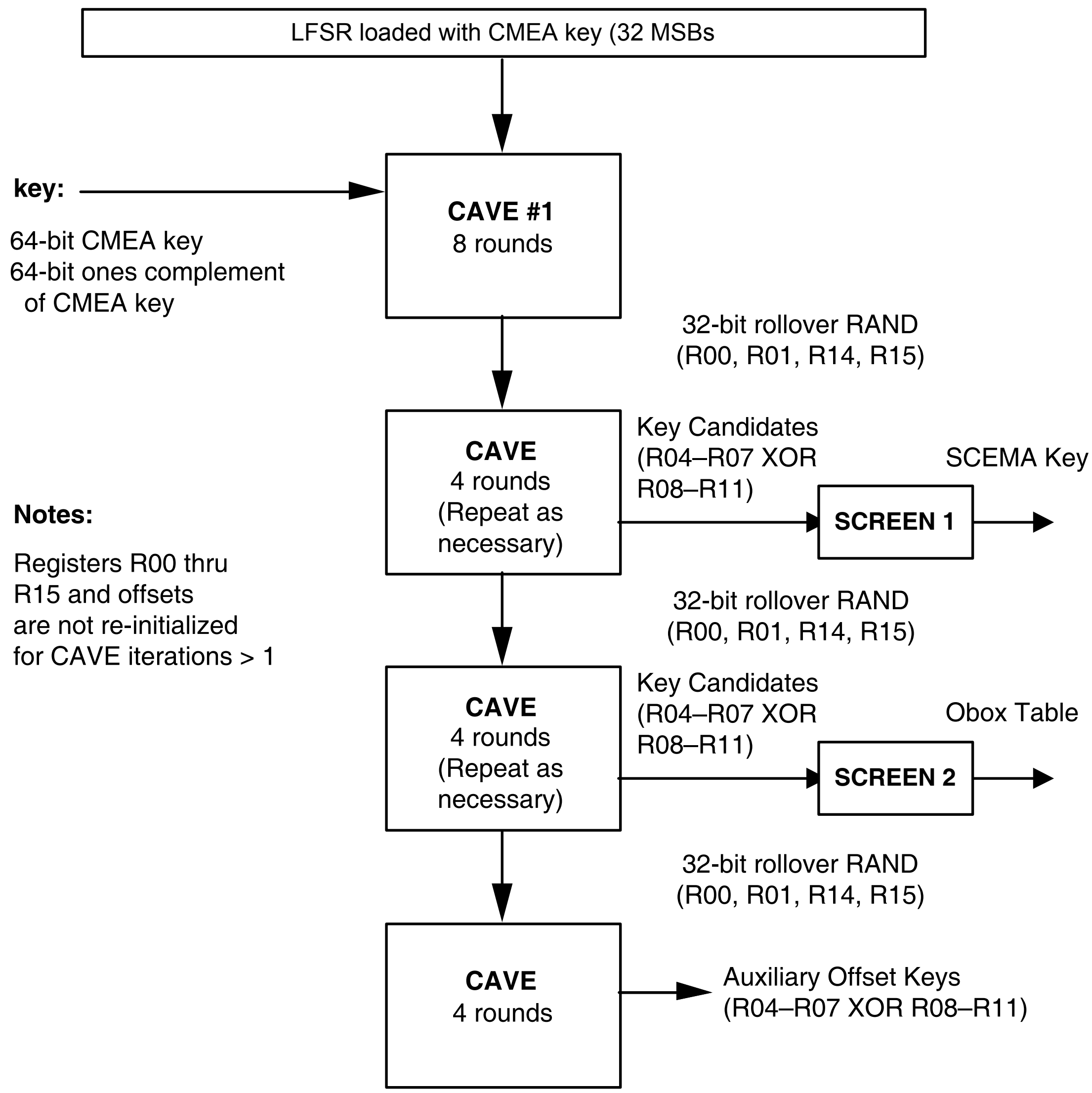


Exhibit 2-52 SCEMA Secret Generation

```
/* SCEMA Financial_Secret_Generation has the same header as SCEMA (see
Exhibit 2-53) */
/* SCEMA_Secret_Generation "scemaKeyGen.c */

#include "cavei.h" /* see Exhibit 2-3 */
#include "scema.h" /* see Exhibit 2-53 */


/* CAVE-related code */

void roll_LFSR_SCEMA(void)
{
   LFSR_A = Register[0];
   LFSR_B = Register[1];
   LFSR_C = Register[14];
   LFSR_D = Register[15];

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      LFSR_A = 0x31;
      LFSR_B = 0x41;
      LFSR_C = 0x59;
      LFSR_D = 0x26;
   }
}


void SCEMA_Secret_Generation(keySched *schedPtr)
{
   int i,j,offset_1,offset_2;

   /* iteration 1, first pass through CAVE */

   for (i = 0; i < 4; i++)
      LFSR[i] = cmeakey[i+4];

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      LFSR_A = 0x31;
      LFSR_B = 0x41;
      LFSR_C = 0x59;
      LFSR_D = 0x26;
   }
   for (i = 0; i < 8; i++)
      Register[i] = cmeakey[i];

   for (i = 8; i < 16; i++)
      Register[i] = ~cmeakey[i-8];

   offset_1 = 0x0;
   offset_2 = 0x0;

   CAVE(8, &offset_1, &offset_2);
```

```
/* Generation of SCEMA_KEY */

   i = 0; j = 4;
   while (i < ScemaKeyLengthCK1)
   {
      /* see if new key material needs to be generated */
      if( j == 4 )
      {
         j = 0;
         roll_LFSR_SCEMA();
         CAVE(4, &offset_1, &offset_2);
      }

      schedPtr->scemaKey[i] = Register[j+4] ^ Register[j+8];
      j++;

      /* advance to next octet of SCEMA_KEY if not zero; otherwise
         generate another value */

      if (schedPtr->scemaKey[i] != 0)
         i++;
   }

   /* Generation of SCEMA Obox Table */

   i = 0; j = 4;
   while (i < 16)
   {
      /* see if new key material needs to be generated */
      if( j == 4 )
      {
         j = 0;
         roll_LFSR_SCEMA();
         CAVE(4, &offset_1, &offset_2);
      }

      schedPtr->obox[i] =
         (int)(((Register[j+4] ^ Register[j+8]) & 0xFF) |
               ((Register[j+5] ^ Register[j+9]) << 8));
      j += 2;

      /* advance to next octet of Obox Table if not zero; otherwise
         generate another value */

      if (((schedPtr->obox[i] + i) & 0x0F) != 0)
         i++;
   }

   /* Generation of SCEMA auxiliary offset keys */

   roll_LFSR_SCEMA();
   CAVE(4, &offset_1, &offset_2);

   schedPtr->offKey[0] = (int)(((Register[4] ^ Register[8]) & 0xFF) |
               ((Register[5] ^ Register[9]) << 8));

   schedPtr->offKey[1] = (int)(((Register[6] ^ Register[10]) & 0xFF) |
               ((Register[7] ^ Register[11]) << 8));

}
```

2.9.2. SCEMA Header File

This section contains the header file used for all of the procedures in EPE. Some of the procedures additionally use CAVE header files, "cave.h" (see Exhibit 2-2) and "cavei.h" (see Exhibit 2-3).

Exhibit 2-53 SCEMA Header File

```
/* SCEMA Header File "scema.h" */

/* Key schedule architecture */

typedef struct _key_sched {
   unsigned char *scemaKey;
   unsigned int *obox;
   unsigned int *offKey;
   unsigned char neededLength;
} keySched;

keySched dtcScheds[];

keySched dcchScheds[];


/* SCEMA procedure/function declarations */

void DTC_Key_Generation(void);

void DCCH_Key_Generation(void);

void SCEMA_Secret_Generation(keySched *schedPtr);

void SCEMA(unsigned char *msg_buf,
           const int octet_count,
           const unsigned char *csync,
         const unsigned char id,
         const unsigned char idMask,
           const unsigned int decrypt,
           keySched *schedPtr);

void SCEMA_KSG(unsigned char *keystreamBuf,
            const unsigned int requestedStreamLen,
            const unsigned char *inputBuf,
            const unsigned int inputLen,
            const unsigned char contentType,
            keySched *schedPtr,
            const unsigned int direction);

void Long_Block_Encryptor(unsigned char *contentBuf,
                     const unsigned char contentType,
                     const unsigned int decrypt,
                     keySched *schedPtr,
                     const unsigned int direction);
```

```
void Short_Block_Encryptor(unsigned char *contentBuf,
               const unsigned int numBits,
               const unsigned char contentType,
               const unsigned char *entropy,
                     const unsigned int decrypt,
                     keySched *schedPtr,
                     const unsigned int direction);

void Enhanced_Message_Encryption(unsigned char *msgBuf,
                     const unsigned int numBits,
                     const unsigned int dcchDTC,
                     const unsigned char *rand,
                     const unsigned char msgType,
                     const unsigned int decrypt,
                     const unsigned int keyGenerator,
                     const unsigned int direction);

void Enhanced_Voice_Privacy(const unsigned int coderVer,
                     unsigned char *speechBuf1,
                     const unsigned int num1aBits,
                     unsigned char *speechBufRem,
                     const unsigned int numRemBits,
                     const unsigned int decrypt,
                     const unsigned int keyGenerator,
                     const unsigned int direction);

void Enhanced_Data_Mask(unsigned char *mask,
                  const unsigned long HOOK,
                  const unsigned int len,
                  const unsigned int keyGenerator);

/* Encryption mode of SCEMA */

#define ENCRYPTING   0
#define DECRYPTING   1

/* Blocksize of plaintext (or ciphertext) */
#define ThreeOctets  3
#define SixOctets 6
#define EightOctets  8

/* Long Block Definitions
Note: The LongBlockArchitecture identity segment forces a one into bit 2
of SCEMA's cryptosync top octet to differentiate the Long Block
Encryptor from all other KSG-type encryptors.
*/

#define LongBlkIdMask         0xFF
#define LongBlockArchitecture 0x04

/* KSG, RLP, and Short Block Definitions
Note: The LongBlockArchitecture identity segment forces a zero into bit
2 of SCEMA's cryptosync top octet.
*/
#define KSGIdMask    0x04
#define KSGArchitecture 0x00

/* Content Types */
#define VoiceContent 0x00
#define MessageContent  0x10
#define RlpContent      0x20
```

```
/* Direction */
#define ForwardChannel 1
#define ReverseChannel 0

/* Instances */
#define Instance1 0x01
#define Instance2 0x02
#define Instance3 0x03

/* DCCH/DTC */
#define DCCH 0
#define DTC 1

/* Message Types */
#define TestMsgType 0x1A
#define TestMsgType2 0x09

/* Used in SCEMA transforms */
#define OFFSETA ((unsigned char)(*offInt & 0xFF))
#define OFFSETB ((unsigned char)((*offInt >> 8) & 0xFF))

/* Miscellaneous */
#define MaxFrameOctetSize  35 /* 278 bits */
#define MaxMessageOctetSize   256 /* 2048 bits */
#define CAVEKey1        1
#define CoderVersionZero   0

#define MAX(A,B) ((A) > (B) ? (A) : (B))
#define MIN(A,B) ((A) < (B) ? (A) : (B))

unsigned int offsetInt[2];

/* Key length determination and individual key schedule architectures
Note: NeededLength must be <= length of scemaKey to prevent
stbox() overflow, and should be >= the key schedule entropy.
Also, it must be even.

If a key schedule of a different strength is required in the future,
replicate the below with "CK1" replaced by the appropriate designator.
*/
/*  CaveKey1 */
#define ScemaKeyLengthCK1 8
#define  NeededLengthCK1 8

#if NeededLengthCK1 > ScemaKeyLengthCK1
   #error NeededLengthCK1 too large
#endif

#if NeededLengthCK1 % 2
   #error NeededLengthCK1 must be an even number
#endif
```

2.9.3. SCEMA Encryption/Decryption Procedure (Level 1)

Procedure name:

SCEMA

Inputs from calling process:

msg_buf[n]	n*8 bits, n > 2
csync[0-1]	32
id	1 octet
idMask	1 octet
decrypt	1 bit
schedPtr	pointer to key schedule containing scemaKey, obox, offKey, and neededLength

Inputs from internal stored data:

None.

Outputs to calling process:

msg_buf[n]	n*8 bits

Outputs to internal stored data:

None.

This algorithm encrypts and decrypts messages that are of length n octets, where n > 2.

The message is first stored in an n-octet buffer called `msg_buf[]`, such that each octet is assigned to one "`msg_buf[]`" value. The input variable `csync` should have a unique value for each message that is encrypted, with the portion that varies quickly in its lower 16 bits. The same value of `csync` is used again for decryption.

The parameters id and idMask allow the internal copy of the top octet of cryptosync to be forced to a given value. idMask defines which bits are forced, and id defines the values of those bits. These inputs allow differentiation of scema instances. In particular, the following are differentiated: instances within a single procedure, and those with different content, direction or architecture. By doing this, a class of attacks is prevented that use recurring encryptor/decryptor outputs. One well-known member of this class are replay attacks.

This SCEMA procedure uses the SCEMA variable-length session key to produce enciphered messages via an enhanced CMEA algorithm. The process of SCMEA key generation is described in §2.9.1.

The `decrypt` variable shall be set to 0 for encryption, and to 1 for decryption.

SCEMA is given a pointer, schedPtr, to the desired key schedule structure. The structure contains the following elements: *scemaKey, *obox, *offKey, and neededLength The first three are pointers to keys (cryptovariables). The fourth, neededLength, generally corresponds to the true entropy of the key. A key generation mechanism may be implemented such that it outputs the scemaKey into a constant buffer size, independent of the true strength of the key. This parameter allows the stbox() function's iterations to track the true strength of the key, which in turn allows for faster operation with lower strength keys.

The function stbox( ) is frequently used in SCEMA. For example, in the case of an 8-octet SCEMA Key, stbox() is defined as:

stbox(z,k) = I(I(I(I(I(I(I(I(I(I(I(I(I(I(z+k0)XOR k1)+k2)XOR k3)+k4)XOR k5)+k6)XOR k7)-k6)XOR k5)-k4)XOR k3)-k2)XOR k1)-k0 3

where "+" denotes modulo 256 addition,

"-" denotes modulo 256 subtraction,

"XOR" is the XOR function,

"z" is the function argument,

k0,. .,k7 are the eight octets of SCEMA key,

and I( ) is the outcome of the ibox 8-bit table look-up (see Exhibit 2-54).

Exhibit 2-54 SCEMA with subtending functions stbox and SCEMA_transform

```
/* SCEMA source including transforms and stbox "scema.c" */

#include "cave.h" /* see Exhibit 2-2 */
#include "scema.h" /* see Exhibit 2-53 */

/* Stbox function
Note: The SCEMA Key Length must be an even number of octets.
The "-1" in the first "while" statement prevents overflow if
ScemaKeyLength is accidentally odd.
*/

unsigned char stbox(const unsigned char z,
               const unsigned char *scema_key,
               const unsigned char len)
{
   unsigned char t = z;
   int i = 0;

   while(i < len - 1)
   {
      t = ibox[(t + scema_key[i++]) & 0xff];
      t = ibox[t ^ scema_key[i++]];
   }

   --i;

   while(i > 1)
   {
      t = ibox[(t - scema_key[--i]) & 0xff];
      t = ibox[t ^ scema_key[--i]];
   }

   t = (t - scema_key[--i]) & 0xff;
   return t;
}
```

```
/* Transformation */

void SCEMA_transform(unsigned char *msg_buf, const int octet_count,
                unsigned int *offInt, const unsigned char *key,
                const unsigned int *obox, const unsigned char len)
{
   unsigned char k, z;
   int msg_index;

   for (msg_index = 0; msg_index < octet_count; msg_index++)
   {
   /* offset generator cycle and involutary lookup of present octet */

      offsetInt[0] += offsetInt[1] + obox[offsetInt[1] & 0x0F];
      offsetInt[1] ^=
         ((offsetInt[0] & 0xFFFF)>>4) + (offsetInt[0]<<4);

      msg_buf[msg_index] = OFFSETB ^
         stbox((unsigned char)(msg_buf[msg_index] ^ OFFSETA),
            key, len);

      /* bit-trade between present octet and the one below */

      if (msg_index > 0)
      {
         k = msg_buf[msg_index - 1] ^ msg_buf[msg_index];
         k &= stbox((unsigned char)(k ^ OFFSETA), key, len);
         msg_buf[msg_index - 1] ^= k;
         msg_buf[msg_index] ^= k;
      }

      /* random octet permutation */
      /* exchange previous octet with a random one below it */

      if (msg_index > 1)
      {
         k = stbox((unsigned char)(msg_buf[msg_index] ^ OFFSETA),
            key, len);
         k = ((msg_index) * k) >> 8;
         z = msg_buf[k];
         msg_buf[k] = msg_buf[msg_index - 1];
         msg_buf[msg_index - 1] = z;
      }
   }

   /* final octet permutation */
   /* exchange last octet with a random one below it */

   k = stbox((unsigned char)(0x37 ^ OFFSETA), key, len);
   k = ((msg_index) * k) >> 8;
   z = msg_buf[k];
   msg_buf[k] = msg_buf[msg_index - 1];
   msg_buf[msg_index - 1] = z;
```

```
    /* final involution and XORing */

    k = stbox(msg_buf[0], key, len);
    for (msg_index = 1; msg_index < octet_count; msg_index++)
    {
       msg_buf[msg_index] = stbox(msg_buf[msg_index], key, len);
       k ^= msg_buf[msg_index];
    }

    msg_buf[0] = k;
    for (msg_index = 1; msg_index < octet_count; msg_index++)
       msg_buf[msg_index] ^= k ;

}
```

```
/* Inverse Transformation */

void SCEMA_inv_transform(unsigned char *msg_buf,
                         const int octet_count,
                         unsigned int *offInt,
                         const unsigned char *key,
                         const unsigned int *obox,
                         const unsigned char len)
{
   unsigned char k, z;
   int msg_index;

   /* inverse of final involution and XORing */

   for (msg_index = 1; msg_index < octet_count; msg_index++)
      msg_buf[msg_index] ^= msg_buf[0];

   for (msg_index = 1; msg_index < octet_count; msg_index++)
   {
      msg_buf[0] ^= msg_buf[msg_index];
      msg_buf[msg_index] = stbox(msg_buf[msg_index], key, len);
   }
   msg_buf[0] = stbox(msg_buf[0], key, len);

   /* initial octet permutation */
   /* exchange last octet with a random one below it */

   k = stbox((unsigned char)(0x37 ^ OFFSETA), key, len);
   k = ((octet_count) * k) >> 8;
   z = msg_buf[k];
   msg_buf[k] = msg_buf[octet_count - 1];
   msg_buf[octet_count - 1] = z;

   for (msg_index = octet_count - 1; msg_index >= 0; msg_index--)
   {
      /* random octet permutation */
      /* exchange previous octet with a random one below it */

      if (msg_index > 1)
      {
         k = stbox((unsigned char)(msg_buf[msg_index] ^ OFFSETA),
            key, len);
         k = ((msg_index) * k) >> 8;
         z = msg_buf[k];
         msg_buf[k] = msg_buf[msg_index - 1];
         msg_buf[msg_index - 1] = z;
      }

      /* bit-trade between present octet and the one below */

      if (msg_index > 0)
      {
         k = msg_buf[msg_index - 1] ^ msg_buf[msg_index];
         k &= stbox((unsigned char)(k ^ OFFSETA), key, len);
         msg_buf[msg_index - 1] ^= k;
         msg_buf[msg_index] ^= k;
      }
```

```
      /* involutary lookup of present octet and offset generator cycle */

         msg_buf[msg_index] = OFFSETA ^
            stbox((unsigned char)(msg_buf[msg_index] ^ OFFSETB),
               key, len);

         offsetInt[1] ^=
            ((offsetInt[0] & 0xFFFF)>>4) + (offsetInt[0]<<4);
         offsetInt[0] -= offsetInt[1] + obox[offsetInt[1] & 0x0F];
      }
   }

   /* SCEMA Algorithm */

   void SCEMA(unsigned char *msg_buf,
               const int octet_count,
               const unsigned char *csync,
               const unsigned char id,
               const unsigned char idMask,
               const unsigned int decrypt,
               keySched *schedPtr)
   {
      unsigned char k, z, offsetc;
      int msg_index;
      unsigned char *key;
      unsigned int *obox, *offKeyAux;
      unsigned char len;
      unsigned char csync3id;
      unsigned int csyncInt[2];

      /* load key schedule element pointers */

      key = schedPtr->scemaKey;
      obox = schedPtr->obox;
      offKeyAux = schedPtr->offKey;
      len = schedPtr->neededLength;


      /* Offset Generator Initialization */

      csync3id = (csync[3] & ~idMask) | (id & idMask);

      csyncInt[0] = (unsigned int)((csync[1] << 8) | (csync[0] & 0xFF));
      csyncInt[1] = (unsigned int)((csync3id << 8) | (csync[2] & 0xFF));

      offsetInt[0] = csyncInt[1] + offKeyAux[0];
      offsetInt[1] = csyncInt[0] + offKeyAux[1];

      offsetInt[0] += obox[offsetInt[1] & 0x0F]
                   + obox[(offsetInt[1] >> 4) & 0x0F]
                   + obox[(offsetInt[1] >> 8) & 0x0F]
                   + obox[(offsetInt[1] >> 12) & 0x0F] ;

      offsetInt[1] += obox[offsetInt[0] & 0x0F]
                   + obox[(offsetInt[0] >> 4) & 0x0F]
                   + obox[(offsetInt[0] >> 8) & 0x0F]
                   + obox[(offsetInt[0] >> 12) & 0x0F] ;
```

```
    /* initial transformation */
    if (decrypt)
       SCEMA_transform(msg_buf, octet_count, offsetInt + 1, key,
          obox, len);
    else
       SCEMA_transform(msg_buf, octet_count, offsetInt, key, obox, len);


    /* CMEA */
    offsetc = (unsigned char)((offsetInt[0] + offsetInt[1]) & 0xFF);
    /* first manipulation (inverse of third) */
    z = 0;
    for (msg_index = 0; msg_index < octet_count; msg_index++)
    {
       k = stbox((unsigned char)(z ^ offsetc), key, len);
       msg_buf[msg_index] += k;
       z = msg_buf[msg_index];
    }

    /* second manipulation (self-inverse) */
    for (msg_index = 0; msg_index < octet_count - 1; msg_index += 2)
       msg_buf[msg_index] ^= msg_buf[msg_index + 1];

    /* third manipulation (inverse of first) */
    z = 0;
    for (msg_index = 0; msg_index < octet_count; msg_index++)
    {
       k = stbox((unsigned char)(z ^ offsetc), key, len);
       z = msg_buf[msg_index];
       msg_buf[msg_index] -= k;
    }

    /* final inverse transformation */
    if (decrypt)
       SCEMA_inv_transform(msg_buf, octet_count, offsetInt, key,
          obox, len);
    else
       SCEMA_inv_transform(msg_buf, octet_count, offsetInt + 1, key,
          obox, len);
}
```

2.9.4. Block and KSG Encryption Primitives (Level 2)

These Level 2 primitives call SCEMA at Level 1 and are called by the voice privacy and message encryption procedures at Level 3.

2.9.4.1. SCEMA KSG

Procedure name:

SCEMA_KSG

Inputs from calling process:

keystreamBuf[n]	n octets, 1 <= n <= 256
requestedStreamLen	1 - 256
inputBuf[n]	1 - 6 octets
inputLen	1 octet
contentType	1 octet defining voice or message
schedPtr	pointer to SCEMA key schedule
direction	1 bit

Inputs from internal stored data:

None.

Outputs to calling process:

keystreamBuf [n]	n octets, 1 <= n <= 256

Outputs to internal stored data:

None.

This encryption primitive generates a buffer of keystream of length requestedStreamLen based on the value of input buffer inputBuf[n] of length inputLen. It runs SCEMA in a KSG mode where the input is fed to both SCEMA's PT (plaintext) input and its CS (cryptosync) input.

The content type variable allows it to generate unique keystream depending upon whether it is used in voice privacy or message encryption. (This primitive is not called in RLP encryption (Enhanced Data Encryption).)

The pointer schedPtr is the SCEMA key schedule pointer described earlier in Section 2.9.

Direction indicates either the forward channel by 1, or the reverse channel by 0.

Exhibit 2-55 SCEMA KSG for Voice and Message Content

```
/* SCEMA KSG for Voice and Message Content "scemaKSG.c" */

#include "scema.h" /* see Exhibit 2-53 */

void SCEMA_KSG(unsigned char *keystreamBuf,
            const unsigned int requestedStreamLen,
            const unsigned char *inputBuf,
            const unsigned int inputLen,
            const unsigned char contentType,
            keySched *schedPtr,
            const unsigned int direction)
{
   unsigned int i;
   unsigned char csync[4];
   unsigned char id;
   unsigned int outputStreamLen;

   /* Generates a minimum of 6 octets of keystream */
   outputStreamLen = MAX(SixOctets,requestedStreamLen);

   /* Combine ID segments */
   id = (unsigned char)(direction << 7) | contentType;

   /* Repeat input across SCEMA's PT field */
   for (i = 0; i < outputStreamLen; i++)
      keystreamBuf[i] =  inputBuf[i % inputLen];

   /*
   Copy 4 least significant octets of PT to CS input.
   ID is XORed in to yield KSGs that are unique with
   respect to content and direction.
   */
   for (i = 0; i < 4; i++)
      csync[i] =  keystreamBuf[i] ^ id;

   SCEMA(keystreamBuf,outputStreamLen,csync,KSGArchitecture,KSGIdMask,
         ENCRYPTING,schedPtr);
}
```

2.9.4.2. Long Block Encryptor

Procedure name:	
Long_Block_Encryptor	
Inputs from calling process:	
contentBuf[n]	6 octets
contentType	1 octet defining voice or message
decrypt	1 bit
schedPtr	pointer to SCEMA key schedule
direction	1 bit
Inputs from internal stored data:	
None.	
Outputs to calling process:	
contentBuf [n]	6 octets
Outputs to internal stored data:	
None.	

This encryption primitive block encrypts or decrypts a 6-octet buffer by running three instances of SCEMA. The content type variable allows it to generate unique keystream depending upon whether it is used in voice privacy or message encryption. (This primitive is not called in RLP encryption (Enhanced Data Encryption).)

The parameter decrypt is set to 0 for encryption and 1 for decryption. It is needed here to determine the instance id number. This number uniquely identifies the particular SCEMA instance to prevent certain types of attacks.

The pointer schedPtr is the SCEMA key schedule pointer described earlier in Section 2.9.

Direction indicates either the forward channel by 1, or the reverse channel by 0.

Exhibit 2-56 Long Block Encryptor for Voice and Message Content

```
/*
Long Block Encryptor (6 octets) for Voice and Message Content
"longBlock.c"
Note: The Long Block Encryptor/Decryptor's LHS and RHS are each 3 octets
in length.
*/
#include "scema.h" /* see Exhibit 2-53 */

void Long_Block_Encryptor(unsigned char *contentBuf,
               const unsigned char contentType,
               const unsigned int decrypt,
               keySched *schedPtr,
               const unsigned int direction)
{
   unsigned char csync[4] = { 0x00, 0x00, 0x00, 0x00 };
   unsigned char id;
   unsigned char instanceId;

/*
Combine ID segments
Note: In particular, the LongBlockArchitecture ID segment forces bit 2
of SCEMA's cryptosync top octet to 1 to differentiate it from all
other uses (i.e.KSG uses) where bit 2 is forced to 0.
*/
   id = (unsigned char)(direction << 7) | contentType |
      LongBlockArchitecture;

/*
SCEMA instance 0: PT <- LHS of contentBuf, CS <- RHS, instance = 0
for encrypt, and 2 for decrypt.

Note: The temporary variable csync is used to prevent buffer overflow
during reading since SCEMA reads in a 4-octet csync buffer. This is
not needed in the second instance since no overflow occurs and since
the highest cync input octet is zeroed by LongBlkIdMask.
*/

   csync[0] = contentBuf[3];
   csync[1] = contentBuf[4];
   csync[2] = contentBuf[5];

   if (decrypt)
      instanceId = id | Instance2;
   else
      instanceId = id;

   SCEMA(contentBuf,ThreeOctets,csync,instanceId,LongBlkIdMask,
         decrypt,schedPtr);

/* SCEMA instance 1: PT <- RHS of contentBuf, CS <- LHS, instance = 1 */

   instanceId = id | Instance1;

   SCEMA(contentBuf + 3,ThreeOctets, contentBuf, instanceId,
      LongBlkIdMask, decrypt, schedPtr);
```

```
/* SCEMA instance 2: PT <- LHS of contentBuf, CS <- RHS, instance = 2 */

   csync[0] = contentBuf[3];
   csync[1] = contentBuf[4];
   csync[2] = contentBuf[5];

   if (decrypt)
      instanceId = id;
   else
      instanceId = id | Instance2;

   SCEMA(contentBuf,ThreeOctets,csync,instanceId,LongBlkIdMask,
         decrypt,schedPtr);

}
```

2.9.4.3. Short Block Encryptor

Procedure name:	
Short_Block_Encryptor	
Inputs from calling process:	
contentBuf[n]	1 - 6 octets, 1 – 47 bits
numBits	1 – 47 number of content bits in contentBuf buffer
contentType	1 octet defining voice or message
entropy[4]	4 octets of possible added entropy
decrypt	1 bit
schedPtr	pointer to SCEMA key schedule
direction	1 bit
Inputs from internal stored data:	
None.	
Outputs to calling process:	
contentBuf [n]	1 - 6 octets, 1 – 47 bits
Outputs to internal stored data:	
None.	

This encryption primitive block encrypts or decrypts a 1- to 6 octet buffer that contains a minimum of 1 bit and a maximum of 47 bits. (48 bits are also acceptable but the Short Block Encryptor will never be called with this amount since the Long Block Encryptor is used for 48 bits.)

The Short Block encryptor and decryptor are formed from four Feistel pieces that run SCEMA in a KSG mode. The Feistel piece contains the following parameters in order: the input buffer, output buffer, a KSG template used for filtering bits, an instance ID used for differentiating SCEMA uses according to instances, direction, and content, entropy from message type and RAND if extant for the type of content being encrypted, and a pointer to the key schedule.

The contentType parameter allows the Short Block Encryptor to generate unique keystream depending upon whether it is used in voice privacy or message encryption. (This primitive is not called in RLP encryption (Enhanced Data Encryption).)

The entropy parameter is used in for message encryption where the variables Message Type, and RAND (for DCCH only) provide added entropy to the encryption.

The parameter decrypt is set to 0 for encryption and 1 for decryption. It is needed here to determine the instance id number. This number uniquely identifies the particular SCEMA instance to prevent certain types of attacks. Also, the encryptor and decryptor architectures are not isomorphic due to the four instances of SCEMA (Feistel pieces), and thus the decryptor parameter is needed to select the architecture.

The pointer schedPtr is the SCEMA key schedule pointer described earlier in Section 2.9.

The direction parameter indicates either the forward channel by 1, or the reverse channel by 0.

Exhibit 2-57 Short Block Encryptor for Voice and Message Content

```
/*
Short Block Encryptor (less than 6 octets) for Voice and Message Content
"shortBlock.c"

Note: The Short Block Encryptor/Decryptor's LHS and RHS are each less
then or equal 3 octets in length. The number of content-bearing bits of
its LHS (left hand side) always equals or is one greater than the number
of content-bearing bits in its RHS.
*/

#include "scema.h" /* see Exhibit 2-53 */

void feistelPiece(const unsigned char *inputBuf,
              unsigned char *outputBuf,
              const unsigned char *ksgTemplate,
              const unsigned char instanceId,
              const unsigned char *entropy,
              keySched *schedPtr)
{
   unsigned int i;
   unsigned char csync[4];
   unsigned char keystreamBuf[3];

   /*
   SCEMA's PT input is tied to CS input with ID differentiator..
   ID is XORed in to yield KSGs that are unique with respect
   to content, direction, and instance.
   */

   for (i = 0; i < 3; i++)
   {
      csync[i] =  inputBuf[i] ^ entropy[i];
      keystreamBuf[i] = csync[i] ^ instanceId;
   }

   csync[3] = entropy[3] ^ instanceId;

   SCEMA(keystreamBuf,ThreeOctets,csync,KSGArchitecture,KSGIdMask,
      ENCRYPTING,schedPtr);
```

```
   /* KSG output is XORed with right buffer. The template passes
   only those bits that correspond to the right buffer's content
   bits.
   */

   for (i = 0; i < 3; i++)
      outputBuf[i] ^= keystreamBuf[i] & ksgTemplate[i];

}


void Short_Block_Encryptor(unsigned char *contentBuf,
               const unsigned int numBits,
               const unsigned char contentType,
               const unsigned char *entropy,
               const unsigned int decrypt,
               keySched *schedPtr,
               const unsigned int direction)
{
   unsigned int i;
   unsigned char id;
   unsigned int numBitsLocal;
   unsigned int octetSize;
   unsigned int numTopBits;

   unsigned char leftBuf[3] = {0x00,0x00,0x00};
   unsigned char rightBuf[3] = {0x00,0x00,0x00};
   unsigned int leftBufNumBits;
   unsigned int rightBufNumBits;

   unsigned char leftKsgTemplate[3] = {0x00,0x00,0x00};
   unsigned char rightKsgTemplate[3] = {0x00,0x00,0x00};

   unsigned char *pContent;
   unsigned char *pLeft;
   unsigned char *pRight;

   /* Prevents accidental buffer overflow */

   numBitsLocal = MIN(numBits,48);
   numBitsLocal = MAX(numBitsLocal,1);

   /*
   Number of octets needed to contain contentBuf bits
   Note: The index of the top octet (the highest one containing
   content) is thus octetSize - 1.
   */

   octetSize = ((numBitsLocal - 1) / 8) + 1;

   /*
   Number of content bits in top octet which occupy the top
   bits of the octet
   */

   numTopBits = numBitsLocal - (8 * (octetSize - 1));

   /* Number of content bits in left buffer */

   leftBufNumBits = (numBitsLocal + 1)/2;

```

```
/* Number of content bits in right buffer */

rightBufNumBits = numBitsLocal/2;

/* Ensure that unused contentBuf octets are zeroed and that
unused bits in the top octet are zeroed.
*/

for (i = octetSize; i < 6; i++)
   contentBuf[i] = 0;

contentBuf[octetSize - 1] >>= (8 - numTopBits);
contentBuf[octetSize - 1] <<= (8 - numTopBits);

/*
Divide contentBuf input bits between left and right buffers
to begin building a Feistel network. If numBitsLocal is even,
both buffers receive an equal number of bits. If numBitsLocal
is odd, the left buffer receives one more bit than the right
buffer.
*/

pContent = contentBuf;
pLeft = &leftBuf[0];
pRight = &rightBuf[0];

for (i = 0; i < 3; i++)
{
   *pLeft |= *pContent & 0xAA;
   *pRight |= (*pContent++ & 0x55) << 1;
   *pLeft++ |= (*pContent & 0xAA) >> 1;
   *pRight++ |= *pContent++ & 0x55;
}

/* Now that the content has been extracted from the contentBuf,
the buffer is re-used temporarily to generate KSG templates.
These templates will be used to pass only those KSG bits
corresponding to the content-bearing left and right buffer bits.
*/

for (i = 0; i < octetSize; i++)
   contentBuf[i] = 0xFF;

for (i = octetSize; i < 6; i++)
   contentBuf[i] = 0;

contentBuf[octetSize - 1] >>= (8 - numTopBits);
contentBuf[octetSize - 1] <<= (8 - numTopBits);

pContent = contentBuf;
pLeft = &leftKsgTemplate[0];
pRight = &rightKsgTemplate[0];
```

```
    for (i = 0; i < 3; i++)
    {
       *pLeft |= *pContent & 0xAA;
       *pRight |= (*pContent++ & 0x55) << 1;
       *pLeft++ |= (*pContent & 0xAA) >> 1;
       *pRight++ |= *pContent++ & 0x55;
    }

    /*
    Combine ID segments. A DCCH/DTC id segment is not needed for
    differentiation   because the two channels use different keys.
    */

    id = (unsigned char)(direction << 7) | contentType;


    /*
    Encryption/Decryption

    */

    if(!decrypt) /* encrypting */
    {
       /*
       Four Feistel-SCEMA instances. The zeroth instance does not
       contain an explicit instance number because the number
       is zero.
       */

       feistelPiece(leftBuf,rightBuf,rightKsgTemplate,
                    id,entropy,schedPtr);

       feistelPiece(rightBuf,leftBuf,leftKsgTemplate,
                    (unsigned char)(id | Instance1),
                    entropy,schedPtr);

       feistelPiece(leftBuf,rightBuf,rightKsgTemplate,
                    (unsigned char)(id | Instance2),
                    entropy,schedPtr);

       feistelPiece(rightBuf,leftBuf,leftKsgTemplate,
                    (unsigned char)(id | Instance3),
                    entropy,schedPtr);

    }


    /*
    Almost everything above is done in reverse order.
    */
```

```
   else  /* decrypting */
   {
      feistelPiece(rightBuf,leftBuf,leftKsgTemplate,
                   (unsigned char)(id | Instance3),
                   entropy,schedPtr);

      feistelPiece(leftBuf,rightBuf,rightKsgTemplate,
                   (unsigned char)(id | Instance2),
                   entropy,schedPtr);

      feistelPiece(rightBuf,leftBuf,leftKsgTemplate,
                   (unsigned char)(id | Instance1),
                   entropy,schedPtr);

      feistelPiece(leftBuf,rightBuf,rightKsgTemplate,
                   id,entropy,schedPtr);

   }

   /*
   Output processing: Load left and right buffers back into content
   buffer.
   */

   for (i = 0; i < 6; i++)
      contentBuf[i] = 0;

   pContent = contentBuf;
   pLeft = &leftBuf[0];
   pRight = &rightBuf[0];

   for (i = 0; i < 3; i++)
   {
      *pContent |= *pLeft & 0xAA;
      *pContent++ |= (*pRight >> 1) & 0x55;
      *pContent |= (*pLeft++ << 1) & 0xAA;
      *pContent++ |= *pRight++ & 0x55;
   }

}
```

2.9.5. Voice, Message, and Data Encryption Procedures (Level 3)

These top-level procedures interface directly TIA/EIA-136-510 and call the Level 2 procedures and, in the case of Enhanced Data Encryption only, the Level 1 (SCEMA) procedure.

2.9.5.1. Enhanced Voice Privacy

Procedure name:

Enhanced_Voice_Privacy

Inputs from calling process:

coderVer	0, 1, 2, etc.
speechBuf1[n]	n octets, 1 <= n <= 256
num1aBits	n >= 1
speechBufRem [n]	n octets, 0 <= n <= 256
numRemBits	n >= 0
decrypt	1 bit
keyGenerator	1,2,3, etc.
direction	1 bit

Inputs from internal stored data:

None.

Outputs to calling process:

speechBuf1[n]	n octets, 1 <= n <= 256
speechBufRem [n]	n octets, 0 <= n <= 256

Outputs to internal stored data:

None.

This Level 3 procedure encrypts or decrypts a frame of speech. The frame is separated into two buffers, speechBuf1 and speechBufRem, containing speech coders' Class 1A and remaining (Class 1B and 2) bits, respectively. Class 1A bits are those that are protected by a CRC in the speech coder algorithm. The respective numbers of these bits are num1aBits and numRemBits.

The parameter coderVer is set to 0 in TIA/EIA-136-510 and is not used here. It comprises a hook in case the CCA would ever need to be revised in the future due to a speech coder architecture incompatible with this current procedure.

The parameter decrypt is set to 0 for encryption and 1 for decryption. The encryptor and decryptor architectures are not isomorphic and thus the decryptor parameter is needed to select the architecture.

The parameter keyGenerator is currently set to 1 in TIA/EIA-136-510 to indicate CaveKey1, a key schedule based on the current CAVE algorithm running at its full strength. Internal to this procedure, the parameter is used to point to the DTCKey CaveKey1.

Direction indicates either the forward channel by 1, or the reverse channel by 0.

If the number of Class 1A bits is 48, then this procedure calls the Long Block Encryptor for these bits. If the number is greater than 48, the excess above 48 are encrypted by the SCEMA KSG. However, prior to encryption, their entropy is folded in to the first 48 bits that are encrypted by the Long Block Encryptor.

If the number of Class 1A bits is less than 48, these bits are encrypted by the Short Block Encryptor.

The remaining bits are encrypted by the SCEMA KSG using the Class 1A ciphertext as input (entropy).

Exhibit 2-58 Enhanced Voice Privacy

```
/* Enhanced Voice Privacy "enhVoicePriv.c" */

#include "scema.h" /* see Exhibit 2-53 */

void Enhanced_Voice_Privacy(const unsigned int coderVer,
                     unsigned char *speechBuf1,
                     const unsigned int num1aBits,
                     unsigned char *speechBufRem,
                     const unsigned int numRemBits,
                     const unsigned int decrypt,
                     const unsigned int keyGenerator,
                     const unsigned int direction)
{
   unsigned int i;
   unsigned char keystreamBuf[MaxFrameOctetSize];
   unsigned int net1aOctetSize;
   unsigned int num1aTopBits;
   unsigned int excess1aOctetSize;
   unsigned int remBitsOctetSize;
   unsigned int numRemTopBits;
   unsigned int ksgInputOctetSize;
   unsigned char nullEntropy[4] = { 0x00, 0x00, 0x00, 0x00 };

   /* Pointers to be set and used later */

   unsigned char *pKeyStream;
   unsigned char *pSpeech;

   /*
   Number of octets that contain the Class 1A bits, and
   number of bits in the 1A bits top octet.
   */

   net1aOctetSize = ((num1aBits - 1) / 8) + 1;
   num1aTopBits = num1aBits - (8 * (net1aOctetSize - 1));
```

```
/*
Number of octets that contain any excess Class 1A bits
beyond the first 6 octets (48 bits). For ACELP and VSELP,
this equals zero.
*/

excess1aOctetSize = MAX(net1aOctetSize ,6) - 6;

/*
Number of octets that contain the remaining bits, namely
those bits not protected by a CRC, usually called Class 1B
and Class 2 bits. Also calculated is the number of bits
in the remaining bits top octet.
*/

remBitsOctetSize = ((numRemBits - 1) / 8) + 1;
numRemTopBits = numRemBits - (8 * (remBitsOctetSize - 1));

/*
If the number of Class 1A bits is greater than or equal
to 48 bits, the 6-octet Long Block Encryptor is used, and
its output feeds the KSG. However, if the number of 1A bits
is less than 48 bits, the Short Block Encryptor is used and
only its output is fed to the KSG. In this latter case, the
KSG input will be repeated as necessary (in SCEMA_KSG()) to
fill SCEMA's plaintext input field.
*/

ksgInputOctetSize = MIN(net1aOctetSize, 6);

/* Input clean up */

/*
Ensure that bits other than the content-containing
1A top bits are zeroed.
*/

speechBuf1[net1aOctetSize - 1] >>= (8 - num1aTopBits);
speechBuf1[net1aOctetSize - 1] <<= (8 - num1aTopBits);

/*Do the same for the remaining bits, i.e the Class 1B and
Class 2 bits.
*/

speechBufRem[remBitsOctetSize - 1] >>= (8 - numRemTopBits);
speechBufRem[remBitsOctetSize - 1] <<= (8 - numRemTopBits);
```

```
    if(!decrypt) /* encrypting */
    {
       /*
       If there are more than 48 1A bits, XOR the excess
       into initial 48 bits to inject added entropy.
       */

       for (i = 0; i < excess1aOctetSize; i++)
          speechBuf1[i % 6] ^=  speechBuf1[i + 6];

       /*
       Use different block encryptors depending on the number
       of 1A bits.
       */
       if(num1aBits >= 48)
       {
          /*
          Block encrypt the first 6 octets of speechBuf1.
          Note: keyGenerator = 1 for CaveKey1.The first
          6 octets of speechBuf1 are replaced by ciphertext.
          */

          Long_Block_Encryptor(speechBuf1,VoiceContent,decrypt,
                               dtcScheds + keyGenerator - 1,
                               direction);
       }

       else /* num1aBits < 48 */
       {
          /*
          Block encrypt num1aBits of speechBuf1 to yield the
          same amount of ciphertext.
          */

          Short_Block_Encryptor (speechBuf1,num1aBits,VoiceContent,
                                 nullEntropy,decrypt,
                                 dtcScheds + keyGenerator - 1,
                                 direction);
       }

       /*
       Form the appropriate amount of keystream with
       speechBuf1 as input. Either the first 6 octets
       of speechBuf1 are used which comprise the output of the
       Long Block Encryptor, or less are used if
       ksgInputOctetSize is set less than 6 octets, namely the
       output of the Short Block Encryptor.
       */

       SCEMA_KSG(keystreamBuf,
                 excess1aOctetSize + remBitsOctetSize,
                 speechBuf1,ksgInputOctetSize,VoiceContent,
                 dtcScheds + keyGenerator - 1,direction);

       /*
       XOR keystream into buffers to yield ciphertext
       Start at zeroth keystream octet
       */

       pKeyStream = &keystreamBuf[0];
```

```
        /* First encrypt excess 1A bits if extant */

        pSpeech = speechBuf1 + 6;

        for (i = 0; i < excess1aOctetSize; i++)
           *pSpeech++ ^=  *pKeyStream++;

        /*
        Ensure that bits other than the content-containing
        (encrypted) (excess) 1A top bits are zeroed.
        */

        speechBuf1[net1aOctetSize - 1] >>= (8 - num1aTopBits);
        speechBuf1[net1aOctetSize - 1] <<= (8 - num1aTopBits);


        /* Then encrypt remaining bits */

        pSpeech = speechBufRem;

        for (i = 0; i < remBitsOctetSize; i++)
           *pSpeech++ ^= *pKeyStream++;

     }


     else  /* decrypting */
     {
        /*
        Almost everything above is done in reverse order.
        The KSG is now first, and the block encryptor second.
        */

        SCEMA_KSG(keystreamBuf,
                  excess1aOctetSize + remBitsOctetSize,
                  speechBuf1,ksgInputOctetSize,VoiceContent,
                  dtcScheds + keyGenerator - 1,direction);

        pKeyStream = &keystreamBuf[0];
        pSpeech = speechBuf1 + 6;

        for (i = 0; i < excess1aOctetSize; i++)
           *pSpeech++ ^=  *pKeyStream++;

        pSpeech = speechBufRem;

        /* Decrypt remaining bits */

        for (i = 0; i < remBitsOctetSize; i++)
           *pSpeech++ ^= *pKeyStream++;

        /* Block encryptor choice */

        if(num1aBits >= 48)
        {
           Long_Block_Encryptor(speechBuf1,VoiceContent,decrypt,
                                dtcScheds + keyGenerator - 1,
                                direction);
        }
```

```
        else /* num1aBits < 48 */
        {
           Short_Block_Encryptor(speechBuf1,num1aBits,VoiceContent,
                                 nullEntropy,decrypt,
                                 dtcScheds + keyGenerator - 1,direction);
        }

        /*
        Ensure that bits other than the content-containing
        (decrypted) 1A top bits are zeroed, and then do
        post-XORing.
        */

        speechBuf1[net1aOctetSize - 1] >>= (8 - num1aTopBits);
        speechBuf1[net1aOctetSize - 1] <<= (8 - num1aTopBits);

        if(num1aBits > 48)
           for (i = 0; i < excess1aOctetSize; i++)
              speechBuf1[i % 6] ^=  speechBuf1[i + 6];

     }

     /*
     Remaining output clean up: Ensure that bits other than the
     content-containing remaining bits (Class 1B and Class 2
     bits) are zeroed.
     */

     speechBufRem[remBitsOctetSize - 1] >>= (8 - numRemTopBits);
     speechBufRem[remBitsOctetSize - 1] <<= (8 - numRemTopBits);

  }
```

2.9.5.2. Enhanced Message Encryption

Procedure name:

Enhanced_Message_Encryption

Inputs from calling process:

msgBuf [n]	n octets, 1 <= n <= 256
numBits	n >= 1
dcchDTC	1 bit
rand[4]	4 octets
msgType	1 octet
decrypt	1 bit
keyGenerator	1,2,3, etc.
direction	1 bit

Inputs from internal stored data:

None.

Outputs to calling process:

msgBuf[n]	n octets, 1 <= n <= 256

Outputs to internal stored data:

None.

This Level 3 procedure encrypts or decrypts the Layer 3 content of a message as a whole. The message and its number of bits are denoted by the parameters msgBuf and numBits respectively.

The parameter dcchDTC indicates to this procedure whether messages are on the DCCH channel (dcchDTC = 0), or on the DTC channel (dcchDTC = 1). For DCCH encryption only, the value rand is used for added entropy in addition to msgType (Message Type). For DTC encryption, only msgType is used.

The parameter decrypt is set to 0 for encryption and 1 for decryption. The encryptor and decryptor architectures are not isomorphic and thus the decryptor parameter is needed to select the architecture.

The parameter keyGenerator is currently set to 1 in TIA/EIA-136-510 to indicate CaveKey1, a key schedule based on the current CAVE algorithm running at its full strength. Internal to this procedure, the parameter is used to point to the DTC CaveKey1 key schedule (DTCKey) for DTC messages, and to the DCCH CaveKey1 key schedule (DCCHKey) for DCCH messages.

Direction indicates either the forward channel by 1, or the reverse channel by 0.

If the number of message bits is 48, then this procedure calls the Long Block Encryptor for these bits. If this number is greater than 48, the excess above 48 are encrypted by the SCEMA KSG. However, prior to encryption, their entropy is folded in to the first 48 bits that are encrypted by the Long Block Encryptor.

If the number of message bits is less than 48, these bits are encrypted by the Short Block Encryptor.

Exhibit 2-59 Enhanced Message Encryption

```
/* Enhanced Message Encryption "enhMsgEnc.c" */

#include "scema.h" /* see Exhibit 2-53 */

void Enhanced_Message_Encryption(unsigned char *msgBuf,
                     const unsigned int numBits,
                     const unsigned int dcchDTC,
                     const unsigned char *rand,
                     const unsigned char msgType,
                     const unsigned int decrypt,
                     const unsigned int keyGenerator,
                     const unsigned int direction)
{
   unsigned int i;
   unsigned char keystreamBuf[MaxMessageOctetSize];
   unsigned int msgBufOctetSize;
   unsigned int numTopBits;
   unsigned int excessOctetSize;
   unsigned int ksgInputOctetSize;
   unsigned char entropy[4] = { 0x00, 0x00, 0x00, 0x00 };

   /* Pointers to be set and used later */

   unsigned char *pKeyStream;
   unsigned char *pMessage;
   keySched *pDcchDtc;

   /* Entropy gathering and key schedule selection*/

   if(dcchDTC) /* DTC channel */
   {
      entropy[0] = msgType;
      pDcchDtc = dtcScheds;
   }

   else /* DCCH channel */
   {
      for (i = 0; i < 4; i++)
         entropy[i] = rand[i];
      entropy[0] ^= msgType;
      pDcchDtc = dcchScheds;
   }
```

```
/*
Number of octets that contain the message bits, and
number of bits in their top octet.
*/

msgBufOctetSize = ((numBits - 1) / 8) + 1;
numTopBits = numBits - (8 * (msgBufOctetSize - 1));

/*
Number of octets that contain any excess message bits
beyond the first 6 octets (48 bits).
*/

excessOctetSize = MAX(msgBufOctetSize ,6) - 6;

/*
If the number of message bits is greater than or equal
to 48 bits, the 6-octet Long Block Encryptor is used, and
its output feeds the KSG. The KSG is run only if excess
bits are present. However, if the number of message bits
is less than 48 bits, only the Short Block Encryptor is
used.
*/

ksgInputOctetSize = MIN(msgBufOctetSize, 6);

/* Input clean up */

/*
Ensure that bits other than the content-containing
top bits are zeroed.
*/

msgBuf[msgBufOctetSize - 1] >>= (8 - numTopBits);
msgBuf[msgBufOctetSize - 1] <<= (8 - numTopBits);

if(!decrypt) /* encrypting */
{
   /*
   If there are more than 48 message bits, XOR the excess
   into initial 48 bits to inject added entropy.
   */

   for (i = 0; i < excessOctetSize; i++)
      msgBuf[i % 6] ^=  msgBuf[i + 6];

   /*
   Use different block encryptors depending on the number
   of message bits.
   */

   if(numBits >= 48)
   {
      /*
      Block encrypt the first 6 octets of msgBuf and
      first inject entropy.
      Note: keyGenerator = 1 for CaveKey1.The first
      6 octets of msgBuf are replaced by ciphertext.
      */
```

```
        for (i = 0; i < 4; i++)
           msgBuf[i] ^=  entropy[i];

        Long_Block_Encryptor(msgBuf,MessageContent,decrypt,
                             pDcchDtc + keyGenerator - 1,
                             direction);

        if(numBits > 48)
        {
           /*
           Form the appropriate amount of keystream with
           msgBuf as input.
           */

           SCEMA_KSG(keystreamBuf, excessOctetSize, msgBuf,
              ksgInputOctetSize, MessageContent,
              pDcchDtc + keyGenerator - 1, direction);

           /*
           XOR keystream into buffers to yield ciphertext
           Start at zeroth keystream octet
           */

           pKeyStream = &keystreamBuf[0];

           /* First encrypt excess message bits if extant */

           pMessage = msgBuf + 6;
           for (i = 0; i < excessOctetSize; i++)
              *pMessage++ ^=  *pKeyStream++;
        }
     }

     else /* numBits < 48 */
     {
        /*
        Block encrypt numBits of msgBuf to yield the
        same amount of ciphertext.
        */

        Short_Block_Encryptor(msgBuf, numBits, MessageContent,
           entropy, decrypt, pDcchDtc + keyGenerator - 1,direction);
     }

     /*
     Ensure that bits other than the content-containing
     (encrypted) (excess) message top bits are zeroed.
     */

     msgBuf[msgBufOctetSize - 1] >>= (8 - numTopBits);
     msgBuf[msgBufOctetSize - 1] <<= (8 - numTopBits);

  }
```

```
    else  /* decrypting */
    {
       /*
       Almost everything above is done in reverse order.
       The KSG is now first, and the block encryptor second.
       */

       if(numBits > 48)
       {


    SCEMA_KSG(keystreamBuf,excessOctetSize,msgBuf,ksgInputOctetSize,
             MessageContent, pDcchDtc + keyGenerator - 1,direction);

             pKeyStream = &keystreamBuf[0];
             pMessage = msgBuf + 6;

             for (i = 0; i < excessOctetSize; i++)
                *pMessage++ ^=  *pKeyStream++;

       }

       /* Block encryptor choice */

       if(numBits >= 48)
       {
          Long_Block_Encryptor(msgBuf,MessageContent,decrypt,
                               pDcchDtc + keyGenerator - 1,
                               direction);

          for (i = 0; i < 4; i++)
             msgBuf[i] ^=  entropy[i];
       }

       else /* numBits < 48 */
       {
          Short_Block_Encryptor(msgBuf,numBits,MessageContent,entropy,
                       decrypt, pDcchDtc + keyGenerator - 1,direction);
       }

       /*
       Ensure that bits other than the content-containing
       (decrypted) message top bits are zeroed, and then do
       post-XORing.
       */

       msgBuf[msgBufOctetSize - 1] >>= (8 - numTopBits);
       msgBuf[msgBufOctetSize - 1] <<= (8 - numTopBits);

       if(numBits > 48)
          for (i = 0; i < excessOctetSize; i++)
             msgBuf[i % 6] ^=  msgBuf[i + 6];

    }

}
```

2.9.5.3. Enhanced Wireless Data Encryption

Procedure name:	
Enhanced_Data_Mask	
Inputs from calling process:	
mask[len]	len octets
HOOK	32 bits
len	1 <= len <= 256
keyGenerator	1,2,3, etc.
Inputs from internal stored data:	
None.	
Outputs to calling process:	
mask[len]	len octets
Outputs to internal stored data:	
None.	

Enhanced data encryption for 136 wireless data services is provided by running SCEMA in the encrypt mode as a KSG. This procedure generates an encryption mask of length len octets, between 1 and 256 inclusive. A pointer for the output value "mask" buffer containing keystream mask of length len octets.

HOOK is a 32-bit value that serves as cryptosync, and is input both to SCEMA’s cryptosync input and repeated across its plaintext field.

The parameter keyGenerator is currently set to 1 in TIA/EIA-136-510 to indicate CaveKey1, a key schedule based on the current CAVE algorithm running at its full strength. Internal to this procedure, the parameter is used to point to the DTC CaveKey1.

Internal to this procedure is a mechanism for differentiating this keystream from that produced by other uses of SCEMA in the KSG mode. To accomplish, it uses the identifier RlpContent.

Exhibit 2-60 Enhanced Data Mask Generation

```
/* Enhanced Data Mask Generation "enhDataMask.c" */

#include "scema.h" /* see Exhibit 2-53 */

void Enhanced_Data_Mask(unsigned char *mask,
                  const unsigned long HOOK,
                  const unsigned int len,
                  const unsigned int keyGenerator)
{
   unsigned int i;
   unsigned char csync[4];
   unsigned char maskSix[6];

   csync[0] = (unsigned char)(HOOK & 0xFF);
   csync[1] = (unsigned char)((HOOK >> 8) & 0xFF);
   csync[2] = (unsigned char)((HOOK >> 16) & 0xFF);
   csync[3] = (unsigned char)((HOOK >> 24) & 0xFF);

   if(len >= 6)
   {

      /* Repeat HOOK across SCEMA's PT field */
      for (i = 0; i < len; i++)
         mask[i] =  csync[i % 4];

      /* Prevents cross-replay effects with other content types */
      for (i = 0; i < 4; i++)
         csync[i] ^=  RlpContent;

      /*
      Note: keyGenerator = 1 for CaveKey1.
      Since RLP encryption uses SCEMA in a KSG mode, the values
      KSGArchitecture and KSGIdMask are passed. This serves to force
      bit 2    in the cryptosync's top octet to zero to differentiate
      the cryptosync from that used in the Long Block Encryptor.
      */
      SCEMA(mask,len,csync,KSGArchitecture,KSGIdMask,ENCRYPTING,
            dtcScheds + keyGenerator - 1);

   }


/*
If requested length is less then 6, create 6 octets of keystream
and output only what is needed
*/

   else
   {
      for (i = 0; i < 6; i++)
         maskSix[i] =  csync[i % 4];

      /* Prevents cross-replay effects with other content types */
      for (i = 0; i < 4; i++)
         csync[i] ^=  RlpContent;

   SCEMA(maskSix,SixOctets,csync,KSGArchitecture,KSGIdMask,ENCRYPTING,
            dtcScheds + keyGenerator - 1);
```

```
        for (i = 0; i < len; i++)
        {
            mask[i] = maskSix[i];
        }

    }
}
```

3. Test Vectors

3.1. CAVE Test Vectors

These two test cases utilize the following fixed input data (expressed in hexadecimal form):

RANDSSD	=	4D	18EE	AA05	895C
Authentication Algorithm Version	=				C7
AUTH_DATA	=			79	2971
ESN	=			D75A	96EC
msg_buf[0] . . msg_buf[5]	=	B6, 2D, A2, 44, FE, 9B			

The following A-key and check digits should be entered in decimal form:

```
14 1421 3562 3730 9504 8808 6500
```

Conversion of the A-key, check digit entry into hex form will produce:

A-key, check bits = C442 F56B E9E1 7158, 1 51E4

The above entry, when combined with RANDSSD, will generate:

SSD_A = CC38 1294 9F4D CD0D

SSD_B = 3105 0234 580E 63B4

3.1.1. Vector 1

If RAND_CHALLENGE = 34A2 B05F:

(Using SSD_AUTH = SSD_A)

```
AUTH_SIGNATURE=   3 66F6
CMEA key k0,. .k7  =  A0 7B 1C D1 02 75 69 14
ECMEA key          =  5D ED AD 53 5B 4A B9 FC
offset_key         =  BD 71 D5 CD
SEED_NF key        =  2F 15 F6 D1 27
ECMEA_NF key       =  73 03 44 3C 55 DF B2 58
offset_nf_key      =  14 6F 91 5B
```

sync = 3D A2

CMEA output = E5 6B 5F 01 65 C6

Mobile station:

```
ECMEA Output = d5 39 d7 45 cd 11
ECMEA_NF Output = 3a 30 6a 40 39 b5
```

Base Station:

```
ECMEA Output = 50 9d c7 9b 19 d1
ECMEA_NF Output = 96 7c 7b e4 9d 34
```

```
VPM = 18 93 94 82 4A 1A 2F 99
      A5 39 F9 5B 4D 22 D5 7C
      EE 32 AC 21 6B 26 0D 36
      A7 C9 63 88 57 8C B9 57
      E2 D6 CA 1D 77 B6 1F D5
      C7 1A 73 A4 17 B2 12 1E
      95 34 70 E3 9B CA 3F D0
      50 BE 4F D6 47 80 CC B8
      DF
```

3.1.2. Vector 2

If RAND_CHALLENGE = 5375 DF99:

(Using SSD_AUTH = SSD_A)

```
AUTH_SIGNATURE=    0 255A
CMEA key k0,. .k7 = F0 06 A8 5A 05 CD B3 2A
ECMEA key       =  B6 DF 9A D0 6E 5A 3D 14
offset_key      =  F9 A4 2C FA
SEED_NF key     =  65 33 AE 92 C7
ECMEA_NF key    =  5C EF 0E E0 80 6A 1F 6B
offset_nf_key   =  C4 74 3C 71
```

```
sync = FF FF
```

```
CMEA output     =  2B AD 16 A9 8F 32
```

Mobile station:

```
ECMEA Output = 91 cf e3 25 5e 44
ECMEA_NF Output = cd 51 22 b2 74 49
```

Base Station:

```
ECMEA Output = f0 b8 9a 4b 06 55
ECMEA_NF Output = 0d fb 93 e4 59 da
```

```
VPM = 20 38 01 6B 89 3C F8 A0
      28 48 98 75 AB 18 65 5A
      49 6E 0B BB D2 CB A8 28
      46 E6 D5 B4 12 B3 8C 9E
      76 6C 9E D4 98 C8 A1 4A
      D2 DC 94 B0 F6 D4 3E E0
      D1 6C 7E 9E AC 6B CA 43
      02 C9 23 63 6F 61 68 E8
      8F
```

3.1.3. Vector 3

If RAND_CHALLENGE = 6c00 0258:

(Using SSD_AUTH = SSD_A)

```
AUTH_SIGNATURE    =  0 8a8a
CMEA key k0,. .k7 =  5A C8 04 25 32 FB 2D 54
ECMEA key         =  20 64 57 F6 EE 60 EB AD
offset_key        =  E9 83 41 FB
SEED_NF key       =  84 AD CF 40 BB
ECMEA_NF key      =  33 37 C8 F3 85 50 C7 03
offset_nf_key     =  E0 2C 66 FA
```

```
sync = FF FF
```

```
CMEA output       =  A3 06 25 D8 3E 21
```

Mobile station:

```
ECMEA Output = 41 ed 74 99 7d 41
ECMEA_NF Output = ab aa 88 7e b6 f3
```

Base Station:

```
ECMEA Output = 6d 73 27 54 3d 9c
ECMEA_NF Output = 8c e1 e2 b4 fd 62
```

```
VPM = ED A7 AA 63 27 EA F8 3D
      30 26 8C C5 18 88 8F 6D
      CD 0D 1D 97 21 06 2D 91
      1D CF 47 1F DD BE E3 E1
      71 18 26 73 7A 5F 09 CC
      13 2A 51 69 27 55 2B 2B
      0B 30 5A 09 F6 15 8F A7
      A9 55 7A 00 23 D8 FD 4C
      3E
```

3.1.4. Test Program

```
#include <stdio.h>
#include "cave.h"   /* see Exhibit 2-2  */
#include "ecmea.h"  /* see Exhibit 2-29 */

/* NAM stored data */

unsigned char ESN[4] = { 0xd7, 0x5a, 0x96, 0xec };
unsigned char MIN1[3] = { 0x79, 0x29, 0x71 };
unsigned char A_key[8];
unsigned char SSD_A_NEW[8], SSD_A[8];
unsigned char SSD_B_NEW[8], SSD_B[8];

/* data received from the network */

unsigned char RANDSSD[7] = { 0x4d, 0x18, 0xee, 0xaa,
                              0x05, 0x89, 0x5c };
unsigned char RAND1[4]    = { 0x34, 0xa2, 0xb0, 0x5f };
unsigned char RAND2[4]    = { 0x53, 0x75, 0xdf, 0x99 };
unsigned char RAND3[4]    = { 0x6c, 0x00, 0x02, 0x58 };

/* cryptosync (meaning is air interface specific) */

unsigned char sync1[2] = { 0x3d, 0xa2 };
unsigned char sync2[2] = { 0xff, 0xff };

/* test plaintext */

unsigned char buf[6] = { 0xb6, 0x2d, 0xa2, 0x44, 0xfe, 0x9b };

/* entered A_key and checksum */

char digits[26] =
   { '1', '4', '1', '4', '2', '1', '3', '5', '6', '2',
     '3', '7', '3', '0', '9', '5', '0', '4', '8', '8',
     '0', '8', '6', '5', '0', '0' };

void pause(void)
{
   printf("Enter to continue\n");
   getchar();
}
```

```
void main(void)
{
   int i, j;
   unsigned char auth_data[3], test_buf[6];
   unsigned long AUTHR;

   /* check A_key and SSD */

   if(A_Key_Verify(digits))
   {
      printf("A key verified ok\n");
   }
   else
   {
      printf("A key verification failed\n");
      return;
   }

   /* check SSD generation process */

   SSD_Generation(RANDSSD);
   SSD_Update();

   printf("SSD_A =");
   for (i = 0; i < 4; i++)
   {
      printf(" ");
      for (j = 0; j < 2; j++)
         printf("%02x",(unsigned int)SSD_A[2*i+j]);
   }
   printf("\n");

   printf("SSD_B =");
   for (i = 0; i < 4; i++)
   {
      printf(" ");
      for (j = 0; j < 2; j++)
         printf("%02x",(unsigned int)SSD_B[2*i+j]);
   }
   printf("\n");

   /* Inputs for test vectors */

   /* put MIN1 into auth_data (no dialed digits for this test) */

   for (i = 0; i < 3; i++)
      auth_data[i] = MIN1[i];

   /* vector 1 */

   printf("\nVector 1\n\n");

   AUTHR = Auth_Signature(RAND1,auth_data,SSD_A,1);

   printf("RAND_CHALLENGE =");
   for (i = 0; i < 2; i++)
   {
      printf(" ");
      for (j = 0; j < 2; j++)
         printf("%02x",(unsigned int)RAND1[2*i+j]);
   }
```

```
    printf("\n");

    printf("AUTH_SIGNATURE = %01lx %04lx\n", AUTHR >> 16,
       AUTHR & 0x0000ffff);

    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];

    Key_VPM_Generation();
    ECMEA_Secret_Generation();
    Non_Financial_Seed_Key_Generation();
    Non_Financial_Secret_Generation();

    printf("      CMEA key =");
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)cmeakey[i]);
    printf("\n");

    printf("     ECMEA key =");
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)ecmea_key[i]);
    printf("\n");

    printf("    offset_key =");
    for (i = 0; i < 4; i++)
       printf(" %02x",(unsigned int)offset_key[i]);
    printf("\n");

    printf("   SEED_NF key =");
    for (i = 0; i < 5; i++)
       printf(" %02x",(unsigned int)seed_nf_key[i]);
    printf("\n");

    printf("  ECMEA_NF key =");
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)ecmea_nf_key[i]);
    printf("\n");

    printf(" offset_nf_key =");
    for (i = 0; i < 4; i++)
       printf(" %02x",(unsigned int)offset_nf_key[i]);
    printf("\n");

    printf("    sync =");
    printf(" %02x %02x\n",(unsigned int)sync1[0],
       (unsigned int)sync1[1]);

    pause();

    printf("          Input =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    CMEA(test_buf,6);

    printf("    CMEA Output =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");
```

```
    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];
    ECMEA(test_buf,6,sync1,0,0);

    printf("   ECMEA Output =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];
    ECMEA(test_buf,6,sync1,0,1);

    printf("ECMEA_NF Output =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    printf("VPM =");
    for (i = 0; i < 65; i++)
    {
       printf(" %02x",(unsigned int)VPM[i]);
       if (((i+1)%8) == 0)
          printf("\n     ");
    }
    printf("\n");

    pause();

    /* vector 2 */

    printf("\nVector 2\n\n");

    AUTHR = Auth_Signature(RAND2,auth_data,SSD_A,1);

    printf("RAND_CHALLENGE =");
    for (i = 0; i < 2; i++)
    {
       printf(" ");
       for (j = 0; j < 2; j++)
          printf("%02x",(unsigned int)RAND2[2*i+j]);
    }
    printf("\n");

    printf("AUTH_SIGNATURE = %01lx %04lx\n", AUTHR >> 16,
       AUTHR & 0x0000ffff);

    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];

    Key_VPM_Generation();
    ECMEA_Secret_Generation();
    Non_Financial_Seed_Key_Generation();
    Non_Financial_Secret_Generation();

    printf("      CMEA key =");
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)cmeakey[i]);
    printf("\n");

    printf("     ECMEA key =");
```

```
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)ecmea_key[i]);
    printf("\n");

    printf("    offset_key =");
    for (i = 0; i < 4; i++)
       printf(" %02x",(unsigned int)offset_key[i]);
    printf("\n");

    printf("   SEED_NF key =");
    for (i = 0; i < 5; i++)
       printf(" %02x",(unsigned int)seed_nf_key[i]);
    printf("\n");

    printf("  ECMEA_NF key =");
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)ecmea_nf_key[i]);
    printf("\n");

    printf(" offset_nf_key =");
    for (i = 0; i < 4; i++)
       printf(" %02x",(unsigned int)offset_nf_key[i]);
    printf("\n");

    printf("    sync =");
    printf(" %02x %02x\n",(unsigned int)sync2[0],
       (unsigned int)sync2[1]);

    pause();

    printf("          Input =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    CMEA(test_buf,6);

    printf("    CMEA Output =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];
    ECMEA(test_buf,6,sync2,0,0);

    printf("   ECMEA Output =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];
    ECMEA(test_buf,6,sync2,0,1);

    printf("ECMEA_NF Output =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    printf("VPM =");
```

```
    for (i = 0; i < 65; i++)
    {
       printf(" %02x",(unsigned int)VPM[i]);
       if (((i+1)%8) == 0)
          printf("\n     ");
    }
    printf("\n");

    pause();

    /* vector 3 */

    printf("\nVector 3\n\n");

    AUTHR = Auth_Signature(RAND3,auth_data,SSD_A,1);

    printf("RAND_CHALLENGE =");
    for (i = 0; i < 2; i++)
    {
       printf(" ");
       for (j = 0; j < 2; j++)
          printf("%02x",(unsigned int)RAND3[2*i+j]);
    }
    printf("\n");

    printf("AUTH_SIGNATURE = %01lx %04lx\n", AUTHR >> 16,
       AUTHR & 0x0000ffff);

    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];

    Key_VPM_Generation();
    ECMEA_Secret_Generation();
    Non_Financial_Seed_Key_Generation();
    Non_Financial_Secret_Generation();

    printf("      CMEA key =");
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)cmeakey[i]);
    printf("\n");

    printf("     ECMEA key =");
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)ecmea_key[i]);
    printf("\n");

    printf("    offset_key =");
    for (i = 0; i < 4; i++)
       printf(" %02x",(unsigned int)offset_key[i]);
    printf("\n");

    printf("   SEED_NF key =");
    for (i = 0; i < 5; i++)
       printf(" %02x",(unsigned int)seed_nf_key[i]);
    printf("\n");

    printf("  ECMEA_NF key =");
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)ecmea_nf_key[i]);
    printf("\n");
```

```
    printf(" offset_nf_key =");
    for (i = 0; i < 4; i++)
       printf(" %02x",(unsigned int)offset_nf_key[i]);
    printf("\n");

    printf("     sync =");
    printf(" %02x %02x\n",(unsigned int)sync2[0],
       (unsigned int)sync2[1]);

    pause();

    printf("            Input =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    CMEA(test_buf,6);

    printf("     CMEA Output =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];
    ECMEA(test_buf,6,sync2,0,0);

    printf("    ECMEA Output =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];
    ECMEA(test_buf,6,sync2,0,1);

    printf("ECMEA_NF Output =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    printf("VPM =");
    for (i = 0; i < 65; i++)
    {
       printf(" %02x",(unsigned int)VPM[i]);
       if (((i+1)%8) == 0)
          printf("\n     ");
    }
    printf("\n");

    pause();
}
```

3.2. Wireless Residential Extension Test Vector

3.2.1. Input data

```
Manufacturer's Key =   2    14 0E 9F 70 50 D7 EA
                            42 D9 C9 00 C9 14 14
                            CF

BID                =   00 00 01 00

Random Challenge   =   7E 49 AE 4F

ACRE Phone Number  =   549-8506

Random WRE         =   3    17 52

ESN                =   ED 07 13 95

Random WIKEY       =   B7 FC 75 5A F0 A4 90

WRE Key            =   CB 60 F9 9F 5B 15 6F AE
```

3.2.2. Test Program

```
#include <stdio.h>
#include "cave.h"  /* see Exhibit 2-2 */
#include "wre.h"   /* see Exhibit 2-31 */

/* NAM stored data */

unsigned char ESN[4] = { 0xd7, 0x5a, 0x96, 0xec };
unsigned char MIN1[3] = { 0x79, 0x29, 0x71 };
unsigned char A_key[8];
unsigned char SSD_A_NEW[8], SSD_A[8];
unsigned char SSD_B_NEW[8], SSD_B[8];


/* Test vector inputs */

unsigned char manufact[16] = { 0x85, 0x03, 0xA7, 0xDC,
                               0x14, 0x35, 0xFA, 0x90,
                               0xB6, 0x72, 0x40, 0x32,
                               0x45, 0x05, 0x33, 0xC0 };

unsigned char baseid[4] = { 0x00, 0x00, 0x01, 0x00 };

unsigned char random_challenge[4] = { 0x7E, 0x49, 0xAE, 0x4F };

unsigned char acre_phone[3] = { 0x49, 0x85, 0xA6 };

unsigned char random_wre[3] = { 0x62, 0xEA, 0x40 };

unsigned char hs_esn[4] = { 0xED, 0x07, 0x13, 0x95 };

unsigned char rand_wikey[7] = { 0xB7, 0xFC, 0x75, 0x5A,
                                0xF0, 0xA4, 0x90 };

/* CAVE outputs */

extern unsigned char    WIKEY[8];
extern unsigned char    WIKEY_NEW[8];
extern unsigned char    WRE_KEY[8];

void main(void)
{
    int i;
    unsigned long auth_sig;

    WIKEY_Generation(manufact,baseid);
    printf("WIKEY = ");
    for(i=0;i<8;i++)
      printf("%02x",(unsigned int)WIKEY[i]);
    printf("\n");

    auth_sig = WI_Auth_Signature(random_challenge,baseid,acre_phone);
    printf("AUTH_SIGNATURE = %05lx\n",auth_sig);

    WRE_KEY[0] = 0xCB;
    WRE_KEY[1] = 0x60;
    WRE_KEY[2] = 0xF9;
    WRE_KEY[3] = 0x9F;
    WRE_KEY[4] = 0x5B;
```

```
        WRE_KEY[5] = 0x15;
        WRE_KEY[6] = 0x6F;
        WRE_KEY[7] = 0xAE;

        auth_sig = WRE_Auth_Signature(random_wre,baseid,hs_esn);
        printf("AUTH_SIGNATURE = %05lx\n",auth_sig);

        WIKEY_Update(rand_wikey,baseid);
        printf("WIKEY_NEW = ");
        for(i=0;i<8;i++)
          printf("%02x",(unsigned int)WIKEY_NEW[i]);
        printf("\n");

    printf("Enter to exit\n");
    getchar();
}
```

3.2.3. Test Program Output

```
WIKEY = cb60f99f5b156fae
AUTH_SIGNATURE = 2cf01
AUTH_SIGNATURE = 12893
WIKEY_NEW = 167ca928358cceba
```

3.3. Basic Data Encryption Test Vector

3.3.1. Input data

```
SSD_B= 1492 5280 1776 1867
RAND = 1234 ABCD
HOOK = CDEF 5678
```
24 octets of mask to be returned

3.3.2. Test Program

```
#include <stdio.h>
#include "cave.h"  /* see Exhibit 2-2 */
#include "oryx.h"  /* see Exhibit 2-45 */

/* NAM stored data */

unsigned char ESN[4] = { 0xd7, 0x5a, 0x96, 0xec };
unsigned char MIN1[3] = { 0x79, 0x29, 0x71 };
unsigned char A_key[8];
unsigned char SSD_A_NEW[8], SSD_A[8];
unsigned char SSD_B_NEW[8], SSD_B[8];

void pause(void)
{
   printf("Enter to continue\n");
   getchar();
}

void main(void)
{
   int i, j;
   unsigned long hook;
   unsigned char buf[24], rand[4];

   rand[0] = 0x12;
   rand[1] = 0x34;
   rand[2] = 0xab;
   rand[3] = 0xcd;

   hook = 0xcdef5678;

   SSD_B[0] = 0x14;
   SSD_B[1] = 0x92;
   SSD_B[2] = 0x52;
   SSD_B[3] = 0x80;
   SSD_B[4] = 0x17;
   SSD_B[5] = 0x76;
   SSD_B[6] = 0x18;
   SSD_B[7] = 0x67;

   printf("\nSSD_B =");
   for (i = 0; i < 4; i++)
   {
      printf(" ");
      for (j = 0; j < 2; j++)
      {
         printf("%02x", (unsigned int)SSD_B[2*i+j]);
```

```
        }
    }

    printf("\nRAND =");
    for (i = 0; i < 2; i++)
    {
        printf(" ");
        for (j = 0; j < 2; j++)
        {
            printf("%02x", (unsigned int)rand[2*i+j]);
        }
    }

    printf("\nHOOK = %04lx %04lx\n", hook >> 16, hook & 0x0000ffff);

    pause();

    printf("24 octets of mask to be returned");

    DataKey = DataKey_Generation();

    printf("\n\nOutput:\n\n");

    printf("\nDataKey = %04lx %04lx\n", DataKey >> 16,
        DataKey & 0x0000ffff);

    LTable_Generation(rand);

    printf("\n\nL:\n\n");

    for(i = 0; i < 16; i++)
    {
        for (j = 0; j < 16; j++)
        {
            printf("%02x ", (unsigned int)L[16*i+j]);
        }
        printf("\n");
    }

    pause();

    Data_Mask(DataKey, hook, 24, buf);

    printf("\n\nmask:\n\n");

    for(i = 0; i < 2; i++)
    {
        for (j = 0; j < 12; j++)
        {
            printf("%02x ", (unsigned int)buf[12*i+j]);
        }
        printf("\n");
    }
    pause();
}
```

3.3.3. Test Program Output

DataKey = 8469 B522

L:

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

mask

```
57 F6 C2 03 7C 78 2F CC 8B 3E E4 0B
E0 4D 73 80 FF 2A 4D 2F 8D 74 8E DB
```

3.4. Enhanced Voice and Data Privacy Test Vectors

3.4.1. Input Data

Data buffer = B6 2D A2 44 FE 9B 23 AB

Vector 1:

CMEA key k0,. .k7 = a0 7b 1c d1 02 75 69 14
sync = 3d 00 a2 00

Vector 2:

CMEA key k0,. .k7 = F0 06 A8 5A 05 CD B3 2A
sync = ff 00 ff 00

3.4.2. Test Program

3.4.2.1. Main program file

```
/*
EPE test file "main.c"

Explicitly contains code for generating vector sets 1 (DTC key
schedule) and 2 (DCCH key schedule). These first two sets also test
SCEMA. The key schedules are needed for generating the remaining
vector sets. However,none of the remaining sets depend upon other sets
being generated.
*/

#include <stdio.h>

#include "cave.h"

#include "scema.h"

void pause(void)
{
   printf("Enter to continue\n");
   getchar();
}

void main(void)
{
   unsigned int i;

/* test plaintext */

   const unsigned char buf[8] = {0xb6,0x2d,0xa2,0x44,
                                 0xfe,0x9b,0x23, 0xab};
   unsigned char testBuf[MaxMessageOctetSize];
   unsigned char testBufTwo[MaxFrameOctetSize];
```

```
/* cryptosync (meaning is air interface specific) */

unsigned char sync1[4] = { 0x3d,0x00,0xa2,0x00 };
unsigned char sync2[4] = { 0xff,0x00,0xff,0x00 };

/* vector set 1 */

cmeakey[0] = 0xA0;
cmeakey[1] = 0x7B;
cmeakey[2] = 0x1C;
cmeakey[3] = 0xD1;
cmeakey[4] = 0x02;
cmeakey[5] = 0x75;
cmeakey[6] = 0x69;
cmeakey[7] = 0x14;

printf("\nVector Set 1 - DTC Key Generation and SCEMA\n\n");

DTC_Key_Generation();

printf("                DTC CMEA key =");
for (i = 0; i < 8; i++)
   printf(" %02x",(unsigned int)cmeakey[i]);
printf("\n");

printf("     DTC scemaKey (CaveKey1) =");
for (i = 0; i < 8; i++)
   printf(" %02x",(unsigned int)(dtcScheds)->scemaKey[i]);
printf("\n");

printf("                        sync =");
printf(" %02x %02x %02x %02x\n",(unsigned int)sync1[0],
    (unsigned int)sync1[1],(unsigned int)sync1[2],
    (unsigned int)sync1[3]);

for (i = 0; i < SixOctets; i++)
   testBuf[i] = buf[i];

printf("                       Input =");
for (i = 0; i < SixOctets; i++)
   printf(" %02x",(unsigned int)testBuf[i]);
printf("\n");

SCEMA(testBuf,SixOctets,sync1,0,0,ENCRYPTING,dtcScheds);

printf("            DTC SCEMA Output =");
for (i = 0; i < SixOctets; i++)
   printf(" %02x",(unsigned int)testBuf[i]);
printf("\n");

pause();

/* vector set 2 */

cmeakey[0] = 0xf0;
cmeakey[1] = 0x06;
cmeakey[2] = 0xa8;
cmeakey[3] = 0x5a;
cmeakey[4] = 0x05;
cmeakey[5] = 0xcd;
```

```
   cmeakey[6] = 0xb3;
   cmeakey[7] = 0x2a;

   printf("\nVector Set 2 - DCCH Key Generation and SCEMA\n\n");

   DCCH_Key_Generation();

   printf("                DCCH CMEA key =");
   for (i = 0; i < 8; i++)
      printf(" %02x",(unsigned int)cmeakey[i]);
   printf("\n");

   printf("      DCCH scemaKey (CaveKey1)=");
   for (i = 0; i < 8; i++)
      printf(" %02x",(unsigned int)(dcchScheds)->scemaKey[i]);
   printf("\n");

   printf("                         sync =");
   printf(" %02x %02x %02x %02x\n",(unsigned int)sync2[0],
      (unsigned int)sync2[1],(unsigned int)sync2[2],
      (unsigned int)sync2[3]);

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   printf("                        Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   SCEMA(testBuf,SixOctets,sync2,0,0,ENCRYPTING,dcchScheds);

   printf("            DCCH SCEMA Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   pause();
/*
Note: None of these remaining tests are mutually dependent, and can
thus be selectively disabled.
*/
/* Vector Set 3 - SCEMA KSG */
#include "vs3scemaKSG.h"

/* Vector Set 4 - Long Block Encryptor */
#include "vs4longBlock.h"

/* Vector Set 5 - Short Block Encryptor */
#include "vs5shortBlock.h"

/* Vector Set 6 - Enhanced Message Encryption */
#include "vs6enhMsgEnc.h"

/* Vector Set 7 - Enhanced Voice Privacy */
#include "vs7enhVoicePriv.h"

/* Vector Set 8 - Enhanced Data Mask Generation */
#include "vs8enhDataMask.h"

}
```

3.4.2.2. Vector set 3

```
/* Vector Set 3 - SCEMA KSG "vs3scemaKSG.h" */

   printf("\nVector Set 3 - SCEMA KSG\n\n");

   /* Voice content, Reverse Channel, 3-octet input, 8-octet output */
   printf("\nVoice content, Reverse Channel, 3-octet input, 8-octet
output\n\n");

   for (i = 0; i < ThreeOctets; i++)
      testBuf[i] = buf[i];

   printf("           Input =");
   for (i = 0; i < ThreeOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   SCEMA_KSG(testBufTwo,EightOctets,testBuf,ThreeOctets,
            VoiceContent,dtcScheds,ReverseChannel);


   printf("SCEMA KSG Output =");
   for (i = 0; i < EightOctets; i++)
      printf(" %02x",(unsigned int)testBufTwo[i]);
   printf("\n\n");


   /* Voice content, Reverse Channel, 6-octet input, 6-octet output */
   printf("\nVoice content, Reverse Channel, 6-octet input, 6-octet
output\n\n");

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   printf("           Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   SCEMA_KSG(testBufTwo,SixOctets,testBuf,SixOctets,
            VoiceContent,dtcScheds,ReverseChannel);


   printf("SCEMA KSG Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBufTwo[i]);
   printf("\n\n");


   /*
   Voice content, Reverse Channel, 6-octet input,
   3-octet requested output, 6 octets delivered
   */
   printf("\nVoice content, Reverse Channel, 6-octet input,\n");
   printf(" 3-octet requested output, 6-octets delivered\n\n");

   for (i = 0; i < SixOctets; i++)
```

```
      testBuf[i] = buf[i];

   printf("           Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   SCEMA_KSG(testBufTwo,ThreeOctets,testBuf,SixOctets,
            VoiceContent,dtcScheds,ReverseChannel);


   printf("SCEMA KSG Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBufTwo[i]);
   printf("\n\n");

   pause();


   printf("\nVector Set 3 - SCEMA KSG cont'd\n\n");

   /* Message content, Reverse Channel, 6-octet input, 6-octet output */
   printf("\nMessage content, Reverse Channel, 6-octet input, 6-octet
output\n\n");

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   printf("           Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   SCEMA_KSG(testBufTwo,SixOctets,testBuf,SixOctets,
            MessageContent,dtcScheds,ReverseChannel);

   printf("SCEMA KSG Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBufTwo[i]);
   printf("\n\n");

   /* Message content, Forward Channel, 6-octet input, 6-octet output */

   printf("\nMessage content, Forward Channel, 6-octet input, 6-octet
output\n\n");

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   printf("           Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   SCEMA_KSG(testBufTwo,SixOctets,testBuf,SixOctets,
            MessageContent,dtcScheds,ForwardChannel);

   printf("SCEMA KSG Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBufTwo[i]);
```

```
   printf("\n\n");


   pause();
```

3.4.2.3. Vector set 4

```
/* Vector Set 4 - Long Block Encryptor "vs4longBlock.h" */

   printf("\nVector Set 4 - Long Block Encryptor\n\n");

   /* Encryption/Decryption (Voice content, Reverse Channel) */
   printf("\nEncryption/Decryption (Voice content, Reverse
Channel)\n\n");

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   printf("                      Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   Long_Block_Encryptor(testBuf,VoiceContent,ENCRYPTING,
                  dtcScheds,ReverseChannel);


   printf("Long Block Encryptor Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   Long_Block_Encryptor(testBuf,VoiceContent,DECRYPTING,
                  dtcScheds,ReverseChannel);

   printf("Long Block Decryptor Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n\n");


   /* Encryption (Message Content,Reverse Channel) */
   printf("\nEncryption (Message Content,Reverse Channel)\n\n");

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   printf("                      Input =");  for (i = 0; i < SixOctets;
i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   Long_Block_Encryptor(testBuf,MessageContent,ENCRYPTING,
                  dtcScheds,ReverseChannel);
```

```
   printf("Long Block Encryptor Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n\n");


   /* Encryption (Voice Content,Forward Channel) */
   printf("\nEncryption (Voice Content,Forward Channel)\n\n");

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   printf("                      Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   Long_Block_Encryptor(testBuf,VoiceContent,ENCRYPTING,
                    dtcScheds,ForwardChannel);


   printf("Long Block Encryptor Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n\n");

   pause();
```

3.4.2.4. Vector set 5

```
/* Vector Set 5 - Short Block Encryptor "vs5shortBlock.h"

Note: The last octets of the decrypted buffers may not match the
original input buffers' last octets. This is legitimate and comprises a
test to ensure that the output clean up code is working to zero out non-
content bearing bits.
*/

   printf("\n\nVector Set 5 - Short Block Encryptor\n");


   /* Encryption/Decryption (47 bits,Voice content,Reverse Channel) */

   printf("\nEncryption/Decryption (47 bits, Voice content,Reverse
Channel)\n\n");

   for (i = 0; i < SixOctets; i++)
   {
      testBuf[i] = buf[i];
      testBufTwo[i] = buf[i + 1];
   }

   printf(" SB Data Mask Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");
```

```
   Short_Block_Encryptor(testBuf,47,VoiceContent,testBufTwo,
                         ENCRYPTING,dtcScheds,ReverseChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Short_Block_Encryptor(testBuf,47,VoiceContent,testBufTwo,
                         DECRYPTING,dtcScheds,ReverseChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   /* Encryption/Decryption (17 bits,Voice content,Reverse Channel) */

   printf("\nEncryption/Decryption (17 bits, Voice content,Reverse
Channel\n\n");

   for (i = 0; i < SixOctets; i++)
   {
      testBuf[i] = buf[i];
      testBufTwo[i] = buf[i + 1];
   }

   printf(" SB Data Mask Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Short_Block_Encryptor(testBuf,17,VoiceContent,testBufTwo,
                         ENCRYPTING,dtcScheds,ReverseChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Short_Block_Encryptor(testBuf,17,VoiceContent,testBufTwo,
                         DECRYPTING,dtcScheds,ReverseChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   pause();


   /* Encryption/Decryption (16 bits,Voice content,Reverse Channel) */

   printf("\nEncryption/Decryption (16 bits, Voice content,Reverse
Channel\n\n");

   for (i = 0; i < SixOctets; i++)
   {
      testBuf[i] = buf[i];
```

```
      testBufTwo[i] = buf[i + 1];
   }

   printf(" SB Data Mask Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Short_Block_Encryptor(testBuf,16,VoiceContent,testBufTwo,
                  ENCRYPTING,dtcScheds,ReverseChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Short_Block_Encryptor(testBuf,16,VoiceContent,testBufTwo,
                  DECRYPTING,dtcScheds,ReverseChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   /* Encryption/Decryption (2 bits,Voice content,Reverse Channel) */

   printf("\nEncryption/Decryption (2 bits, Voice content,Reverse
Channel\n\n");

   for (i = 0; i < SixOctets; i++)
   {
      testBuf[i] = buf[i];
      testBufTwo[i] = buf[i + 1];
   }

   printf(" SB Data Mask Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Short_Block_Encryptor(testBuf,2,VoiceContent,testBufTwo,
                  ENCRYPTING,dtcScheds,ReverseChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Short_Block_Encryptor(testBuf,2,VoiceContent,testBufTwo,
                  DECRYPTING,dtcScheds,ReverseChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   pause();


   /* Encryption,47 bits,Voice content,Forward Channel */
```

```
   printf("\nEncryption,47 bits,Voice content,Forward Channel\n\n");

   for (i = 0; i < SixOctets; i++)
   {
      testBuf[i] = buf[i];
      testBufTwo[i] = buf[i + 1];
   }

   printf(" SB Data Mask Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Short_Block_Encryptor(testBuf,47,VoiceContent,testBufTwo,
                        ENCRYPTING,dtcScheds,ForwardChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   /* Encryption,47 bits,Message content,Forward Channel */

   printf("\nEncryption,47 bits,Message content,Forward Channel\n\n");

   for (i = 0; i < SixOctets; i++)
   {
      testBuf[i] = buf[i];
      testBufTwo[i] = buf[i + 1];
   }

   printf(" SB Data Mask Input =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Short_Block_Encryptor(testBuf,47,MessageContent,testBufTwo,
                        ENCRYPTING,dtcScheds,ForwardChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   /*
   Encryption,47 bits,Message content,Forward Channel,different entropy
*/

   printf("\nEncryption,47 bits,Message content,Forward
Channel,different entropy\n\n");

   for (i = 0; i < SixOctets; i++)
   {
      testBuf[i] = buf[i];
      testBufTwo[i] = ~buf[i + 1];
   }

   printf(" SB Data Mask Input =");
```

```
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Short_Block_Encryptor(testBuf,47,MessageContent,testBufTwo,
                  ENCRYPTING,dtcScheds,ForwardChannel);

   printf("SB Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   pause();
```

3.4.2.5. Vector set 6

```
/* Vector Set 6 - Enhanced Message Encryption "vs6enhMsgEnc.h"

Note: The last octets of the decrypted buffers may not match the
original input buffers' last octets. This is legitimate and comprises a
test to ensure that the output clean up code is working to zero out non-
content bearing bits.
*/

   printf("\n\nVector Set 6 - Enhanced Message Encryption\n");

   /* 48 bits */

   printf("\n48 bits\n\n");

   printf("     Message input =");

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   for (i = 0; i < 4; i++)
      testBufTwo[i] = ~buf[i];


   /* Encrypting */
   Enhanced_Message_Encryption(testBuf,48,DCCH,testBufTwo,TestMsgType,
                  ENCRYPTING,CAVEKey1,ForwardChannel);


   printf("  Encryptor output =");

   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");


   /* Decrypting */
   Enhanced_Message_Encryption(testBuf,48,DCCH,testBufTwo,TestMsgType,
                  DECRYPTING,CAVEKey1,ForwardChannel);
```

```
    printf("  Decryptor output =");

    for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf("\n");


    pause();


    /* 256 Octets (2047 bits) */

    printf("\n256 Octets (2047 bits)\n\n");

    printf("   Last P/O Message input =");

    for (i = 0; i < 256; i++)
       testBuf[i] = buf[i % EightOctets];

    for (i = 0; i < EightOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i + 248]);
    printf("\n");

    for (i = 0; i < 4; i++)
       testBufTwo[i] = ~buf[i];


    /* Encrypting */

    Enhanced_Message_Encryption(testBuf,2047,DCCH,testBufTwo,
       TestMsgType,ENCRYPTING,CAVEKey1,ForwardChannel);


    printf("Last P/O Encryptor output =");

    for (i = 0; i < EightOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i + 248]);
    printf("\n");


    /* Decrypting */

    Enhanced_Message_Encryption(testBuf,2047,DCCH,testBufTwo,
       TestMsgType,DECRYPTING,CAVEKey1,ForwardChannel);


    printf("Last P/O Decryptor output =");

    for (i = 0; i < EightOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i + 248]);
    printf("\n");


    pause();


    /* 44 bits */

    printf("\n44 bits\n\n");
```

```
    printf("     Message input =");

    for (i = 0; i < SixOctets; i++)
       testBuf[i] = buf[i];

    for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf("\n");

    for (i = 0; i < 4; i++)
       testBufTwo[i] = ~buf[i];


    /* Encrypting */
    Enhanced_Message_Encryption(testBuf,44,DCCH,testBufTwo,TestMsgType,
                      ENCRYPTING,CAVEKey1,ForwardChannel);


    printf("  Encryptor output =");

    for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf("\n");

    /* Decrypting */
    Enhanced_Message_Encryption(testBuf,44,DCCH,testBufTwo,TestMsgType,
                      DECRYPTING,CAVEKey1,ForwardChannel);


    printf("  Decryptor output =");

    for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf("\n");


    pause();

    /* 48 bits, Forward Channel -> Reverse Channel */

    printf("\n48 bits, Forward Channel -> Reverse Channel\n\n");

    printf("     Message input =");

    for (i = 0; i < SixOctets; i++)
       testBuf[i] = buf[i];

 for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf("\n");

    for (i = 0; i < 4; i++)
       testBufTwo[i] = ~buf[i];


    /* Encrypting */
    Enhanced_Message_Encryption(testBuf,48,DCCH,testBufTwo,TestMsgType,
                      ENCRYPTING,CAVEKey1,ReverseChannel);

    printf("  Encryptor output =");

```

```
    for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf("\n");

    /* 48 bits, DCCH -> DTC */

    printf("\n48 bits, DCCH -> DTC\n\n");

    printf("     Message input =");

    for (i = 0; i < SixOctets; i++)
       testBuf[i] = buf[i];

    for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf("\n");

    for (i = 0; i < 4; i++)
       testBufTwo[i] = ~buf[i];

    /* Encrypting */
    Enhanced_Message_Encryption(testBuf,48,DTC,testBufTwo,TestMsgType,
                      ENCRYPTING,CAVEKey1,ForwardChannel);


    printf("  Encryptor output =");

    for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf("\n");

    /* 48 bits, different RAND */

    printf("\n48 bits, different RAND\n\n");

    printf("     Message input =");

    for (i = 0; i < SixOctets; i++)
       testBuf[i] = buf[i];

    for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf("\n");

    for (i = 0; i < 4; i++)
       testBufTwo[i] = buf[i];

    /* Encrypting */
    Enhanced_Message_Encryption(testBuf,48,DCCH,testBufTwo,TestMsgType,
                      ENCRYPTING,CAVEKey1,ForwardChannel);

    printf("  Encryptor output =");

    for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf("\n");

    /* 44 bits, different RAND */

    printf("\n44 bits, different RAND\n\n");
```

```
   printf("     Message input =");

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   for (i = 0; i < 4; i++)
      testBufTwo[i] = buf[i];

   /* Encrypting */
   Enhanced_Message_Encryption(testBuf,44,DCCH,testBufTwo,TestMsgType,
                    ENCRYPTING,CAVEKey1,ForwardChannel);

   printf("  Encryptor output =");

   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   /* 48 bits, different Message Type */

   printf("\n48 bits, different Message Type\n\n");

   printf("     Message input =");

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   for (i = 0; i < 4; i++)
      testBufTwo[i] = ~buf[i];

/* Encrypting */
   Enhanced_Message_Encryption(testBuf,48,DCCH,testBufTwo,
                              TestMsgType2, ENCRYPTING, CAVEKey1,
                              ForwardChannel);

   printf("  Encryptor output =");

   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   pause();
```

3.4.2.6. Vector set 7

```
/* Vector Set 7 - Enhanced Voice Privacy "vs7enhVoicePriv.h"

Note 1: The current coder standards' bit allocations as listed in
TIA/EIA-136-510 are: The Number of {Class 1A bits, remaining bits, CRC
bits} for 136 speech coders are: 136-410 ACELP {48, 100, 7}, 136-420
VSELP {12, 147, 7}, and 136-430 US1 {81, 163, 8}.
```

```
Note 2: The last octets of the decrypted buffers may not match the
original input buffers' last octets. This is legitimate and comprises a
test to ensure that the output clean up code is working to zero out non-
content bearing bits.
*/

   printf("\n\nVector Set 7 - Enhanced Voice Privacy\n");

   /* 48 Class 1A bits, 100 remaining bits */

   printf("\n48 Class 1A bits, 100 remaining bits\n\n");

   printf("1A/Rem. bits input =");

   for (i = 0; i < SixOctets; i++)
      testBuf[i] = buf[i];

   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf(" /");

   for (i = 0; i < ((100 - 1) / 8) + 1; i++)
      testBufTwo[i] = ~buf[i % EightOctets];

   for (i = 0; i < ((100 - 1) / 8) + 1; i++)
      printf(" %02x",(unsigned int)testBufTwo[i]);
   printf("\n");


   /* Encrypting */
   Enhanced_Voice_Privacy(CoderVersionZero,testBuf,48,testBufTwo,100,
                      ENCRYPTING,CAVEKey1,ForwardChannel);

   printf("  Encryptor output =");

   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf(" /");

   for (i = 0; i < ((100 - 1) / 8) + 1; i++)
      printf(" %02x",(unsigned int)testBufTwo[i]);
   printf("\n");

   /* Decrypting */
   Enhanced_Voice_Privacy(CoderVersionZero,testBuf,48,testBufTwo,100,
                      DECRYPTING,CAVEKey1,ForwardChannel);

   printf("  Decryptor output =");

   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf(" /");

   for (i = 0; i < ((100 - 1) / 8) + 1; i++)
      printf(" %02x",(unsigned int)testBufTwo[i]);
   printf("\n");

   pause();

   /* 81 Class 1A bits, 163 remaining bits */
```

```
    printf("\n81 Class 1A bits, 163 remaining bits\n\n");

    printf("1A/Rem. bits input =");

    for (i = 0; i < ((81 - 1) / 8) + 1; i++)
       testBuf[i] = buf[i % EightOctets];

    for (i = 0; i < ((81 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf(" /");

    for (i = 0; i < ((163 - 1) / 8) + 1; i++)
       testBufTwo[i] = ~buf[i % EightOctets];

    for (i = 0; i < ((163 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBufTwo[i]);
    printf("\n");

    /* Encrypting */
    Enhanced_Voice_Privacy(CoderVersionZero,testBuf,81,testBufTwo,163,
                      ENCRYPTING,CAVEKey1,ForwardChannel);

    printf("  Encryptor output =");

    for (i = 0; i < ((81 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf(" /");

    for (i = 0; i < ((163 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBufTwo[i]);
    printf("\n");

    /* Decrypting */
    Enhanced_Voice_Privacy(CoderVersionZero,testBuf,81,testBufTwo,163,
                      DECRYPTING,CAVEKey1,ForwardChannel);

    printf("  Decryptor output =");

    for (i = 0; i < ((81 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf(" /");

    for (i = 0; i < ((163 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBufTwo[i]);
    printf("\n");

    pause();

    /* 12 Class 1A bits, 147 remaining bits */

    printf("\n12 Class 1A bits, 147 remaining bits\n\n");

    printf("1A/Rem. bits input =");

    for (i = 0; i < ((12 - 1) / 8) + 1; i++)
       testBuf[i] = buf[i % EightOctets];

    for (i = 0; i < ((12 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf(" /");
```

```
    for (i = 0; i < ((147 - 1) / 8) + 1; i++)
       testBufTwo[i] = ~buf[i % EightOctets];

    for (i = 0; i < ((147 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBufTwo[i]);
    printf("\n");

    /* Encrypting */
    Enhanced_Voice_Privacy(CoderVersionZero,testBuf,12,testBufTwo,147,
                      ENCRYPTING,CAVEKey1,ForwardChannel);

    printf("  Encryptor output =");

    for (i = 0; i < ((12 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf(" /");

    for (i = 0; i < ((147 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBufTwo[i]);
    printf("\n");

    /* Decrypting */
    Enhanced_Voice_Privacy(CoderVersionZero,testBuf,12,testBufTwo,147,
                      DECRYPTING,CAVEKey1,ForwardChannel);

    printf("  Decryptor output =");

    for (i = 0; i < ((12 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf(" /");
    for (i = 0; i < ((147 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBufTwo[i]);
    printf("\n");

    pause();

    /* Reverse Channel, 48 Class 1A bits, 100 remaining bits */

    printf("\nReverse Channel, 48 Class 1A bits, 100 remaining
bits\n\n");

    printf("1A/Rem. bits input =");

    for (i = 0; i < SixOctets; i++)
       testBuf[i] = buf[i];

    for (i = 0; i < SixOctets; i++)
       printf(" %02x",(unsigned int)testBuf[i]);
    printf(" /");

    for (i = 0; i < ((100 - 1) / 8) + 1; i++)
       testBufTwo[i] = ~buf[i % EightOctets];

    for (i = 0; i < ((100 - 1) / 8) + 1; i++)
       printf(" %02x",(unsigned int)testBufTwo[i]);
    printf("\n");

    /* Encrypting */
    Enhanced_Voice_Privacy(CoderVersionZero,testBuf,48,testBufTwo,100,
                      ENCRYPTING,CAVEKey1,ReverseChannel);
```

```
   printf("  Encryptor output =");

   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf(" /");

   for (i = 0; i < ((100 - 1) / 8) + 1; i++)
      printf(" %02x",(unsigned int)testBufTwo[i]);
   printf("\n");
```

3.4.2.7. Vector set 8

```
/* Vector Set 8 - Enhanced Data Mask Generation "vs8enhDataMask.h" */

   printf("\nVector Set 8 - Enhanced Data Mask Generation\n\n");

   Enhanced_Data_Mask(testBuf, 0x87654321, SixOctets, CAVEKey1);

   printf("Enhanced Data Mask Output =");
   for (i = 0; i < SixOctets; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n");

   Enhanced_Data_Mask(testBuf, 0x87654321, 3, CAVEKey1);

   printf("   Output,with short Mask =");
   for (i = 0; i < 3; i++)
      printf(" %02x",(unsigned int)testBuf[i]);
   printf("\n\n");

   pause();
```

3.4.3. Test Program Input and Output

```
Vector Set 1 - DTC Key Generation and SCEMA

           DTC CMEA key = a0 7b 1c d1 02 75 69 14
DTC scemaKey (CaveKey1) = 5d ed ad 53 5b 4a b9 fc
                   sync = 3d 00 a2 00
                  Input = b6 2d a2 44 fe 9b
       DTC SCEMA Output = 63 f0 21 7a 3c 97


Vector Set 2 - DCCH Key Generation and SCEMA

          DCCH CMEA key = f0 06 a8 5a 05 cd b3 2a
          DCCH CMEA key = f0 06 a8 5a 05 cd b3 2a
DCCH scemaKey (CaveKey1)= b6 df 9a d0 6e 5a 3d 14
DCCH scemaKey (CaveKey1)= b6 df 9a d0 6e 5a 3d 14
                   sync = ff 00 ff 00
                  Input = b6 2d a2 44 fe 9b
      DCCH SCEMA Output = 4c 3d 77 13 e9 a0
```

Vector Set 3 - SCEMA KSG

Voice content, Reverse Channel, 3-octet input, 8-octet output

```
             Input = b6 2d a2
SCEMA KSG Output = f4 bc 1e 9b 27 a1 54 fa
```

Voice content, Reverse Channel, 6-octet input, 6-octet output

```
             Input = b6 2d a2 44 fe 9b
SCEMA KSG Output = 26 08 0c fa d2 7d
```

Voice content, Reverse Channel, 6-octet input, 3-octet requested output, 6-octets delivered

```
             Input = b6 2d a2 44 fe 9b
SCEMA KSG Output = 26 08 0c fa d2 7d
```

Message content, Reverse Channel, 6-octet input, 6-octet output

```
             Input = b6 2d a2 44 fe 9b
SCEMA KSG Output = df 39 6c 92 c8 63
```

Message content, Forward Channel, 6-octet input, 6-octet output

```
             Input = b6 2d a2 44 fe 9b
SCEMA KSG Output = 8c a4 9a f5 54 53
```

Vector Set 4 - Long Block Encryptor

Encryption/Decryption (Voice content, Reverse Channel)

```
                      Input = b6 2d a2 44 fe 9b
Long Block Encryptor Output = 59 fe 84 59 ec 18
Long Block Decryptor Output = b6 2d a2 44 fe 9b
```

Encryption (Message Content,Reverse Channel)

```
                      Input = b6 2d a2 44 fe 9b
Long Block Encryptor Output = 53 7e d4 c6 37 98
```

Encryption (Voice Content,Forward Channel)

```
                      Input = b6 2d a2 44 fe 9b
Long Block Encryptor Output = bd 5e 36 a5 8c 07
```

Vector Set 5 - Short Block Encryptor

Encryption/Decryption (47 bits,Voice content,Reverse Channel)

```
 SB Data Mask Input = b6 2d a2 44 fe 9b
SB Encryptor Output = af f8 41 7e 5d f2
SB Decryptor Output = b6 2d a2 44 fe 9a
```

Encryption/Decryption (17 bits, Voice content,Reverse Channel

```
 SB Data Mask Input = b6 2d a2 44 fe 9b
SB Encryptor Output = b7 ed 80 00 00 00
SB Decryptor Output = b6 2d 80 00 00 00
```

Encryption/Decryption (16 bits, Voice content,Reverse Channel

```
 SB Data Mask Input = b6 2d a2 44 fe 9b
SB Encryptor Output = 9b a8 00 00 00 00
SB Decryptor Output = b6 2d 00 00 00 00
```

Encryption/Decryption (2 bits, Voice content,Reverse Channel

```
 SB Data Mask Input = b6 2d a2 44 fe 9b
SB Encryptor Output = 00 00 00 00 00 00
SB Decryptor Output = 80 00 00 00 00 00
```

Encryption,47 bits,Voice content,Forward Channel

```
 SB Data Mask Input = b6 2d a2 44 fe 9b
SB Encryptor Output = 9e df 05 a8 43 34
```

Encryption,47 bits,Message content,Forward Channel

```
 SB Data Mask Input = b6 2d a2 44 fe 9b
SB Encryptor Output = 4f 89 f7 09 29 a8
```

Encryption,47 bits,Message content,Forward Channel,different entropy

```
 SB Data Mask Input = b6 2d a2 44 fe 9b
SB Encryptor Output = c8 fe da 7d 87 da
```

Vector Set 6 - Enhanced Message Encryption

48 bits

```
    Message input = b6 2d a2 44 fe 9b
 Encryptor output = 87 ce 86 a0 f1 86
 Decryptor output = b6 2d a2 44 fe 9b
```

256 Octets (2047 bits)

```
   Last P/O Message input = b6 2d a2 44 fe 9b 23 ab
Last P/O Encryptor output = 2b 52 46 a6 da 82 f2 f0
Last P/O Decryptor output = b6 2d a2 44 fe 9b 23 aa
```

44 bits

```
    Message input = b6 2d a2 44 fe 9b
 Encryptor output = b4 5b 16 d1 c2 10
 Decryptor output = b6 2d a2 44 fe 90
```

48 bits, Forward Channel -> Reverse Channel

```
    Message input = b6 2d a2 44 fe 9b
 Encryptor output = 28 09 3e fe 49 06
```

48 bits, DCCH -> DTC

```
    Message input = b6 2d a2 44 fe 9b
 Encryptor output = 28 a4 ed a0 68 0a
```

48 bits, different RAND

```
    Message input = b6 2d a2 44 fe 9b
 Encryptor output = 3c cf 9e 23 a5 7c
```

44 bits, different RAND

```
    Message input = b6 2d a2 44 fe 9b
 Encryptor output = a7 03 f3 42 2b 10
```

48 bits, different Message Type

```
    Message input = b6 2d a2 44 fe 9b
 Encryptor output = dc 27 53 82 d5 77
```

Vector Set 7 - Enhanced Voice Privacy

48 Class 1A bits, 100 remaining bits

```
1A/Rem. bits input = b6 2d a2 44 fe 9b / 49 d2 5d bb 01 64 dc 54 49 d2
5d bb 01
  Encryptor output = bd 5e 36 a5 8c 07 / 87 58 05 c7 38 37 0f 68 e2 3f
d4 5c 30
  Decryptor output = b6 2d a2 44 fe 9b / 49 d2 5d bb 01 64 dc 54 49 d2
5d bb 00
```

81 Class 1A bits, 163 remaining bits

```
1A/Rem. bits input = b6 2d a2 44 fe 9b 23 ab b6 2d a2 / 49 d2 5d bb 01
64 dc 54 49 d2 5d bb 01 64 dc 54 49 d2 5d bb 01
  Encryptor output = 5b 68 57 98 42 83 81 92 b2 1f 80 / e8 52 c6 f6 60
39 16 a0 80 c7 b0 59 fb 5c 6e 23 91 08 bc d2 a0
  Decryptor output = b6 2d a2 44 fe 9b 23 ab b6 2d 80 / 49 d2 5d bb 01
64 dc 54 49 d2 5d bb 01 64 dc 54 49 d2 5d bb 00
```

12 Class 1A bits, 147 remaining bits

```
1A/Rem. bits input = b6 2d / 49 d2 5d bb 01 64 dc 54 49 d2 5d bb 01 64
dc 54 49 d2 5d
  Encryptor output = ed 20 / 5b 3c 7e 6a 21 18 f1 69 82 87 f7 d8 92 51
c3 f9 d6 db e0
  Decryptor output = b6 20 / 49 d2 5d bb 01 64 dc 54 49 d2 5d bb 01 64
dc 54 49 d2 40
```

Reverse Channel, 48 Class 1A bits, 100 remaining bits

```
1A/Rem. bits input = b6 2d a2 44 fe 9b / 49 d2 5d bb 01 64 dc 54 49 d2
5d bb 01
  Encryptor output = 59 fe 84 59 ec 18 / e0 6d a0 79 0a 89 6a 05 7d 2a
a3 19 e0
```

Vector Set 8 - Enhanced Data Mask Generation

```
Enhanced Data Mask Output = 45 b0 15 31 d6 e0
  Output, with short mask = 45 b0 15
```