Часть V

Исходные коды

1. **DES**
2. **LOKI91**
3. **IDEA**
4. **GOST**
5. **BLOWFISH**
6. **3-WAY**
7. **RC5**
8. **A5**
9. **SEAL**

DES

```
#define EN0   0      /* MODE == encrypt */
#define DE1   1      /* MODE == decrypt */

typedef struct {
        unsigned long ek[32];
        unsigned long dk[32];
} des_ctx;

extern void deskey(unsigned char *, short);
/*                    hexkey[8]     MODE
 * Sets the internal key register according to the hexadecimal
 * key contained in the 8 bytes of hexkey, according to the DES,
 * for encryption or decryption according to MODE.
 */

extern void usekey(unsigned long *);
/*                  cookedkey[32]
 * Loads the internal key register with the data in cookedkey.
 */

extern void cpkey(unsigned long *);
/*                 cookedkey[32]
 * Copies the contents of the internal key register into the storage
 * located at &cookedkey[0].
 */

extern void des(unsigned char *, unsigned char *);
/*                  from[8]          to[8]
 * Encrypts/Decrypts (according to the key currently loaded in the
 * internal key register) one block of eight bytes at address 'from'
 * into the block at address 'to'.  They can be the same.
 */

static void scrunch(unsigned char *, unsigned long *);
static void unscrun(unsigned long *, unsigned char *);
static void desfunc(unsigned long *, unsigned long *);
static void cookey(unsigned long *);
```

```
static unsigned long KnL[32] = { 0L };
static unsigned long KnR[32] = { 0L };
static unsigned long Kn3[32] = { 0L };
static unsigned char Df_Key[24] = {
	0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
	0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10,
	0x89,0xab,0xcd,0xef,0x01,0x23,0x45,0x67 };

static unsigned short bytebit[8]    = {
	0200, 0100, 040, 020, 010, 04, 02, 01 };

static unsigned long bigbyte[24] = {
	0x800000L,    0x400000L,     0x200000L,    0x100000L,
	0x80000L,     0x40000L,      0x20000L,     0x10000L,
	0x8000L,      0x4000L,       0x2000L,      0x1000L,
	0x800L,              0x400L,               0x200L,               0x100L,
	0x80L,               0x40L,                0x20L,                0x10L,
	0x8L,         0x4L,          0x2L,         0x1L   };

/* Use the key schedule specified in the Standard (ANSI X3.92-1981). */

static unsigned char pc1[56] = {
	56, 48, 40, 32, 24, 16,  8,   0, 57, 49, 41, 33, 25, 17,
	 9,  1, 58, 50, 42, 34, 26,  18, 10,  2, 59, 51, 43, 35,
	62, 54, 46, 38, 30, 22, 14,   6, 61, 53, 45, 37, 29, 21,
	13,  5, 60, 52, 44, 36, 28,  20, 12,  4, 27, 19, 11,  3 };

static unsigned char totrot[16] = {
	1,2,4,6,8,10,12,14,15,17,19,21,23,25,27,28 };

static unsigned char pc2[48] = {
	13, 16, 10, 23,  0,  4,       2, 27, 14,  5, 20,  9,
	22, 18, 11,  3, 25,  7,      15,  6, 26, 19, 12,  1,
	40, 51, 30, 36, 46, 54,      29, 39, 50, 44, 32, 47,
	43, 48, 38, 55, 33, 52,      45, 41, 49, 35, 28, 31 };

void deskey(key, edf)       /* Thanks to James Gillogly & Phil Karn! */
unsigned char *key;
short edf;
{
	register int i, j, l, m, n;
	unsigned char pc1m[56], pcr[56];
	unsigned long kn[32];

	for ( j = 0; j < 56; j++ ) {
		l = pc1[j];
		m = l & 07;
		pc1m[j] = (key[l >> 3] & bytebit[m]) ? 1 : 0;
		}
	for( i = 0; i < 16; i++ ) {
		if( edf == DE1 ) m = (15 - i) << 1;
		else m = i << 1;
		n = m + 1;
		kn[m] = kn[n] = 0L;
		for( j = 0; j < 28; j++ ) {
			l = j + totrot[i];
			if( l < 28 ) pcr[j] = pc1m[l];
			else pcr[j] = pc1m[l - 28];
			}
		for( j = 28; j < 56; j++ ) {
		    l = j + totrot[i];
```

```
                  if( l < 56 ) pcr[j] = pc1m[l];
                  else pcr[j] = pc1m[l - 28];
                  }
            for( j = 0; j < 24; j++ ) {
                  if( pcr[pc2[j]] ) kn[m] |= bigbyte[j];
                  if( pcr[pc2[j+24]] ) kn[n] |= bigbyte[j];
                  }
            }
      cookey(kn);
      return;
}

static void cookey(raw1)
register unsigned long *raw1;
{
      register unsigned long *cook, *raw0;
      unsigned long dough[32];
      register int i;

      cook = dough;
      for( i = 0; i < 16; i++, raw1++ ) {
            raw0 = raw1++;
            *cook   = (*raw0 & 0x00fc0000L) << 6;
            *cook  |= (*raw0 & 0x00000fc0L) << 10;
            *cook  |= (*raw1 & 0x00fc0000L) >> 10;
            *cook++       |= (*raw1 & 0x00000fc0L) >> 6;
            *cook   = (*raw0 & 0x0003f000L) << 12;
            *cook  |= (*raw0 & 0x0000003fL) << 16;
            *cook  |= (*raw1 & 0x0003f000L) >> 4;
            *cook++       |= (*raw1 & 0x0000003fL);
            }
      usekey(dough);
      return;
}

void cpkey(into)
register unsigned long *into;
{
      register unsigned long *from, *endp;

      from = KnL, endp = &KnL[32];
      while( from < endp ) *into++ = *from++;
      return;
}

void usekey(from)
register unsigned long *from;
{
      register unsigned long *to, *endp;

      to = KnL, endp = &KnL[32];
      while( to < endp ) *to++ = *from++;
      return;
}

void des(inblock, outblock)
unsigned char *inblock, *outblock;
{
      unsigned long work[2];

      scrunch(inblock, work);
      desfunc(work, KnL);
```

```
        unscrun(work, outblock);
        return;
}

static void scrunch(outof, into)
register unsigned char *outof;
register unsigned long *into;
{
        *into   = (*outof++ & 0xffL) << 24;
        *into  |= (*outof++ & 0xffL) << 16;
        *into  |= (*outof++ & 0xffL) << 8;
        *into++ |= (*outof++ & 0xffL);
        *into   = (*outof++ & 0xffL) << 24;
        *into  |= (*outof++ & 0xffL) << 16;
        *into  |= (*outof++ & 0xffL) << 8;
        *into  |= (*outof   & 0xffL);
        return;
}

static void unscrun(outof, into)
register unsigned long *outof;
register unsigned char *into;
{
        *into++ = (*outof >> 24) & 0xffL;
        *into++ = (*outof >> 16) & 0xffL;
        *into++ = (*outof >>  8) & 0xffL;
        *into++ =  *outof++           & 0xffL;
        *into++ = (*outof >> 24) & 0xffL;
        *into++ = (*outof >> 16) & 0xffL;
        *into++ = (*outof >>  8) & 0xffL;
        *into   =  *outof     & 0xffL;
        return;
}

static unsigned long SP1[64] = {
        0x01010400L, 0x00000000L, 0x00010000L, 0x01010404L,
        0x01010004L, 0x00010404L, 0x00000004L, 0x00010000L,
        0x00000400L, 0x01010400L, 0x01010404L, 0x00000400L,
        0x01000404L, 0x01010004L, 0x01000000L, 0x00000004L,
        0x00000404L, 0x01000400L, 0x01000400L, 0x00010400L,
        0x00010400L, 0x01010000L, 0x01010000L, 0x01000404L,
        0x00010004L, 0x01000004L, 0x01000004L, 0x00010004L,
        0x00000000L, 0x00000404L, 0x00010404L, 0x01000000L,
        0x00010000L, 0x01010404L, 0x00000004L, 0x01010000L,
        0x01010400L, 0x01000000L, 0x01000000L, 0x00000400L,
        0x01010004L, 0x00010000L, 0x00010400L, 0x01000004L,
        0x00000400L, 0x00000004L, 0x01000404L, 0x00010404L,
        0x01010404L, 0x00010004L, 0x01010000L, 0x01000404L,
        0x01000004L, 0x00000404L, 0x00010404L, 0x01010400L,
        0x00000404L, 0x01000400L, 0x01000400L, 0x00000000L,
        0x00010004L, 0x00010400L, 0x00000000L, 0x01010004L };

static unsigned long SP2[64] = {
        0x80108020L, 0x80008000L, 0x00008000L, 0x00108020L,
        0x00100000L, 0x00000020L, 0x80100020L, 0x80008020L,
        0x80000020L, 0x80108020L, 0x80108000L, 0x80000000L,
        0x80008000L, 0x00100000L, 0x00000020L, 0x80100020L,
        0x00108000L, 0x00100020L, 0x80008020L, 0x00000000L,
        0x80000000L, 0x00008000L, 0x00108020L, 0x80100000L,
        0x00100020L, 0x80000020L, 0x00000000L, 0x00108000L,
        0x00008020L, 0x80108000L, 0x80100000L, 0x00008020L,
        0x00000000L, 0x00108020L, 0x80100020L, 0x00100000L,
```

```
	0x80008020L, 0x80100000L, 0x80108000L, 0x00008000L,
	0x80100000L, 0x80008000L, 0x00000020L, 0x80108020L,
	0x00108020L, 0x00000020L, 0x00008000L, 0x80000000L,
	0x00008020L, 0x80108000L, 0x00100000L, 0x80000020L,
	0x00100020L, 0x80008020L, 0x80000020L, 0x00100020L,
	0x00108000L, 0x00000000L, 0x80008000L, 0x00008020L,
	0x80000000L, 0x80100020L, 0x80108020L, 0x00108000L };

static unsigned long SP3[64] = {
	0x00000208L, 0x08020200L, 0x00000000L, 0x08020008L,
	0x08000200L, 0x00000000L, 0x00020208L, 0x08000200L,
	0x00020008L, 0x08000008L, 0x08000008L, 0x00020000L,
	0x08020208L, 0x00020008L, 0x08020000L, 0x00000208L,
	0x08000000L, 0x00000008L, 0x08020200L, 0x00000200L,
	0x00020200L, 0x08020000L, 0x08020008L, 0x00020208L,
	0x08000208L, 0x00020200L, 0x00020000L, 0x08000208L,
	0x00000008L, 0x08020208L, 0x00000200L, 0x08000000L,
	0x08020200L, 0x08000000L, 0x00020008L, 0x00000208L,
	0x00020000L, 0x08020200L, 0x08000200L, 0x00000000L,
	0x00000200L, 0x00020008L, 0x08020208L, 0x08000200L,
	0x08000008L, 0x00000200L, 0x00000000L, 0x08020008L,
	0x08000208L, 0x00020000L, 0x08000000L, 0x08020208L,
	0x00000008L, 0x00020208L, 0x00020200L, 0x08000008L,
	0x08020000L, 0x08000208L, 0x00000208L, 0x08020000L,
	0x00020208L, 0x00000008L, 0x08020008L, 0x00020200L };

static unsigned long SP4[64] = {
	0x00802001L, 0x00002081L, 0x00002081L, 0x00000080L,
	0x00802080L, 0x00800081L, 0x00800001L, 0x00002001L,
	0x00000000L, 0x00802000L, 0x00802000L, 0x00802081L,
	0x00000081L, 0x00000000L, 0x00800080L, 0x00800001L,
	0x00000001L, 0x00002000L, 0x00800000L, 0x00802001L,
	0x00000080L, 0x00800000L, 0x00002001L, 0x00002080L,
	0x00800081L, 0x00000001L, 0x00002080L, 0x00800080L,
	0x00002000L, 0x00802080L, 0x00802081L, 0x00000081L,
	0x00800080L, 0x00800001L, 0x00802000L, 0x00802081L,
	0x00000081L, 0x00000000L, 0x00000000L, 0x00802000L,
	0x00002080L, 0x00800080L, 0x00800081L, 0x00000001L,
	0x00802001L, 0x00002081L, 0x00002081L, 0x00000080L,
	0x00802081L, 0x00000081L, 0x00000001L, 0x00002000L,
	0x00800001L, 0x00002001L, 0x00802080L, 0x00800081L,
	0x00002001L, 0x00002080L, 0x00800000L, 0x00802001L,
	0x00000080L, 0x00800000L, 0x00002000L, 0x00802080L };

static unsigned long SP5[64] = {
	0x00000100L, 0x02080100L, 0x02080000L, 0x42000100L,
	0x00080000L, 0x00000100L, 0x40000000L, 0x02080000L,
	0x40080100L, 0x00080000L, 0x02000100L, 0x40080100L,
	0x42000100L, 0x42080000L, 0x00080100L, 0x40000000L,
	0x02000000L, 0x40080000L, 0x40080000L, 0x00000000L,
	0x40000100L, 0x42080100L, 0x42080100L, 0x02000100L,
	0x42080000L, 0x40000100L, 0x00000000L, 0x42000000L,
	0x02080100L, 0x02000000L, 0x42000000L, 0x00080100L,
	0x00080000L, 0x42000100L, 0x00000100L, 0x02000000L,
	0x40000000L, 0x02080000L, 0x42000100L, 0x40080100L,
	0x02000100L, 0x40000000L, 0x42080000L, 0x02080100L,
	0x40080100L, 0x00000100L, 0x02000000L, 0x42080000L,
	0x42080100L, 0x00080100L, 0x42000000L, 0x42080100L,
	0x02080000L, 0x00000000L, 0x40080000L, 0x42000000L,
	0x00080100L, 0x02000100L, 0x40000100L, 0x00080000L,
	0x00000000L, 0x40080000L, 0x02080100L, 0x40000100L };
```

```
static unsigned long SP6[64] = {
	0x20000010L, 0x20400000L, 0x00004000L, 0x20404010L,
	0x20400000L, 0x00000010L, 0x20404010L, 0x00400000L,
	0x20004000L, 0x00404010L, 0x00400000L, 0x20000010L,
	0x00400010L, 0x20004000L, 0x20000000L, 0x00004010L,
	0x00000000L, 0x00400010L, 0x20004010L, 0x00004000L,
	0x00404000L, 0x20004010L, 0x00000010L, 0x20400010L,
	0x20400010L, 0x00000000L, 0x00404010L, 0x20404000L,
	0x00004010L, 0x00404000L, 0x20404000L, 0x20000000L,
	0x20004000L, 0x00000010L, 0x20400010L, 0x00404000L,
	0x20404010L, 0x00400000L, 0x00004010L, 0x20000010L,
	0x00400000L, 0x20004000L, 0x20000000L, 0x00004010L,
	0x20000010L, 0x20404010L, 0x00404000L, 0x20400000L,
	0x00404010L, 0x20404000L, 0x00000000L, 0x20400010L,
	0x00000010L, 0x00004000L, 0x20400000L, 0x00404010L,
	0x00004000L, 0x00400010L, 0x20004010L, 0x00000000L,
	0x20404000L, 0x20000000L, 0x00400010L, 0x20004010L };

static unsigned long SP7[64] = {
	0x00200000L, 0x04200002L, 0x04000802L, 0x00000000L,
	0x00000800L, 0x04000802L, 0x00200802L, 0x04200800L,
	0x04200802L, 0x00200000L, 0x00000000L, 0x04000002L,
	0x00000002L, 0x04000000L, 0x04200002L, 0x00000802L,
	0x04000800L, 0x00200802L, 0x00200002L, 0x04000800L,
	0x04000002L, 0x04200000L, 0x04200800L, 0x00200002L,
	0x04200000L, 0x00000800L, 0x00000802L, 0x04200802L,
	0x00200800L, 0x00000002L, 0x04000000L, 0x00200800L,
	0x04000000L, 0x00200800L, 0x00200000L, 0x04000802L,
	0x04000802L, 0x04200002L, 0x04200002L, 0x00000002L,
	0x00200002L, 0x04000000L, 0x04000800L, 0x00200000L,
	0x04200800L, 0x00000802L, 0x00200802L, 0x04200800L,
	0x00000802L, 0x04000002L, 0x04200802L, 0x04200000L,
	0x00200800L, 0x00000000L, 0x00000002L, 0x04200802L,
	0x00000000L, 0x00200802L, 0x04200000L, 0x00000800L,
	0x04000002L, 0x04000800L, 0x00000800L, 0x00200002L };

static unsigned long SP8[64] = {
	0x10001040L, 0x00001000L, 0x00040000L, 0x10041040L,
	0x10000000L, 0x10001040L, 0x00000040L, 0x10000000L,
	0x00040040L, 0x10040000L, 0x10041040L, 0x00041000L,
	0x10041000L, 0x00041040L, 0x00001000L, 0x00000040L,
	0x10040000L, 0x10000040L, 0x10001000L, 0x00001040L,
	0x00041000L, 0x00040040L, 0x10040040L, 0x10041000L,
	0x00001040L, 0x00000000L, 0x00000000L, 0x10040040L,
	0x10000040L, 0x10001000L, 0x00041040L, 0x00040000L,
	0x00041040L, 0x00040000L, 0x10041000L, 0x00001000L,
	0x00000040L, 0x10040040L, 0x00001000L, 0x00041040L,
	0x10001000L, 0x00000040L, 0x10000040L, 0x10040000L,
	0x10040040L, 0x10000000L, 0x00040000L, 0x10001040L,
	0x00000000L, 0x10041040L, 0x00040040L, 0x10000040L,
	0x10040000L, 0x10001000L, 0x10001040L, 0x00000000L,
	0x10041040L, 0x00041000L, 0x00041000L, 0x00001040L,
	0x00001040L, 0x00040040L, 0x10000000L, 0x10041000L };

static void desfunc(block, keys)
register unsigned long *block, *keys;
{
	register unsigned long fval, work, right, leftt;
	register int round;

	leftt = block[0];
	right = block[1];
```

```
work = ((leftt >> 4) ^ right) & 0x0f0f0f0fL;
right ^= work;
leftt ^= (work << 4);
work = ((leftt >> 16) ^ right) & 0x0000ffffL;
right ^= work;
leftt ^= (work << 16);
work = ((right >> 2) ^ leftt) & 0x33333333L;
leftt ^= work;
right ^= (work << 2);
work = ((right >> 8) ^ leftt) & 0x00ff00ffL;
leftt ^= work;
right ^= (work << 8);
right = ((right << 1) | ((right >> 31) & 1L)) & 0xffffffffL;
work = (leftt ^ right) & 0xaaaaaaaaL;
leftt ^= work;
right ^= work;
leftt = ((leftt << 1) | ((leftt >> 31) & 1L)) & 0xffffffffL;

for( round = 0; round < 8; round++ ) {
       work  = (right << 28) | (right >> 4);
       work ^= *keys++;
       fval  = SP7[ work             & 0x3fL];
       fval |= SP5[(work >>  8) & 0x3fL];
       fval |= SP3[(work >> 16) & 0x3fL];
       fval |= SP1[(work >> 24) & 0x3fL];
       work  = right ^ *keys++;
       fval |= SP8[ work             & 0x3fL];
       fval |= SP6[(work >>  8) & 0x3fL];
       fval |= SP4[(work >> 16) & 0x3fL];
       fval |= SP2[(work >> 24) & 0x3fL];
       leftt ^= fval;
       work  = (leftt << 28) | (leftt >> 4);
       work ^= *keys++;
       fval  = SP7[ work             & 0x3fL];
       fval |= SP5[(work >>  8) & 0x3fL];
       fval |= SP3[(work >> 16) & 0x3fL];
       fval |= SP1[(work >> 24) & 0x3fL];
       work  = leftt ^ *keys++;
       fval |= SP8[ work             & 0x3fL];
       fval |= SP6[(work >>  8) & 0x3fL];
       fval |= SP4[(work >> 16) & 0x3fL];
       fval |= SP2[(work >> 24) & 0x3fL];
       right ^= fval;
       }

right = (right << 31) | (right >> 1);
work = (leftt ^ right) & 0xaaaaaaaaL;
leftt ^= work;
right ^= work;
leftt = (leftt << 31) | (leftt >> 1);
work = ((leftt >> 8) ^ right) & 0x00ff00ffL;
right ^= work;
leftt ^= (work << 8);
work = ((leftt >> 2) ^ right) & 0x33333333L;
right ^= work;
leftt ^= (work << 2);
work = ((right >> 16) ^ leftt) & 0x0000ffffL;
leftt ^= work;
right ^= (work << 16);
work = ((right >> 4) ^ leftt) & 0x0f0f0f0fL;
leftt ^= work;
right ^= (work << 4);
```

```
        *block++ = right;
        *block = leftt;
        return;
}

/* Validation sets:
 *
 * Single-length key, single-length plaintext -
 * Key    : 0123 4567 89ab cdef
 * Plain  : 0123 4567 89ab cde7
 * Cipher : c957 4425 6a5e d31d
 *
 **********************************************************************/

void des_key(des_ctx *dc, unsigned char *key){
        deskey(key,EN0);
        cpkey(dc->ek);
        deskey(key,DE1);
        cpkey(dc->dk);
}

/* Encrypt several blocks in ECB mode.  Caller is responsible for
   short blocks. */
void des_enc(des_ctx *dc, unsigned char *data, int blocks){
        unsigned long work[2];
        int i;
        unsigned char *cp;

        cp = data;
        for(i=0;i<blocks;i++){
                scrunch(cp,work);
                desfunc(work,dc->ek);
                unscrun(work,cp);
                cp+=8;
        }
}

void des_dec(des_ctx *dc, unsigned char *data, int blocks){
        unsigned long work[2];
        int i;
        unsigned char *cp;

        cp = data;
        for(i=0;i<blocks;i++){
                scrunch(cp,work);
                desfunc(work,dc->dk);
                unscrun(work,cp);
                cp+=8;
        }
}

void main(void){
        des_ctx dc;
        int i;
        unsigned long data[10];
        char *cp,key[8] = {0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef};
        char x[8] = {0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xe7};


        cp = x;

        des_key(&dc,key);
```

```
        des_enc(&dc,cp,1);
        printf("Enc(0..7,0..7) = ");
        for(i=0;i<8;i++) printf("%02x ", ((unsigned int) cp[i])&0x00ff);
        printf("\n");

        des_dec(&dc,cp,1);

        printf("Dec(above,0..7) = ");
        for(i=0;i<8;i++) printf("%02x ",((unsigned int)cp[i])&0x00ff);
        printf("\n");

        cp = (char *) data;
        for(i=0;i<10;i++)data[i]=i;

        des_enc(&dc,cp,5); /* Enc 5 blocks. */
        for(i=0;i<10;i+=2) printf("Block %01d = %08lx %08lx.\n",
                             i/2,data[i],data[i+1]);

        des_dec(&dc,cp,1);
        des_dec(&dc,cp+8,4);
        for(i=0;i<10;i+=2) printf("Block %01d = %08lx %08lx.\n",
                             i/2,data[i],data[i+1]);

}
```

LOKI91

```
#include <stdio.h>

#define LOKIBLK        8              /* No of bytes in a LOKI data-block
*/
#define ROUNDS         16             /* No of LOKI rounds
*/

typedef unsigned long          Long;   /* type specification for aligned
LOKI blocks */

extern Long    lokikey[2];     /* 64-bit key used by LOKI routines          */
extern char    *loki_lib_ver;         /* String with version no. &
copyright       */

#ifdef __STDC__                       /* declare prototypes for library
functions  */
extern void enloki(char *b);
extern void deloki(char *b);
extern void setlokikey(char key[LOKIBLK]);
#else                          /* else just declare library functions extern
*/
extern void enloki(), deloki(), setlokikey();
#endif __STDC__

char P[32] = {
        31, 23, 15, 7, 30, 22, 14, 6,
        29, 21, 13, 5, 28, 20, 12, 4,
        27, 19, 11, 3, 26, 18, 10, 2,
        25, 17, 9, 1, 24, 16, 8, 0
        };
```

```
typedef        struct {
        short  gen;            /* irreducible polynomial used in this field */
        short  exp;            /* exponent used to generate this s function */
        } sfn_desc;

sfn_desc sfn[] = {
        { /* 101110111 */ 375, 31}, { /* 101111011 */ 379, 31},
        { /* 110000111 */ 391, 31}, { /* 110001011 */ 395, 31},
        { /* 110001101 */ 397, 31}, { /* 110011111 */ 415, 31},
        { /* 110100011 */ 419, 31}, { /* 110101001 */ 425, 31},
        { /* 110110001 */ 433, 31}, { /* 110111101 */ 445, 31},
        { /* 111000011 */ 451, 31}, { /* 111001111 */ 463, 31},
        { /* 111010111 */ 471, 31}, { /* 111011101 */ 477, 31},
        { /* 111100111 */ 487, 31}, { /* 111110011 */ 499, 31},
        { 00, 00}      };

typedef struct {
        Long loki_subkeys[ROUNDS];
} loki_ctx;

static Long    f();                   /* declare LOKI function f */
static short   s();                   /* declare LOKI S-box fn s */

#define ROL12(b) b = ((b << 12) | (b >> 20));
#define ROL13(b) b = ((b << 13) | (b >> 19));

#ifdef  LITTLE_ENDIAN
#define bswap(cb) {                             \
        register char   c;                      \
        c = cb[0]; cb[0] = cb[3]; cb[3] = c;    \
        c = cb[1]; cb[1] = cb[2]; cb[2] = c;    \
        c = cb[4]; cb[4] = cb[7]; cb[7] = c;    \
        c = cb[5]; cb[5] = cb[6]; cb[6] = c;    \
}
#endif

void
setlokikey(loki_ctx *c, char *key)
{
        register       i;
        register Long  KL, KR;

#ifdef LITTLE_ENDIAN
        bswap(key);                   /* swap bytes round if little-endian */
#endif
        KL = ((Long *)key)[0];
        KR = ((Long *)key)[1];

        for (i=0; i<ROUNDS; i+=4) {           /* Generate the 16 subkeys */
            c->loki_subkeys[i] = KL;
            ROL12 (KL);
            c->loki_subkeys[i+1] = KL;
            ROL13 (KL);
            c->loki_subkeys[i+2] = KR;
            ROL12 (KR);
            c->loki_subkeys[i+3] = KR;
            ROL13 (KR);
        }

#ifdef LITTLE_ENDIAN
        bswap(key);                   /* swap bytes back if little-endian */
#endif
```

```
}

void
enloki (loki_ctx *c, char *b)
{
        register        i;
        register Long   L, R;           /* left & right data halves  */

#ifdef LITTLE_ENDIAN
        bswap(b);                       /* swap bytes round if little-endian */
#endif

        L = ((Long *)b)[0];
        R = ((Long *)b)[1];

        for (i=0; i<ROUNDS; i+=2) {             /* Encrypt with the 16 subkeys
*/
            L ^= f (R, c->loki_subkeys[i]);
            R ^= f (L, c->loki_subkeys[i+1]);
        }

        ((Long *)b)[0] = R;             /* Y = swap(LR) */
        ((Long *)b)[1] = L;

#ifdef LITTLE_ENDIAN
        bswap(b);                       /* swap bytes round if little-endian */
#endif
}

void
deloki(loki_ctx *c, char *b)
{
        register        i;
        register Long   L, R;                   /* left & right data halves  */

#ifdef LITTLE_ENDIAN
        bswap(b);                       /* swap bytes round if little-endian */
#endif

        L = ((Long *)b)[0];                     /* LR = X XOR K */
        R = ((Long *)b)[1];

        for (i=ROUNDS; i>0; i-=2) {                     /* subkeys in reverse
order */
            L ^= f(R, c->loki_subkeys[i-1]);
            R ^= f(L, c->loki_subkeys[i-2]);
        }

        ((Long *)b)[0] = R;                     /* Y = LR XOR K */
        ((Long *)b)[1] = L;
}

#define MASK12          0x0fff                  /* 12 bit mask for expansion E
*/

static Long
f(r, k)
register Long  r;       /* Data value R(i-1) */
Long           k;       /* Key     K(i)   */
{
        Long   a, b, c;                 /* 32 bit S-box output, & P output */
```

```
        a = r ^ k;                      /* A = R(i-1) XOR K(i) */

        /* want to use slow speed/small size version */
        b = ((Long)s((a         & MASK12))      ) | /* B = S(E(R(i-1))^K(i))
*/
            ((Long)s(((a >>  8) & MASK12)) <<  8) |
            ((Long)s(((a >> 16) & MASK12)) << 16) |
            ((Long)s((((a >> 24) | (a << 8)) & MASK12)) << 24);

        perm32(&c, &b, P);              /* C = P(S( E(R(i-1)) XOR K(i))) */

        return(c);                      /* f returns the result C */
}

static short s(i)
register Long i;        /* return S-box value for input i */
{
        register short r, c, v, t;
        short  exp8();                  /* exponentiation routine for GF(2^8) */

        r = ((i>>8) & 0xc) | (i & 0x3);                 /* row value-top 2 &
bottom 2 */
        c = (i>>2) & 0xff;                              /* column
value-middle 8 bits */
        t = (c + ((r * 17) ^ 0xff)) & 0xff;             /* base value for Sfn */
        v = exp8(t, sfn[r].exp, sfn[r].gen);            /* Sfn[r] = t ^ exp
mod gen */
        return(v);
}

#define        MSB     0x80000000L             /* MSB of 32-bit word */

perm32(out, in , perm)
Long    *out;           /* Output 32-bit block to be permuted               */
Long    *in;            /* Input  32-bit block after permutation            */
char    perm[32];       /* Permutation array                                */
{
        Long   mask = MSB;                     /* mask used to set bit in
output    */
        register int   i, o, b;        /* input bit no, output bit no, value */
        register char  *p = perm;      /* ptr to permutation array  */

        *out = 0;                       /* clear output block */
        for (o=0; o<32; o++) {                 /* For each output bit
position o */
                i =(int)*p++;                  /* get input bit permuted to
output o */
                b = (*in >> i) & 01;           /* value of input bit i */
                if (b)                  /* If the input bit i is set */
                        *out |= mask;                   /*  OR in mask to
output i */
                mask >>= 1;                             /* Shift mask to next
bit    */
        }
}

#define SIZE 256                /* 256 elements in GF(2^8) */

short mult8(a, b, gen)
short   a, b;           /* operands for multiply */
short   gen;            /* irreducible polynomial generating Galois Field */
{
```

```
        short  product = 0;            /* result of multiplication */

        while(b != 0) {                         /* while multiplier is
non-zero */
                if (b & 01)
                        product ^= a;           /*   add multiplicand if LSB
of b set */
                a <<= 1;                /*   shift multiplicand one place */
                if (a >= SIZE)
                        a ^= gen;       /*   and modulo reduce if needed */
                b >>= 1;                /*   shift multiplier one place  */
        }
        return(product);
}

short exp8(base, exponent, gen)
short   base;           /* base of exponentiation      */
short   exponent;       /* exponent                    */
short   gen;            /* irreducible polynomial generating Galois Field */
{
        short  accum = base;           /* superincreasing sequence of base */
        short  result = 1;     /* result of exponentiation       */

        if (base == 0)                 /* if zero base specified then      */
                return(0);      /* the result is "0" if base = 0    */

        while (exponent != 0) {        /* repeat while exponent non-zero */
                if (( exponent & 0x0001) == 0x0001)           /* multiply if
exp 1 */
                        result = mult8(result, accum, gen);
                exponent >>= 1;                /* shift exponent to next
digit */
                accum = mult8(accum, accum, gen);            /* & square  */
        }
        return(result);
}

void loki_key(loki_ctx *c, unsigned char *key){
        setlokikey(c,key);
}

void loki_enc(loki_ctx *c, unsigned char *data, int blocks){
        unsigned char *cp;
        int i;

        cp = data;
        for(i=0;i<blocks;i++){
                enloki(c,cp);
                cp+=8;
        }
}

void loki_dec(loki_ctx *c, unsigned char *data, int blocks){
        unsigned char *cp;
        int i;

        cp = data;
        for(i=0;i<blocks;i++){
                deloki(c,cp);
                cp+=8;
        }
}
```

```
void main(void){
        loki_ctx lc;
        unsigned long data[10];
        unsigned char *cp;
        unsigned char key[] = {0,1,2,3,4,5,6,7};
        int i;

        for(i=0;i<10;i++) data[i]=i;

        loki_key(&lc,key);

        cp = (char *)data;
        loki_enc(&lc,cp,5);
        for(i=0;i<10;i+=2) printf("Block %01d = %08lx %08lx\n",
                        i/2,data[i],data[i+1]);
        loki_dec(&lc,cp,1);
        loki_dec(&lc,cp+8,4);
        for(i=0;i<10;i+=2) printf("Block %01d = %08lx %08lx\n",
                        i/2,data[i],data[i+1]);


}
```

IDEA

```
typedef unsigned char boolean;      /* values are TRUE or FALSE */
typedef unsigned char byte; /* values are 0-255 */
typedef byte *byteptr;      /* pointer to byte */
typedef char *string;/* pointer to ASCII character string */
typedef unsigned short word16;      /* values are 0-65535 */
typedef unsigned long word32;       /* values are 0-4294967295 */

#ifndef TRUE
#define FALSE 0
#define TRUE (!FALSE)
#endif /* if TRUE not already defined */

#ifndef min   /* if min macro not already defined */
#define min(a,b) ( (a)<(b) ? (a) : (b) )
#define max(a,b) ( (a)>(b) ? (a) : (b) )
#endif /* if min macro not already defined */

#define IDEAKEYSIZE 16
#define IDEABLOCKSIZE 8

#define IDEAROUNDS 8
#define IDEAKEYLEN (6*IDEAROUNDS+4)

typedef struct{
       word16 ek[IDEAKEYLEN],dk[IDEAKEYLEN];
}idea_ctx;

/* End includes for IDEA.C */
#ifdef IDEA32        /* Use >16-bit temporaries */
#define low16(x) ((x) & 0xFFFF)
typedef unsigned int uint16;/* at LEAST 16 bits, maybe more */
#else
#define low16(x) (x) /* this is only ever applied to uint16's */
typedef word16 uint16;
```

```
#endif

#ifdef SMALL_CACHE
static uint16
mul(register uint16 a, register uint16 b)
{
	register word32 p;

	p = (word32)a * b;
	if (p) {
		b = low16(p);
		a = p>>16;
		return (b - a) + (b < a);
	} else if (a) {
		return 1-b;
	} else {
		return 1-a;
	}
} /* mul */
#endif /* SMALL_CACHE */

static uint16
mulInv(uint16 x)
{
	uint16 t0, t1;
	uint16 q, y;

	if (x <= 1)
		return x;	/* 0 and 1 are self-inverse */
	t1 = 0x10001L / x;	/* Since x >= 2, this fits into 16 bits */
	y = 0x10001L % x;
	if (y == 1)
		return low16(1-t1);
	t0 = 1;
	do {
		q = x / y;
		x = x % y;
		t0 += q * t1;
		if (x == 1)
			return t0;
		q = y / x;
		y = y % x;
		t1 += q * t0;
	} while (y != 1);
	return low16(1-t1);
} /* mukInv */

static void
ideaExpandKey(byte const *userkey, word16 *EK)
{
	int i,j;

	for (j=0; j<8; j++) {
		EK[j] = (userkey[0]<<8) + userkey[1];
		userkey += 2;
	}
	for (i=0; j < IDEAKEYLEN; j++) {
		i++;
		EK[i+7] = EK[i & 7] << 9 | EK[i+1 & 7] >> 7;
		EK += i & 8;
		i &= 7;
	}
```

```
} /* ideaExpandKey */

static void
ideaInvertKey(word16 const *EK, word16 DK[IDEAKEYLEN])
{
	int i;
	uint16 t1, t2, t3;
	word16 temp[IDEAKEYLEN];
	word16 *p = temp + IDEAKEYLEN;

	t1 = mulInv(*EK++);
	t2 = -*EK++;
	t3 = -*EK++;
	*--p = mulInv(*EK++);
	*--p = t3;
	*--p = t2;
	*--p = t1;

	for (i = 0; i < IDEAROUNDS-1; i++) {
		t1 = *EK++;
		*--p = *EK++;
		*--p = t1;

		t1 = mulInv(*EK++);
		t2 = -*EK++;
		t3 = -*EK++;
		*--p = mulInv(*EK++);
		*--p = t2;
		*--p = t3;
		*--p = t1;
	}
	t1 = *EK++;
	*--p = *EK++;
	*--p = t1;

	t1 = mulInv(*EK++);
	t2 = -*EK++;
	t3 = -*EK++;
	*--p = mulInv(*EK++);
	*--p = t3;
	*--p = t2;
	*--p = t1;
/* Copy and destroy temp copy */
	memcpy(DK, temp, sizeof(temp));
	for(i=0;i<IDEAKEYLEN;i++)temp[i]=0;
} /* ideaInvertKey */

#ifdef SMALL_CACHE
#define MUL(x,y) (x = mul(low16(x),y))
#else /* !SMALL_CACHE */
#ifdef AVOID_JUMPS
#define MUL(x,y) (x = low16(x-1), t16 = low16((y)-1), \
		t32 = (word32)x*t16 + x + t16 + 1, x = low16(t32), \
		t16 = t32>>16, x = (x-t16) + (x<t16) )
#else /* !AVOID_JUMPS (default) */
#define MUL(x,y) \
	((t16 = (y)) ? \
		(x=low16(x)) ? \
			t32 = (word32)x*t16, \
			x = low16(t32), \
			t16 = t32>>16, \
			x = (x-t16)+(x<t16) \
```

```
		: \
			(x = 1-t16) \
	: \
		(x = 1-x))
#endif
#endif

static void
ideaCipher(byte *inbuf, byte *outbuf, word16 *key)
{
	register uint16 x1, x2, x3, x4, s2, s3;
	word16 *in, *out;
#ifndef SMALL_CACHE
	register uint16 t16; /* Temporaries needed by MUL macro */
	register word32 t32;
#endif
	int r = IDEAROUNDS;

	in = (word16 *)inbuf;
	x1 = *in++;  x2 = *in++;
	x3 = *in++;  x4 = *in;
#ifndef HIGHFIRST
	x1 = (x1 >>8) | (x1<<8);
	x2 = (x2 >>8) | (x2<<8);
	x3 = (x3 >>8) | (x3<<8);
	x4 = (x4 >>8) | (x4<<8);
#endif
	do {
		MUL(x1,*key++);
		x2 += *key++;
		x3 += *key++;
		MUL(x4, *key++);

		s3 = x3;
		x3 ^= x1;
		MUL(x3, *key++);
		s2 = x2;
		x2 ^= x4;
		x2 += x3;
		MUL(x2, *key++);
		x3 += x2;

		x1 ^= x2;  x4 ^= x3;

		x2 ^= s3;  x3 ^= s2;
	} while (--r);
	MUL(x1, *key++);
	x3 += *key++;
	x2 += *key++;
	MUL(x4, *key);

	out = (word16 *)outbuf;
#ifdef HIGHFIRST
	*out++ = x1;
	*out++ = x3;
	*out++ = x2;
	*out = x4;
#else /* !HIGHFIRST */
	*out++ = (x1 >>8) | (x1<<8);
	*out++ = (x3 >>8) | (x3<<8);
	*out++ = (x2 >>8) | (x2<<8);
	*out = (x4 >>8) | (x4<<8);
```

```
#endif
} /* ideaCipher */

void idea_key(idea_ctx *c, unsigned char *key){
	ideaExpandKey(key,c->ek);
	ideaInvertKey(c->ek,c->dk);
}

void idea_enc(idea_ctx *c, unsigned char *data, int blocks){
	int i;
	unsigned char *d = data;

	for(i=0;i<blocks;i++){
		ideaCipher(d,d,c->ek);
		d+=8;
	}
}

void idea_dec(idea_ctx *c, unsigned char *data, int blocks){
	int i;
	unsigned char *d = data;

	for(i=0;i<blocks;i++){
		ideaCipher(d,d,c->dk);
		d+=8;
	}
}

#include <stdio.h>

#ifndef BLOCKS
#ifndef KBYTES
#define KBYTES 1024
#endif
#define BLOCKS (64*KBYTES)
#endif

int
main(void)
{	/* Test driver for IDEA cipher */
	int i, j, k;
	idea_ctx c;
	byte userkey[16];
	word16 EK[IDEAKEYLEN], DK[IDEAKEYLEN];
	byte XX[8], YY[8], ZZ[8];
	word32 long_block[10]; /* 5 blocks */
	long l;
	char *lbp;

	/* Make a sample user key for testing... */
	for(i=0; i<16; i++)
		userkey[i] = i+1;

	idea_key(&c,userkey);

	/* Make a sample plaintext pattern for testing... */
	for (k=0; k<8; k++)
		XX[k] = k;

	idea_enc(&c,XX,1); /* encrypt */

	lbp = (unsigned char *) long_block;
```

```
        for(i=0;i<10;i++) long_block[i] = i;
        idea_enc(&c,lbp,5);
        for(i=0;i<10;i+=2) printf("Block %01d = %08lx %08lx.\n",
                                i/2,long_block[i],long_block[i+1]);

        idea_dec(&c,lbp,3);
        idea_dec(&c,lbp+24,2);

        for(i=0;i<10;i+=2) printf("Block %01d = %08lx %08lx.\n",
                                i/2,long_block[i],long_block[i+1]);

        return 0;       /* normal exit */
} /* main */
```

GOST

```
typedef unsigned long u4;
typedef unsigned char byte;

typedef struct {
        u4 k[8];
        /* Constant s-boxes -- set up in gost_init(). */
        char k87[256],k65[256],k43[256],k21[256];
} gost_ctx;

/* Note:  encrypt and decrypt expect full blocks--padding blocks is
          caller's responsibility.  All bulk encryption is done in
          ECB mode by these calls.  Other modes may be added easily
          enough.                                                    */
void gost_enc(gost_ctx *, u4 *, int);
void gost_dec(gost_ctx *, u4 *, int);
void gost_key(gost_ctx *, u4 *);
void gost_init(gost_ctx *);
void gost_destroy(gost_ctx *);

#ifdef __alpha  /* Any other 64-bit machines? */
typedef unsigned int word32;
#else
typedef unsigned long word32;
#endif


kboxinit(gost_ctx *c)
{
        int i;

        byte k8[16] = {14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6,
                       12,  5,  9,  0,  7 };
        byte k7[16] = {15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2,
                       13, 12,  0,  5, 10 };
        byte k6[16] = {10,  0,  9, 14,  6,  3, 15,  5,  1, 13, 12,
                        7, 11,  4,  2,  8 };
        byte k5[16] = { 7, 13, 14,  3,  0,  6,  9, 10,  1,  2,  8,
                        5, 11, 12,  4, 15 };
        byte k4[16] = { 2, 12,  4,  1,  7, 10, 11,  6,  8,  5,  3,
                       15, 13,  0, 14,  9 };
        byte k3[16] = {12,  1, 10, 15,  9,  2,  6,  8,  0, 13,  3,
                        4, 14,  7,  5, 11 };
```

```
        byte k2[16] = { 4, 11,  2, 14, 15,  0,  8, 13,  3, 12,  9,
                        7,  5, 10,  6,  1 };
        byte k1[16] = {13,  2,  8,  4,  6, 15, 11,  1, 10,  9,  3,
                       14,  5,  0, 12,  7 };

     for (i = 0; i < 256; i++) {
                c->k87[i] = k8[i >> 4] << 4 | k7[i & 15];
                c->k65[i] = k6[i >> 4] << 4 | k5[i & 15];
                c->k43[i] = k4[i >> 4] << 4 | k3[i & 15];
                c->k21[i] = k2[i >> 4] << 4 | k1[i & 15];
     }
}

static word32
f(gost_ctx *c,word32 x)
{
        x = c->k87[x>>24 & 255] << 24 | c->k65[x>>16 & 255] << 16 |
            c->k43[x>> 8 & 255] <<  8 | c->k21[x & 255];

     /* Rotate left 11 bits */
     return x<<11 | x>>(32-11);
}

void gostcrypt(gost_ctx *c, word32 *d){
     register word32 n1, n2; /* As named in the GOST */

        n1 = d[0];
        n2 = d[1];

     /* Instead of swapping halves, swap names each round */
        n2 ^= f(c,n1+c->k[0]); n1 ^= f(c,n2+c->k[1]);
        n2 ^= f(c,n1+c->k[2]); n1 ^= f(c,n2+c->k[3]);
        n2 ^= f(c,n1+c->k[4]); n1 ^= f(c,n2+c->k[5]);
        n2 ^= f(c,n1+c->k[6]); n1 ^= f(c,n2+c->k[7]);

        n2 ^= f(c,n1+c->k[0]); n1 ^= f(c,n2+c->k[1]);
        n2 ^= f(c,n1+c->k[2]); n1 ^= f(c,n2+c->k[3]);
        n2 ^= f(c,n1+c->k[4]); n1 ^= f(c,n2+c->k[5]);
        n2 ^= f(c,n1+c->k[6]); n1 ^= f(c,n2+c->k[7]);

        n2 ^= f(c,n1+c->k[0]); n1 ^= f(c,n2+c->k[1]);
        n2 ^= f(c,n1+c->k[2]); n1 ^= f(c,n2+c->k[3]);
        n2 ^= f(c,n1+c->k[4]); n1 ^= f(c,n2+c->k[5]);
        n2 ^= f(c,n1+c->k[6]); n1 ^= f(c,n2+c->k[7]);

        n2 ^= f(c,n1+c->k[7]); n1 ^= f(c,n2+c->k[6]);
        n2 ^= f(c,n1+c->k[5]); n1 ^= f(c,n2+c->k[4]);
        n2 ^= f(c,n1+c->k[3]); n1 ^= f(c,n2+c->k[2]);
        n2 ^= f(c,n1+c->k[1]); n1 ^= f(c,n2+c->k[0]);

        d[0] = n2; d[1] = n1;
}

void
gostdecrypt(gost_ctx *c, u4 *d){
     register word32 n1, n2; /* As named in the GOST */

        n1 = d[0]; n2 = d[1];

        n2 ^= f(c,n1+c->k[0]); n1 ^= f(c,n2+c->k[1]);
        n2 ^= f(c,n1+c->k[2]); n1 ^= f(c,n2+c->k[3]);
        n2 ^= f(c,n1+c->k[4]); n1 ^= f(c,n2+c->k[5]);
```

```
        n2 ^= f(c,n1+c->k[6]); n1 ^= f(c,n2+c->k[7]);

        n2 ^= f(c,n1+c->k[7]); n1 ^= f(c,n2+c->k[6]);
        n2 ^= f(c,n1+c->k[5]); n1 ^= f(c,n2+c->k[4]);
        n2 ^= f(c,n1+c->k[3]); n1 ^= f(c,n2+c->k[2]);
        n2 ^= f(c,n1+c->k[1]); n1 ^= f(c,n2+c->k[0]);

        n2 ^= f(c,n1+c->k[7]); n1 ^= f(c,n2+c->k[6]);
        n2 ^= f(c,n1+c->k[5]); n1 ^= f(c,n2+c->k[4]);
        n2 ^= f(c,n1+c->k[3]); n1 ^= f(c,n2+c->k[2]);
        n2 ^= f(c,n1+c->k[1]); n1 ^= f(c,n2+c->k[0]);

        n2 ^= f(c,n1+c->k[7]); n1 ^= f(c,n2+c->k[6]);
        n2 ^= f(c,n1+c->k[5]); n1 ^= f(c,n2+c->k[4]);
        n2 ^= f(c,n1+c->k[3]); n1 ^= f(c,n2+c->k[2]);
        n2 ^= f(c,n1+c->k[1]); n1 ^= f(c,n2+c->k[0]);

        d[0] = n2; d[1] = n1;
}

void gost_enc(gost_ctx *c, u4 *d, int blocks){
        int i;

        for(i=0;i<blocks;i++){
                gostcrypt(c,d);
                d+=2;
        }
}

void gost_dec(gost_ctx *c, u4 *d, int blocks){
        int i;

        for(i=0;i<blocks;i++){
                gostdecrypt(c,d);
                d+=2;
        }
}

void gost_key(gost_ctx *c, u4 *k){
        int i;
        for(i=0;i<8;i++) c->k[i]=k[i];
}

void gost_init(gost_ctx *c){
        kboxinit(c);
}

void gost_destroy(gost_ctx *c){
        int i;
        for(i=0;i<8;i++) c->k[i]=0;
}

void main(void){
        gost_ctx gc;
        u4 k[8],data[10];
        int i;

        /* Initialize GOST context. */
        gost_init(&gc);

        /* Prepare key--a simple key should be OK, with this many rounds! */
        for(i=0;i<8;i++) k[i] = i;
```

```
        gost_key(&gc,k);

        /* Try some test vectors. */
        data[0] = 0; data[1] = 0;
        gostcrypt(&gc,data);
        printf("Enc of zero vector:  %08lx %08lx\n",data[0],data[1]);
        gostcrypt(&gc,data);
        printf("Enc of above:        %08lx %08lx\n",data[0],data[1]);
        data[0] = 0xffffffff; data[1] = 0xffffffff;
        gostcrypt(&gc,data);
        printf("Enc of ones vector:  %08lx %08lx\n",data[0],data[1]);
        gostcrypt(&gc,data);
        printf("Enc of above:        %08lx %08lx\n",data[0],data[1]);

        /* Does gost_dec() properly reverse gost_enc()?  Do
           we deal OK with single-block lengths passed in gost_dec()?
           Do we deal OK with different lengths passed in? */

        /* Init data */
        for(i=0;i<10;i++) data[i]=i;

        /* Encrypt data as 5 blocks. */
        gost_enc(&gc,data,5);

        /* Display encrypted data. */
        for(i=0;i<10;i+=2) printf("Block %02d = %08lx %08lx\n",
                                  i/2,data[i],data[i+1]);

        /* Decrypt in different sized chunks. */
        gost_dec(&gc,data,1);
        gost_dec(&gc,data+2,4);
        printf("\n");

        /* Display decrypted data. */
        for(i=0;i<10;i+=2) printf("Block %02d = %08lx %08lx\n",
                                  i/2,data[i],data[i+1]);

        gost_destroy(&gc);
}
```

BLOWFISH

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

#ifdef little_endian   /* Eg: Intel */
   #include <alloc.h>
#endif

#include <ctype.h>

#ifdef little_endian   /* Eg: Intel */
   #include <dir.h>
   #include <bios.h>
#endif
```

```
#ifdef big_endian
   #include <Types.h>
#endif

typedef struct {
        unsigned long S[4][256],P[18];
} blf_ctx;


#define MAXKEYBYTES 56          /* 448 bits */
// #define little_endian 1              /* Eg: Intel */
#define big_endian 1            /* Eg: Motorola */

void Blowfish_encipher(blf_ctx *,unsigned long *xl, unsigned long *xr);
void Blowfish_decipher(blf_ctx *,unsigned long *xl, unsigned long *xr);

#define N               16
#define noErr            0
#define DATAERROR         -1
#define KEYBYTES         8

FILE*         SubkeyFile;


unsigned long F(blf_ctx *bc, unsigned long x)
{
   unsigned short a;
   unsigned short b;
   unsigned short c;
   unsigned short d;
   unsigned long  y;

   d = x & 0x00FF;
   x >>= 8;
   c = x & 0x00FF;
   x >>= 8;
   b = x & 0x00FF;
   x >>= 8;
   a = x & 0x00FF;
   //y = ((S[0][a] + S[1][b]) ^ S[2][c]) + S[3][d];
   y = bc->S[0][a] + bc->S[1][b];
   y = y ^ bc->S[2][c];
   y = y + bc->S[3][d];

   return y;
}

void Blowfish_encipher(blf_ctx *c,unsigned long *xl, unsigned long *xr)
{
   unsigned long  Xl;
   unsigned long  Xr;
   unsigned long  temp;
   short          i;

   Xl = *xl;
   Xr = *xr;

   for (i = 0; i < N; ++i) {
      Xl = Xl ^ c->P[i];
      Xr = F(c,Xl) ^ Xr;

      temp = Xl;
```

```
      Xl = Xr;
      Xr = temp;
   }

   temp = Xl;
   Xl = Xr;
   Xr = temp;

   Xr = Xr ^ c->P[N];
   Xl = Xl ^ c->P[N + 1];

   *xl = Xl;
   *xr = Xr;
}

void Blowfish_decipher(blf_ctx *c, unsigned long *xl, unsigned long *xr)
{
   unsigned long  Xl;
   unsigned long  Xr;
   unsigned long  temp;
   short          i;

   Xl = *xl;
   Xr = *xr;

   for (i = N + 1; i > 1; --i) {
      Xl = Xl ^ c->P[i];
      Xr = F(c,Xl) ^ Xr;

      /* Exchange Xl and Xr */
      temp = Xl;
      Xl = Xr;
      Xr = temp;
   }

   /* Exchange Xl and Xr */
   temp = Xl;
   Xl = Xr;
   Xr = temp;

   Xr = Xr ^ c->P[1];
   Xl = Xl ^ c->P[0];

   *xl = Xl;
   *xr = Xr;
}

short InitializeBlowfish(blf_ctx *c, char key[], short keybytes)
{
   short          i;
   short          j;
   short          k;
   short          error;
   short          numread;
   unsigned long  data;
   unsigned long  datal;
   unsigned long  datar;

unsigned long ks0[] = {
0xd1310ba6, 0x98dfb5ac, 0x2ffd72db, 0xd01adfb7, 0xb8e1afed, 0x6a267e96,
0xba7c9045, 0xf12c7f99, 0x24a19947, 0xb3916cf7, 0x0801f2e2, 0x858efc16,
0x636920d8, 0x71574e69, 0xa458fea3, 0xf4933d7e, 0x0d95748f, 0x728eb658,
```

```
0x718bcd58, 0x82154aee, 0x7b54a41d, 0xc25a59b5, 0x9c30d539, 0x2af26013,
0xc5d1b023, 0x286085f0, 0xca417918, 0xb8db38ef, 0x8e79dcb0, 0x603a180e,
0x6c9e0e8b, 0xb01e8a3e, 0xd71577c1, 0xbd314b27, 0x78af2fda, 0x55605c60,
0xe65525f3, 0xaa55ab94, 0x57489862, 0x63e81440, 0x55ca396a, 0x2aab10b6,
0xb4cc5c34, 0x1141e8ce, 0xa15486af, 0x7c72e993, 0xb3ee1411, 0x636fbc2a,
0x2ba9c55d, 0x741831f6, 0xce5c3e16, 0x9b87931e, 0xafd6ba33, 0x6c24cf5c,
0x7a325381, 0x28958677, 0x3b8f4898, 0x6b4bb9af, 0xc4bfe81b, 0x66282193,
0x61d809cc, 0xfb21a991, 0x487cac60, 0x5dec8032, 0xef845d5d, 0xe98575b1,
0xdc262302, 0xeb651b88, 0x23893e81, 0xd396acc5, 0x0f6d6ff3, 0x83f44239,
0x2e0b4482, 0xa4842004, 0x69c8f04a, 0x9e1f9b5e, 0x21c66842, 0xf6e96c9a,
0x670c9c61, 0xabd388f0, 0x6a51a0d2, 0xd8542f68, 0x960fa728, 0xab5133a3,
0x6eef0b6c, 0x137a3be4, 0xba3bf050, 0x7efb2a98, 0xa1f1651d, 0x39af0176,
0x66ca593e, 0x82430e88, 0x8cee8619, 0x456f9fb4, 0x7d84a5c3, 0x3b8b5ebe,
0xe06f75d8, 0x85c12073, 0x401a449f, 0x56c16aa6, 0x4ed3aa62, 0x363f7706,
0x1bfedf72, 0x429b023d, 0x37d0d724, 0xd00a1248, 0xdb0fead3, 0x49f1c09b,
0x075372c9, 0x80991b7b, 0x25d479d8, 0xf6e8def7, 0xe3fe501a, 0xb6794c3b,
0x976ce0bd, 0x04c006ba, 0xc1a94fb6, 0x409f60c4, 0x5e5c9ec2, 0x196a2463,
0x68fb6faf, 0x3e6c53b5, 0x1339b2eb, 0x3b52ec6f, 0x6dfc511f, 0x9b30952c,
0xcc814544, 0xaf5ebd09, 0xbee3d004, 0xde334afd, 0x660f2807, 0x192e4bb3,
0xc0cba857, 0x45c8740f, 0xd20b5f39, 0xb9d3fbdb, 0x5579c0bd, 0x1a60320a,
0xd6a100c6, 0x402c7279, 0x679f25fe, 0xfb1fa3cc, 0x8ea5e9f8, 0xdb3222f8,
0x3c7516df, 0xfd616b15, 0x2f501ec8, 0xad0552ab, 0x323db5fa, 0xfd238760,
0x53317b48, 0x3e00df82, 0x9e5c57bb, 0xca6f8ca0, 0x1a87562e, 0xdf1769db,
0xd542a8f6, 0x287effc3, 0xac6732c6, 0x8c4f5573, 0x695b27b0, 0xbbca58c8,
0xe1ffa35d, 0xb8f011a0, 0x10fa3d98, 0xfd2183b8, 0x4afcb56c, 0x2dd1d35b,
0x9a53e479, 0xb6f84565, 0xd28e49bc, 0x4bfb9790, 0xe1ddf2da, 0xa4cb7e33,
0x62fb1341, 0xcee4c6e8, 0xef20cada, 0x36774c01, 0xd07e9efe, 0x2bf11fb4,
0x95dbda4d, 0xae909198, 0xeaad8e71, 0x6b93d5a0, 0xd08ed1d0, 0xafc725e0,
0x8e3c5b2f, 0x8e7594b7, 0x8ff6e2fb, 0xf2122b64, 0x8888b812, 0x900df01c,
0x4fad5ea0, 0x688fc31c, 0xd1cff191, 0xb3a8c1ad, 0x2f2f2218, 0xbe0e1777,
0xea752dfe, 0x8b021fa1, 0xe5a0cc0f, 0xb56f74e8, 0x18acf3d6, 0xce89e299,
0xb4a84fe0, 0xfd13e0b7, 0x7cc43b81, 0xd2ada8d9, 0x165fa266, 0x80957705,
0x93cc7314, 0x211a1477, 0xe6ad2065, 0x77b5fa86, 0xc75442f5, 0xfb9d35cf,
0xebcdaf0c, 0x7b3e89a0, 0xd6411bd3, 0xae1e7e49, 0x00250e2d, 0x2071b35e,
0x226800bb, 0x57b8e0af, 0x2464369b, 0xf009b91e, 0x5563911d, 0x59dfa6aa,
0x78c14389, 0xd95a537f, 0x207d5ba2, 0x02e5b9c5, 0x83260376, 0x6295cfa9,
0x11c81968, 0x4e734a41, 0xb3472dca, 0x7b14a94a, 0x1b510052, 0x9a532915,
0xd60f573f, 0xbc9bc6e4, 0x2b60a476, 0x81e67400, 0x08ba6fb5, 0x571be91f,
0xf296ec6b, 0x2a0dd915, 0xb6636521, 0xe7b9f9b6, 0xff34052e, 0xc5855664,
0x53b02d5d, 0xa99f8fa1, 0x08ba4799, 0x6e85076a};
unsigned long ks1[] = {
0x4b7a70e9, 0xb5b32944, 0xdb75092e, 0xc4192623, 0xad6ea6b0, 0x49a7df7d,
0x9cee60b8, 0x8fedb266, 0xecaa8c71, 0x699a17ff, 0x5664526c, 0xc2b19ee1,
0x193602a5, 0x75094c29, 0xa0591340, 0xe4183a3e, 0x3f54989a, 0x5b429d65,
0x6b8fe4d6, 0x99f73fd6, 0xa1d29c07, 0xefe830f5, 0x4d2d38e6, 0xf0255dc1,
0x4cdd2086, 0x8470eb26, 0x6382e9c6, 0x021ecc5e, 0x09686b3f, 0x3ebaefc9,
0x3c971814, 0x6b6a70a1, 0x687f3584, 0x52a0e286, 0xb79c5305, 0xaa500737,
0x3e07841c, 0x7fdeae5c, 0x8e7d44ec, 0x5716f2b8, 0xb03ada37, 0xf0500c0d,
0xf01c1f04, 0x0200b3ff, 0xae0cf51a, 0x3cb574b2, 0x25837a58, 0xdc0921bd,
0xd19113f9, 0x7ca92ff6, 0x94324773, 0x22f54701, 0x3ae5e581, 0x37c2dadc,
0xc8b57634, 0x9af3dda7, 0xa9446146, 0x0fd0030e, 0xecc8c73e, 0xa4751e41,
0xe238cd99, 0x3bea0e2f, 0x3280bba1, 0x183eb331, 0x4e548b38, 0x4f6db908,
0x6f420d03, 0xf60a04bf, 0x2cb81290, 0x24977c79, 0x5679b072, 0xbcaf89af,
0xde9a771f, 0xd9930810, 0xb38bae12, 0xdccf3f2e, 0x5512721f, 0x2e6b7124,
0x501adde6, 0x9f84cd87, 0x7a584718, 0x7408da17, 0xbc9f9abc, 0xe94b7d8c,
0xec7aec3a, 0xdb851dfa, 0x63094366, 0xc464c3d2, 0xef1c1847, 0x3215d908,
0xdd433b37, 0x24c2ba16, 0x12a14d43, 0x2a65c451, 0x50940002, 0x133ae4dd,
0x71dff89e, 0x10314e55, 0x81ac77d6, 0x5f11199b, 0x043556f1, 0xd7a3c76b,
0x3c11183b, 0x5924a509, 0xf28fe6ed, 0x97f1fbfa, 0x9ebabf2c, 0x1e153c6e,
0x86e34570, 0xeae96fb1, 0x860e5e0a, 0x5a3e2ab3, 0x771fe71c, 0x4e3d06fa,
0x2965dcb9, 0x99e71d0f, 0x803e89d6, 0x5266c825, 0x2e4cc978, 0x9c10b36a,
0xc6150eba, 0x94e2ea78, 0xa5fc3c53, 0x1e0a2df4, 0xf2f74ea7, 0x361d2b3d,
```

```
0x1939260f, 0x19c27960, 0x5223a708, 0xf71312b6, 0xebadfe6e, 0xeac31f66,
0xe3bc4595, 0xa67bc883, 0xb17f37d1, 0x018cff28, 0xc332ddef, 0xbe6c5aa5,
0x65582185, 0x68ab9802, 0xeecea50f, 0xdb2f953b, 0x2aef7dad, 0x5b6e2f84,
0x1521b628, 0x29076170, 0xecdd4775, 0x619f1510, 0x13cca830, 0xeb61bd96,
0x0334fe1e, 0xaa0363cf, 0xb5735c90, 0x4c70a239, 0xd59e9e0b, 0xcbaade14,
0xeecc86bc, 0x60622ca7, 0x9cab5cab, 0xb2f3846e, 0x648b1eaf, 0x19bdf0ca,
0xa02369b9, 0x655abb50, 0x40685a32, 0x3c2ab4b3, 0x319ee9d5, 0xc021b8f7,
0x9b540b19, 0x875fa099, 0x95f7997e, 0x623d7da8, 0xf837889a, 0x97e32d77,
0x11ed935f, 0x16681281, 0x0e358829, 0xc7e61fd6, 0x96dedfa1, 0x7858ba99,
0x57f584a5, 0x1b227263, 0x9b83c3ff, 0x1ac24696, 0xcdb30aeb, 0x532e3054,
0x8fd948e4, 0x6dbc3128, 0x58ebf2ef, 0x34c6ffea, 0xfe28ed61, 0xee7c3c73,
0x5d4a14d9, 0xe864b7e3, 0x42105d14, 0x203e13e0, 0x45eee2b6, 0xa3aaabea,
0xdb6c4f15, 0xfacb4fd0, 0xc742f442, 0xef6abbb5, 0x654f3b1d, 0x41cd2105,
0xd81e799e, 0x86854dc7, 0xe44b476a, 0x3d816250, 0xcf62a1f2, 0x5b8d2646,
0xfc8883a0, 0xc1c7b6a3, 0x7f1524c3, 0x69cb7492, 0x47848a0b, 0x5692b285,
0x095bbf00, 0xad19489d, 0x1462b174, 0x23820e00, 0x58428d2a, 0x0c55f5ea,
0x1dadf43e, 0x233f7061, 0x3372f092, 0x8d937e41, 0xd65fecf1, 0x6c223bdb,
0x7cde3759, 0xcbee7460, 0x4085f2a7, 0xce77326e, 0xa6078084, 0x19f8509e,
0xe8efd855, 0x61d99735, 0xa969a7aa, 0xc50c06c2, 0x5a04abfc, 0x800bcadc,
0x9e447a2e, 0xc3453484, 0xfdd56705, 0x0e1e9ec9, 0xdb73dbd3, 0x105588cd,
0x675fda79, 0xe3674340, 0xc5c43465, 0x713e38d8, 0x3d28f89e, 0xf16dff20,
0x153e21e7, 0x8fb03d4a, 0xe6e39f2b, 0xdb83adf7};
unsigned long ks2[] = {
0xe93d5a68, 0x948140f7, 0xf64c261c, 0x94692934, 0x411520f7, 0x7602d4f7,
0xbcf46b2e, 0xd4a20068, 0xd4082471, 0x3320f46a, 0x43b7d4b7, 0x500061af,
0x1e39f62e, 0x97244546, 0x14214f74, 0xbf8b8840, 0x4d95fc1d, 0x96b591af,
0x70f4ddd3, 0x66a02f45, 0xbfbc09ec, 0x03bd9785, 0x7fac6dd0, 0x31cb8504,
0x96eb27b3, 0x55fd3941, 0xda2547e6, 0xabca0a9a, 0x28507825, 0x530429f4,
0x0a2c86da, 0xe9b66dfb, 0x68dc1462, 0xd7486900, 0x680ec0a4, 0x27a18dee,
0x4f3ffea2, 0xe887ad8c, 0xb58ce006, 0x7af4d6b6, 0xaace1e7c, 0xd3375fec,
0xce78a399, 0x406b2a42, 0x20fe9e35, 0xd9f385b9, 0xee39d7ab, 0x3b124e8b,
0x1dc9faf7, 0x4b6d1856, 0x26a36631, 0xeae397b2, 0x3a6efa74, 0xdd5b4332,
0x6841e7f7, 0xca7820fb, 0xfb0af54e, 0xd8feb397, 0x454056ac, 0xba489527,
0x55533a3a, 0x20838d87, 0xfe6ba9b7, 0xd096954b, 0x55a867bc, 0xa1159a58,
0xcca92963, 0x99e1db33, 0xa62a4a56, 0x3f3125f9, 0x5ef47e1c, 0x9029317c,
0xfdf8e802, 0x04272f70, 0x80bb155c, 0x05282ce3, 0x95c11548, 0xe4c66d22,
0x48c1133f, 0xc70f86dc, 0x07f9c9ee, 0x41041f0f, 0x404779a4, 0x5d886e17,
0x325f51eb, 0xd59bc0d1, 0xf2bcc18f, 0x41113564, 0x257b7834, 0x602a9c60,
0xdff8e8a3, 0x1f636c1b, 0x0e12b4c2, 0x02e1329e, 0xaf664fd1, 0xcad18115,
0x6b2395e0, 0x333e92e1, 0x3b240b62, 0xeebeb922, 0x85b2a20e, 0xe6ba0d99,
0xde720c8c, 0x2da2f728, 0xd0127845, 0x95b794fd, 0x647d0862, 0xe7ccf5f0,
0x5449a36f, 0x877d48fa, 0xc39dfd27, 0xf33e8d1e, 0x0a476341, 0x992eff74,
0x3a6f6eab, 0xf4f8fd37, 0xa812dc60, 0xa1ebddf8, 0x991be14c, 0xdb6e6b0d,
0xc67b5510, 0x6d672c37, 0x2765d43b, 0xdcd0e804, 0xf1290dc7, 0xcc00ffa3,
0xb5390f92, 0x690fed0b, 0x667b9ffb, 0xcedb7d9c, 0xa091cf0b, 0xd9155ea3,
0xbb132f88, 0x515bad24, 0x7b9479bf, 0x763bd6eb, 0x37392eb3, 0xcc115979,
0x8026e297, 0xf42e312d, 0x6842ada7, 0xc66a2b3b, 0x12754ccc, 0x782ef11c,
0x6a124237, 0xb79251e7, 0x06a1bbe6, 0x4bfb6350, 0x1a6b1018, 0x11caedfa,
0x3d25bdd8, 0xe2e1c3c9, 0x44421659, 0x0a121386, 0xd90cec6e, 0xd5abea2a,
0x64af674e, 0xda86a85f, 0xbebfe988, 0x64e4c3fe, 0x9dbc8057, 0xf0f7c086,
0x60787bf8, 0x6003604d, 0xd1fd8346, 0xf6381fb0, 0x7745ae04, 0xd736fccc,
0x83426b33, 0xf01eab71, 0xb0804187, 0x3c005e5f, 0x77a057be, 0xbde8ae24,
0x55464299, 0xbf582e61, 0x4e58f48f, 0xf2ddfda2, 0xf474ef38, 0x8789bdc2,
0x5366f9c3, 0xc8b38e74, 0xb475f255, 0x46fcd9b9, 0x7aeb2661, 0x8b1ddf84,
0x846a0e79, 0x915f95e2, 0x466e598e, 0x20b45770, 0x8cd55591, 0xc902de4c,
0xb90bace1, 0xbb8205d0, 0x11a86248, 0x7574a99e, 0xb77f19b6, 0xe0a9dc09,
0x662d09a1, 0xc4324633, 0xe85a1f02, 0x09f0be8c, 0x4a99a025, 0x1d6efe10,
0x1ab93d1d, 0x0ba5a4df, 0xa186f20f, 0x2868f169, 0xdcb7da83, 0x573906fe,
0xa1e2ce9b, 0x4fcd7f52, 0x50115e01, 0xa70683fa, 0xa002b5c4, 0x0de6d027,
0x9af88c27, 0x773f8641, 0xc3604c06, 0x61a806b5, 0xf0177a28, 0xc0f586e0,
0x006058aa, 0x30dc7d62, 0x11e69ed7, 0x2338ea63, 0x53c2dd94, 0xc2c21634,
0xbbcbee56, 0x90bcb6de, 0xebfc7da1, 0xce591d76, 0x6f05e409, 0x4b7c0188,
```

```
0x39720a3d, 0x7c927c24, 0x86e3725f, 0x724d9db9, 0x1ac15bb4, 0xd39eb8fc,
0xed545578, 0x08fca5b5, 0xd83d7cd3, 0x4dad0fc4, 0x1e50ef5e, 0xb161e6f8,
0xa28514d9, 0x6c51133c, 0x6fd5c7e7, 0x56e14ec4, 0x362abfce, 0xddc6c837,
0xd79a3234, 0x92638212, 0x670efa8e, 0x406000e0};
unsigned long ks3[] = {
0x3a39ce37, 0xd3faf5cf, 0xabc27737, 0x5ac52d1b, 0x5cb0679e, 0x4fa33742,
0xd3822740, 0x99bc9bbe, 0xd5118e9d, 0xbf0f7315, 0xd62d1c7e, 0xc700c47b,
0xb78c1b6b, 0x21a19045, 0xb26eb1be, 0x6a366eb4, 0x5748ab2f, 0xbc946e79,
0xc6a376d2, 0x6549c2c8, 0x530ff8ee, 0x468dde7d, 0xd5730a1d, 0x4cd04dc6,
0x2939bbdb, 0xa9ba4650, 0xac9526e8, 0xbe5ee304, 0xa1fad5f0, 0x6a2d519a,
0x63ef8ce2, 0x9a86ee22, 0xc089c2b8, 0x43242ef6, 0xa51e03aa, 0x9cf2d0a4,
0x83c061ba, 0x9be96a4d, 0x8fe51550, 0xba645bd6, 0x2826a2f9, 0xa73a3ae1,
0x4ba99586, 0xef5562e9, 0xc72fefd3, 0xf752f7da, 0x3f046f69, 0x77fa0a59,
0x80e4a915, 0x87b08601, 0x9b09e6ad, 0x3b3ee593, 0xe990fd5a, 0x9e34d797,
0x2cf0b7d9, 0x022b8b51, 0x96d5ac3a, 0x017da67d, 0xd1cf3ed6, 0x7c7d2d28,
0x1f9f25cf, 0xadf2b89b, 0x5ad6b472, 0x5a88f54c, 0xe029ac71, 0xe019a5e6,
0x47b0acfd, 0xed93fa9b, 0xe8d3c48d, 0x283b57cc, 0xf8d56629, 0x79132e28,
0x785f0191, 0xed756055, 0xf7960e44, 0xe3d35e8c, 0x15056dd4, 0x88f46dba,
0x03a16125, 0x0564f0bd, 0xc3eb9e15, 0x3c9057a2, 0x97271aec, 0xa93a072a,
0x1b3f6d9b, 0x1e6321f5, 0xf59c66fb, 0x26dcf319, 0x7533d928, 0xb155fdf5,
0x03563482, 0x8aba3cbb, 0x28517711, 0xc20ad9f8, 0xabcc5167, 0xccad925f,
0x4de81751, 0x3830dc8e, 0x379d5862, 0x9320f991, 0xea7a90c2, 0xfb3e7bce,
0x5121ce64, 0x774fbe32, 0xa8b6e37e, 0xc3293d46, 0x48de5369, 0x6413e680,
0xa2ae0810, 0xdd6db224, 0x69852dfd, 0x09072166, 0xb39a460a, 0x6445c0dd,
0x586cdecf, 0x1c20c8ae, 0x5bbef7dd, 0x1b588d40, 0xccd2017f, 0x6bb4e3bb,
0xdda26a7e, 0x3a59ff45, 0x3e350a44, 0xbcb4cdd5, 0x72eacea8, 0xfa6484bb,
0x8d6612ae, 0xbf3c6f47, 0xd29be463, 0x542f5d9e, 0xaec2771b, 0xf64e6370,
0x740e0d8d, 0xe75b1357, 0xf8721671, 0xaf537d5d, 0x4040cb08, 0x4eb4e2cc,
0x34d2466a, 0x0115af84, 0xe1b00428, 0x95983a1d, 0x06b89fb4, 0xce6ea048,
0x6f3f3b82, 0x3520ab82, 0x011a1d4b, 0x277227f8, 0x611560b1, 0xe7933fdc,
0xbb3a792b, 0x344525bd, 0xa08839e1, 0x51ce794b, 0x2f32c9b7, 0xa01fbac9,
0xe01cc87e, 0xbcc7d1f6, 0xcf0111c3, 0xa1e8aac7, 0x1a908749, 0xd44fbd9a,
0xd0dadecb, 0xd50ada38, 0x0339c32a, 0xc6913667, 0x8df9317c, 0xe0b12b4f,
0xf79e59b7, 0x43f5bb3a, 0xf2d519ff, 0x27d9459c, 0xbf97222c, 0x15e6fc2a,
0x0f91fc71, 0x9b941525, 0xfae59361, 0xceb69ceb, 0xc2a86459, 0x12baa8d1,
0xb6c1075e, 0xe3056a0c, 0x10d25065, 0xcb03a442, 0xe0ec6e0e, 0x1698db3b,
0x4c98a0be, 0x3278e964, 0x9f1f9532, 0xe0d392df, 0xd3a0342b, 0x8971f21e,
0x1b0a7441, 0x4ba3348c, 0xc5be7120, 0xc37632d8, 0xdf359f8d, 0x9b992f2e,
0xe60b6f47, 0x0fe3f11d, 0xe54cda54, 0x1edad891, 0xce6279cf, 0xcd3e7e6f,
0x1618b166, 0xfd2c1d05, 0x848fd2c5, 0xf6fb2299, 0xf523f357, 0xa6327623,
0x93a83531, 0x56cccd02, 0xacf08162, 0x5a75ebb5, 0x6e163697, 0x88d273cc,
0xde966292, 0x81b949d0, 0x4c50901b, 0x71c65614, 0xe6c6c7bd, 0x327a140a,
0x45e1d006, 0xc3f27b9a, 0xc9aa53fd, 0x62a80f00, 0xbb25bfe2, 0x35bdd2f6,
0x71126905, 0xb2040222, 0xb6cbcf7c, 0xcd769c2b, 0x53113ec0, 0x1640e3d3,
0x38abbd60, 0x2547adf0, 0xba38209c, 0xf746ce76, 0x77afa1c5, 0x20756060,
0x85cbfe4e, 0x8ae88dd8, 0x7aaaf9b0, 0x4cf9aa7e, 0x1948c25c, 0x02fb8a8c,
0x01c36ae4, 0xd6ebe1f9, 0x90d4f869, 0xa65cdea0, 0x3f09252d, 0xc208e69f,
0xb74e6132, 0xce77e25b, 0x578fdfe3, 0x3ac372e6};

/* Initialize s-boxes without file read. */
        for(i=0;i<256;i++){
                c->S[0][i] = ks0[i];
                c->S[1][i] = ks1[i];
                c->S[2][i] = ks2[i];
                c->S[3][i] = ks3[i];
        }


        j = 0;
        for (i = 0; i < N + 2; ++i) {
                data = 0x00000000;
                for (k = 0; k < 4; ++k) {
```

```
                        data = (data << 8) | key[j];
                        j = j + 1;
                        if (j >= keybytes) {
                                j = 0;
                        }
                }
        c->P[i] = c->P[i] ^ data;
        }

     datal = 0x00000000;
        datar = 0x00000000;

        for (i = 0; i < N + 2; i += 2) {
                Blowfish_encipher(c,&datal, &datar);

                c->P[i] = datal;
                c->P[i + 1] = datar;
        }

        for (i = 0; i < 4; ++i) {
                for (j = 0; j < 256; j += 2) {

                        Blowfish_encipher(c,&datal, &datar);

                        c->S[i][j] = datal;
                        c->S[i][j + 1] = datar;
                }
        }
}

void blf_key(blf_ctx *c, char *k, int len){
        InitializeBlowfish(c,k,len);
}

void blf_enc(blf_ctx *c, unsigned long *data, int blocks){
        unsigned long *d;
        int i;

        d = data;
        for(i=0;i<blocks;i++){
                Blowfish_encipher(c,d,d+1);
                d += 2;
        }
}

void blf_dec(blf_ctx *c, unsigned long *data, int blocks){
        unsigned long *d;
        int i;

        d = data;
        for(i=0;i<blocks;i++){
                Blowfish_decipher(c,d,d+1);
                d += 2;
        }
}

void main(void){
        blf_ctx c;
        char key[]="AAAAA";
        unsigned long data[10];
        int i;
```

```
        for(i=0;i<10;i++) data[i] = i;

        blf_key(&c,key,5);
        blf_enc(&c,data,5);
        blf_dec(&c,data,1);
        blf_dec(&c,data+2,4);
        for(i=0;i<10;i+=2) printf("Block %01d decrypts to: %08lx %08lx.\n",
                                  i/2,data[i],data[i+1]);
}
```

3-WAY

```
#define   STRT_E   0x0b0b /* round constant of first encryption round */
#define   STRT_D   0xb1b1 /* round constant of first decryption round */
#define     NMBR       11 /* number of rounds is 11                    */

typedef   unsigned long int  word32 ;
                 /* the program only works correctly if long = 32bits */
typedef unsigned long u4;
typedef unsigned char u1;

typedef struct {
        u4 k[3],ki[3], ercon[NMBR+1],drcon[NMBR+1];
} twy_ctx;

/* Note:  encrypt and decrypt expect full blocks--padding blocks is
          caller's responsibility.  All bulk encryption is done in
          ECB mode by these calls.  Other modes may be added easily
          enough.                                                    */

/* destroy:  Context. */
/* Scrub context of all sensitive data. */
void twy_destroy(twy_ctx *);

/* encrypt:  Context, ptr to data block, # of blocks. */
void twy_enc(twy_ctx *, u4 *, int);

/* decrypt:  Context, ptr to data block, # of blocks. */
void twy_dec(twy_ctx *, u4 *, int);

/* key:  Context, ptr to key data. */
void twy_key(twy_ctx *, u4 *);

/* ACCODE----------------------------------------------------------- */
/* End of AC code prototypes and structures.                         */
/* ----------------------------------------------------------------- */

void mu(word32 *a)       /* inverts the order of the bits of a */
{
int i ;
word32 b[3] ;

b[0] = b[1] = b[2] = 0 ;
for( i=0 ; i<32 ; i++ )
   {
   b[0] <<= 1 ; b[1] <<= 1 ; b[2] <<= 1 ;
   if(a[0]&1) b[2] |= 1 ;
   if(a[1]&1) b[1] |= 1 ;
```

```
    if(a[2]&1) b[0] |= 1 ;
    a[0] >>= 1 ; a[1] >>= 1 ; a[2] >>= 1 ;
    }

a[0] = b[0] ;      a[1] = b[1] ;      a[2] = b[2] ;
}


void gamma(word32 *a)   /* the nonlinear step */
{
word32 b[3] ;

b[0] = a[0] ^ (a[1]|(~a[2])) ;
b[1] = a[1] ^ (a[2]|(~a[0])) ;
b[2] = a[2] ^ (a[0]|(~a[1])) ;

a[0] = b[0] ;      a[1] = b[1] ;      a[2] = b[2] ;
}


void theta(word32 *a)    /* the linear step */
{
word32 b[3];

b[0] = a[0] ^  (a[0]>>16) ^ (a[1]<<16) ^     (a[1]>>16) ^ (a[2]<<16) ^
               (a[1]>>24) ^ (a[2]<<8)  ^     (a[2]>>8)  ^ (a[0]<<24) ^
               (a[2]>>16) ^ (a[0]<<16) ^     (a[2]>>24) ^ (a[0]<<8)  ;
b[1] = a[1] ^  (a[1]>>16) ^ (a[2]<<16) ^     (a[2]>>16) ^ (a[0]<<16) ^
               (a[2]>>24) ^ (a[0]<<8)  ^     (a[0]>>8)  ^ (a[1]<<24) ^
               (a[0]>>16) ^ (a[1]<<16) ^     (a[0]>>24) ^ (a[1]<<8)  ;
b[2] = a[2] ^  (a[2]>>16) ^ (a[0]<<16) ^     (a[0]>>16) ^ (a[1]<<16) ^
               (a[0]>>24) ^ (a[1]<<8)  ^     (a[1]>>8)  ^ (a[2]<<24) ^
               (a[1]>>16) ^ (a[2]<<16) ^     (a[1]>>24) ^ (a[2]<<8)  ;

a[0] = b[0] ;      a[1] = b[1] ;      a[2] = b[2] ;
}

void pi_1(word32 *a)
{
a[0] = (a[0]>>10) ^ (a[0]<<22);
a[2] = (a[2]<<1)  ^ (a[2]>>31);
}

void pi_2(word32 *a)
{
a[0] = (a[0]<<1)  ^ (a[0]>>31);
a[2] = (a[2]>>10) ^ (a[2]<<22);
}

void rho(word32 *a)    /* the round function       */
{
theta(a) ;
pi_1(a) ;
gamma(a) ;
pi_2(a) ;
}

void rndcon_gen(word32 strt,word32 *rtab)
{                            /* generates the round constants */
int i ;

for(i=0 ; i<=NMBR ; i++ )
```

```
    {
    rtab[i] = strt ;
    strt <<= 1 ;
    if( strt&0x10000 ) strt ^= 0x11011 ;
    }
}

/* Modified slightly to fit the caller's needs. */
void encrypt(twy_ctx *c, word32 *a)
{
char i ;
for( i=0 ; i<NMBR ; i++ )
    {
    a[0] ^= c->k[0] ^ (c->ercon[i]<<16) ;
    a[1] ^= c->k[1] ;
    a[2] ^= c->k[2] ^ c->ercon[i] ;
    rho(a) ;
    }
a[0] ^= c->k[0] ^ (c->ercon[NMBR]<<16) ;
a[1] ^= c->k[1] ;
a[2] ^= c->k[2] ^ c->ercon[NMBR] ;
theta(a) ;
}

/* Modified slightly to meet caller's needs. */
void decrypt(twy_ctx *c, word32 *a)
{
char i ;

mu(a) ;
for( i=0 ; i<NMBR ; i++ )
    {
    a[0] ^= c->ki[0] ^ (c->drcon[i]<<16) ;
    a[1] ^= c->ki[1] ;
    a[2] ^= c->ki[2] ^ c->drcon[i] ;
    rho(a) ;
    }
a[0] ^= c->ki[0] ^ (c->drcon[NMBR]<<16) ;
a[1] ^= c->ki[1] ;
a[2] ^= c->ki[2] ^ c->drcon[NMBR] ;
theta(a) ;
mu(a) ;
}

void twy_key(twy_ctx *c, u4 *key){
        c->ki[0] = c->k[0] = key[0];
        c->ki[1] = c->k[1] = key[1];
        c->ki[2] = c->k[2] = key[2];
        theta(c->ki);
        mu(c->ki);
        rndcon_gen(STRT_E,c->ercon);
        rndcon_gen(STRT_D,c->drcon);

}

/* Encrypt in ECB mode. */
void twy_enc(twy_ctx *c, u4 *data, int blkcnt){
        u4 *d;
        int i;

        d = data;
        for(i=0;i<blkcnt;i++) {
```

```
                encrypt(c,d);
                d +=3;
        }
}


/* Decrypt in ECB mode. */
void twy_dec(twy_ctx *c, u4 *data, int blkcnt){
        u4 *d;
        int i;

        d = data;
        for(i=0;i<blkcnt;i++){
                decrypt(c,d);
                d+=3;
        }
}


/* Scrub sensitive values from memory before deallocating. */
void twy_destroy(twy_ctx *c){
        int i;

        for(i=0;i<3;i++) c->k[i] = c->ki[i] = 0;
}

void printvec(char *chrs, word32 *d){
        printf("%20s : %08lx %08lx %08lx \n",chrs,d[2],d[1],d[0]);
}


main()
{
twy_ctx gc;
word32 a[9],k[3];
int i;

/* Test vector 1. */

k[0]=k[1]=k[2]=0;
a[0]=a[1]=a[2]=1;
twy_key(&gc,k);

printf("**********\n");
printvec("KEY = ",k);
printvec("PLAIN = ",a);
encrypt(&gc,a);
printvec("CIPHER = ",a);


/* Test vector 2. */

k[0]=6;k[1]=5;k[2]=4;
a[0]=3;a[1]=2;a[2]=1;
twy_key(&gc,k);

printf("**********\n");
printvec("KEY = ",k);
printvec("PLAIN = ",a);
encrypt(&gc,a);
printvec("CIPHER = ",a);
```

```
/* Test vector 3. */

k[2]=0xbcdef012;k[1]=0x456789ab;k[0]=0xdef01234;
a[2]=0x01234567;a[1]=0x9abcdef0;a[0]=0x23456789;
twy_key(&gc,k);

printf("**********\n");
printvec("KEY = ",k);
printvec("PLAIN = ",a);
encrypt(&gc,a);
printvec("CIPHER = ",a);


/* Test vector 4. */

k[2]=0xcab920cd;k[1]=0xd6144138;k[0]=0xd2f05b5e;
a[2]=0xad21ecf7;a[1]=0x83ae9dc4;a[0]=0x4059c76e;
twy_key(&gc,k);

printf("**********\n");
printvec("KEY = ",k);
printvec("PLAIN = ",a);
encrypt(&gc,a);
printvec("CIPHER = ",a);

/*  TEST VALUES

key        : 00000000 00000000 00000000
plaintext  : 00000001 00000001 00000001
ciphertext : ad21ecf7 83ae9dc4 4059c76e

key        : 00000004 00000005 00000006
plaintext  : 00000001 00000002 00000003
ciphertext : cab920cd d6144138 d2f05b5e

key        : bcdef012 456789ab def01234
plaintext  : 01234567 9abcdef0 23456789
ciphertext : 7cdb76b2 9cdddb6d 0aa55dbb

key        : cab920cd d6144138 d2f05b5e
plaintext  : ad21ecf7 83ae9dc4 4059c76e
ciphertext : 15b155ed 6b13f17c 478ea871

*/

/* Enc/dec test: */
for(i=0;i<9;i++) a[i]=i;
twy_enc(&gc,a,3);
for(i=0;i<9;i+=3) printf("Block %01d encrypts to %08lx %08lx %08lx\n",
                         i/3,a[i],a[i+1],a[i+2]);

twy_dec(&gc,a,2);
twy_dec(&gc,a+6,1);

 for(i=0;i<9;i+=3) printf("Block %01d decrypts to %08lx %08lx %08lx\n",
                         i/3,a[i],a[i+1],a[i+2]);
}
```

RC5

```
#include <stdio.h>

/* An RC5 context needs to know how many rounds it has, and its subkeys. */
typedef struct {
        u4 *xk;
        int nr;
} rc5_ctx;

/* Where possible, these should be replaced with actual rotate instructions.
   For Turbo C++, this is done with _lrotl and _lrotr. */

#define ROTL32(X,C) (((X)<<(C))|((X)>>(32-(C))))
#define ROTR32(X,C) (((X)>>(C))|((X)<<(32-(C))))

/* Function prototypes for dealing with RC5 basic operations. */
void rc5_init(rc5_ctx *, int);
void rc5_destroy(rc5_ctx *);
void rc5_key(rc5_ctx *, u1 *, int);
void rc5_encrypt(rc5_ctx *, u4 *, int);
void rc5_decrypt(rc5_ctx *, u4 *, int);


/* Function implementations for RC5. */

/* Scrub out all sensitive values. */
void rc5_destroy(rc5_ctx *c){
        int i;
     for(i=0;i<(c->nr)*2+2;i++) c->xk[i]=0;
     free(c->xk);
}

/* Allocate memory for rc5 context's xk and such. */
void rc5_init(rc5_ctx *c, int rounds){
     c->nr = rounds;
     c->xk = (u4 *) malloc(4*(rounds*2+2));
}

void rc5_encrypt(rc5_ctx *c, u4 *data, int blocks){
        u4 *d,*sk;
        int h,i,rc;

     d = data;
        sk = (c->xk)+2;
        for(h=0;h<blocks;h++){
                d[0] += c->xk[0];
                d[1] += c->xk[1];
                for(i=0;i<c->nr*2;i+=2){
                        d[0] ^= d[1];
                        rc = d[1] & 31;
                        d[0] = ROTL32(d[0],rc);
                        d[0] += sk[i];
               d[1] ^= d[0];
                        rc = d[0] & 31;
                        d[1] = ROTL32(d[1],rc);
                        d[1] += sk[i+1];
/*printf("Round %03d : %08lx %08lx  sk= %08lx %08lx\n",i/2,
                                d[0],d[1],sk[i],sk[i+1]);*/
                }
          d+=2;
        }
```

```
}

void rc5_decrypt(rc5_ctx *c, u4 *data, int blocks){
     u4 *d,*sk;
        int h,i,rc;

     d = data;
        sk = (c->xk)+2;
     for(h=0;h<blocks;h++){
                for(i=c->nr*2-2;i>=0;i-=2){
/*printf("Round %03d: %08lx %08lx  sk: %08lx %08lx\n",
        i/2,d[0],d[1],sk[i],sk[i+1]); */
                        d[1] -= sk[i+1];
                        rc = d[0] & 31;
                        d[1] = ROTR32(d[1],rc);
                        d[1] ^= d[0];

                        d[0] -= sk[i];
                        rc = d[1] & 31;
                        d[0] = ROTR32(d[0],rc);
               d[0] ^= d[1];
                }
                d[0] -= c->xk[0];
                d[1] -= c->xk[1];
        d+=2;
     }
}

void rc5_key(rc5_ctx *c, u1 *key, int keylen){
     u4 *pk,A,B; /* padded key */
     int xk_len, pk_len, i, num_steps,rc;
     u1 *cp;

     xk_len = c->nr*2 + 2;
     pk_len = keylen/4;
     if((keylen%4)!=0) pk_len += 1;

     pk = (u4 *) malloc(pk_len * 4);
     if(pk==NULL) {
          printf("An error occurred!\n");
          exit(-1);
     }

     /* Initialize pk -- this should work on Intel machines, anyway.... */
     for(i=0;i<pk_len;i++) pk[i]=0;
     cp = (u1 *)pk;
     for(i=0;i<keylen;i++) cp[i]=key[i];

     /* Initialize xk. */
     c->xk[0] = 0xb7e15163; /* P32 */
     for(i=1;i<xk_len;i++) c->xk[i] = c->xk[i-1] + 0x9e3779b9; /* Q32 */

     /* TESTING */
     A = B = 0;
     for(i=0;i<xk_len;i++) {
          A = A + c->xk[i];
          B = B ^ c->xk[i];
     }

     /* Expand key into xk. */
     if(pk_len>xk_len) num_steps = 3*pk_len;else num_steps = 3*xk_len;
```

```
     A = B = 0;
     for(i=0;i<num_steps;i++){
          A = c->xk[i%xk_len] = ROTL32(c->xk[i%xk_len] + A + B,3);
          rc = (A+B) & 31;
          B = pk[i%pk_len] = ROTL32(pk[i%pk_len] + A + B,rc);

     }

     /* Clobber sensitive data before deallocating memory. */
     for(i=0;i<pk_len;i++) pk[i] =0;

     free(pk);
}

void main(void){
     rc5_ctx c;
     u4 data[8];
     char key[] = "ABCDE";
     int i;

     printf("-------------------------------------------------\n");

        for(i=0;i<8;i++) data[i] = i;
     rc5_init(&c,10); /* 10 rounds */
     rc5_key(&c,key,5);

        rc5_encrypt(&c,data,4);
        printf("Encryptions:\n");
        for(i=0;i<8;i+=2) printf("Block %01d = %08lx %08lx\n",
                                 i/2,data[i],data[i+1]);
        rc5_decrypt(&c,data,2);
     rc5_decrypt(&c,data+4,2);
        printf("Decryptions:\n");
        for(i=0;i<8;i+=2) printf("Block %01d = %08lx %08lx\n",
                                 i/2,data[i],data[i+1]);

}
```

A5

```
typedef struct {
        unsigned long r1,r2,r3;
} a5_ctx;

static int threshold(r1, r2, r3)
unsigned int r1;
unsigned int r2;
unsigned int r3;
{
int total;

  total = (((r1 >>  9) & 0x1) == 1) +
          (((r2 >> 11) & 0x1) == 1) +
          (((r3 >> 11) & 0x1) == 1);

  if (total > 1)
    return (0);
  else
```

```
    return (1);
}

unsigned long clock_r1(ctl, r1)
int ctl;
unsigned long r1;
{
unsigned long feedback;

  ctl ^= ((r1 >> 9) & 0x1);
  if (ctl)
  {
    feedback = (r1 >> 18) ^ (r1 >> 17) ^ (r1 >> 16) ^ (r1 >> 13);
    r1 = (r1 << 1) & 0x7ffff;
    if (feedback & 0x01)
      r1 ^= 0x01;
  }
  return (r1);
}

unsigned long clock_r2(ctl, r2)
int ctl;
unsigned long r2;
{
unsigned long feedback;

  ctl ^= ((r2 >> 11) & 0x1);
  if (ctl)
  {
    feedback = (r2 >> 21) ^ (r2 >> 20) ^ (r2 >> 16) ^ (r2 >> 12);
    r2 = (r2 << 1) & 0x3fffff;
    if (feedback & 0x01)
      r2 ^= 0x01;
  }
  return (r2);
}

unsigned long clock_r3(ctl, r3)
int ctl;
unsigned long r3;
{
unsigned long feedback;

  ctl ^= ((r3 >> 11) & 0x1);
  if (ctl)
  {
    feedback = (r3 >> 22) ^ (r3 >> 21) ^ (r3 >> 18) ^ (r3 >> 17);
    r3 = (r3 << 1) & 0x7fffff;
    if (feedback & 0x01)
      r3 ^= 0x01;
  }
  return (r3);
}

int keystream(key, frame, alice, bob)
unsigned char *key;   /* 64 bit session key              */
unsigned long frame;  /* 22 bit frame sequence number    */
unsigned char *alice; /* 114 bit Alice to Bob key stream */
unsigned char *bob;   /* 114 bit Bob to Alice key stream */
{
unsigned long r1;   /* 19 bit shift register */
unsigned long r2;   /* 22 bit shift register */
```

```
unsigned long r3;   /* 23 bit shift register */
int i;              /* counter for loops     */
int clock_ctl;      /* xored with clock enable on each shift register */
unsigned char *ptr; /* current position in keystream */
unsigned char byte; /* byte of keystream being assembled */
unsigned int bits;  /* number of bits of keystream in byte */
unsigned int bit;   /* bit output from keystream generator */

  /* Initialise shift registers from session key */

  r1 = (key[0] | (key[1] << 8) | (key[2] << 16) ) & 0x7ffff;
  r2 = ((key[2] >> 3) | (key[3] << 5) | (key[4] << 13) | (key[5] << 21)) &
0x3fffff;
  r3 = ((key[5] >> 1) | (key[6] << 7) | (key[7] << 15) ) & 0x7fffff;


  /* Merge frame sequence number into shift register state, by xor'ing it
   * into the feedback path
   */

  for (i=0;i<22;i++)
  {
    clock_ctl = threshold(r1, r2, r2);
    r1 = clock_r1(clock_ctl, r1);
    r2 = clock_r2(clock_ctl, r2);
    r3 = clock_r3(clock_ctl, r3);
    if (frame & 1)
    {
      r1 ^= 1;
      r2 ^= 1;
      r3 ^= 1;
    }
    frame = frame >> 1;
  }

  /* Run shift registers for 100 clock ticks to allow frame number to
   * be diffused into all the bits of the shift registers
   */

  for (i=0;i<100;i++)
  {
    clock_ctl = threshold(r1, r2, r2);
    r1 = clock_r1(clock_ctl, r1);
    r2 = clock_r2(clock_ctl, r2);
    r3 = clock_r3(clock_ctl, r3);
  }

  /* Produce 114 bits of Alice->Bob key stream */

  ptr = alice;
  bits = 0;
  byte = 0;
  for (i=0;i<114;i++)
  {
    clock_ctl = threshold(r1, r2, r2);
    r1 = clock_r1(clock_ctl, r1);
    r2 = clock_r2(clock_ctl, r2);
    r3 = clock_r3(clock_ctl, r3);

    bit = ((r1 >> 18) ^ (r2 >> 21) ^ (r3 >> 22)) & 0x01;
    byte = (byte << 1) | bit;
    bits++;
```

```
    if (bits == 8)
    {
      *ptr = byte;
      ptr++;
      bits = 0;
      byte = 0;
    }
  }
  if (bits)
    *ptr = byte;

  /* Run shift registers for another 100 bits to hide relationship between
   * Alice->Bob key stream and Bob->Alice key stream.
   */

  for (i=0;i<100;i++)
  {
    clock_ctl = threshold(r1, r2, r2);
    r1 = clock_r1(clock_ctl, r1);
    r2 = clock_r2(clock_ctl, r2);
    r3 = clock_r3(clock_ctl, r3);
  }

  /* Produce 114 bits of Bob->Alice key stream */

  ptr = bob;
  bits = 0;
  byte = 0;
  for (i=0;i<114;i++)
  {
    clock_ctl = threshold(r1, r2, r2);
    r1 = clock_r1(clock_ctl, r1);
    r2 = clock_r2(clock_ctl, r2);
    r3 = clock_r3(clock_ctl, r3);

    bit = ((r1 >> 18) ^ (r2 >> 21) ^ (r3 >> 22)) & 0x01;
    byte = (byte << 1) | bit;
    bits++;
    if (bits == 8)
    {
      *ptr = byte;
      ptr++;
      bits = 0;
      byte = 0;
    }
  }
  if (bits)
    *ptr = byte;

  return (0);

}

void a5_key(a5_ctx *c, char *k){
        c->r1 = k[0]<<11|k[1]<<3 | k[2]>>5           ; /* 19 */
        c->r2 = k[2]<<17|k[3]<<9 | k[4]<<1 | k[5]>>7; /* 22 */
        c->r3 = k[5]<<15|k[6]<<8 | k[7]              ; /* 23 */
}

/* Step one bit in A5, return 0 or 1 as output bit. */
int a5_step(a5_ctx *c){
        int control;
```

```
        control = threshold(c->r1,c->r2,c->r3);
        c->r1 = clock_r1(control,c->r1);
        c->r2 = clock_r2(control,c->r2);
        c->r3 = clock_r3(control,c->r3);
        return( (c->r1^c->r2^c->r3)&1);
}

/* Encrypts a buffer of len bytes. */
void a5_encrypt(a5_ctx *c, char *data, int len){
        int i,j;
        char t;

        for(i=0;i<len;i++){
                for(j=0;j<8;j++) t = t<<1 | a5_step(c);
                data[i]^=t;
        }
}

void a5_decrypt(a5_ctx *c, char *data, int len){
        a5_encrypt(c,data,len);
}

void main(void){
        a5_ctx c;
        char data[100];
        char key[] = {1,2,3,4,5,6,7,8};
        int i,flag;

        for(i=0;i<100;i++) data[i] = i;

        a5_key(&c,key);
        a5_encrypt(&c,data,100);

        a5_key(&c,key);
        a5_decrypt(&c,data,1);
        a5_decrypt(&c,data+1,99);

        flag = 0;
        for(i=0;i<100;i++) if(data[i]!=i)flag = 1;
        if(flag)printf("Decrypt failed\n"); else printf("Decrypt
succeeded\n");
}
```

SEAL

```
#undef SEAL_DEBUG

#define ALG_OK 0
#define ALG_NOTOK 1
#define WORDS_PER_SEAL_CALL 1024

typedef struct {
     unsigned long t[520]; /* 512 rounded up to a multiple of 5 + 5*/
     unsigned long s[265]; /* 256 rounded up to a multiple of 5 + 5*/
     unsigned long r[20];  /* 16 rounded up to multiple of 5 */
        unsigned long counter; /* 32-bit synch value. */
        unsigned long ks_buf[WORDS_PER_SEAL_CALL];
        int ks_pos;
} seal_ctx;
```

```
#define ROT2(x) (((x) >> 2) | ((x) << 30))
#define ROT9(x) (((x) >> 9) | ((x) << 23))
#define ROT8(x) (((x) >> 8) | ((x) << 24))
#define ROT16(x) (((x) >> 16) | ((x) << 16))
#define ROT24(x) (((x) >> 24) | ((x) << 8))
#define ROT27(x) (((x) >> 27) | ((x) << 5))

#define WORD(cp)  ((cp[0] << 24)|(cp[1] << 16)|(cp[2] << 8)|(cp[3]))

#define F1(x, y, z) (((x) & (y)) | ((~(x)) & (z)))
#define F2(x, y, z) ((x)^(y)^(z))
#define F3(x, y, z) (((x) & (y)) | ((x) & (z)) | ((y) & (z)))
#define F4(x, y, z) ((x)^(y)^(z))

int g(in, i, h)
unsigned char *in;
int i;
unsigned long *h;
{
unsigned long h0;
unsigned long h1;
unsigned long h2;
unsigned long h3;
unsigned long h4;
unsigned long a;
unsigned long b;
unsigned long c;
unsigned long d;
unsigned long e;
unsigned char *kp;
unsigned long w[80];
unsigned long temp;

     kp = in;
     h0 = WORD(kp); kp += 4;
     h1 = WORD(kp); kp += 4;
     h2 = WORD(kp); kp += 4;
     h3 = WORD(kp); kp += 4;
     h4 = WORD(kp); kp += 4;

     w[0] = i;
     for (i=1;i<16;i++)
          w[i] = 0;
     for (i=16;i<80;i++)
          w[i] = w[i-3]^w[i-8]^w[i-14]^w[i-16];

     a = h0;
     b = h1;
     c = h2;
     d = h3;
     e = h4;

     for (i=0;i<20;i++)
     {
          temp = ROT27(a) + F1(b, c, d) + e + w[i] + 0x5a827999;
          e = d;
          d = c;
          c = ROT2(b);
          b = a;
          a = temp;
     }
```

```
	for (i=20;i<40;i++)
	{
		temp = ROT27(a) + F2(b, c, d) + e + w[i] + 0x6ed9eba1;
		e = d;
		d = c;
		c = ROT2(b);
		b = a;
		a = temp;
	}
	for (i=40;i<60;i++)
	{
		temp = ROT27(a) + F3(b, c, d) + e + w[i] + 0x8f1bbcdc;
		e = d;
		d = c;
		c = ROT2(b);
		b = a;
		a = temp;
	}
	for (i=60;i<80;i++)
	{
		temp = ROT27(a) + F4(b, c, d) + e + w[i] + 0xca62c1d6;
		e = d;
		d = c;
		c = ROT2(b);
		b = a;
		a = temp;
	}
	h[0] = h0+a;
	h[1] = h1+b;
	h[2] = h2+c;
	h[3] = h3+d;
	h[4] = h4+e;

	return (ALG_OK);
}

unsigned long gamma(a, i)
unsigned char *a;
int i;
{
unsigned long h[5];

	(void) g(a, i/5, h);
	return h[i % 5];
}

int	seal_init(seal_ctx *result, unsigned char *key)
{
int i;
unsigned long h[5];

	for (i=0;i<510;i+=5)
		g(key, i/5, &(result->t[i]));
	/* horrible special case for the end */
	g(key, 510/5, h);
	for (i=510;i<512;i++)
		result->t[i] = h[i-510];
	/* 0x1000 mod 5 is +1, so have horrible special case for the start */
	g(key, (-1+0x1000)/5, h);
	for (i=0;i<4;i++)
		result->s[i] = h[i+1];
	for (i=4;i<254;i+=5)
```

```
          g(key, (i+0x1000)/5, &(result->s[i]));
     /* horrible special case for the end */
     g(key, (254+0x1000)/5, h);
     for (i=254;i<256;i++)
          result->s[i] = h[i-254];
     /* 0x2000 mod 5 is +2, so have horrible special case at the start */
     g(key, (-2+0x2000)/5, h);
     for (i=0;i<3;i++)
          result->r[i] = h[i+2];
     for (i=3;i<13;i+=5)
          g(key, (i+0x2000)/5, &(result->r[i]));
     /* horrible special case for the end */
     g(key, (13+0x2000)/5, h);
     for (i=13;i<16;i++)
          result->r[i] = h[i-13];
     return (ALG_OK);
}

int seal(seal_ctx *key, unsigned long in, unsigned long *out)
{
int i;
int j;
int l;
unsigned long a;
unsigned long b;
unsigned long c;
unsigned long d;
unsigned short p;
unsigned short q;
unsigned long n1;
unsigned long n2;
unsigned long n3;
unsigned long n4;
unsigned long *wp;

     wp = out;

     for (l=0;l<4;l++)
     {
          a = in ^ key->r[4*l];
          b = ROT8(in) ^ key->r[4*l+1];
          c = ROT16(in) ^ key->r[4*l+2];
          d = ROT24(in) ^ key->r[4*l+3];

          for (j=0;j<2;j++)
          {
               p = a & 0x7fc;
               b += key->t[p/4];
               a = ROT9(a);

               p = b & 0x7fc;
               c += key->t[p/4];
               b = ROT9(b);

               p = c & 0x7fc;
               d += key->t[p/4];
               c = ROT9(c);

               p = d & 0x7fc;
               a += key->t[p/4];
               d = ROT9(d);
```

```
  }
  n1 = d;
  n2 = b;
  n3 = a;
  n4 = c;

  p = a & 0x7fc;
  b += key->t[p/4];
  a = ROT9(a);

  p = b & 0x7fc;
  c += key->t[p/4];
  b = ROT9(b);

  p = c & 0x7fc;
  d += key->t[p/4];
  c = ROT9(c);

  p = d & 0x7fc;
  a += key->t[p/4];
  d = ROT9(d);

/* This generates 64 32-bit words, or 256 bytes of keystream. */
  for (i=0;i<64;i++)
  {
       p = a & 0x7fc;
       b += key->t[p/4];
       a = ROT9(a);
       b ^= a;

       q = b & 0x7fc;
       c ^= key->t[q/4];
       b = ROT9(b);
       c += b;

       p = (p+c) & 0x7fc;
       d += key->t[p/4];
       c = ROT9(c);
       d ^= c;

       q = (q+d) & 0x7fc;
       a ^= key->t[q/4];
       d = ROT9(d);
       a += d;

       p = (p+a) & 0x7fc;
       b ^= key->t[p/4];
       a = ROT9(a);

       q = (q+b) & 0x7fc;
       c += key->t[q/4];
       b = ROT9(b);

       p = (p+c) & 0x7fc;
       d ^= key->t[p/4];
       c = ROT9(c);

       q = (q+d) & 0x7fc;
       a += key->t[q/4];
       d = ROT9(d);

       *wp = b + key->s[4*i];
```

```
                wp++;
                *wp = c ^ key->s[4*i+1];
                wp++;
                *wp = d + key->s[4*i+2];
                wp++;
                *wp = a ^ key->s[4*i+3];
                wp++;

                if (i & 1)
                {
                     a += n3;
                     c += n4;
                }
                else
                {
                     a += n1;
                     c += n2;
                }

          }
     }
     return (ALG_OK);
}

/* Added call to refill ks_buf and reset counter and ks_pos. */
void seal_refill_buffer(seal_ctx *c){
        seal(c,c->counter,c->ks_buf);
        c->counter++;
        c->ks_pos = 0;
}

void seal_key(seal_ctx *c, unsigned char *key){
        seal_init(c,key);
        c->counter = 0;  /* By default, init to zero. */
        c->ks_pos = WORDS_PER_SEAL_CALL;
                /* Refill keystream buffer on next call. */
}

/* This encrypts the next w words with SEAL. */
void seal_encrypt(seal_ctx *c, unsigned long *data_ptr, int w){
        int i;

        for(i=0;i<w;i++){
                if(c->ks_pos>=WORDS_PER_SEAL_CALL) seal_refill_buffer(c);
                data_ptr[i]^=c->ks_buf[c->ks_pos];
                c->ks_pos++;
        }
}

void seal_decrypt(seal_ctx *c, unsigned long *data_ptr, int w) {
        seal_encrypt(c,data_ptr,w);
}

void seal_resynch(seal_ctx *c, unsigned long synch_word){
        c->counter = synch_word;
        c->ks_pos = WORDS_PER_SEAL_CALL;
}


void main(void){
        seal_ctx sc;
        unsigned long buf[1000],t;
```

```
        int i,flag;
        unsigned char key[] =
                {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};

        printf("1\n");
        seal_key(&sc,key);

        printf("2\n");
        for(i=0;i<1000;i++) buf[i]=0;
        printf("3\n");
        seal_encrypt(&sc,buf,1000);
        printf("4\n");
        t = 0;
        for(i=0;i<1000;i++) t = t ^ buf[i];
                printf("XOR of buf is %08lx.\n",t);

        seal_key(&sc,key);
        seal_decrypt(&sc,buf,1);
        seal_decrypt(&sc,buf+1,999);
        flag = 0;
        for(i=0;i<1000;i++) if(buf[i]!=0)flag=1;
        if(flag) printf("Decrypt failed.\n");
        else printf("Decrypt succeeded.\n");


}
```