Исходные коды алгоритмов

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]);

}
```

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);
}
```