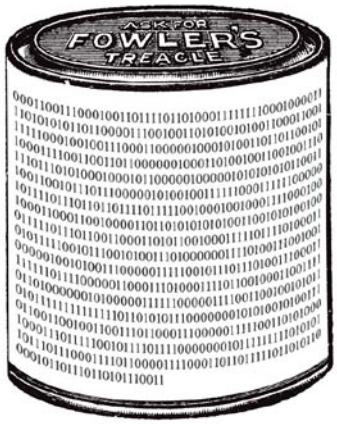
Designing a crypto attack on the Ccrp (bit shuffling) cipher

Dale Thorn



A bitdump (after encryption) of Mary had a little lamb, its fleece was white as snow, and everywhere that Mary went the lamb was sure to go. Note the variable clustering of 1 and 0 bits. This is what actual random ciphertext should look like.

Ccrp was designed to be a highly secure private key encryptor for small files and messages, and uses bit-move logic as the primary means of "scrambling" the plaintext. Ccrp also uses a lookup table instead of a pseudorandom bit generator, and so to obtain good security with that method, the performance of the code is more on the order of a public key program than the private key types that people use for whole-disk encryption.

This article will demonstrate a unique and secure method of encrypting files, where the code contains clear examples of bit manipulation, fast sorting, buffer and file manipulation, and some simple user interface validation. There is little that the reader will have to know beyond the simplest level of programming, if the reader is willing to trace the execution of the code while trying to encrypt a very small example file of, say, two bytes.

Ccrp uses two arrays of variable length for random block size bit moves. The two arrays might start off like the following:

After sorting by the values in the random number array:

Note that after sorting, we discard the lookup table numbers, and use the randomized sequential array to move the bits from the old positions (sequential array index) to the new positions (sequential array contents). No mathematics or "hash" values are required, since all bit positions are accounted for with none missing and no duplicates (more on this below).

Conventional crypto attacks

Conventional attacks range from the physical (trojan horse, keylogger, RFI monitoring) to the electronic (differential analysis, brute force, etc.) to social engineering scams such as the Man In

What you will learn...

- Crypto vulnerabilities and how to address some of them,
- How to manipulate bits,
- How to generate randomness from a lookup table, which is similar to playing card decks and lottery tumblers.

What you should know...

- Have an understanding of the relationship between bits and bytes,
- Be able to read C language code, on a beginner level at least.

The Middle attack for public key systems. In conventional hosted crypto attacks that I'm aware of, the participants presumably use any means available to them to crack the ciphertext, short of physical spying or interrogation and torture, to name a couple of methods that would certainly be disallowed in a major public contest.

In crypto attacks that I've hosted, the challengers focused mainly on getting around the serial numbering (see below) "session key" method I used to make each encryption unique, rather than seriously investigate the possibility of finding a shortcut for decrypting multiple bit shuffled layers in a single pass. While I don't blame anyone for using any legal method to win a crypto challenge where actual prize money is offered, the real purpose of hosting a contest to crack the Ccrp cipher is to determine whether there is a fatal weakness in the cipher itself, rather than an arbitrary implementation detail.

Crypto attack hosting

Some of the best-known Crypto attacks are the Chosen Plaintext Attack and the Chosen Ciphertext Attack. For this article, I'll describe some chosen plaintext attacks I've hosted, and some suggestions for how to improve the methods to have a better chance of success. The primary difference between plaintext and ciphertext attacks, from my point of view, is how to create the crude equivalent of a "session key"; in other words, a computerized modification of the passwords so that each encryption of several files is done differently, even though the passwords entered are the same and the contents of the files may be identical.

The method I've used for plaintext attacks is to use the filename to reiterate the passwords. In a chosen ciphertext attack, the encryption server (ex: an ATM machine) would add a unique serial number to each transaction/file, and that serial number or some iteration of it would modify the encryption, although the serial number itself would not be encrypted. That way, the ATM's decryption process would simply read the plaintext serial number, and in conjunction with the ciphertext, the passwords, and the decryption code, reproduce the plaintext. In my plaintext attack hosting, I've chosen to use the filename rather than add a unique serial number, for simplicity in testing. In a more formal challenge, I would switch to the serial numbering to preclude any tampering, or to alleviate any suspicions about the filename method.

Table 1. *Before Sorting*

Index	Sequential array contents	"Random" array filled from lookup table
0	0	5743
1	1	13496
2	2	17729
3	3	8933
4	4	10150
5	5	14584
6	6	22362
7	7	31955
8	8	2867
9	9	16383

Table 2. *After Sorting*

Index	Sequential array	"Random" array
0	8	2867
1	0	5743
2	3	8933
3	4	10150
4	1	13496
5	5	14584
6	9	16383
7	2	17729
8	6	22362
9	7	31955

Preliminary considerations

In conventional (XOR) cryptography, there is little point in running multiple sessions on a single file (i.e. adding "layers"), since all layers can be decrypted in a single pass by creating and applying a decryption "mask", at least in theory. Multi-layer bit shuffled ciphertext cannot be decrypted in one pass, since there is no XOR process, and the shuffle pattern is different for each layer. You could create a mask after the processing is complete of course, but there would be little point in storing that mask anywhere, unless you created a two-step mask using another plaintext, so that you could "decrypt" your ciphertext to something like "Mary had a little lamb..." etc., the usefulness of which requires no explanation.

The simplest attack on the Ccrp cipher would be to send the host 'n' number of files, where 'n' equals the number of bits in the encrypted contest file. Each of the 'n' files would have one bit on and all other bits off, and the on bit would occupy a different position in each file, i.e. zero through n-1. If there were no filename or serial number used to modify the encryption for each file, then when the host returned the 'n' files to the attacker, encrypted with the same passwords as the encrypted contest file, the attacker could merely look at where each bit was moved to, and un-move the bits in the contest file

the same way, producing the secret plaintext and winning the contest!

At this point in the explanation, most cryptographers would assume that the process is weak, hanging as it were on one little factor, the filename or serial number. But an actual implementation of the cipher is not so simple. In one implementation I'm currently using, the first layer is actually an XOR layer from a bitstream created using the same lookup table as the main algorithm. This layer serves two purposes - one, to obfuscate the nature of the plaintext, should the plaintext have many more zero than one bits or vice-versa, and two, to prevent trial rearrangements of the bits to see if any one pattern comes close to success.

Again at this point, one is tempted to ask - why use this cipher at all? One, because it's based on the randomizing logic that's used in casinos and lotteries, two, because it doesn't incorporate math shortcuts that will allow easy decryption by quantum computers, and three, because the code is so simple that any average technical person can "own" the process and understand every aspect of it. This last point is absolutely vital to security, as history has shown time and again when people trust their vital secrets to erstwhile "trusted" agents.

Preparing the attack

In the current implementation, bits are moved a maximum distance of 189-1/2 bytes (1516 bits) per encryption layer. In a 12-layer encryption, bits would be moved a maximum distance of 2274 bytes, although the vast majority would be reshuffled back and forth to a final destination not far from their original positions. Any attempt to do analysis by shuffling followed by lexical parsing of the result must note that sampling just a few bytes at a time has no possibility of success unless all layers are decrypted in the correct sequence prior to sampling.

Taking into account all of the foregoing, I would not know how to describe a possible attack on the Ccrp cipher if the filename/serial number feature and the first-layer XOR feature were both implemented in the encryption. If, however, we can ignore those features in the following text, and put aside the simple 'n' number of files attack described above, we can examine the algorithm at its core, and if we find that we can successfully compromise that, we can then turn our attention to the implementation features for further analysis.

Ccrp uses the lookup table numbers only to sort the sequential number array, and then the lookup table numbers are discarded. What this means is that the bit move-to positions are based on the relative size of the lookup table numbers compared to each other within the current bit "group" selected by the code. I would guess that we could create an array or lattice representing those relative values for each bit group, where the group size is the first lookup table number selected, and the group members are the next <size> numbers from the lookup table. In real-world terms, this lattice would be quite large for the 1048576 numbers in the current lookup table, and exponentially larger for a lookup table of 1048576 numbers squared, which is a possible implementation. The big lattice would not be a problem for a quantum computer, but it would be unworkable (I think) on a conventional computer.

To simplify this analysis, let's visualize a lookup table of only 32 numbers, similar to the number of balls in a lottery tumbler, or the 52 cards in a standard deck:

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

Again for simplicity, the first number we grab is 5, so the first group size is 5, and the five members of the group are 6, 17, 14, 10, and 26. The relative sizes of these numbers are 1, 4, 3, 2, and 5. Therefore, the first "row" of the lattice would be 1, 4, 3, 2, 5. To generate the second row, we will begin with the second number in the lookup table, and the second row will then have the values 3, 2, 1, 6, 5, 4. Note that when the program reaches the end of the lookup table and requires more values to fill out the bit group size, it simply wraps around to the beginning of the table.

I will leave it to the readers to determine whether a mathematical shortcut can be substituted for the lattice just described, but in any case, it's the key to understanding what occurs within the Ccrp algorithm. If that lattice or the equivalent can be applied to a multi-layer Ccrp ciphertext in a reasonable time frame, then the filename/serial number logic will become irrelevant and perhaps only the above described first-layer XOR coding will prevent the failure of the Ccrp code, or maybe not.

Listing 1 is the DOS-code 'C' language listing. Note the typedefs that are used in the code. Different operating systems may require resizing some variables, and if so, you may have to resize one or more of the 'malloc()' allocations in the 'ifn_cryp' routine. There is a Windows version available in Visual Basic, and the VB code looks nearly identical to the 'C' code presented in Listing 1.

This program is called from a DOS (etc.) command line for encryption as follows:

CCRP filename /e passwordno1 passwordno2 passwordno3 ....

Decryption is called as follows:

CCRP filename /d passwordno1 passwordno2 passwordno3 .... ●

About the Author

Dale Thorn is a software engineer by profession, since 1988. Prior to that he purchased some early personal computers and learned programming and database development while working in a manufacturing environment. The encryption program described here evolved from an original design by Dale for a simple password encryptor. Ccrp has been vetted on the cypherpunks forum of the 1990's, and against the various crypto FAQ's such as the sci.crypt FAQ's.

The contact with the author: *d_t_h_o_r_n@yahoo.com*

Listing 1. DOS- code 'C' language listing

```
/* CCRP.H */
typedef char     C;                      /* char (strings, null-terminated) */
typedef double   D;                      /* double float (double precision) */
typedef float    F;                            /* float (single precision) */
typedef int      I;                               /* short integer (signed) */
typedef long     L;                                /* long integer (signed) */
typedef unsigned int U;                          /* short integer (unsigned) */
typedef unsigned char UC;                              /* unsigned character */
typedef void     V;                                        /* void data type */

I bitget(C *cstr, I ibit);
V bitput(C *cstr, I ibit, I iput);
V ifn_cryp(U ibuf, FILE *ebuf, I iopr, L llof, L lrnd);
V ifn_msgs(C *cmsg, I iofs, I irow, I icol, I ibrp, I iext);
V ifn_read(C *cbuf, L lbyt, U ibuf, FILE *ebuf);
V ifn_sort(I *int1, L *lnt2, I *istk, I imax);
V ifn_write(C *cbuf, L lbyt, U ibuf, FILE *ebuf);
U io_vadr(I inop);
V io_vcls(I iclr);
V io_vcsr(I irow, I icol, I icsr);
V io_vdsp(C *cdat, I irow, I icol, I iclr);
L ltable(L lrnd);

union REGS rg;                           /* DOS registers declaration (video) */
U _far *uvadr = 0;                                 /* video display pointer */

/* CCRP.C */
#include "stdlib.h"
#include "string.h"
#include "stdio.h"
#include "dos.h"
#include "io.h"
#include "ccrp.h"

V main(I argc, C **argv) {               /* get user's command-line arguments */
   C cmsg[64];                          /* initialize the User message string */
   C cwrd[58] = "!#$%&'()+-.0123456789@ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_`{}~□";
   C cwrx[58] = "                                                          ";
   U ibeg;                               /* initialize the loop-begin variable */
   U ibuf = 2048;                        /* set the maximum file buffer length */
   C *cchr;                      /* initialize a temporary character variable */
   U idot;                     /* initialize the filename extension separator */
   U idx2;                             /* initialize a temporary loop variable */
   U iend;                              /* initialize the loop-ending variable */
   U ilen;                            /* initialize a temporary length variable */
   U incr;                            /* initialize the loop-increment variable */
   U indx;                              /* initialize a temporary loop variable */
   I iopr;                                    /* initialize the operation code */
   U iwrd = strlen(cwrd);               /* initialize length of filename chars */
   L llof;                              /* initialize the file length variable */
   L lrnd;                                /* initialize the lookup table value */
   FILE *ebuf;                           /* get next available DOS file handle */
   U _far *uvadr = 0;                                 /* video display pointer */
   I int1[58];                           /* allocate filename sort index array */
   L lnt2[58];                          /* allocate filename sort lookup array */
   I istk[58];                           /* allocate filename sort stack array */

   if (argc == 1) {                        /* a command line was not supplied */
      strcpy(cmsg, "Usage:  CCRP(v4.3)  filename  [/e /d]  [key1  key2  ....]");
      ifn_msgs(cmsg, 4, 24, 79, 0, 1);     /* display usage message and exit */
   }
   if (argc < 4 || argc > 15) {       /* no. of seed keys should be one to 12 */
      ifn_msgs("Invalid number of parameters", 4, 24, 79, 1, 1);
   }                                 /* display error message [above] and exit */
   if (argv[2][0] != '/') {           /* slash preceding opcode param missing */
```

```
   ifn_msgs("Invalid operation parameter", 4, 24, 79, 1, 1);
}                               /* display error message [above] and exit */
strupr(argv[1]);                          /* uppercase the target filename */
strupr(argv[2]);                           /* uppercase the operation code */
if (strchr("ED", argv[2][1]) == NULL) {       /* invalid opcode parameter */
   ifn_msgs("Invalid operation parameter", 4, 24, 79, 1, 1);
}                               /* display error message [above] and exit */
idot = strcspn(argv[1], "."); /* position of filename extension separator */
ilen = strlen(argv[1]);                        /* length of target filename */
if (idot == 0 || idot > 8 || ilen - idot > 4) {        /* filename is bad */
   ifn_msgs("Invalid filename", 4, 24, 79, 1, 1);      /* filename is bad */
}                               /* display error message [above] and exit */
if (idot < ilen) {                 /* filename extension separator found! */
   if (strcspn(argv[1] + idot + 1, ".") < ilen - idot - 1) {
      ifn_msgs("Invalid filename", 4, 24, 79, 1, 1);/* 2nd '.' was found! */
   }                            /* display error message [above] and exit */
   if (idot == ilen - 1) {      /* extension separator at end of filename */
      ilen--;                    /* decrement length of target filename */
      argv[1][ilen] = '\0';      /* decrement length of target filename */
   }
}
ebuf = fopen(argv[1], "rb+");                    /* open the selected file */
llof = filelength(fileno(ebuf));           /* get length of selected file */
if (ebuf == NULL || llof == -1L || llof == 0) {/* length=0 or call failed */
   fclose(ebuf);                                /* close the selected file */
   remove(argv[1]);                            /* kill the zero-length file */
   strcpy(cmsg, argv[1]);                       /* copy filename to message */
   strcat(cmsg, " not found");                 /* add "not found" to message */
   ifn_msgs(cmsg, 4, 24, 79, 1, 1);             /* display message and exit */
}
iopr = argv[2][1] - 68;                   /* opcode (1=encrypt, 0=decrypt) */
if (iopr == 1) {                          /* this is the encrypt operation */
   ibeg = 3;                                /* set the loop-begin variable */
   iend = argc;                             /* set the loop-ending variable */
   incr = 1;                               /* set the loop-increment variable */
} else {                                    /* this is the decrypt operation */
   ibeg = argc - 1;                         /* set the loop-begin variable */
   iend = 2;                                /* set the loop-ending variable */
   incr = -1;                              /* set the loop-increment variable */
}
for (indx = ibeg; indx != iend; indx += incr) {/* loop thru #of seed keys */
   lrnd = atol(argv[indx]) % (L)1048576;      /* get lookup table seed key */
   for (idx2 = 0; idx2 < iwrd; idx2++) {   /* loop through array elements */
      int1[idx2] = idx2;                 /* offsets from current byte offset */
      lrnd = ltable(lrnd);                /* get the next lookup table value */
      lnt2[idx2] = lrnd;                   /* put lookup value to sort array */
   }
   ifn_sort(int1, lnt2, istk, iwrd - 1);              /* sort lookup array */
   for (idx2 = 0; idx2 < iwrd; idx2++) {      /* loop thru filename chars */
      cwrx[int1[idx2]] = cwrd[idx2];
   }                      /* shuffle bytes in valid filename chars [above] */
   lrnd = atol(argv[indx]) % (L)1048576;      /* get lookup table seed key */
   for (idx2 = 0; idx2 < ilen; idx2++) {      /* loop thru filename chars */
      cchr = strchr(cwrx, argv[1][idx2]);      /* filename char. position */
      if (cchr == NULL) {                /* character not found in filename */
         ifn_msgs("Invalid character in filename", 4, 24, 79, 1, 1);
      }                            /* display error message [above] and exit */
      lrnd = (lrnd + (cchr - cwrx + 1)) % (L)1048576;/* add value to seed */
      lrnd = ltable(lrnd);                  /* reiterate value of seed key */
   }
   if (iopr == 1) {                         /* encrypt operation specified */
      ifn_msgs("Encrypting layer", 4, 24, 79, 0, 0);       /* encrypt msg. */
   } else {                                 /* decrypt operation specified */
      ifn_msgs("Decrypting layer", 4, 24, 79, 0, 0);       /* decrypt msg. */
   }
   itoa(indx - 2, cmsg, 10);                   /* convert 'indx' to string */
   ifn_msgs(cmsg, -21, 24, 79, 0, 0);         /* show layer number message */
   ifn_cryp(ibuf, ebuf, iopr, llof, lrnd);           /* encrypt or decrypt */
```

```
   }
   ifn_msgs("Translation complete", 4, 24, 79, 0, 1);
}

V ifn_cryp(U ibuf, FILE *ebuf, I iopr, L llof, L lrnd) {  /* encrypt routine */
   C cmsg[64];                          /* initialize the User message string */
   U ibit = 0;                           /* initialize the bit offset in cbuf */
   I ieof = 0;                                    /* initialize the EOF flag */
   U ilen;                           /* initialize a temporary length variable */
   U indx;                             /* initialize the for-next loop counter */
   L lbyt;                             /* initialize the file pointer variable */
   C *cbuf = (C *)malloc(2048);                /* initialize the file buffer */
   C *ctmp = (C *)malloc(2048);                /* initialize the temp buffer */
   I *int1 = (I *)malloc(3074);             /* allocate the sort index array */
   L *lnt2 = (L *)malloc(6148);          /* allocate sort lookup number array */
   I *istk = (I *)malloc(3074);             /* allocate the sort stack array */

   for (lbyt = 0; lbyt < llof; lbyt += ibuf) {    /* process in ibuf segments */
      if (llof > (L)ibuf) {                     /* so we don't divide by zero */
         ltoa(lbyt / (llof / 100), cmsg, 10);      /* convert pct. to string */
         strcat(cmsg, "%");                   /* append '%' symbol to message */
         ifn_msgs("    ", -24, 24, 79, 0, 0);    /* erase prev.complete msg. */
         ifn_msgs(cmsg, -24, 24, 79, 0, 0);      /* show pct. completed msg. */
      }
      if (lbyt + ibuf >= llof) {     /* current file pointer + ibuf spans EOF */
         ibuf = (U)(llof - lbyt);                /* reset file buffer length */
         ieof = 1;                                    /* set the EOF flag ON */
      }
      ifn_read(cbuf, lbyt, ibuf, ebuf);    /* read data into the file buffer */
      while (1) {                        /* loop to process bit groups in cbuf */
         lrnd = ltable(lrnd);              /* get the next lookup table value */
         ilen = (U)(lrnd / 832 + 256);     /* buffer bitlen: 256<=ilen<=1516 */
         if (ibit + ilen > ibuf * 8) {  /* curr. bit-pointer+ilen spans cbuf */
            if (ieof) {                                    /* EOF flag is ON */
               ilen = ibuf * 8 - ibit; /* reset bit-length of buffer segment */
            } else {                  /* EOF flag is OFF; adjust file pointer */
               ifn_write(cbuf, lbyt, ibuf, ebuf);  /* write data to the file */
               lbyt -= (ibuf - ibit / 8);      /* set lbyt to load from ibit */
               ibit %= 8;             /* set ibit to first byte of <new> cbuf */
               break;                   /* exit loop to reload cbuf from lbyt */
            }
         }                 /* encrypt or decrypt the current segment [below] */
         for (indx = 0; indx < ilen; indx++) {/* loop through array elements */
            int1[indx] = indx;        /* bit offsets from current ibit offset */
            lrnd = ltable(lrnd);           /* get the next lookup table value */
            lnt2[indx] = lrnd;             /* lookup values for sort function */
         }
         ifn_sort(int1, lnt2, istk, ilen - 1);          /* sort lookup array */
         memcpy(ctmp, cbuf, 2048);       /* copy data buffer to dest. buffer */
         if (iopr) {                           /* this is the encrypt operation */
            for (indx = 0; indx < ilen; indx++) {  /* loop through bit group */
               bitput(ctmp, indx + ibit, bitget(cbuf, int1[indx] + ibit));
            }                      /* move bits to "random" positions [above] */
         } else {                              /* this is the decrypt operation */
            for (indx = 0; indx < ilen; indx++) {  /* loop through bit group */
               bitput(ctmp, int1[indx] + ibit, bitget(cbuf, indx + ibit));
            }                 /* restore bits from "random" positions [above] */
         }
         memcpy(cbuf, ctmp, 2048);       /* copy dest. buffer to data buffer */
         ibit += ilen;                  /* increment ibit to next bit-segment */
         if (ibit == ibuf * 8) {         /* loop until ibit == length of cbuf */
            ifn_write(cbuf, lbyt, ibuf, ebuf); /* put current buffer to file */
            ibit = 0;                /* set ibit to first byte of <new> cbuf */
            break;                     /* ibit == length of cbuf; exit loop */
         }
      }
   }
   free(cbuf);                                  /* deallocate the file buffer */
```

```
   free(ctmp);                                /* deallocate the temp buffer */
   free(int1);                               /* deallocate the sort index array */
   free(lnt2);                              /* deallocate the sort lookup array */
   free(istk);                               /* deallocate the sort stack array */
}

I bitget(C *cstr1, I ibit) {                  /* get a bit-value from a string */
   I ival;                                      /* initialize the bit value */

   switch (ibit % 8) {                       /* switch on bit# within character */
      case 0:                                   /* bit #0 in target character */
         ival = 1;                                    /* value of bit #0 */
         break;
      case 1:                                   /* bit #1 in target character */
         ival = 2;                                    /* value of bit #1 */
         break;
      case 2:                                   /* bit #2 in target character */
         ival = 4;                                    /* value of bit #2 */
         break;
      case 3:                                   /* bit #3 in target character */
         ival = 8;                                    /* value of bit #3 */
         break;
      case 4:                                   /* bit #4 in target character */
         ival = 16;                                   /* value of bit #4 */
         break;
      case 5:                                   /* bit #5 in target character */
         ival = 32;                                   /* value of bit #5 */
         break;
      case 6:                                   /* bit #6 in target character */
         ival = 64;                                   /* value of bit #6 */
         break;
      case 7:                                   /* bit #7 in target character */
         ival = 128;                                  /* value of bit #7 */
         break;
      default:
         break;
   }
   return ((cstr1[ibit / 8] & ival) != 0);
}                              /* return the value of the target bit [above] */

V bitput(C *cstr1, I ibit, I iput) {          /* put a bit-value to a string */
   I ival;                                      /* initialize the bit value */
   I ipos = ibit / 8;                       /* position of 8-bit char. in cstr1 */

   switch (ibit % 8) {                       /* switch on bit# within character */
      case 0:                                   /* bit #0 in target character */
         ival = 1;                                    /* value of bit #0 */
         break;
      case 1:                                   /* bit #1 in target character */
         ival = 2;                                    /* value of bit #1 */
         break;
      case 2:                                   /* bit #2 in target character */
         ival = 4;                                    /* value of bit #2 */
         break;
      case 3:                                   /* bit #3 in target character */
         ival = 8;                                    /* value of bit #3 */
         break;
      case 4:                                   /* bit #4 in target character */
         ival = 16;                                   /* value of bit #4 */
         break;
      case 5:                                   /* bit #5 in target character */
         ival = 32;                                   /* value of bit #5 */
         break;
      case 6:                                   /* bit #6 in target character */
         ival = 64;                                   /* value of bit #6 */
         break;
      case 7:                                   /* bit #7 in target character */
         ival = 128;                                  /* value of bit #7 */
         break;
```

```
      default:
         break;
   }
   if (iput) {                                    /* OK to set the bit ON */
      if (!(cstr1[ipos] & ival)) {                /* bit is NOT already ON */
         cstr1[ipos] += ival;                   /* set bit ON by adding ival */
      }
   } else {                                       /* OK to set the bit OFF */
      if (cstr1[ipos] & ival) {                  /* bit is NOT already OFF */
         cstr1[ipos] -= ival;                   /* set bit OFF by subt. ival */
      }
   }
}

V ifn_sort(I *int1, L *lnt2, I *istk, I imax) {  /* array Quicksort function */
   I iex1;                              /* initialize the outer-loop exit flag */
   I iex2;                              /* initialize the inner-loop exit flag */
   I ilap;                                 /* initialize the low array pointer */
   I ilsp;                                 /* initialize the low stack pointer */
   I irdx = 0;                                    /* initialize the sort radix */
   I itap;                                 /* initialize the top array pointer */
   I itsp;                                 /* initialize the top stack pointer */
   I iva1;                  /* initialize array value from low stack pointer */
   L lva2;                  /* initialize array value from low stack pointer */

   istk[0] = 0;                            /* initialize the low array pointer */
   istk[1] = imax;                         /* initialize the top array pointer */
   while (irdx >= 0) {                            /* loop until sort radix < 0 */
      ilsp = istk[irdx + irdx];                   /* set the low stack pointer */
      itsp = istk[irdx + irdx + 1];               /* set the top stack pointer */
      irdx--;                                      /* decrement the sort radix */
      iva1 = int1[ilsp];            /* get array value from low stack pointer */
      lva2 = lnt2[ilsp];            /* get array value from low stack pointer */
      ilap = ilsp;                                /* set the low array pointer */
      itap = itsp + 1;                            /* set the top array pointer */
      iex1 = 0;                            /* initialize the outer-loop exit flag */
      while (!iex1) {                      /* loop to sort within the radix limit */
         itap--;                              /* decrement the top array pointer */
         if (itap == ilap) {           /* top array pointer==low array pointer */
            iex1 = 1;                         /* set the outer-loop exit flag ON */
         } else if (lva2 > lnt2[itap]) {  /* value @low ptr > value @top ptr */
            int1[ilap] = int1[itap];          /* swap low and top array values */
            lnt2[ilap] = lnt2[itap];          /* swap low and top array values */
            iex2 = 0;                    /* initialize the inner-loop exit flag */
            while (!iex2) {           /* loop to compare and swap array values */
               ilap++;                        /* increment the low array pointer */
               if (itap == ilap) {    /* top array pointer==low array pointer */
                  iex1 = 1;                   /* set the outer-loop exit flag ON */
                  iex2 = 1;                   /* set the inner-loop exit flag ON */
               } else if (lva2 < lnt2[ilap]) {/* value@low ptr<value@low ptr */
                  int1[itap] = int1[ilap];  /* swap top and low array values */
                  lnt2[itap] = lnt2[ilap];  /* swap top and low array values */
                  iex2 = 1;                   /* set the inner-loop exit flag ON */
               }
            }
         }
      }
      int1[ilap] = iva1;            /* put array value from low stack pointer */
      lnt2[ilap] = lva2;            /* put array value from low stack pointer */
      if (itsp - ilap > 1) {                      /* low segment-width is > 1 */
         irdx++;                                  /* increment the sort radix */
         istk[irdx + irdx] = ilap + 1;             /* reset low array pointer */
         istk[irdx + irdx + 1] = itsp;             /* reset top array pointer */
      }
      if (itap - ilsp > 1) {                     /* top segment-width is > 1 */
         irdx++;                                 /* increment the sort radix */
         istk[irdx + irdx] = ilsp;                /* reset low array pointer */
         istk[irdx + irdx + 1] = itap - 1;        /* reset top array pointer */
```

```
      }
   }
}

V ifn_msgs(C *cmsg, I iofs, I irow, I icol, I ibrp, I iext) {/* display msgs */
   if (iofs >= 0) {                                    /* OK to clear screen */
      io_vcls(7);                                        /* clear the screen */
   }

   io_vdsp(cmsg, 4, abs(iofs), 7);                 /* display the user message */
   if (ibrp) {                                  /* OK to sound user-alert (beep) */
      printf("\a");                                     /* sound the user-alert */
   }
   if (iext) {                                      /* OK to exit the program */
      io_vcsr(5, 0, 0);                                /* relocate the cursor */
      fcloseall();                                    /* close all open files */
      exit(0);                                               /* return to DOS */
   } else {                                          /* do NOT exit the program */
      io_vcsr(irow, icol, 0);                           /* 'hide' the cursor */
   }
}

L ltable(L lrnd) {                                  /* get next lookup table no.*/
   L l1;                                            /* initialize temp value #1 */
   L l2;                                            /* initialize temp value #2 */
   L l3;                                            /* initialize temp value #3 */
   L l4;                                            /* initialize temp value #4 */

   l1 = lrnd % 8;                          /* These 5 lines are an integer-only */
   l2 = (lrnd - l1) % 16;                   /* equivalent to the floating-point */
   l3 = (lrnd - l1 - l2) % 64;             /* operations formerly used in this, */
   l4 = (lrnd - l1 - l2 - l3);            /* the 16-bit DOS version of the code */
   return (l1 * 214013 + l2 * 82941 + l3 * 17405 + l4 * 1021 + 2531011) % 1048576;
}

V ifn_read(C *cbuf, L lbyt, U ibuf, FILE *ebuf) {    /* read from binary file */
   fseek(ebuf, lbyt, SEEK_SET);                  /* set the buffer-read pointer */
   fread((V *)cbuf, 1, ibuf, ebuf);           /* read data from the binary file */
}

V ifn_write(C *cbuf, L lbyt, U ibuf, FILE *ebuf) {   /* write to binary file */
   fseek(ebuf, lbyt, SEEK_SET);                 /* set the buffer-write pointer */
   fwrite((V *)cbuf, 1, ibuf, ebuf);           /* write data to the binary file */
}

U io_vadr(I inop) {                          /* get video address (color or b/w) */
   rg.h.ah = 15;                                     /* video-address function */
   int86(0x10, &rg, &rg);                         /* call DOS for video address */
   if (rg.h.al == 7) {                                  /* register A-low is 7 */
      return(0xb000);                                    /* return b/w address */
   } else {                                         /* register A-low is NOT 7 */
      return(0xb800);                                 /* return color address */
   }
}

V io_vcls(I iclr) {                                    /* clear screen function */
   I irow;                                  /* initialize the row number variable */
   C cdat[81];                                 /* initialize the row data buffer */

   memset(cdat, ' ', 80);                           /* clear the row data buffer */
   cdat[80] = '\0';                              /* terminate the row data buffer */
   for (irow = 0; irow < 25; irow++) {              /* loop thru the screen rows */
      io_vdsp(cdat, irow, 0, iclr);          /* display each <blank> screen row */
   }
}

V io_vcsr(I irow, I icol, I icsr) {          /* set cursor position [and size] */
   rg.h.ah = 2;                                        /* cursor-position function */
   rg.h.bh = 0;                                                /* video page zero */
```

```
   rg.h.dh = (C)irow;                                    /* row number */
   rg.h.dl = (C)icol;                                   /* column number */
   int86(0x10, &rg, &rg);                        /* call DOS to position cursor */
   if (icsr) {                                        /* cursor-size specified */
      rg.h.ah = 1;                                     /* cursor-size function */
      rg.h.ch = (C)(13 - icsr);                       /* set cursor-begin line */
      rg.h.cl = 12;                                     /* set cursor-end line */
      int86(0x10, &rg, &rg);                     /* call DOS to set cursor size */
   }
}

V io_vdsp(C *cdat, I irow, I icol, I iclr) {        /* display data on screen */
   I ilen = strlen(cdat);                 /* length of string to be displayed */
   I iptr;                                /* byte-counter for displayed string */
   U uclr = iclr * 256;                  /* unsigned attribute high-byte value */

   if (!uvadr) {                                 /* video pointer segment not set */
      FP_SEG(uvadr) = io_vadr(0);                 /* set video pointer segment */
   }
   FP_OFF(uvadr) = irow * 160 + icol * 2;         /* set video pointer offset */
   for (iptr = 0; iptr < ilen; iptr ++) {       /* loop thru displayed string */
      *uvadr = uclr + (UC)cdat[iptr];             /* put data to video memory */
      uvadr++;                                 /* increment video display pointer */
   }
}
```