****

COMPUTER
SUPPLEMENT #16

****

In this issue:

A SURVEY OF DATA ENCRYPTION — A review of different cryptographic systems and their application to computers.

A THREE ROTOR ENCRYPTION SYSTEM — MONIOC details a rotor system based on the Japanese Purple code.

SOUNDEX PATTERNS — Another way to pattern words when some letters may be unknown.

LETTER COUNTING — BASIC and C programs to compute letter frequency and variety of contact.

SON OF PERMUTE — TATTERS gives a faster permutation program.

Plus: News and notes for computerists interested in cryptography, and cryptographers interested in computers.

Published for the AMERICAN CRYPTOGRAM ASSOCATION

INTRODUCTORY MATERIAL

The ACA and Your Computer (1p). Background on the ACA for computerists. (As printed in *ACA and You*, 1988 edition; [Also on Issue Disk #11]

Using Your Home Computer (1p). Ciphering at the ACA level with a computer. (As printed in *ACA and You*, 1988 edition).

Frequently Asked Questions (approx. 20p) with answers, from the Usenet newsgroup `sci.crypt`.

REFERENCE MATERIAL

BASICBUGS - Bugs and errors in GWBASIC (1p). [Also on Issue Disk #11].
BIBLIOG — A bibliography of computer magazine articles and books dealing with cryptography (2p). (Updated August 89). [available on Issue Disk #11].

CRYPTOSUB - Complete listing of Cryptographic Substitution Program as published by PHOENIX in sections in *The Cryptogram* 1983–1985. (With updates from CS #2,3). [available on Issue Disk #3].

DISKEX - A list of programs and reference data available on disk in various formats (Apple—Atari—TRS80—Commodore—IBM—Mac). Revised March 1990.

ERRATA sheet and program index for Caxton Foster's *Cryptanalysis for Microcomputers* (3p). (Reprint from CS #5,6,7 and 9) [disk available from TATTERS with revised programs].

BACK ISSUES

$2.50 per copy. All back issues prior to 13 have been exhausted, and are awaiting reprinting. Contact the Editor for current availability.

ISSUE DISKS

$5 per disk; specify issue(s), format and density required. All issues are presently available on two IBM High Density 1.2M disks, archived with PKZIP. For other disk formats, ask. Disk One — Issues 1 - 10; Disk Two — issues 11 to current. Disks contain ONLY programs and data discussed in the issue. Programs are generally BASIC or Pascal, and almost all executables are for IBM PC–compatible computers. Issue text in TEX format is available for issues 16 to current. Available from the Editor.

TO OBTAIN THESE MATERIALS

Write to:	Or via Electronic Mail:
Dan Veeneman PO Box 2442 Columbia, Maryland 21045-2442, USA.	dan%decode.UUCP@uunet.uu.net or uunet!anagld!decode!dan

Allow 6–8 weeks for delivery. No charge for hard copies, but contributions to postage appreciated. Disk charge $5 per disk; specify format and density required. ACA Issue Disks and additional crypto material resides on `Decode`, the ACA Bulletin Board system, `+1 410 730 6734`, available 24 hours a day, 7 days a week, 300/1200/2400/9600 baud, 8 bits, No Parity, 1 stop bit. All callers welcome.

SUBSCRIPTION

Subscriptions are open to paid-up members of the American Cryptogram Association at the rate of US$2.50 per issue. Contact the Editor for non-member rates. Published three times a year or as submitted material warrants. Write to `Dan Veeneman, PO Box 2442, Columbia, MD, 21045-2442, USA.` Make checks payable to Dan Veeneman. UK subscription requests may be sent to G4EGG.

CHECK YOUR SUBSCRIPTION EXPIRATION by looking at the `Last Issue =` number on your address label. You have paid for issues up to and including this number.

SURVEY OF DATA ENCRYPTION

John A. Thomas
CompuServe: 75236,3536
101 N.W. Eighth St.
Grand Prairie, TX 75050

Introduction

The following article is a survey of data encryption. It is intended to provoke discussion among the members of this forum and perhaps lead to a creative exchange of ideas. Although the basics of the subject seem to be known to few programmers, it embraces many interesting and challenging programming problems, ranging from the optimization of machine code for maximum throughput to the integration of encryption routines into editors, communications packages, and perhaps products as yet not invented. Governments have dominated this technology up until the last few years, but now the need for privacy and secrecy in the affairs of a computer–using public has made it essential that programmers understand and apply the fundamentals of data encryption.

Some Cryptographic Basics

A few definitions are appropriate first. We use the term "encryption" to refer to the general process of making plain information secret and making secret information plain. To "encipher" a file is to transform the information in the file so that it is no longer directly intelligible. The file is then said to be in "ciphertext". To "decipher" a file is to transform it so that it is directly intelligible; that is, to recover the "plaintext."

The two general devices of encryption are "ciphers" and "codes" A cipher works on the individual letters of an alphabet, while a code operates on some higher semantic level, such as whole words or phrases. Cipher systems may work by transposition (shuffling the characters in a message into some new order), or by substitution (exchanging each character in the message for a different character according to some rule), or a combination of both. In modern usage, transposition is often called "permutation." A cipher which employs both transposition and substitution is called a "product" cipher. In general, product ciphers are stronger than those using transposition or substitution alone. Shannon[5] referred to substitution as "confusion" because the output is a non-linear function of the input, thus creating confusion as to the set of input characters. He referred to transposition as "diffusion" because it spreads the dependence of the output from a small number of input positions to a larger number.

Every encryption system has two essential parts: an algorithm for enciphering and deciphering, and a "key" which consists of information to be combined with the plaintext according to the dictates of the algorithm. In any modern encryption system, the algorithm is assumed to be known to an opponent, and the security of the system rests entirely in the secrecy of the key.

Our goal is to translate the language of the plaintext to a new "language" which cannot convey meaning without the additional information in the key. Those fa-

miliar with the concept of "entropy" in physics may be surprised to learn that it is also useful in information theory and cryptography. Entropy is a measure of the amount of disorder in a physical system, or the relative absence of information in a communication system. A natural language such as English has a low entropy because of its redundancies and statistical regularities. Even if many of the characters in a sentence are missing or garbled, we can usually make a good guess as to its meaning. Conversely, we want the language of our ciphertext to have as high an entropy as possible; ideally, it should be utterly random. Our guiding principle is that we must increase the uncertainty of the cryptanalyst as much as possible. His uncertainty should be so great that he cannot make any meaningful statement about the plaintext after examining the ciphertext; also, he must be just as uncertain about the key, even if he has the plaintext itself and the corresponding ciphertext (In practice, it is impossible to keep all plaintext out of his hands).

A prime consideration in the security of an encryption system is the length of the key. If a short key (i.e., short compared with the length of the plaintext) is used, then the statistical properties of the language will begin to "show through" in the ciphertext as the key is used over and over, and a cryptanalyst will be able to derive the key if he has enough ciphertext to work with. On the other hand, we want a relatively short key, so that it can be easily stored or even remembered by a human. The government or a large corporation may have the means to generate and store long binary keys, but we cannot assume that the personal computer user will be able to do so.

The other important fact about the keys is that there must be very many of them. If our system allows only 10,000 different keys, for example, it is not secure, because our opponent could try every possible key in a reasonable amount of time. This introduces the concept of the "work factor" required to break an encryption system. We may not have a system unbreakable in principle, but if we can make the work factor for breaking so high it is not practical for our opponent to do so, then it is irrelevant that the system may be less strong than the ideal. What constitutes an adequate work factor depends essentially on the number of uncertainties the cryptanalyst must resolve before he can derive plaintext or a key. In these days of constantly improving computers, that number should probably exceed 2^{128}. It is easy to quantify the work factor if we are talking about exhaustive key trial, but few modern ciphers are likely to be broken by key trial, since it is too easy to make the key space very large. Most likely they will be broken because of internal periodicities and subtle dependency of output on input which give the cryptanalyst enough information to reduce his uncertainty by orders of magnitude.

A corollary to work factor is the rule that a system need only be strong enough to protect the information for however long it has value. If a system can be broken in a week, but not sooner, then it may be good enough, if the information has no value to an opponent after a week.

Cryptanalysis

Cryptanalysis is the science of deriving plaintext without the key information. Anyone intending to design an encryption system must acquaint himself to some degree with cryptanalytic methods. The

methods of attack may range from sophisticated statistical analysis of ciphertext to breaking into the opponent's office and stealing his keys ("practical cryptanalysis"). There are no rules of fair play. The cryptanalyist is free to use his puzzle–solving ingenuity to the utmost, even to the point of applying the knowledge that your dog's name is "Pascal", and that you might be lazy enough to use that as your key for the day.

The cryptanalyst may have only ciphertext to work with, or he may have both ciphertext and the corresponding plaintext, or he may be able to obtain the encipherment of chosen plaintext. Some cryptographic systems are fairly strong if the analyst is limited to ciphertext, but fail completely if he has corresponding plaintext. Your system should be strong enough to resist attack even if your opponent has both plaintext and ciphertext.

Computer power can greatly aid cryptanalysis, but many systems that appear strong can be broken with pencil-and-paper methods. For example, the Vigenere family of polyalphabetic ciphers was generally believed to be unbreakable up until the late nineteenth century. A polyalphabetic cipher is a substitution cipher in which a different alphabet is used for each character of plaintext. In these systems, the key determines the order of the substitution alphabets, and the cycle repeats with a period equal to the length of the key. This periodicity is a fatal weakness, since fairly often a repeated letter or word of plaintext will be enciphered with the same key letters, giving identical blocks of ciphertext. This exposes the length of the key. Once we have the length of the key, we use the known letter frequencies of the language to gradually build and test hypotheses about the key. Vigenere ciphers can be easily implemented on computers, but they are worthless today. A designer without knowledge of cryptanalysis however, might be just as ignorant of this fact as his colleagues of the last century. Please see the references at the end of this article for information on cryptanalytic technique.

A Survey of Cryptographic systems

We now review some representative encryption schemes, starting with traditional ones and proceeding to the systems which are only feasible to implement on computers.

The infinite–key cipher, also known as the "one time pad," is simple in concept. We first generate a key which is random and at least the same length as our message. Then, for each character of plaintext, we add the corresponding character of the key, to give the ciphertext. By "addition," we mean some reversible operation; the usual choice is the exclusive-or. A little reflection will show that given a random key at least the size of the plaintext (i.e., "infinite" with respect to the plaintext because it is never repeated), then the resulting cipher is unbreakable, even in principle. This scheme is in use today for the most secret government communications, but it presents a serious practical problem with its requirement for a long random key for each message and the need to somehow send the lengthy key to the recipient. Thus the ideal infinite key system is not practical for large volumes of message traffic. It is certainly not practical for file encryption on computers, since where would the key be stored ? Be wary of schemes which use software random–number generators to supply the "infinite" key. Typical random–number algorithms use the preceeding random number to generate the succeeding

number, and can thus be solved if only one number in the sequence is found.

Some ciphers have been built to approximate the infinite–key system by expanding a short key. The Vernam system for telegraph transmission used long paper tapes containing random binary digits (Baudot code, actually) which were exclusively-or'ed with the message digits. To achieve a long key stream, Vernam and others used two or more key tapes of relatively prime lengths, giving a composite key equal to their product. The system is still not ideal, since eventually the key stream will repeat, allowing the analyst to derive the length and composition of the keys, given enough ciphertext. There are other ways to approach the infinite–key ideal, some of which are suggested in the author's article (with Joan Thersites) in the August 1984 issue of *Doctor Dobbs Journal*.

The "rotor" systems take their name from the electromechanical devices of World War II, the best known being perhaps the German ENIGMA. The rotors are wheels with characters inscribed on their edges, and with electrical contacts corresponding to the letters on both sides. A plaintext letter enters on one side of the rotor and is mapped to a different letter on the other side before passing to the next rotor, and so on. All of the rotors (and there may be few or many) are then stepped, so that the next substitution is different. The key is the arrangement and initial setting of the rotor disks. These devices are easy to implement in software and are fairly strong. They can be broken however; the British solution of the ENIGMA is an interesting story outside the scope of this note. If you implement a rotor system, consider having it operate on bits or nybbles instead of bytes, consider adding permutation stages, and consider how you are going to generate the rotor tables, since you must assume these will become known to an opponent.

In 1977 the National Bureau of Standards promulgated the Data Encryption Standard (DES) as the encryption system to be used by all federal agencies (except for those enciphering data classified under any of the National Security Acts). The standard is available in a government publication and also in a number of books. The DES was intended to be implemented only in hardware, probably because its designers did not want users to make changes to its internal tables. However, DES has been implemented in software and is available in several microcomputer products (such as Borland's Superkey or IBM's Data Encoder). [Editor's Note: DES implementations, including source code, are also available on Apres, the ACA bulletin board system. DMV]

The DES is a product cipher using 16 stages of permutation and substitution on blocks of 64 bits each. The permutation tables are fixed, and the substitutions are determined by bits from a 56–bit key and the message block. This short key has caused some experts to question the security of DES. Controversy also exists regarding the involvement of the National Security Agency in parts of the DES design. The issues are interesting, but beyond the scope of this note.

Since DES was intended for hardware implementation, it is relatively slow in software. Software implementations of DES are challenging because of the bit–manipulation required in the key scheduling and permutation routines of the algorithm. Some implementations gain speed at the expense of code size by using large pre–computed tables.

The public key cipher is an interesting new development which shows potential for making other encryption systems obsolete. It takes its name from the fact that the key information is divided into two parts, one of which can be made public. A person with the public key can encipher messages, but only one with the private key can decipher them. All of the public key systems rely on the existence of certain functions for which the inverse is very difficult to compute without the information in the private key. These schemes do not appear to be practical for microcomputers if their strength is fully exploited, at least for eight-bit machines. One variety of public key system (the "knap–sack") has been broken by solution of its enciphering function, but this is no reflection on other systems, such as the RSA scheme, which use different enciphering functions. All public–key systems proposed to date require heavy computation, such as the exponentiation and division of very large numbers (200 decimal digits for the RSA scheme). On the other hand, a public–key system that worked at only 10 bytes/sec might be useful if all we are sending are the keys for some other system, such as the DES.

Some random thoughts

To wrap up this too-lengthy exposition, I append a few questions for the readers:

Must we operate on blocks instead of bytes ? Block ciphers seem stronger, since they allow for permutation. On the other hand, they make life difficult when file size is not an integral multiple of the block size.

Can we make a file encryption system independent of the Operating System ? This is related to the question above on blocks vs bits. How do we define the end-of-file if the plaintext is ASCII and the ciphertext can be any 8–bit value ?

Can we find an efficient way to generate and store a random key for the infinite–key system ? Hardware random–number generators are not hard to build, but would they be of any use ?

Bit–fiddling is expensive. Can it be avoided and still leave a secure system ? What are the relative costs of manipulating bits on the Z80 vs the 68000, for example ?

No file–encryption system can erase a file logically and be considered secure. The information can be recovered until it is overwritten. Overwriting files adds to processing time. I am informed that it is possible to reliably extract information even from sectors that HAVE been overwritten. Is this so ? [Editor's Note: Yes, it is possible to recover information from disks that have been overwritten. For more information on the subject, see National Computer Security Center publication NCSC-TG-025, *A Guide to Understanding Data Remnance in Automated Information Systems* from September 1991. DMV] If it is, what is the solution ?

How do we integrate encryption systems into different tools ? Should a telecommunications program transparently encrypt data if the correspondent is compatible ? What about an editor–encryption system wherein plaintext would never exist on the disk, only on the screen ? How would we manage to encipher/decipher text as we scroll through it and make changes, and still get acceptable performance ?

By their nature, encryption schemes are difficult to test. In practice, we can only have confidence that a system is strong after it has been subjected to repeated attack

and remained unbroken. What test might we subject a system to that would increase our confidence in it ?

References

Here are a few useful books and articles. This is by no means a complete bibliography of the subject:

[1] Kahn, David. *The Code Breakers*. The basic reference for the history of cryptography and cryptanalysis. Use it to learn where others have gone wrong.

[2] Konheim, Alan G. *Cryptography, A Primer*. Survey of cryptographic systems from a mathematical perspective. Discusses rotor systems and the DES in great detail.

[3] Sinkov, Abraham. *Elementary Cryptanalysis*. Very basic, but very useful, introduction to the mathematical concepts of cryptanalysis.

[4] Foster, Caxton C. *Cryptanalysis for Microcomputers*. Covers the cryptanalysis of simple systems, but still a good introduction to cryptanalytic technique. Describes the operation of many traditional systems in detail.

[5] Shannon, Claude. *Communication Theory of Secrecy Systems*. Bell System Technical Journal (October 1949) : 656-715. Discusses secrecy systems from viewpoint of information theory. No practical tips, but useful orientation.

[6] Rivest, R. et al. *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*. Comm. of the ACM, Vol. 21, No. 2, (February 1978) : 120-126. This article describes what has come to be known as the RSA public-key system.

[7] *Data Encryption Standard*, Federal Information Processing Standard (FIPS), Publication No. 46, National Bureau of Standards, U.S. Dept. of Commerce, January, 1977.

CRYPTOSYSTEMS JOURNAL

Tony Patti has once again come out with an excellent issue of *Cryptosystems Journal*. Volume 2 Number 2 covers the SUMMIT cryptosystem, and includes articles on the *Ranger* device, computer graphics, reviews of books and products, and many other interesting topics. The tome runs some 118 pages, and comes with a program diskette.

Cryptosystems Journal is available from:

```
Tony Patti
P.O. Box 188
Newtown, PA  18940--0188
USA
(215) 579--9888
```

SOUNDEX

In large databases containing many names, there may be several names that are spelled almost identically, but refer to distinctly different people. Locating a person when one is unsure of the exact spelling is a common problem. One technique used to overcome this problem is the Soundex Method. This algorithm converts similar-sounding names into identical codes that allow matching based on less-than-exact criteria.

For instance, the last names "Smith", "Smyth", and "Smithe" sound the same, but are spelled differently. The Soundex Method would convert each of these names into the same code (S530), allowing someone looking for last names sounding like "Smith" to search for the Soundex code "S530".

For some of you in the United States, the Soundex code may appear as the first part of your Driver's License number. For instance, in Illinois my Driver's Licence begins as V555.

This technique can also be extended to crypto work, to provide a somewhat different method of pattern searching. If you have a tryout word that almost fits, you could convert the tryout word to Soundex, and do a search for other words that match the Soundex code.

This article will present an implementation of the Soundex algorithm in both C and BASIC. These programs are available on on Issue Disk 16.

The algorithm to generate the codes:

1. Retain the first letter of the name, and drop all occurrences of a,e,h,i,o,u,w,y in other positions.

2. Assign the following numbers to the remaining letters after the first

```
b,f,p,v -> 1
c,g,j,k,q,s,x,z -> 2
d,t -> 3
l -> 4
m,n -> 5
r -> 6
```

3. If two or more letters with the same code were adjacent in the original word (before step 1), omit all but the first.

4. Convert to the form "letter, digit, digit, digit" by adding trailing zeros (if there are less than three digits), or by dropping rightmost digits (if there are more than three).

SOUNDEX.BAS

```
1000 ' SOUNDEX.BAS
1010 ' Generate soundex values for given names
1020 '
1030 ' References:
1040 '  Knuth, Donald  The Art of Computer Programming:
1050 '                       Searching and Sorting, pp 391-392
1060 '
1070 ' Originally developed by Margaret K. Odell and Robert C. Russell
1080 '  US Patents 1261167 (1918), 1435663 (1922)
1090 '
1100 '  1.  Retain the first letter of the name, and drop all occurrences
1110 '      of a,e,h,i,o,u,w,y in other positions.
1120 '
1130 '  2.  Assign the following numbers to the remaining letters after
1140 '      the first:
1150 '        b,f,p,v -> 1                      l -> 4
1160 '        c,g,j,k,q,s,x,z -> 2              m,n -> 5
1170 '        d,t -> 3                          r -> 6
1180 '
1190 '  3.  If two or more letters with the same code were adjacent in the
1200 '      original name (before step 1), omit all but the first.
1210 '
1220 '  4.  Convert to the form "letter, digit, digit, digit" by adding
1230 '      trailing zeroes (if there are less than three digits), or
1240 '      by dropping rightmost digits (if there are more than three).
1250 '
1260 '  Euler = E460   Gauss = G200  Hilbert = H416  Knuth = K530
1270 '  Lloyd = L300   Lukasiewicz = L222
1280 '
1290 '
1300 '  Variables:   N$   name as entered
1310 '               W$   working string
1320 '               R$   result
1330 '
1340 T$ = "01230120022455012623010202"
1350 '   abcdefghijklmnopqrstuvwxyz
1360 '
1370 PRINT "Enter name ";
1380 INPUT N$
1390 ' Uppercase all the letters in the name
1400 W$ = ""
1410 FOR P = 1 TO LEN(N$)
1420 V = ASC(MID$(N$,P,1)) AND 223
1430 I = V - ASC("A") + 1    ' Convert letter to an offset (index into T$)
1440 ' If the index doesn't point a letter (1 to 26), skip it
1450 IF (I < 1) OR (I > 26) THEN 1490
```

```
1460 ' Put the code into the result
1470 W$ = W$ + MID$(T$,I,1)
1480 '
1490 NEXT P
1500 '
1510 PRINT "Work string is ";W$
1520 '
1530 ' Result begins with the first letter of the name
1540 R$ = CHR$(ASC(LEFT$(N$,1)) AND 223)
1550 FOR P = 2 TO LEN(W$)
1560 ' If the digit is a zero, this is a letter to skip
1570 IF MID$(W$,P,1) = "0" THEN 1620
1580 ' If this digit is the same as the previous one, skip it
1590 IF MID$(W$,P,1) = MID$(W$,P-1,1) THEN 1620
1600 ' Put the code into the result
1610 R$ = R$ + MID$(W$,P,1)
1620 NEXT P
1630 '
1640 ' pad with zeros and cut to the correct length
1650 R$ = R$ + "0000"
1660 R$ = LEFT$(R$,4)
1670 PRINT N$;" gives a code of ";R$
1680 END
```

SOUNDEX.C

```
/* SOUNDEX.C
** Generate soundex values for given names
**
** References:
**  Knuth, Donald  The Art of Computer Programming: Searching and Sorting
**    pp 391-392
**
** Originally developed by Margaret K. Odell and Robert C. Russell
**  US Patents 1261167 (1918), 1435663 (1922)
**
**  1.  Retain the first letter of the name, and drop all occurrences
**      of a,e,h,i,o,u,w,y in other positions.
**
**  2.  Assign the following numbers to the remaining letters after
**      the first:
**        b,f,p,v -> 1                      l -> 4
**        c,g,j,k,q,s,x,z -> 2              m,n -> 5
```

```
**       d,t -> 3                          r -> 6
**
**  3.  If two or more letters with the same code were adjacent in the
**      original name (before step 1), omit all but the first.
**
**  4.  Convert to the form "letter, digit, digit, digit" by adding
**      trailing zeroes (if there are less than three digits), or
**      by dropping rightmost digits (if there are more than three).
**
**  Euler = E460   Gauss = G200  Hilbert = H416  Knuth = K530
**  Lloyd = L300   Lukasiewicz = L222
*/


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

#define STRING_SIZE  40
#define SOUNDEX_SIZE 4

const char table[] = "01230120022455012623010202";
/*                    abcdefghijklmnopqrstuvwxyz  */

void  soundex( char *result, char *source, int length )
{
  int  pos, index, result_length;
  char  name[STRING_SIZE + 1];
  char  working[STRING_SIZE + 1];

  /* Uppercase all the letters in the name */

  for (pos = 0; pos < strlen(source); pos++)
    name[pos] = toupper(source[pos]);

  name[strlen(source)] = '\0';    /* End of string marker */

  result_length = 0;

  for ( pos = 0; pos < strlen(name); pos++ )
  {
    /* Convert letter to an offset */
    index = name[pos] - 'A';

    /* If index points to a letter replace the letter
      with the corresponding table entry */

    if ((index >= 0) && (index <= 25))
```

```
      working[ result_length++ ] = table[ index ];
  }

  working[ result_length ] = '\0';  /* Place an end of string marker */

  printf("Working string is %s\n", working);

  /* result starts with the first letter of the name */
  result[0] = name[0];

  /* Transfer code to result if it's not zero or a repeat */

  for (pos = 1, result_length = 1;
    (pos < strlen(working)) && (result_length < length); pos++)
    if (( working[pos] != '0' ) && (working[pos] != working[pos-1]))
      result[ result_length++ ] = working[pos];

  /* Pad the result with zeros */
  while ( result_length < length )
    result[ result_length++ ] = '0';

  /* Trim the result to the required length */
  result[length] = '\0';    /* Place end of string marker */

}


main( int argc, char *argv[] )
{
  char  name[STRING_SIZE + 1], soundex_code[STRING_SIZE + 1];

  printf("Enter name: ");
   gets( name );

  soundex( soundex_code , name , SOUNDEX_SIZE );

  printf("%s gives a code of %s\n", name, soundex_code );

}
```

THREE ROTOR ENCRYPTION SYSTEM

MONIOC

This program is based on the Japanese Purple code described by David Kahn in *The CodeBreakers*. (As an aside, I just about inhaled this book when I read it for the first time !). In his description the machine used five rotary discs and I thought three would be enough to prove the point. The number of matrices can be increased easily by repeating and adding to lines 400–600. (The entire program was written in GW-BASIC on an AT clone.) Another modification was to extend the number of characters to 81. This was needed to make the 9x9 array which I used.

Some observations of my own on the performance of this cipher. I find that the CODing and DECoding works well without error if the DECode characters are entered in the right sequence. However, if a character is incorrect the remainder of the text is garbled. I have tried to find periodicity within the cipher text but I am sure that it exists and would provide a penetration point for deciphering. If the message always starts with the same character group the cipher will show the same character string.

One way to prevent this is to include new values for B1, B2 and B3 at some random point within the message so that they may be used in the succeeding message(s). This can be done at each message and effectively produce a "one-time pad."

Of course, the program has significant weaknesses. That is if the program is recovered with the attendant matrices shown then all bets are off.

The following is a description of the program functions which can be helpful in tracking the execution as it progresses:

```
5-30            Initialization of variables
40              Sets the sequence of change for each matrix
60-120          Operator inputs.  If NEW MATRICES is selected the
                present cipher cannot be recovered.  It should
                only be used when initiated at the start of a
                sequence of messages or when the cipher has been
                compromised.
200-290         Message entry.  Up to 512 characters can be
                entered for the version.  It can be extended if
                necessary.
300-650         Sets up the matrix search for each character and
                indexes each change in each matrix before
                searching for the next character in the succeeding
                matrix.  This is the attempt to rotate the discs
                as in the Purple machine (see line 40).
650-700         Print instructions to the screen.

5000-5100       This routine insures that if the change sequence
```

goes past the matrix limit it is returned to that matrix. This is to imitate a wheel used in some enciphering devices.

6000-6300 These are to get the right matrix in order to search for the next character.

7000-7200 These lines are used to retrieve the character set in order to generate the matrices.

7300-8400 These lines generate the three matrices used in this program. The three matrices are stored on disc for use.

9000-9100 This sequence is the heart of the ciphering technique. The shuffling of the characters is based on a pseudo-random generator within the computer. It is not really necessary to have a true random generator as long as the matrices are sufficiently in structure.

9500-9840 This code is used to retrieve the matrix data and send it to the printer.

COD-DEC6.BAS

```
5 REM COD/DEC 6
10 DIM E$(85),F$(85),W$(10)
20 DIM Z$(10),U$(10),V$(10),S$(81)
30 HH=0:V=0:DIM LL$(300):WW$=CHR$(32)
40 B1=3:B2=-5:B3=4
50 CLS:LOCATE 10,25: INPUT "COD OR DEC"  ;H$
60 IF H$="COD" THEN K1=1 ELSE K1=-1
80 CLS:LOCATE 10,25: INPUT "PRINT MATRICES" ;QP$
90 IF QP$="Y" THEN 9500 ELSE 110
110 CLS:LOCATE 10,25: INPUT "NEW MATRICES" ;Q$
120 IF Q$="Y" THEN GOSUB 7000 ELSE 140
140 CLS:GOSUB 6000
200 LOCATE 5,10: INPUT OO$
202 IF LEN(OO$)>254 THEN INPUT KK$
206 JJ$=OO$
208 LPRINT TAB(12);OO$+KK$:LPRINT:LPRINT
218 FOR CC= 1 TO LEN(JJ$+KK$)
220 J$=MID$(JJ$,CC,1)
290 IF LEN(J$)>1 THEN 200
300 IF H$="COD" THEN FOR X=1 TO 3:ON X GOSUB 400,500,600:NEXT X
320 IF H$="DEC" THEN FOR X=1 TO 3:ON X GOSUB 600,500,400:NEXT X
```

```
340 GOTO 650
400 FOR I=1 TO 9:FOR J=1 TO 9
410 IF H$="COD" AND MID$(W$(I),J,1)=J$ THEN 430
415 IF H$="DEC" AND MID$(W$(I),J,1)=N$ THEN 430
420 NEXT J,I
430 I=I+B1*K1:J=J+B1*K1:IF (I>9 OR J>9) THEN GOSUB 5000
431 REM LOCATE 5,25:PRINT"I=";I;" ";"J=";J;"B1=";B1;" ";"431"
435 IF(I<=0 OR J<=0) THEN GOSUB 5000
440 Z$= MID$(W$(I),J,1)
450 B1=B1+1:IF B1>6 THEN B1=1
460 RETURN
500 FOR I=1 TO 9:FOR J=1 TO 9
510 IF H$="COD" AND MID$(U$(I),J,1)=Z$ THEN 530
515 IF H$="DEC" AND MID$(U$(I),J,1)=RR$ THEN 530
520 NEXT J,I
530 I=I+B2*K1:J=J+B2*K1:IF (I>9 OR J>9) THEN GOSUB 5000
535 IF (I<1 OR J<1) THEN GOSUB 5000
540 N$= MID$(U$(I),J,1)
542 B2=B2-1:IF B2<1 THEN B2=5
560 RETURN
600 FOR I=1 TO 9:FOR J=1 TO 9
610 IF H$="COD" AND MID$(V$(I),J,1)=N$ THEN 630
615 IF H$="DEC" AND MID$(V$(I),J,1)=J$ THEN 630
620 NEXT J,I
630 I=I+B3*K1:J=J+B3*K1:IF (I>9 OR J>9) THEN GOSUB 5000
635 IF (I<1 OR J<1) THEN GOSUB 5000
640 RR$= MID$(V$(I),J,1)
642 B3=B3+1:IF B3>5 THEN B3=2
645 RETURN
650 IF H$="COD" AND INT(HH/6)=HH/6 THEN HH=HH+1
660 IF H$="COD" THEN LOCATE(10+VV),(10+HH):PRINT RR$;
662 IF H$="COD" THEN LL$=RR$
665 IF H$="DEC" THEN LL$=Z$
670 IF H$="DEC" THEN LOCATE(10+VV),(10+HH):PRINT Z$;
675 IF HH>=50 THEN VV=VV+2
677 IF HH>=50 THEN HH=0
678 IF VV=24 THEN VV=10
680 HH=HH+1
682 IF H$="COD" THEN LPRINT TAB(10+HH);LL$;
685 IF H$="DEC" THEN LPRINT TAB(10+HH);LL$;
690 NEXT CC
700 END

5000 REM WRAP-AROUND ROUTINE
5020 IF J>9 THEN J=J-9
5040 IF J<0 THEN J=J+9
5045 IF J=0 THEN J=J+9
5060 IF I>9 THEN I=I-9
5080 IF I<0 THEN I=I+9
5085 IF I=0 THEN I=I+9
5100 RETURN
```

```
6000 REM MATRIX1A
6010 T$="MATRIX1A"
6020 OPEN "R",#1,T$,9
6030 FIELD #1,9 AS B$
6040 FOR X=1 TO 9
6050 CODE%=X
6060 GET #1,CODE%
6070 REM A=CVS(D$)
6080 W$(X)=B$
6090 NEXT X: CLOSE
6100 REM MATRIX2A
6110 T$="MATRIX2A"
6120 OPEN "R",#1,T$,9
6130 FIELD #1,9 AS B$
6140 FOR X=1 TO 9
6150 CODE%=X
6160 GET #1,CODE%
6170 REM A=CVS(D$)
6180 U$(X)=B$
6190 NEXT X : CLOSE
6200 REM MATRIX3A
6210 T$="MATRIX3A"
6220 OPEN "R",#1,T$,9
6230 FIELD #1,9 AS B$
6240 FOR X=1 TO 9
6250 CODE%=X
6260 GET #1,CODE%
6270 REM A=CVS(D$)
6280 V$(X)=B$
6290 NEXT X : CLOSE
6300 RETURN
7000 REM MATRIX GEN SHORT
7020 T$="CHARSET"
7040 OPEN "R",#1,T$,1
7050 FIELD #1,1 AS B$
7060 FOR X=1 TO 81
7065 CODE%=X
7070 GET #1,CODE%
7080 REM A=CVS(D$)
7090 E$(X)=B$
7100 NEXT X: CLOSE
7140 FOR X=1 TO 81
7170 IF E$(X)=" " THEN E$(X)=CHR$(32)
7180 F$(X)=E$(X)
7190 PRINT F$(X);
7195 NEXT X
7200 REM MATRIX GEN
7280 GOSUB 9000
7300 I=1:W$=""
7310 FOR J=1 TO 81
7320 W$(I)=W$(I)+F$(J)
7330 IF INT(J/9)=J/9 THEN I=I+1
7340 IF I=10 THEN 7360
7350 NEXT J
7360 FOR X=1 TO 9:PRINT W$(X),:NEXT X
7440 REM MATRIX1A
7460 T$="MATRIX1A"
7480 OPEN "R",#1,T$,9
7500 FIELD #1,9 AS B$
7520 FOR X=1 TO 9
7540 CODE%=X
7560 LSET B$=W$(X)
7580 REM LSET D$=MKS$(X)
7590 PUT #1,CODE%
7600 NEXT X
7610 CLOSE
7620 REM MATRIX GEN
7622 FOR X=1 TO 9:U$(X)="":NEXT X
7625 GOSUB 9000
7630 I=1:U$=""
7640 FOR J=1 TO 81
7650 U$(I)=U$(I)+F$(J)
7660 IF INT(J/9)=J/9 THEN I=I+1
7670 IF I=10 THEN 7690
7680 NEXT J
7690 PRINT:PRINT
7695 FOR X=1 TO 9:PRINT U$(X),:NEXT X
7800 REM MATRIX2A
7820 T$="MATRIX2A"
7840 OPEN "R",#1,T$,9
7860 FIELD #1,9 AS B$
7880 FOR X=1 TO 9
7900 CODE%=X
7910 LSET B$=U$(X)
7920 REM LSET D$=MKS$(X)
7940 PUT #1,CODE%
7960 NEXT X
7980 CLOSE
8000 REM MATRIX GEN
8010 I=1:V$=""
8012 FOR X=1 TO 9:V$(X)="":NEXT X
8020 GOSUB 9000
8030 FOR J=1 TO 81
8040 V$(I)=V$(I)+ F$(J)
```

```
8050 IF INT(J/9)=J/9 THEN I=I+1
8060 IF I=10 THEN 8080
8070 NEXT J
8080 FOR X=1 TO 9:PRINT V$(X),:NEXT X
8180 REM MATRIX 3A
8200 T$="MATRIX3A"
8220 OPEN "R",#1,T$,9
8240 FIELD #1,9 AS B$
8260 FOR X=1 TO 9
8280 CODE%=X
8300 LSET B$=V$(X)
8320 REM LSET D$=MKS$(X)
8340 PUT #1,CODE%
8350 NEXT X
8360 CLOSE
8370 STOP
9000 REM SHUFFLE ROUTINE
9010 N=1
9015 RANDOMIZE TIMER
9020 FOR X=N TO 81
9030 REM Y=INT(RND*(81))+1
9034 Y=INT(RND*81+1)
9036 SWAP F$(Y),F$(N)
9070 N=N+1:IF N>81 THEN N=81
9080 NEXT X
9081 PRINT
9084 FOR X=1 TO 81
9086 PRINT F$(X);
9088 NEXT X
9090 RETURN
9100 REM LEN=4215 BYTES
9110 STOP
9500 REM READ MATRIX1A
9510 T$="MATRIX1A"
9520 OPEN "R",#1,T$,9
9530 FIELD #1,9 AS B$
9540 FOR X=1 TO 9
9550 CODE%=X
9555 GET #1,CODE%
9565 W$(X)=B$
9570 NEXT X
9580 CLOSE
9600 REM READ MATRIX2A
9610 T$="MATRIX2A"
9620 OPEN "R",#1,T$,9
9630 FIELD #1,9 AS B$
9640 FOR X=1 TO 9
9650 CODE%=X
9655 GET #1,CODE%
9665 U$(X)=B$
9670 NEXT X
9680 CLOSE
9700 REM READ MATRIX 3A
9710 T$="MATRIX3A"
9720 OPEN "R",#1,T$,9
9730 FIELD #1,9 AS B$
9740 FOR X= 1 TO 9
9750 CODE%=X
9755 GET #1,CODE%
9765 V$(X)=B$
9770 NEXT X:CLOSE
9780 CLS
9790 FOR X=1 TO 9
9800 LOCATE (X+10),10:LPRINT W$(X),
9810 LOCATE (X+10),22:LPRINT U$(X),
9820 LOCATE (X+10),34:LPRINT V$(X)
9830 NEXT X
9840 END
```

CHARGEN.BAS

```
5 REM DEFINE CHAR SET FOR MATRICES
20 DIM E$(81)
30 T$="CHARSET"
40 FOR X=1 TO 81
60 CLS
100 LOCATE 8,15
120 INPUT "ENTER A CHARACTER";O$
140 E$(X)=O$
180 NEXT X
300 OPEN "R",#1,T$,6
320 FIELD #1,6 AS B$
340 FOR X=1 TO 81
360 CODE%=X
380 LSET B$=E$(X)
400 REM LSET D$=MKS$(X)
420 PUT #1,CODE%
440 NEXT X
460 CLOSE
500 FOR X=1 TO 81
520 PRINT E$(X);" ";
540 NEXT X
560 REM END
580 T$="CHARGEN"
600 OPEN "R",#1,T$,6
610 FIELD #1,6 AS B$
620 FOR X=1 TO 81
630 CODE%=X
640 GET #1,CODE%
650 PRINT B$;" ";
655 NEXT X
660 CLOSE
670 END
```

CHARGEN1.BAS

```
5 REM DEFINE CHAR SET FOR MATRICES
20 DIM E$(81)
30 T$="CHARSET1"
40 FOR X=1 TO 26
60 CLS
100 LOCATE 8,15
120 INPUT "ENTER A CHARACTER";O$
140 E$(X)=O$
180 NEXT X
300 OPEN "R",#1,T$,1
320 FIELD #1,1 AS B$
340 FOR X=1 TO 26
360 CODE%=X
380 LSET B$=E$(X)
400 REM LSET D$=MKS$(X)
420 PUT #1,CODE%
440 NEXT X
460 CLOSE
500 FOR X=1 TO 26
520 PRINT E$(X);" ";
540 NEXT X
560 REM END
580 T$="CHARSET1"
600 OPEN "R",#1,T$,1
610 FIELD #1,1 AS B$
620 FOR X=1 TO 81
630 CODE%=X
640 GET #1,CODE%
650 PRINT B$;" ";
655 NEXT X
660 CLOSE
670 END
```

CHARSET.BAS

```
5 REM DEFINE CHAR SET FOR MATRICES
20 DIM E$(81)
30 T$="CHARSET"
40 FOR X=1 TO 81
60 CLS
100 LOCATE 8,15
120 INPUT "ENTER A CHARACTER";O$
140 E$(X)=O$
180 NEXT X
300 OPEN "R",#1,T$,1
320 FIELD #1,1 AS B$
340 FOR X=1 TO 81
360 CODE%=X
380 LSET B$=E$(X)
400 REM LSET D$=MKS$(X)
420 PUT #1,CODE%
440 NEXT X
460 CLOSE
500 FOR X=1 TO 81
520 PRINT E$(X),
540 NEXT X
560 END
570 DIM E$(81)
580 T$="CHARSET"
600 OPEN "R",#1,T$,1
610 FIELD #1,1 AS B$
620 FOR X=1 TO 81
630 CODE%=X
640 GET #1,CODE%
650 REM A=CVS(D$)
660 E$(X)=B$
670 PRINT E$(X);
680 NEXT X
690 CLOSE
```

RDMATRX.BAS

```
5 REM READ MATRIX
6 DIM A$(9)
10 PRINT "WHICH MATRIX DO YOU WANT?"
20 INPUT T$
30 OPEN "R",#1,T$,9
40 FIELD #1,9 AS B$
50 FOR X=1 TO 9
60 CODE%=X
70 GET #1,CODE%
80 A$(X)=B$
90 NEXT X
100 CLOSE
110 FOR X=1 TO 9:PRINT A$(X),:NEXT X
```

WHAT THE OTHER GUY IS DOING

Tom Martin is looking for a program that will automatically break Aristocrats. Anyone have one that's fairly well documented ?

John Eubanks is a Commodore C64 enthusiast, and is looking for others of the Krewe that share the same interest and would be willing to exchange programs and ideas.

TATTERS (**Caxton Foster**) writes: "... does anyone in the Krewe have any references to Change Ringing ? I'll gladly pay for photocopies since our library is very proud of the fact that they keep in-depth back copies of the *New York Times* — all the way back to last week. I have Wilson's *Change Ringing* but would welcome any other."

TEOWOC (**Gary Thomas**) uses Hewlett-Packard's HP–UX at work on a Series 300, and writes in 80C196 assembly language. At home he uses a Macintosh SE and writes in SPITBOL.

Richard Kummer is looking for more information about vowel searching. He's familiar with the entry in the January 1986 *Cryptologia*, but is trying to get some BASIC programs in electronic format. **PHOENIX** has helped out with an offer to put his programs on disk; anyone else have something that could help ?

KARL (**Waldo Boyd**) is working in QuickBASIC, and thought it might be interesting to see a comparison of the current crop of BASIC programming tools and products. Any volunteers ?

A DES DONGLE

The GL306060 Hardlock–DES parallel port adapter incorporates an application–specific IC that encrypts and decrypts data using the DES algorithm. The GlenDES chip features nonvolatile 8–byte internal EEPROM for storing the DES key.

The Hardlock–DES, which connects directly to your PC's printer port, is transparent to normal printer operation. The unit's key–management features support private and public key modes.

Price: $149.
Contact: Glenco Engineering, Inc.
270 Lexington Drive
Buffalo Grove, Illinois 60089
(708) 808–0300
fax (708) 808–0313

LETTER COUNTING

One popular method for analyzing ciphers is to generate a frequency count of letters in the text. A frequency count answers the question "How many times does a certain letter appear in the text ?" Frequency counts are one of the basic computations performed by cryptanalysts. This article will describe a tool to produce simple letter counts.

In any language certain letters are used more often than others. In English, for example, it is far more common to see the letters E, T, and N than say X, Q, and Z. Kahn[1] gives the following percentages for likelihood of appearance for letters in the English language:

```
A    B    C    D    E    F    G
8   1.5   3    4    13   2   1.5

H    I    J    K    L    M
6   6.5  0.5  0.5  3.5   3

N    O    P    Q    R    S    T
7    8    2   0.25 6.5   6    9

U    V    W    X    Y    Z
3    1   1.5  0.5   2   0.25
```

This property often gives the cryptanalyst a head–start in solving a cipher.

Gaines[2] suggests an additional metric for cracking substitution ciphers, which she calls Variety of Contact. It is essentially a count of letters that are found next to other letters. For instance, in the text "ENCODE THE WORD" a frequency count would give us the following information:

```
E: 3   N: 1   C: 1

O: 2   D: 2   T: 1

H: 1   W: 1   R: 1
```

and a Variety of Contact would tell us:

```
LETTER  FOUND NEXT TO
------  -------------
  E       N, D, H.
  N       E, C.
  C       N, O.
  O       C, D, W, R.
  D       O, E, R.
  T       H.
  etc.
```

In tabular format it would look like this:

```
C (1):  N1 O1
D (2):  E1 O1 R1
E (3):  D1 H1 N1
H (1):  E1 T1
N (1):  C1 E1
O (2):  C1 D1 R1 W1
R (1):  D1 O1
T (1):  H1
W (1):  O1
```

The cryptanalyst would use this chart to look for patterns of pairs that would decrypt into common common digraphs (TH, AN, NE, etc.). See Gaines for an explanation and example of this technique.

Presented below is a BASIC program to compute frequency and variety of contact, and a C language program with options for sorting the results. [Editor's Note: Both of these programs are available on Issue Disk

16. DMV] I hope these programs will serve as a starting point for further discussion of statistical properties of ciphers.

References

[1] Kahn, David; *The CodeBreakers*, 1967; Macmillan Publishing, New York

[2] Gaines, Helen Fouches; *Cryptanalysis*, 1956; Dover Publications, New York

---

FQCT.BAS

```
1000 ' FREQCONT.BAS
1010 ' Report frequency count and variety of contact
1020 '
1030 A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
1040 AL = LEN(A$)
1050 DIM F(26), C(26,26,2)
1060 '
1070 ' Zero the counters
1080 '
1090 FOR I = 1 TO AL
1100 F(I) = 0
1110 FOR J = 1 TO AL
1120 C(I,J,1) = 0
1130 C(I,J,2) = 0
1140 NEXT J
1150 NEXT I
1160 '
1170 ' Open the input file
1180 '
1190 PRINT "Filename";
1200 INPUT F$
1210 OPEN F$ FOR INPUT AS #1
1220 '
1230 IF EOF(1) THEN 1610
1240 ' read in the next line
1250 INPUT #1, B$
1260 ' set the last character holder to SPACE
1270 LC$=" "
1280 '
1290 FOR L = 1 TO LEN(B$)
1300 '
1310 ' Get the next character and make it uppercase, if necessary
1320 '
1330 C = ASC(MID$(B$, L, 1))
```

```
1340 IF C >= ASC("a") AND C <= ASC("z") THEN C = C - 32
1350 C$ = CHR$(C)
1360 P = INSTR(A$, C$)
1370 IF P = 0 THEN 1550
1380 '
1390 ' character C$ appears in the alphabet list, so count it
1400 '
1410 F(P) = F(P) + 1
1420 '
1430 ' check if the last character is also in the alphabet list
1440 '
1450 LP = INSTR(A$, LC$)
1460 IF LP = 0 THEN 1550
1470 '
1480 ' It is, so increment the contact pointers for each letter
1490 '
1500 C(P,LP,1) = C(P,LP,1) + 1
1510 C(LP,P,2) = C(LP,P,2) + 1
1520 '
1530 ' set last character to this character for the next pass
1540 '
1550 LC$ = C$
1560 NEXT L
1570 GOTO 1230
1580 '
1590 ' Input file is exhausted, so close it and report results
1600 '
1610 CLOSE #1
1620 FOR I = 1 TO AL
1630 IF F(I) = 0 THEN 1960
1640 '
1650 ' report letter and frequency
1660 '
1670 N$=STR$(F(I))
1680 N$=MID$(N$,2,LEN(N$)-1)
1690 PRINT MID$(A$,I,1);" (";N$;"): ";
1700 '
1710 ' Check for contact counts
1720 '
1730 FOR J = 1 TO AL
1740 IF C(I,J,1) + C(I,J,2) = 0 THEN 1940
1750 '
1760 PRINT "  ";
1770 '
1780 ' contacts before letter
1790 '
1800 IF C(I,J,2) = 0 THEN 1850
```

```
1810 N$=STR$(C(I,J,2))
1820 N$=MID$(N$,2,LEN(N$)-1)
1830 PRINT N$;
1840 ’
1850 PRINT MID$(A$,J,1);
1860 ’
1870 ’ contacts after letter
1880 ’
1890 IF C(I,J,1) = 0 THEN 1940
1900 N$=STR$(C(I,J,1))
1910 N$=MID$(N$,2,LEN(N$)-1)
1920 PRINT N$;
1930 ’
1940 NEXT J
1950 PRINT
1960 NEXT I
1970 ’
1980 END
```

FREQCONT.C

```
/* FREQCONT.C
** Reports frequency count and Variety of Contact
**
**  Uses ANSI-style function prototyping
*/

#include <stdio.h>

#define FALSE  0
#define TRUE  1

/* tags for position-sensitive contact */
#define BEFORE  0
#define AFTER  1

/* tags for type of report sorting */
#define ALPHABETICAL  0
#define FREQUENCY  1
#define CONTACT    2
```

```
/* length and characters recognized */
#define ALPHABET_LENGTH 26

char  alphabet[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

#define SPACE  ((char) 32)

#define CARRIAGE_WIDTH  70
#define INTERVAL_SPACE  "  "

/* globals for simple print formatting */

int  carriage_position;
int  initial_tab;


/* output routines to format reports */

void  output_header( char *h )
{
  printf(h);
  initial_tab = strlen(h);
  carriage_position = initial_tab;
}


void  output_char( char c )
{
  printf("%c", c);
  carriage_position++;
}

void  output_string( char *s )
{
  int  i;
  char  text[40];

  for (i = 0; i < strlen(s); i++ )
    output_char( s[i] );

  if (carriage_position > CARRIAGE_WIDTH)
    if (!strcmp(s, INTERVAL_SPACE))
    {
      printf("\n");
      for (i = 0; i < initial_tab; i++)
        text[i] = SPACE;
      text[i] = '\0';
```

```
      output_header(text);
    }
}


void  output_number( int n )
{
  char  text[10];

  sprintf(text, "%d", n );
  output_string( text );
}


/* Quick and dirty routine to return position of character in a string */

int  strInPos( char *source, char c )
{
  int  pos;

  pos = 0;
  while( source[pos] )
  {
    if (source[pos] == c)
      return(pos);

    pos++;
  }

  return( -1 );
}



main(int argc, char *argv[])
{
  FILE  *fp_in;

  int  frequency[ ALPHABET_LENGTH ];
  int  contact[ ALPHABET_LENGTH][ALPHABET_LENGTH ][2];
  int  value[ ALPHABET_LENGTH ], index[ ALPHABET_LENGTH ];

  int  i, j;
  int  index_loop;
  int  count, back;
  int  contacts_before, contacts_after, contacts_total;
```

```
int  carriage_position, initial_tab;

char  buffer[2], text[20], previous_character;
int  report_position;
int  sort_by;
char  *filename;


/* default options */

report_position = FALSE;
sort_by = ALPHABETICAL;
filename = (char *) NULL;


/* search argument list for input filename and options */

for ( i = 1; i < argc; i++ )
  if ((argv[i][0] == '/') || (argv[i][0] == '-'))
  {
    switch( toupper(argv[i][1]) ) {
    case 'C':
      sort_by = CONTACT;
      break;

    case 'F':
      sort_by = FREQUENCY;
      break;

    case 'P':
      report_position = TRUE;
      break;

    default:
      printf("Ignoring unknown option %c.\n",
        toupper(argv[i][1]));
      break;
    }
  }
  else
    filename = argv[i];


/* If there was no filename on the command line, print help */

if (!filename)
{
```

```
  puts("Usage: FREQCONT [options] input_file");
  puts("         input_file  is ASCII text file to count");
  puts("         Options:");
  puts("           -p  report contact position as nXm");
  puts("                 where X = contacted letter");
  puts("                       n = letter count before");
  puts("                       m = letter count after");
  puts("           -c  sort report by variety of contact");
  puts("           -f  sort report by letter frequency");
  exit(-1);
}


/* Open the input file */

if ((fp_in = fopen( filename , "rb")) == NULL)
{
  printf("Can't open %s\n", filename );
  exit(-1);
}


/* Zero the counters */

for (i = 0; i < ALPHABET_LENGTH; i++)
{
  frequency[i] = 0;
  for (j = 0; j < ALPHABET_LENGTH; j++)
  {
    contact[i][j][BEFORE] = 0;
    contact[i][j][AFTER] = 0;
  }
}


/* Read each character in the file and count it properly */

previous_character = SPACE;
while(1)
{
  /* read one character */
  count = fread( buffer, 1, 1, fp_in );


  /* exit the loop if there aren't any left */
  if (!count)
    break;
```

```
  /* make the character uppercase */
  buffer[0] = toupper(buffer[0]);

  /* Is it a recognized character ? */
  i = strInPos( alphabet , buffer[0] );
  if (i > -1)
  {
    /* count it */
    frequency[i]++;

    /* count contacts */
    back = strInPos( alphabet , previous_character );
    if (back > -1)
    {
      contact[ i ][ back ][BEFORE]++;
      contact[ back ][ i ][AFTER]++;
    }
  }

  previous_character = buffer[0];
}

/* We're done with the file, so close it up */

fclose(fp_in);


/* Set up the value array for use by the sorting routine */

switch( sort_by ) {
case ALPHABETICAL:
  puts("ALPHABETICAL");
  for (i = 0; i < ALPHABET_LENGTH; i++)
    value[i] = ALPHABET_LENGTH - i;
  break;

case CONTACT:
  puts("CONTACT");
  for (i = 0; i < ALPHABET_LENGTH; i++)
  {
    value[i] = 0;
    for (j = 0; j < ALPHABET_LENGTH; j++)
    {
      value[i] += contact[i][j][BEFORE];
      value[i] += contact[i][j][AFTER];
    }
```

```
    }
    break;

  case FREQUENCY:
    puts("FREQUENCY");
    for (i = 0; i < ALPHABET_LENGTH; i++)
      value[i] = frequency[i];
    break;
  }


  /* Generic search for highest values */

  for (i = 0; i < ALPHABET_LENGTH; i++)
  {
    index[i] = 0;
    for (j = 0; j < ALPHABET_LENGTH; j++)
    {
      if (value[index[i]] < value[j])
        index[i] = j;
    }
    value[index[i]] = -1;
  }


  /* Report results */

  for (index_loop = 0; index_loop < ALPHABET_LENGTH; index_loop++)
  {
    i = index[index_loop];

    if (frequency[i])
    {
      contacts_total = 0;
      contacts_before = 0;
      contacts_after = 0;

      sprintf(text, "%c (%d):", alphabet[i], frequency[i]);
      output_header( text );

      for (j = 0; j < ALPHABET_LENGTH; j++)
        if ((contact[i][j][BEFORE]) ||
          (contact[i][j][AFTER]))
        {
          contacts_total++;
          output_string(INTERVAL_SPACE);
```

```
          if (report_position == TRUE)
          {
            if (contact[i][j][AFTER])
            {
              contacts_after++;
              output_number(contact[i][j][AFTER]);
            }
            output_char(alphabet[j]);
            if (contact[i][j][BEFORE])
            {
              contacts_before++;
              output_number(contact[i][j][BEFORE]);
            }
          }
          else
          {
            output_number( contact[i][j][BEFORE] + contact[i][j][AFTER] );
            output_char( alphabet[j] );
          }
        }

      if (report_position == TRUE)
      {
        sprintf(text, "  [%d %d]", contacts_after,
          contacts_before);

        output_string( text );
      }

      sprintf(text, "  [%d]\n", contacts_total );
      output_string( text );
    }
  }
}
```

FASTER PERMUTATION ROUTINE

TATTERS

This program permutes the numbers in cells C(1) ... C(N). The "Initialize" subroutine sets up the numbers 1 through N in those cells. Each time the Generate subroutine is called it returns the "alphabetically next" permutation in the C's.

In lines 210–270 the algorithm begins with cell N and searches backwards toward cell 1 looking for a "down step" — that is, a cell K such that the contents of cell K is less than the contents of cell $K+1$. In 310–360 it marks off as busy the numbers in cells 1 through $K-1$. Then in 370–410 it finds the smallest non-busy number larger than the current contents of C(K).

Lastly in 430–470 it puts the remaining non-busy numbers into the cells $K+1$ through N, in ascending order.

Time to generate a new permutation is order N as opposed to at least order N^2 for PERMUTE.BAS.

```
10 REM MAIN PROG
20 CLS
30 INPUT "N=";N
40 FOR I=1 TO N
42 PRINT I;
44 NEXT I
46 PRINT
50 GOSUB 150
60 GOSUB 210
70 IF FG=1 THEN 130
80 FOR I=1 TO N
90 PRINT C(I);
100 NEXT I
110 PRINT
120 GOTO 60
130 PRINT "ALL DONE"
140 END
150 REM INITIALIZE
160 DIM C(20)
170 FOR I=1 TO N
180 C(I)=I
190 NEXT I
200 RETURN
210 REM GENERATE NEW PERMUTATION
220 PV=0
230 K=N
240 IF C(K) < PV THEN 300
250 PV=C(K)
260 K=K-1
270 IF K>0 THEN 240
280 FG=1
290 RETURN
300 REM FILL IN
310 FOR I=1 TO N
320 D(I)=0
330 NEXT I
340 FOR I=1 TO K-1
350 D(C(I))=1
360 NEXT I
370 FOR I=C(K)+1 TO N
380 IF D(I)=0 THEN F1=I:I=N
390 NEXT I
400 D(F1)=1
410 C(K)=F1
420 K=K+1
430 FOR I=1 TO N
440 IF D(I)=1 THEN 470
450 C(K)=I
460 K=K+1
470 NEXT I
480 RETURN
```

THE PUBLIC KEY

Volume 1 Issue 3 of *The Public Key* is available, containing information and programs relating to Public Key cryptography. This publication from the United Kingdom discusses DES security, digital signature standards, and other modern systems. There are also articles concerning electronic funds and automated teller machine fraud, and concludes with a message about cryptographic legislation in the United States.

The editors work very hard to describe relatively complicated processes in a clear and concise manner. It is well worth a copy if just to read the excellent explanation of how Public Key cryptography works. *The Public Key* is available from:

```
George Foot
Waterfall, Uvedale Road,
Oxted, Surrey
RH8 0EW   United Kingdom
```

ABOUT THIS ISSUE

This issue was produced using Donald Knuth's amazing TEX typesetting program, with help from various style files, especially **multicol.sty**. The **.tex** file that produced this issue is about 60,000 bytes, and is edited with a simple ASCII text editor (MKS Toolkit's **vi**). The pcTEX version I have runs fine on my IBM–PC 8086 clone in 640K of memory !

The utilities **ptihp** and **ptispool** were used to print this on a Hewlett-Packard LaserJet IIIP, a 300 dpi laser printer. I've retired my HP DeskJet after three years of faithful service, an replaced it with the LaserJet for about $1100 (US). Besides the usual outstanding HP quality, it gives me scalable fonts and many programmable features, including HP's PCL 5 Printer Language. Highly recommended !